2 Creating Simple Agent-Based Models

One of my central … tenets is that the construction that takes place "in the head" often happens especially felicitously when it is supported by construction of a more public sort "in the world"—a sand castle or a cake, a Lego house or a corporation, a computer program, a poem, or a theory of the universe. Part of what I mean by "in the world" is that the product can be shown, discussed, examined, probed, and admired. It is out there.

—Seymour Papert (1991)

It can scarcely be denied that 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.

—Albert Einstein (1933, p. 165)

In this chapter, we will learn to construct a few simple agent-based models. These simple models, sometimes referred to as "toy models," are not meant to be models of real phenomena, but instead are intended as "thought experiments." They are offered, as Seymour Papert puts it, as "objects to think with" (1980). Our purpose is to show that it is relatively easy to create simple agent-based models, yet these simple models still exhibit interesting and surprising emergent behavior. We will construct three such models: "Life," "Heroes and Cowards," and "Simple Economy." All three models can be found in the chapter 2 folder of the IABM Textbook folder in the NetLogo models library.

If you have not yet completed the NetLogo tutorials, this would be a good time to do so. Although we review some of that material in this chapter, we go through it more quickly here and do not describe all the steps in detail.

Life

In 1970, the British mathematician John Horton Conway (described in Conway, 1976) created a cellular automaton that he called the "Game of Life." Martin Gardner (1970) popularized this game in his *Scientific American* column. Subsequently, millions of readers played the game.

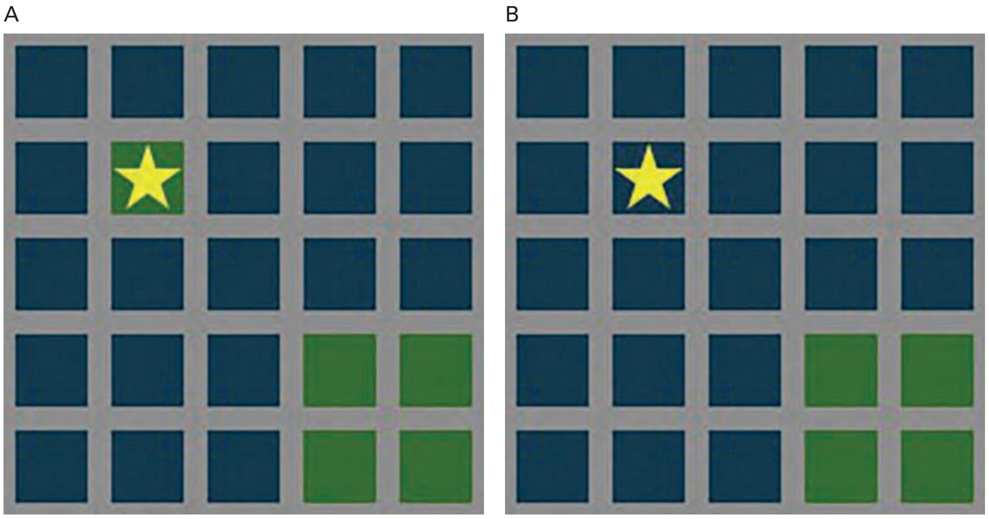


Figure 2.1
(A) Before. (B) After.

The game is played on a large grid, such as a checkerboard or graph paper. Let's say we are playing on a square grid with 51 squares[1] (or "cells") on a side. Each cell can be either "alive" or "dead." This is called the "state" of the cell. Every cell is surrounded by eight "neighbor" cells. The grid is considered to "wrap around" so that a cell on the left edge has three (3) neighbor cells on the right edge and, similarly, a cell on the top edge has three (3) neighbor cells on the bottom edge. There is a central clock. The clock ticks establish a unit of time. In the game of Life, the unit of time is called a generation. More generally, in agent-based models, the unit of time is referred to a tick. Whenever the clock ticks, each cell updates its state according to the following rules:

Each cell checks the state of itself and its eight neighbors and then sets itself to either alive or dead. In the rule descriptions that follow, blue cells are "dead," green cells are "alive" and the yellow stars indicate the cells affected by the rule described.

(1) If the cell has less than two (2) alive neighbors, it dies (figure 2.1).
(2) If it has more than three (3) alive neighbors, it also dies (figure 2.2).
(3) If it has exactly two (2) alive neighbors, the cell remains in the state it is in (figure 2.3).
(4) If it has exactly three (3) alive neighbors, the cell becomes alive if it is dead, or stays alive if it is already alive (figure 2.4).

1. The standard setup for the NetLogo grid is to have an odd number of cells, so that there is always a center cell. Other setups are possible and are described in chapter 5.

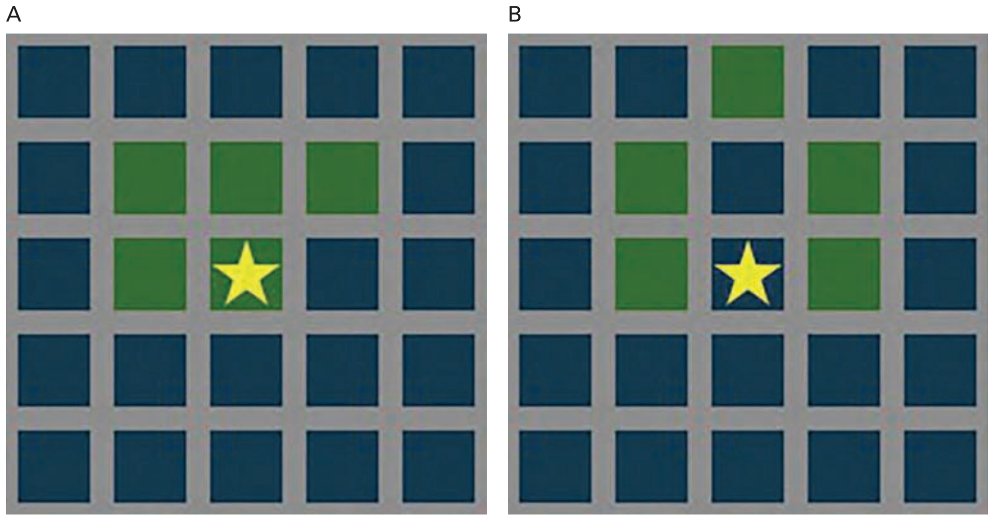


Figure 2.2
(A) Before. (B) After.

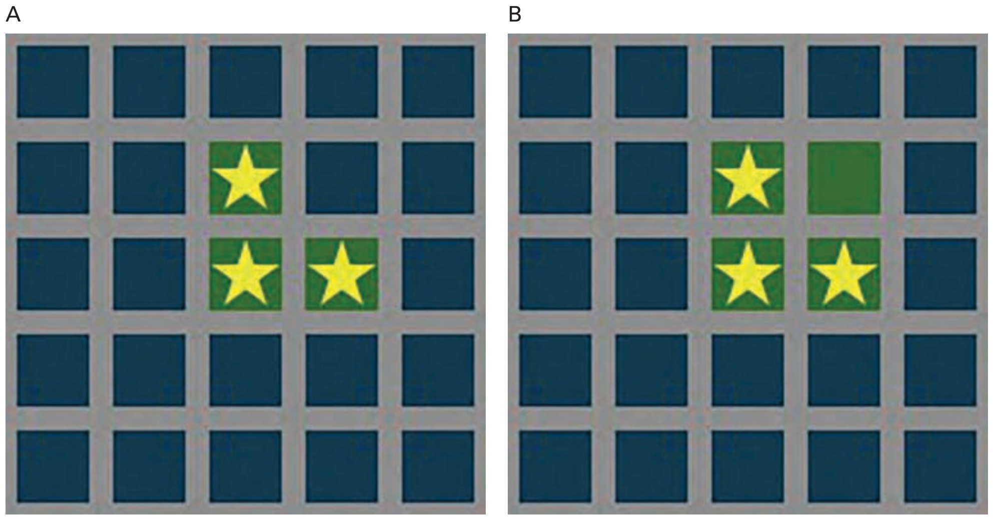


Figure 2.3
(A) Before. (B) After.

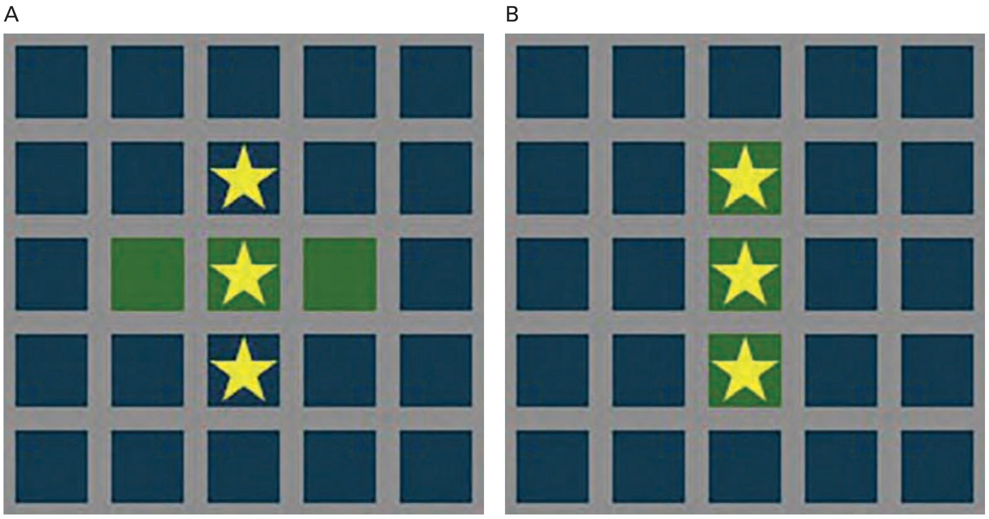


Figure 2.4
(A) Before. (B) After.

Since Gardner's publication of Conway's Game of Life, many people have explored it and have been surprised by the wide diversity of shapes and patterns that "emerge" from these simple rules. We will now use NetLogo to construct the Game of Life.

We will begin by reviewing the basic NetLogo elements. We start by opening the NetLogo application. The application opens with a blank interface with a large black square in it (see figure 2.5). The black square is the known as the "view" and is the area in which we will play the Game of Life. The surrounding white area is known as the "interface" and is where we can set up user-interface elements such as buttons and sliders. Right-click on the view and select "edit" from the drop-down menu. You will see the Model Settings dialog (figure 2.6) where we can configure some basic settings for our NetLogo model.

Every NetLogo model consists of three tabs.[2] The tab that you are looking at now is the *Interface* tab, where we work with widgets and observe model runs. Next we will work with the *Code* tab, where we write the model procedures.

A third very important tab is the *Info* tab. In this chapter, we will not work with the *Info* tab in detail, but it is an important part of any model. This is where model authors put the information about their models. Details on how the Info tab is structured can be found in the "Sections of the Info Tab" box in chapter five. The Info tab is a very useful resource for exploring a NetLogo model and we recommend that Textbook readers read them carefully when working with models and take the time to write good Info tabs for models you create.

2. In some versions of NetLogo, there is a fourth "Review" tab.

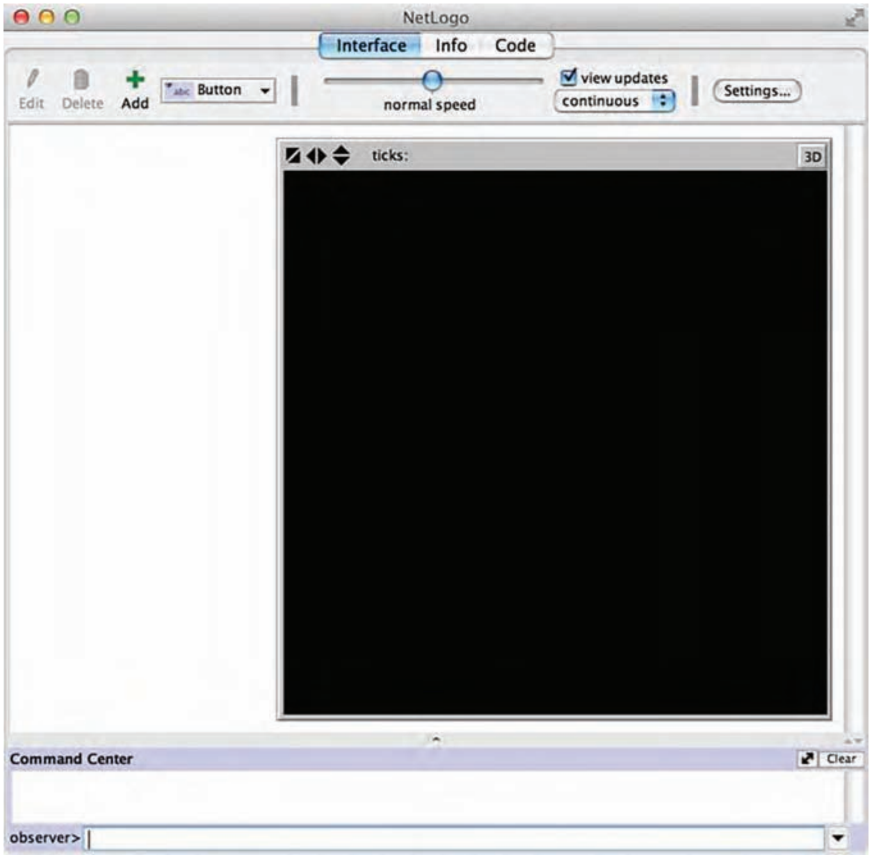
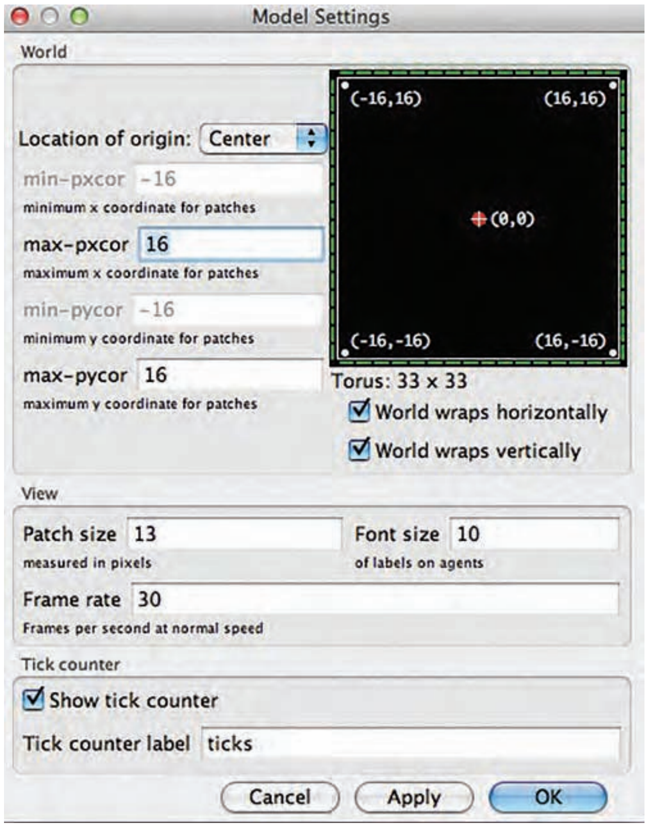


Figure 2.5
The NetLogo application at startup.

Now let us return to the Interface tab, and begin developing our Life-simple model. The view is composed of a grid of cells known, in NetLogo parlance, as patches. Click Settings in the toolbar of the Interface tab. By default, the view origin is at the center of the view and its current maximum x-coordinate and y-coordinate is 16. To begin developing our Life-simple model, we will change the values of MAX-PXCOR and MAX-PYCOR to 25, which will give us a grid of 51 by 51 patches, for a total of 2,601 patches. This creates a much larger world enabling more space to create the elements of the Game of Life. To keep the view a manageable size on the screen, we will change the default patch size from 13 to 8. As the Game of Life is played in a wrapping grid, we keep the wrapping check-boxes checked (as they are by default, we will explain these in more detail in chapter 5). We can now click OK and save these new settings. (See figure 2.7.)



Model Settings

World

Location of origin: Center

min-pxcor -16
minimum x coordinate for patches

max-pxcor 16
maximum x coordinate for patches

min-pycor -16
minimum y coordinate for patches

max-pycor 16
maximum y coordinate for patches

Torus: 33 x 33

✓ World wraps horizontally

✓ World wraps vertically

View

Patch size 13
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

✓ Show tick counter

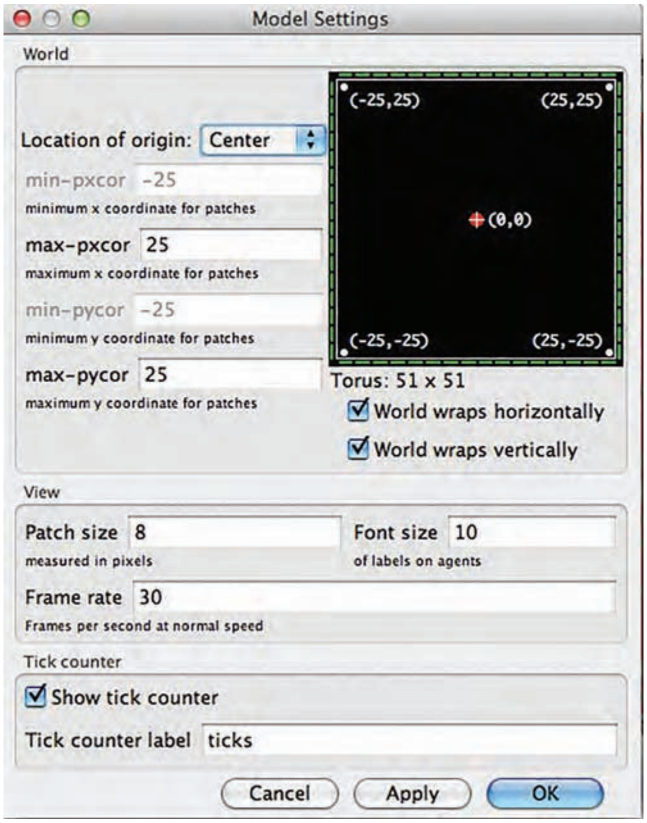
Tick counter label ticks

Figure 2.6
The NetLogo Model Settings dialog before adjusting the settings.

The Game of Life is played on a grid of cells, so we will consider each NetLogo patch as a different cell. Life has two kinds of cells in it, “live” cells and “dead” cells. We choose to model live cells as green patches and dead cells as blue patches. Once we have thought through what the model will look like we still need to create the model instructions. To do this, we will need to write NetLogo instructions (or code) in the Code tab. We select the Code tab and begin to write our code. NetLogo code takes the form of modules known as “procedures.” Each procedure has a name and begins with the word TO and ends with the word END.

Our Life model (Life-Simple in the IABM Textbook folder of the NetLogo models library) will consist of two procedures: SETUP, which initializes the game, and GO, which advances the clock by one tick.

We create the SETUP procedure as follows:


Model Settings

World

Location of origin: Center

min-pxcor -25
minimum x coordinate for patches

max-pxcor 25
maximum x coordinate for patches

min-pycor -25
minimum y coordinate for patches

max-pycor 25
maximum y coordinate for patches

(-25,25) (25,25)
(0,0)
(-25,-25) (25,-25)

Torus: 51 x 51

☑ World wraps horizontally

☑ World wraps vertically

View

Patch size 8
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

☑ Show tick counter

Tick counter label ticks

Figure 2.7
After configuring the Model Settings dialog for the Life Simple model.

```
to setup
    clear-all
    ask patches
;; create 10% alive patches
[
       set pcolor blue  ;; blue cells are dead
       if random 100 < 10
           [set pcolor green] ;; green cells are alive
    ]
    reset-ticks
end
```

This code has three sections. The first section is one line, the command CLEAR-ALL, which clears the view (setting the color of all its patches to the default color black) in case there are leftover elements from our previous play of the game. Thus, we can run multiple

Box 2.1
Wrapping

One thing to note is that the world in the Game of Life model has been set up to "wrap." In a wrapping world, a patch on the left-edge of the world is neighbor to three patches on the right edge of the world. It is often convenient to set the world to wrap "without boundary conditions," avoiding special code for patches at the edge.

A

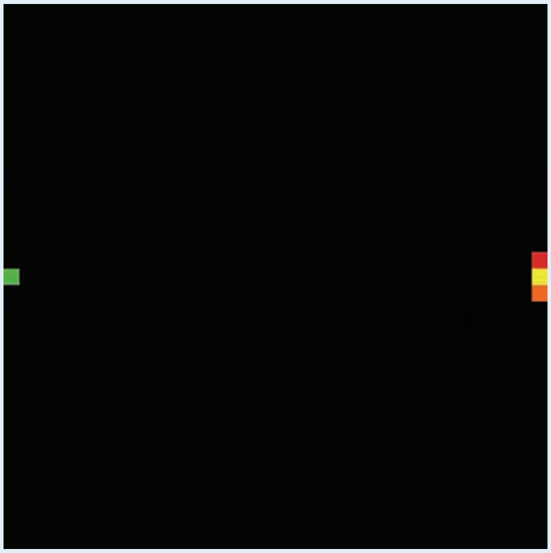
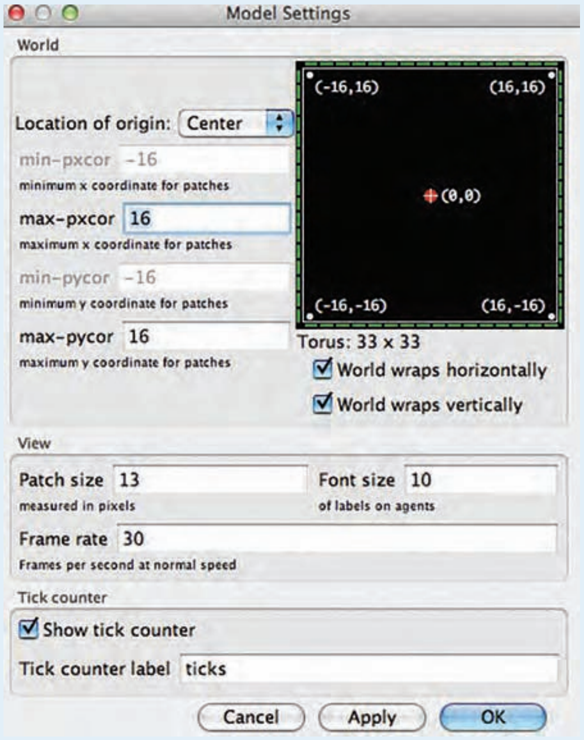


Figure 2.8
(A) With wrapping turned on, the green colored patch at the left edge of the world is a neighbor of the red, yellow, and orange colored patches at the right edge of the world. (B) The Model Settings dialog with the world set to wrap vertically and horizontally.

To set the world to wrap, select the view object and edit it. You will bring up the "Model Settings" dialog. The dialog has two checkboxes for wrapping. Checking the first one sets the world to wrap horizontally, as in the figure above. Checking the second checkbox sets the world to wrap vertically, so patches at the top of the view are neighbors with patches at the bottom. In ABM it is often useful to use a wrapping world, so NetLogo's default setting allows for the world to wrap.

Box 2.1
(continued)

B



World

Location of origin: Center

min-pxcor -16
minimum x coordinate for patches

max-pxcor 16
maximum x coordinate for patches

min-pycor -16
minimum y coordinate for patches

max-pycor 16
maximum y coordinate for patches

Torus: 33 x 33

☑ World wraps horizontally

☑ World wraps vertically

View

Patch size 13
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

☑ Show tick counter

Tick counter label ticks

Figure 2.8
(continued)

versions of our game sequentially without having to close NetLogo and open it back up. The second section issues commands to (or makes requests of) all of the patches. In NetLogo, we are polite in our interactions with agents, so to issue commands to the patches we use the form ASK PATCHES. We then enclose our requests to the patches in brackets. However, do not mistake our politeness—the patches have no choice but to do as they are asked. As a result, you will often find that in this book, we will slip interchangeably between the language of commands and requests as synonymous. In the second section of our SETUP code, there are two commands/requests. The first asks each patch to set its color (PCOLOR which stands for patch color) to blue, making all the cells dead. When creating agent-based models, it is useful to think of commands to agents in an agent-centric way. This will help you to understand how the model works, as described in the box that follows:

Box 2.2
Agent-Centric Thinking

When asking the agents to execute some commands, it is useful to think in an agent-centric way. That is, thinking from the point of view of the agent. Instead of thinking from the perspective of telling all the agents what to do, think of each agent as individually receiving the instructions and behaving accordingly.

For example, if we issue the command:

```
ask patches [if pxcor < 0 [set pcolor blue]]
```

One way of thinking about it is that we are commanding all the patches whose x-coordinate is less than zero to turn blue. The agent-centric way to think about the same command is to think about it from the patch's perspective; each patch wakes up when it "hears" ASK PATCHES and listens for its command. In this case, the patch asks itself, "Is my x-coordinate less than zero? If it isn't, I won't do anything, if it is, I'll set my color to blue."

At first, thinking in this way may seem unnatural, but as you become more experienced at agent-based modeling, you will see the value of agent-centric thinking.

The second request is a little trickier. It asks some of the patches to turn green; that is, it seeds the game with some live cells. The way it does this is also best approached through agent-centric thinking. Each patch executes the code RANDOM 100, which can be thought of as each patch rolling a hundred-sided die.[3] If a patch rolls any number less than ten (10), it turns green. Since the dice throws are random, we cannot know exactly which patches will roll a number less than ten (10), hence do not know exactly which particular cells will turn green. However, we can expect that approximately 10 percent of the cells will turn green. NetLogo does have a way of telling exactly 10 percent of the patches to turn green. However, in designing ABMs, we often use a probabilistic procedure because most natural phenomena we model have some stochasticity or variability, so having our procedure use a randomized process is a closer model of reality.

The third section of our SETUP procedure is also just one line, RESET-TICKS. This command resets the NetLogo clock.

Returning to the Interface tab, we create a button and name it SETUP.[4] Pressing this button will now cause the SETUP procedure to run. Try it a few times. Each time you should see a different set of green cells on a blue background (see figure 2.9).

3. The metaphor of a die is one way we can understand the action of the "random" primitive in NetLogo. Many other metaphors are commonly used. Another useful one is a spinner, with the numbers 0 to 99 on the outside. You spin the spinner arrow and it turns till it points to one of the one hundred numbers.

4. If you don't remember how to create *Interface* elements from the tutorials, please refer to the *Interface* Guide of the NetLogo Programming Manual (http://ccl.northwestern.edu/netlogo/docs/interface.html).

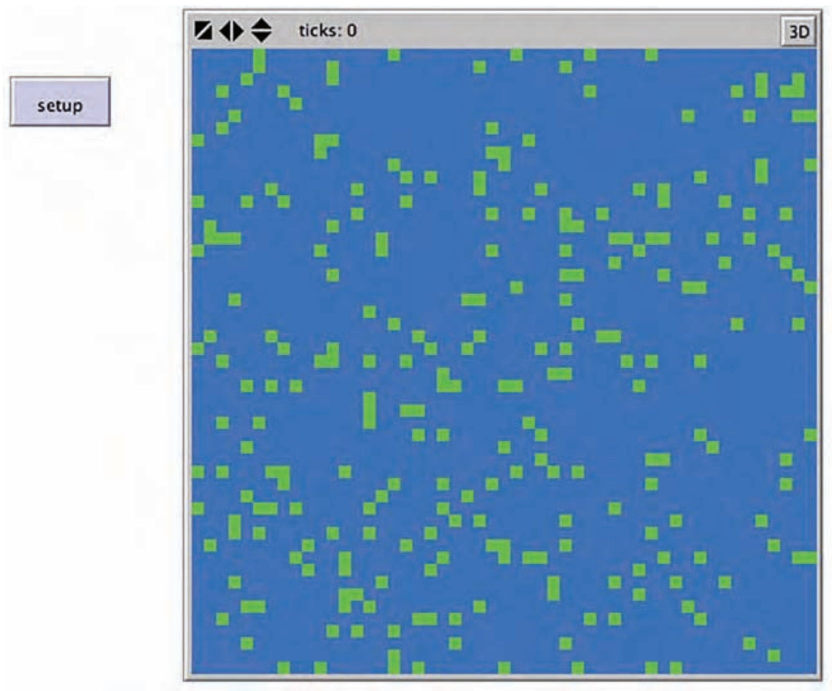


Figure 2.9
A typical random initial configuration of the Game of Life. Approximately 10 percent of the cells are alive and colored green.

Now let's take a look at the GO procedure, which advances the clock for one tick.

```
to go
    ask patches [
           ;; each patch counts its number of green neighboring patches
           ;; and stores the value in its live-neighbors variable
       set live-neighbors count neighbors with [pcolor = green]
       ]
    ask patches [
           ;; patches with 3 green neighbors, turn (or stay) green
       if live-neighbors = 3 [ set pcolor green ]
           ;; patches with 0 or 1 green neighbors turn (or stay) blue
           ;; from isolation
       if (live-neighbors = 0) or (live-neighbors = 1) [ set pcolor blue ]
           ;; patches with 4 or more green neighbors turn (or stay) blue
           ;; from overcrowding
       if live-neighbors >= 4 [set pcolor blue]
           ;; patches with exactly 2 green neighbors keep their color
    ]
    tick
end
```

Box 2.3
Neighbors

In agent-based modeling, we often model local interactions. This is accomplished by having cells in the grid talk to their neighbor cells. Most often, we are using two-dimensional grids or lattices with square cells (though later in this textbook we will see three-dimensional grids and two-dimensional grids of triangular or hexagonal lattices). In a square lattice, there are two commonly used neighborhoods: the von Neumann neighborhood and the Moore neighborhood. A cell's *von Neumann neighborhood* consists of the four cells that share an edge with it, the cells to the north, south, east and west (the green squares in the right figure that follows). A *Moore neighborhood* consists of the eight cells that touch it, adding the cells to the northeast, southeast, northwest, and southwest (the entire green area in the left figure that follows). In NetLogo, the primitive NEIGHBORS refers to a patch's Moore neighbors, and NEIGHBORS4 refers to a patch's von Neumann neighbors.

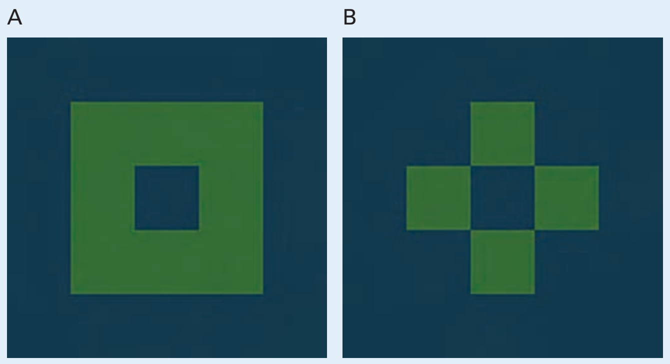


Figure 2.10
(A) Moore Neighborhood. (B) von Neumann Neighborhood. This will be further elaborated in chapter 5.

The GO procedure consists of three sections: the first two ask the patches to "behave" (i.e., they specify the rules that we described at the beginning of this section), and the third section simply advances the clock by one "tick." This third section is the standard way to end a GO procedure. First, let's examine the first two sections. The first section asks all the patches to execute one command. We assign to each patch a LIVE-NEIGHBORS variable that we will need to "declare"; that is, we will need to tell NetLogo about the existence of this patch variable. The command asks each patch to store (SET) a value in the LIVE-NEIGHBORS variable. The value to be stored is COUNT NEIGHBORS WITH [PCOLOR = GREEN]. That expression tells the patch to examine its eight neighbor patches and count

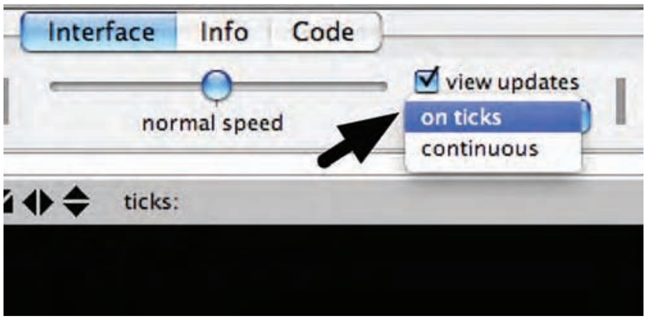


Figure 2.11
Pull-down menu to set display updating frequency.

how many are green. Thus, at the end of the first section, each patch has a variable LIVE-NEIGHBORS that has the value of the number of its live neighbors.

To declare the LIVE-NEIGHBORS variable, we add the following line to the top of the Code tab:

```
patches-own [live-neighbors]
```

The second section of the GO procedure consists of three "if" conditions.

The first asks each patch to check if its value for LIVE-NEIGHBORS is three, in other words, if it has exactly three live neighbors. If it does, then, according to the Life rules, the cell should be alive, so the patch turns green to symbolize birth (if it already was green, it remains so). If the patch's value for LIVE-NEIGHBORS is either zero or one, the patch turns blue to symbolize death from isolation. If the patch's value for LIVE-NEIGHBORS is four or more, the patch also turns blue to symbolize death from overcrowding. That is the end of the second section. Notice that the second section does not tell the patch what to do if it has exactly two (2) live neighbors. Since the patch was not asked to do anything in that case, it will not do anything, so the patch will remain the color it was before this code was executed. That ends the GO procedure.

Returning to the interface, we now create a new button called GO. We set up the GO button to call the GO procedure just like we did for the SETUP button. We also need to make sure the "view updates" pull-down is set to "on ticks"[5] (see figure 2.11). This means we see the world only after a full iteration of the GO procedure has completed.

5. For most NetLogo models, it is best to set this pull-down to "ticks," which will cause the view to update on each tick. But when exploring in the command center, it is often useful to set it to "continuous," so you can view behaviors that might occur between ticks. For a detailed discussion of the differences between continuous and tick-based view updates, see the VIEW UPDATES section of the PROGRAMMING GUIDE in the NetLogo users' manual, http://ccl.northwestern.edu/netlogo/docs/programming.html#updates.

Box 2.4
Variables

In most programming languages, the term *variable* is used to refer to a symbol that can have many different possible values. That is also true in NetLogo, but in NetLogo there is a distinction between a global variable and an agent variable. A *global variable* has only one value regardless of the agent that is accessing it. All agents can access that variable. In contrast, an *agent variable* can have different values for each agent of a certain type. The core agent-types in NetLogo are turtles, patches and links. Each of these agent-types can have an agent-variable, namely a turtle, patch, or link variable respectively. Each turtle has its *own* value for every turtle variable. The same goes for patches and links.

Some variables are built into NetLogo. For example, all turtles and links have a COLOR variable, and all patches have a PCOLOR variable. If you set the variable, the turtle or patch changes color. Other built-in turtle variables including XCOR, YCOR, and HEADING. Other built-in patch variables include PXCOR and PYCOR.

You can also define your own variables. You can make a global variable by adding a switch, slider, chooser, or input box to your model interface, or by using the GLOBALS keyword at the beginning of your code.

For instance, if you want to keep track of a global score for your model that all agents can access, you can create a global variable in which to store it by writing the following line at the top of your Code tab:

```
globals [score]
```

You can also define new turtle, patch, and link variables using the TURTLES-OWN, PATCHES-OWN and LINKS-OWN keywords. If you write:

```
turtles-own [speed]
```

each turtle will get a speed variable and each turtle can have a different value for speed. Similarly, if you write:

```
patches-own [friction]
```

then each patch can have a different friction with which to slow down the turtle's speed.

There are also local variables defined by the "let" primitive. We will explore their use in the next chapter.

See http://ccl.northwestern.edu/netlogo/docs/programming.html#variables in the NetLogo user's manual programming guide for more information on variables.

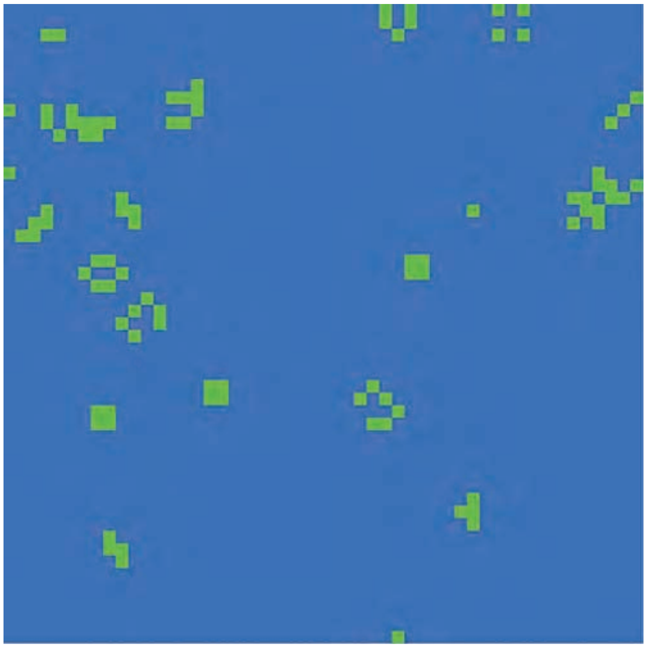


Figure 2.12
The Life-simple model view after two ticks.

When we press the GO-ONCE button once, the colors of the patches change according to the rules of the Game of Life. Each time you press it, you get another generation of the Game of Life. You'll also notice that on the gray bar above the view, the ticks counter will advance by one, recording how many times the clock has ticked. (See figure 2.12.) The initial random distribution of live cells morphs into a small set of live-cell configurations or "shapes."

When you press the GO button several times, you will notice that the original random distribution of green patches will morph into a set of structures. One reason for this is that the isolated green patches die from "loneliness" according to the first of the Life rules (cells with less than two neighbors die), leaving only clusters of cells to live on. Some of these structures remain stable over several generations.

As you can see from letting this model run, there are many different structures that can emerge in the Game of Life. Players of the Game of Life are frequently astounded by the amazing diversity and complexity of the structures that can emerge from different initial configurations. The model we have created always initializes the game to start with a random 10 percent of live cells. But this is an unnecessary constraint. Any initial configuration of live cells can be played. (See the Life model in the Computer Science section of the NetLogo models library for a version that allows you to "paint" the live cells.)

Box 2.5
Inspectors/Agent Monitors

One way to keep close track of the state of a live cell is to use an inspector, also known as an *agent monitor*. You can use an agent inspector/monitor to view the properties of any agent type. Let's look at an example of using a patch inspector. Consider the block shape we saw in figure 2.1.

We can inspect the state of any patch in the world. Let's inspect the green colored patch on the lower left. We can inspect it either by right-clicking on the patch and selecting INSPECT or by issuing the command INSPECT and specifying which patch we wish to inspect. When we inspect the lower left green colored patch, we will see an inspector window appear:

The fields in the inspector include all the built-in patch variables (PXCOR, PYCOR, PCOLOR, PLABEL, and PLABEL-COLOR) as well as any patch variables we have declared. In this case, you can see the LIVE-NEIGHBORS variable that we declared earlier. Agent monitors have many capabilities. To learn more about their capabilities, see http://ccl.northwestern.edu/netlogo/docs/interface.html#agentmonitors in the NetLogo interface guide.

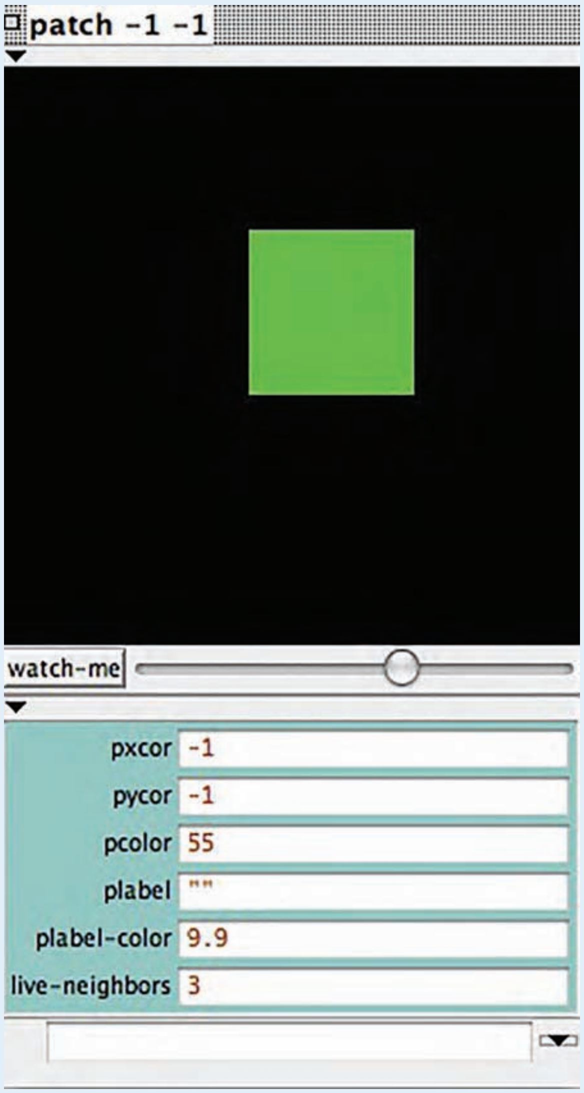
Before we go further with this model, we will make one more small change to the code. Many people find the combination of the default blue and green colors that come with NetLogo to be a bit hard on the eyes. So, we will modify the color of the dead cells throughout the model to a different shade of blue, as described by the expression "blue—3." (All NetLogo colors are numbers and can be added to or subtracted from to darken or lighten their shade. For a comprehensive discussion of the NetLogo color scheme, see ccl.northwestern.edu/netlogo/docs/programming.html#colors.) After doing that, now edit the GO button and check the "forever" box in the top middle. Now when you press the GO button, it will continue advancing the clock until you press the button again.[6] In this way, you can quickly watch thousands of generations of the game. Each initial configuration leads to different life trajectories. Some trajectories quickly stabilize in just a few generations, and some go on for much longer. It is worthwhile to examine many different trajectories and see what patterns you can detect. (See figure 2.14.)

There are at least three general classes of stable Life patterns: Still Lifes, Oscillators, and Spaceships.

1. Still Lifes: *Still Life shapes* are stable unless other shapes collide with them. The most common still life shape is the block, which keeps its shape from one generation to the

6. If you still want a button that only executes one step at a time, then you can add another button called GO-ONCE and not click the forever button. This is what we did in the Life Simple model included in the IABM Models Folder.

Box 2.5
(continued)



pxcor -1
pycor -1
pcolor 55
plabel ""
plabel-color 9.9
live-neighbors 3

Figure 2.13
(Top) A set of patches in the Game of Life. (Bottom) An inspector window for the patch at (–1,–1).

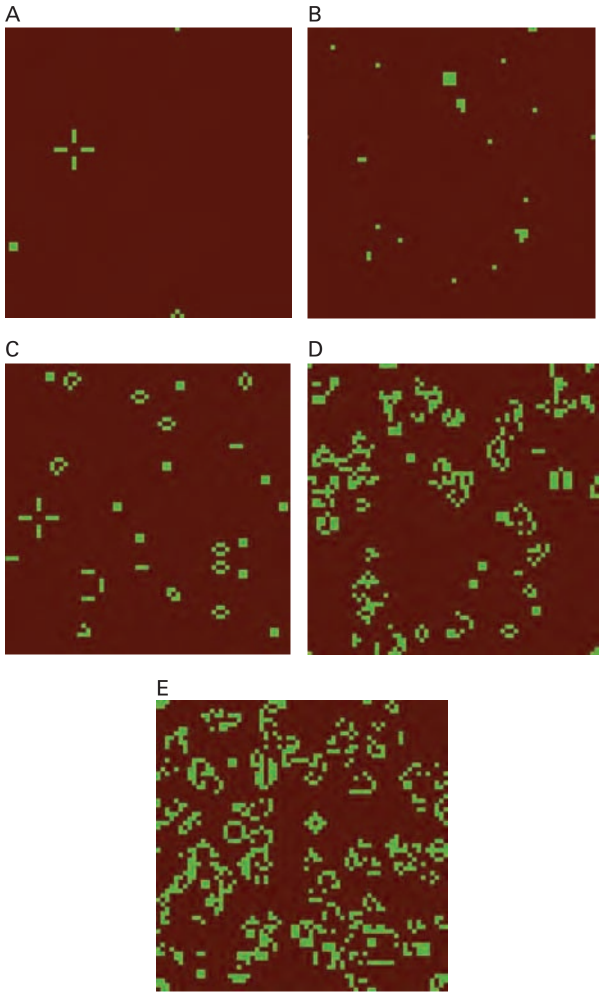


Figure 2.14 (A–E)
Some sample trajectories of the Life-Simple model (with dark red for the dead cells). Some initial conditions stabilize in just a few generations. Others take many thousands. Final states will typically include some oscillators and spaceships (see figures 2.15 and 2.16) that will continue to cycle through their different states.

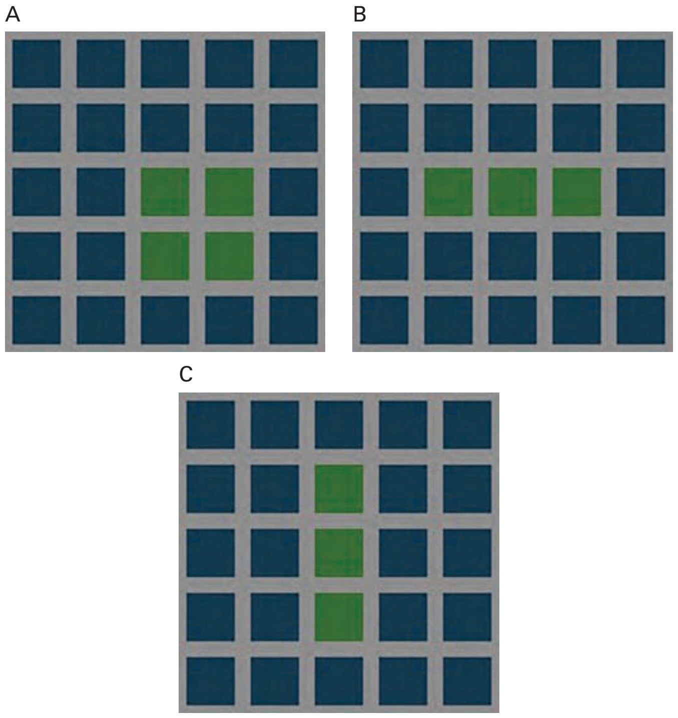


Figure 2.15
(A) A four-cell block shape is a "still-life"; it doesn't change from one Life generation to the next. Each live cell (green) has exactly three live neighbor cells, so it stays alive in the next generation. Each dead cell (blue) has one or two live neighbor cells, so it stays dead in the next generation. (B, C) The two states of the "blinker." It oscillates between these two states, alternating vertical and horizontal orientation from one generation to the next.

next (figure 2.15A). Other well-known still-lifes are the beehive, the loaf, the boat and the ship (see exploration 5).

2. Oscillators: *Oscillator patterns* repeat over time. They may have one shape in the first tick (t) and another shape in the next tick (t + 1), eventually getting back to the original shape after *n* ticks (*n* is known as the period of the oscillator). For instance, a blinker is a period two (2) oscillator. It consists of a configuration of three cells (either up and down or left and right) that rotates between horizontal and vertical orientations. (See figure 2.15B,C.)

3. Spaceships: Some Life shapes can move across the Life world. These are called *Spaceships*. For example, the "glider" is composed of five (5) cells that form a small arrowhead shape (see figure 2.16A).

At each tick, the glider moves from one state to the next. After four (4) ticks (remember, each tick represents a "generation" of the Game of Life), the glider is back to its original

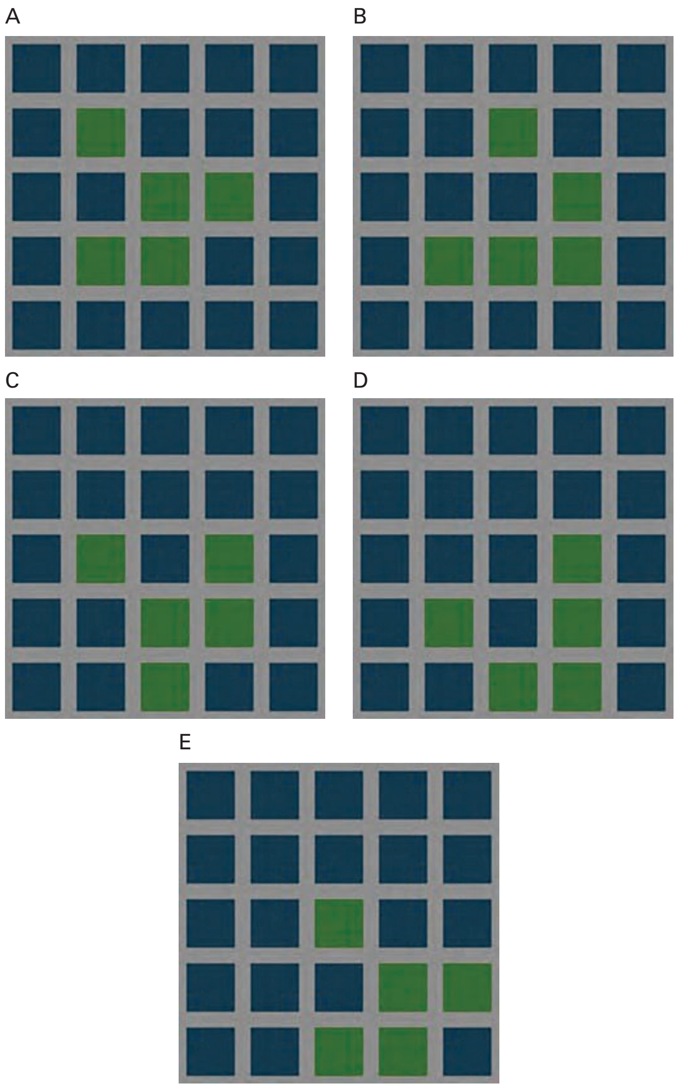


Figure 2.16
(A) The initial glider shape. (B–E) The four states of the glider. The glider changes from one state to the next. After traversing the four states, it comes back to its original shape, but has moved one step right and one step down (rightmost panel).

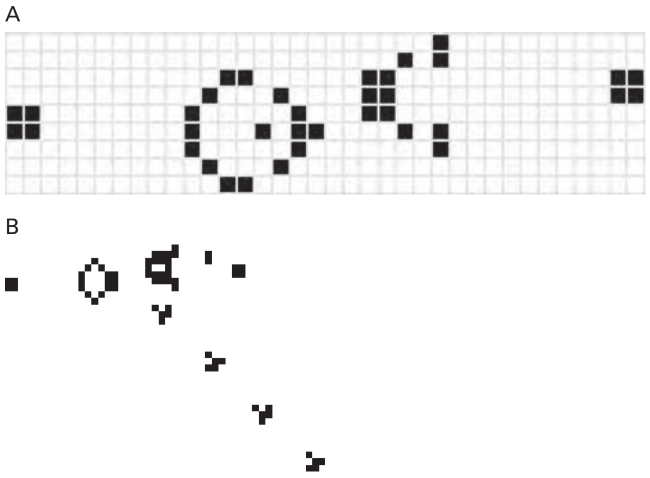


Figure 2.17
(A) Initial state of Gosper's Glider Gun. (B) The Glider Gun in action. As discussed earlier, the variety and diversity of forms that can arise in the game of Life is astonishing. See e.g., http://www.youtube.com/watch?v=C2vgICfQawE&feature=fvwp for a video of a range of forms.

shape but is moved over one cell to the right and one cell down. The glider acts as a stable shape "moving" across the grid. (See figure 2.16B–E.)

In addition to the three stable classes in the game of Life, there are also "guns." A *gun* is a pattern with a main part that repeats periodically, like an oscillator, and that also periodically emits spaceships. There are then two periods that may be considered: the period of the spaceship output, and the period of the gun itself, which must be a multiple of the spaceship output's period. It is possible to create guns that emit gliders, known as "glider guns." In 1970, the self-described "hacker" Bill Gosper discovered the first glider gun (see figure 2.17). The discovery of the glider gun eventually led to the proof that Conway's Game of Life was universal and could function as a Turing machine.[7]

Since Conway's publication of the Life model, the field of cellular automata has continued to develop and grow. Researchers have studied and simulated cellular automata in one (1), two (2), and three (3) dimensions as well as higher dimensional cellular automata. Evidence of cellular automata-like mechanisms has been abundantly found in nature, including the shapes of shells and flowers, the colorations and striations of animals, and the structure of organisms such as fibroblasts. (See figures 2.18–2.20.)

In 1969, computer pioneer Konrad Zuse published *Calculating Space*, which contained the revolutionary proposition that the physical laws of the universe are discrete, and that

7. For a fascinating video of the Game of Life simulating itself, see http://www.youtube.com/watch?v=xP5-iIeKXE8.

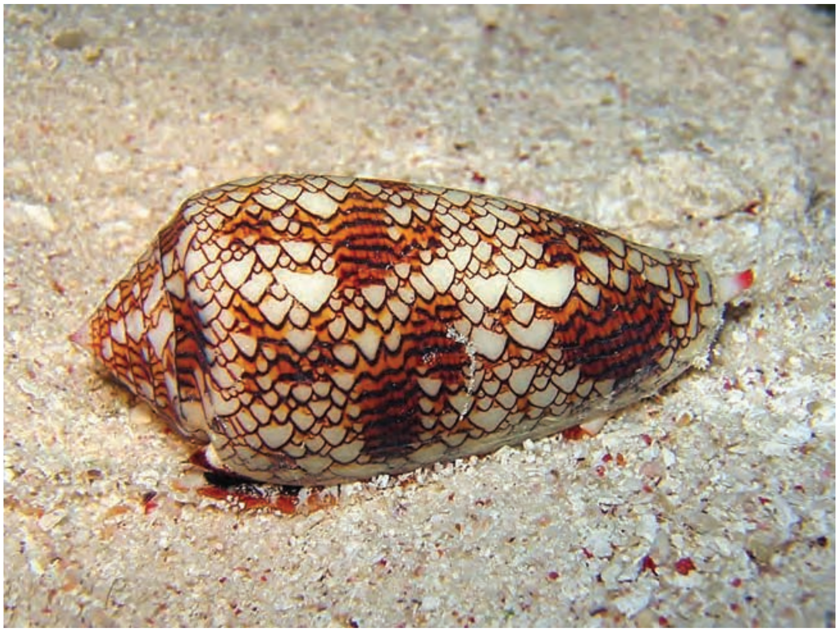


Figure 2.18
Textile cone and snail shell (http://en.wikipedia.org/wiki/Puka_shell).

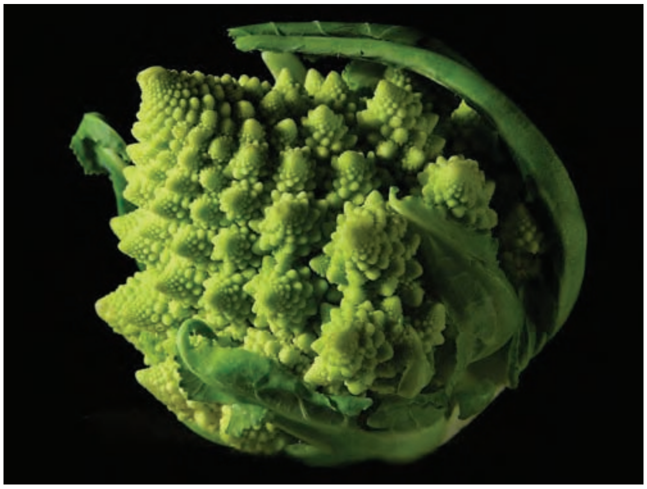


Figure 2.19
Romanesco broccoli (http://en.wikipedia.org/wiki/File:Fractal_Broccoli.jpg).

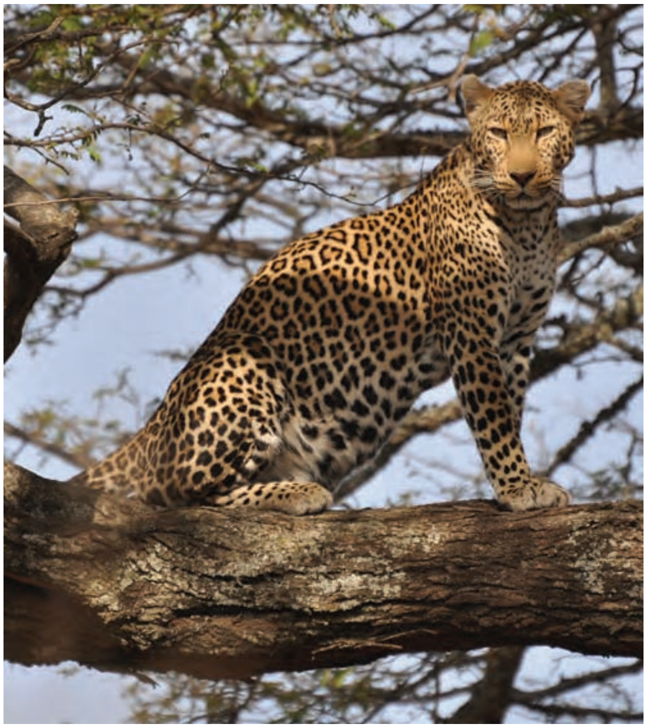


Figure 2.20
Spotted leopard. Some patterns in nature that can be generated by a cellular automaton (http://commons.wikimedia.org/wiki/File:Leopard_standing_in_tree_2.jpg).

the universe can be viewed as the output of a computation on a giant cellular automaton. In writing this book, Zuse founded the field of digital physics. In the words of another prominent digital physicist, Ed Fredkin (1990): "We hypothesize that there will be found a single cellular automaton rule that models all of microscopic physics; and models it exactly. We call this field DM, for digital mechanics."

The prodigal physicist and mathematical software entrepreneur Stephen Wolfram (1983) wrote a paper that investigated a basic class of elementary cellular automata. These CAs were the simplest possible. They were one-dimensional, and each cell could only take in to account one neighbor on either side of it. Nevertheless, they exhibited a remarkable amount of complexity. The unexpected complexity of the behavior of these elementary CAs caused Wolfram, much like Zuse before him, to hypothesize that complexity in nature may be due to simple CA-like mechanisms. Wolfram classified the 256 possible rules and categorized them into four behavioral regimes (homogenous, periodic, chaotic and complex). He showed that the behavior of some rules (chaotic) was indistinguishable from randomness—that is, that when you apply the rules repeatedly the patterns they generate look like random noise.[8] He also found that other rules generated behavior that led to patterns that were intricately complex[9] and that CAs could generate many patterns found in nature as in the previous figure (Wolfram, 2002). Matthew Cook, a research assistant with Wolfram, proved that one of Wolfram's 1D CAs (rule 110) is universal—that is, any computation that can be done by any computer can also be done by that cellular automaton (Cook, 2004). In 2002 Wolfram published a voluminous and controversial book, *A New Kind of Science*, which argues that the discoveries about cellular automata have major significance for all disciplines of science.

At this time, you may want to open the Life Simple model from the chapter 1 folder of the IABM Textbook folder in the NetLogo models library. There are also several other cellular automata models in the models library that you can peruse and explore.

Heroes and Cowards

We are now going to create a model of another game, a game we call Heroes and Cowards. The origins of this game are difficult to pin down. In the 1980s and 1990s, an Italian troupe called the Fratelli theater group used the game as an improvisation activity. In the 1999 conference Embracing Complexity, held in Cambridge, Massachusetts, the Fratelli group ran this game with the conference participants, and this seems to be the first recorded public instance of the game. The game is also related to a game called "Party Planner" proposed by A. K. Dewdney in his "Computer Recreations" column in *Scientific American* (September 1987) and reprinted in the book *The Magic Machine* (1990).

To play the game you need a group of people. The game starts by asking each person to pick one other person to be their "friend" and another to be their "enemy." There are two stages to the game. In the first stage, everyone is told to act like a "coward." To act like a coward you move so as to make sure your friend is always between you and your enemy (effectively, hiding from your enemy behind your friend in a cowardly manner). When people played this stage of the game, the center of the room became empty as people "fled" from their enemy. In the second stage of the game, people were asked to behave as heroes, that is, to move in between their friend and enemy (effectively protecting your friend from your enemy in a heroic manner). When people played the second stage, the center of the room got very crowded. It was a dramatic difference and generated laughter and curiosity.

From the conference, the game spread to the nascent community of complexity scholars, since it was a nice example of surprising emergent behavior. Eric Bonabeau, the president

8. This proof lent more credence to some claims of Digital Mechanics, which could now show that the sources of randomness found in the universe could be generated by deterministic automata.

9. The NetLogo models library contains several models of cellular automata, including versions of Wolfram's 1D CA models. See, for example, CA 1D elementary (*Wilensky, 1998*) in the Computer Science section of the library.

of Icosystems and a well-known complexity scientist and agent-based modeler, created a version of the game while at BIOS (a company with complexity ties) in 2001. Later, Bonabeau created an agent-based model of the game that runs on the Icosystems website (Bonabeau, 2012). Stephen Guerin, the president of the Redfish group, also created an early version of the game and presented it in 2002 at the Twelfth Annual International Conference of the Society for Chaos Theory in Psychology & Life Sciences. Some images of the game being played by people (alongside the equivalent simulations) are in figure 2.21.[10]

Since then, many different versions of the game have appeared in various places within the complex systems community. Besides coding early versions, Eric Bonabeau, Stephen Guerin, and others have led people in playing it at conferences. A three-player variant of the game appears in the Systems Thinking Playbook (Sweeney & Meadows, 2010). The game does not always appear under the name "Heroes and Cowards"; in fact, it has appeared under many different names. Bonabeau calls it "Aggressors and Defenders" (Bonabeau & Meyer, 2001). We have also heard it called "Friends and Enemies" and "Swords and Shields." We prefer the name "Heroes and Cowards," since that most closely matches the agent behavior.

Building the model is fairly straightforward, so we will create a model of the game here and explore some variations of the original game. As in the LIFE model, we will create two main procedures: SETUP to initialize the model and GO to run it. This time, we will primarily use the turtle agents rather than the patches.

The SETUP procedure is as follows:

```
to setup
    clear-all
    ask patches [ set pcolor white ] ;; create a blank background
    create-turtles number [
       setxy random-xcor random-ycor

       ;; set the turtle personalities based on chooser
       if (personalities = "cowards") [ set color blue ]
       if (personalities = "heroes")  [ set color red ]

       ;; choose friend and enemy targets
       set friend one-of other turtles
       set enemy one-of other turtles
    ]
    reset-ticks
end
```

The SETUP procedure has four main sections. The first two should be familiar from the Game of Life. The view is cleared so as to start with an empty black screen area; then we

10. Pictures courtesy of Eric Bonabeau.

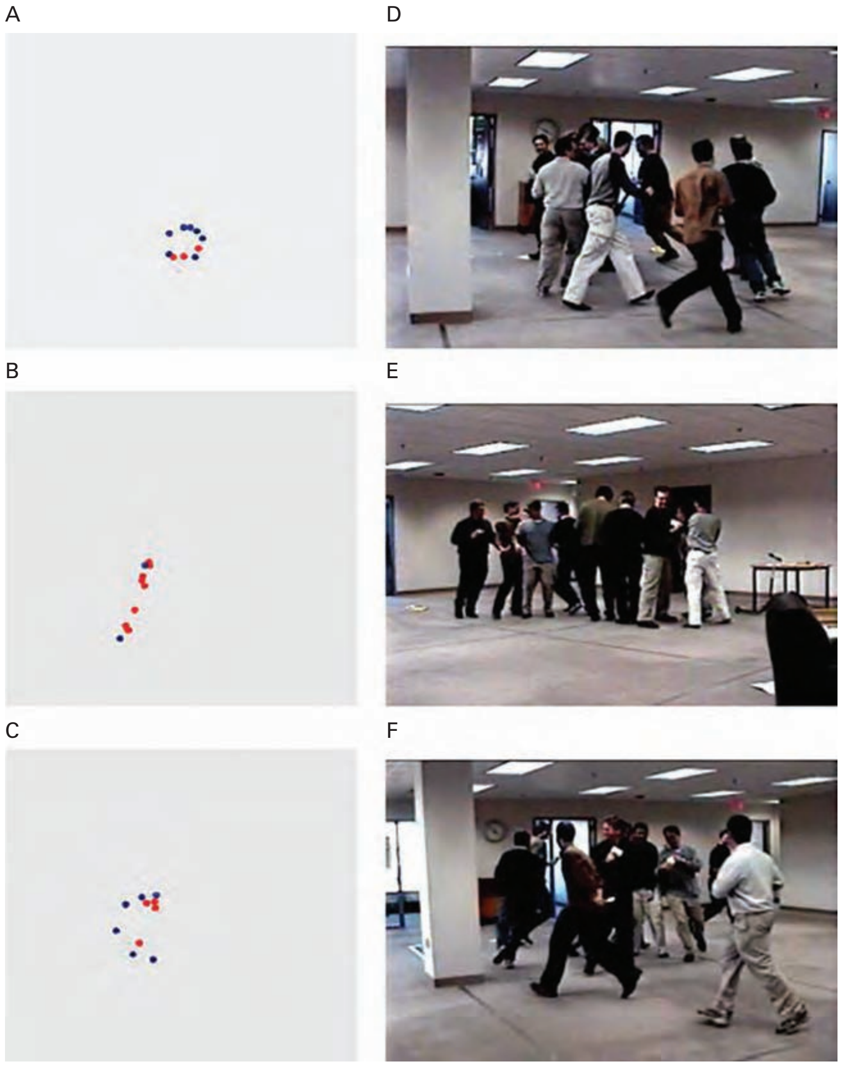


Figure 2.21
Three rules designed for swarms of ten agents, evolved using the IEC interface (A–C) and then given to a group of people (D–F). Rule “circle” (A,D makes all agents run around in a circle; rule “align” (B,E) made them form a straight line; and rule “Chinese streamer” resulted in a central cluster with a tail or “ribbon” circling behind. (From Bonabeau et al., 2003.)

color all the patches white in order to make it easier to observe the actions of the agents. You should also recognize the fourth block, RESET-TICKS, which initializes the NetLogo clock. The new and interesting action is in the third block.

The third block is one long command. The CREATE-TURTLES command creates a number of turtles and then gives them some commands. We will use the global variable NUMBER to designate the number of turtles we start with. We will control the variable with a slider in the interface. In this case, we set the NUMBER slider to 68 turtles (the size of a party or small conference) and give them some commands.[11] The first command we give the turtles is:

```
setxy random-xcor random-ycor
```

This command sets the XCOR and YCOR values of the turtles, which has the effect of scattering the turtle agents randomly about the view. The second command is:

```
if (personalities = "cowards") [ set color blue ]
```

This command depends on a variable called PERSONALITIES, which we will create together later. PERSONALITIES tells us if we are in stage one or stage two in the game, i.e., whether the agents are playing as “heroes” or as “cowards.” If it is set to the value “cowards,” then we are playing stage one, where all the agents will act cowardly. To visualize the cowardly agents, we color them blue. Similarly, the third command is:

```
if (personalities = "heroes")  [ set color red ]
```

If the personalities variable is set to “heroes,” then we are playing the second stage, where all the agents act like heroes. To visualize the heroic agents, we color them red.

Next the agents need to pick an agent to be their friend and another to be their enemy. To accomplish this, we use “turtle variables.” We will use two such variables, FRIEND and ENEMY. We will need to “indicate” (also called “declare”) in the Code tab that we have given the turtles these two variables. We do that by putting the following line at the top of the Code tab.

```
turtles-own [ friend enemy ]
```

11. You can get similar results with a large range in the number of agents, but we have tuned the sample model in the models library to work with this exact number.

This gives every turtle two variables. One variable is named FRIEND, and one is named ENEMY. Now that these variables are established, we can set them.

```
set friend one-of other turtles
set enemy one-of other turtles
```

These commands tell each turtle to choose another turtle from the set of all turtles and set the FRIEND variable to take on the value of that turtle. This is a slightly different use of variables than we have seen before. In this case, FRIEND is not a number; instead, it is a reference to another agent in the model. In NetLogo, as in many agent-based modeling languages, we can use variables to refer directly to other agents, which makes our code much easier to write.

Once the value for the FRIEND variable has been decided, the code asks the turtle to choose another turtle from the set of all turtles and set the enemy variable to be equal to the selected turtle.

The final command is reset-ticks, which, as we saw, initializes the NetLogo clock. That completes our SETUP procedure. This code is enough to set up the model as initially described. A common variation we will explore is to have a "mixed" game; that is, if PERSONALITIES = "mixed," we would like some of the agents to act bravely and others to act cowardly. To do this, we add one line to the SETUP procedure.

```
if (personalities = "mixed")     [ set color one-of [ red blue ] ]
```

The ONE-OF primitive picks a random element of a list, which in this case is the list [RED BLUE]. This command will randomly set some of the agents to be red and cowardly, others to be blue and brave. The final form for the SETUP procedure is:

```
to setup
    ca
    ask patches [ set pcolor white ] ;; create a white background
    create-turtles number [
       setxy random-xcor random-ycor

       ;; set the turtle personalities based on chooser value
       if (personalities = "brave")    [ set color blue ]
       if (personalities = "cowardly") [ set color red ]
       if (personalities = "mixed")    [ set color one-of [ red blue ] ]

       ;; choose friend and enemy targets
       set friend one-of other turtles
       set enemy one-of other turtles
    ]
    reset-ticks
end
```

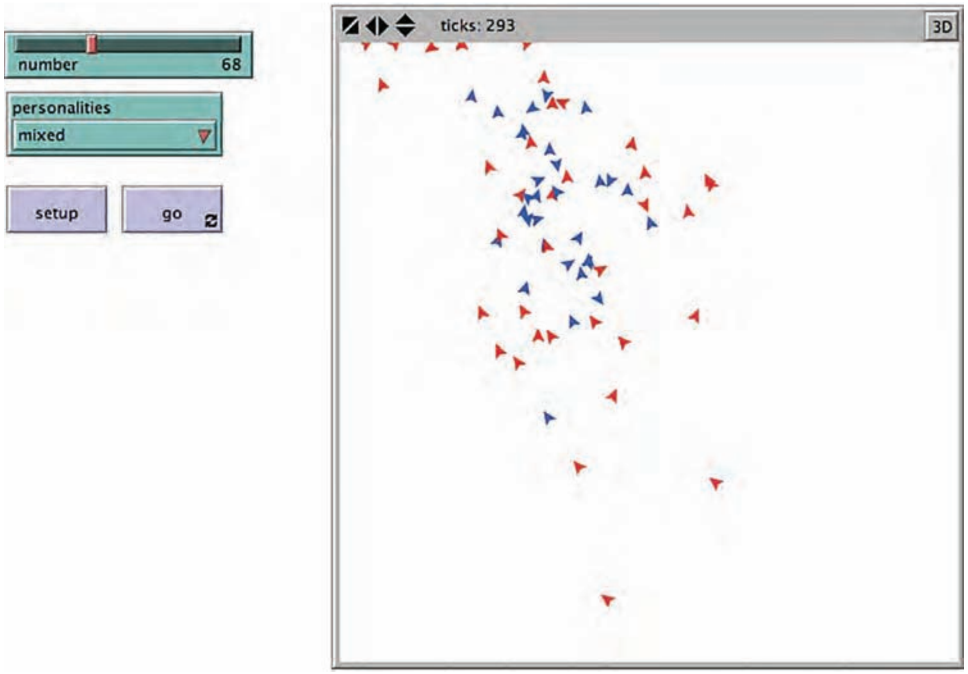


Figure 2.22
The setup for the Heroes and Cowards model. The hero agents are blue; the cowards are red.

To create the interface for the model, we go to the NetLogo *Interface* tab and create two widgets: a button that says SETUP and a chooser that has three values: HEROES, COWARDS and MIXED. The chooser variable name is the PERSONALITIES variable we have been branching on. If you press the SETUP button, you should see a screen like the one shown in figure 2.22.

Now that we have verified that our SETUP procedure seems to be working well, we are ready for our GO procedure.

Here is the GO procedure for the model.

```
to go
    ask turtles [
       if (color = blue) [ act-bravely ]
       if (color = red)  [ act-cowardly ]
    ]
    tick
end
```

Box 2.6
Turtle Monitors and Links

Similar to the patch monitors we explored in the Game of Life, we can use a turtle monitor to inspect the properties of any turtle. If we right-click on the blue turtle near the center of the view and select inspect, a turtle monitor pops up.

Looking at its properties, we see that its COLOR is 105, which is blue. We also see that it has a FRIEND property and that its FRIEND is turtle 45, and its ENEMY is turtle 41.

It may be handy to keep track of the relative position of our turtle with its FRIEND and ENEMY. One way to do that is to create a link agent linking the turtle to its FRIEND and ENEMY. Link agents are a core agent class in NetLogo, just like turtles and patches. We will go over link agents in detail in chapter 6. We can create link agents directly in the turtle monitor. We type into the command center of the turtle monitor, the command:

```
create-link-with turtle 45 [set color green]
```

This would be equivalent to writing:

```
create-link-with friend [set color green]
```

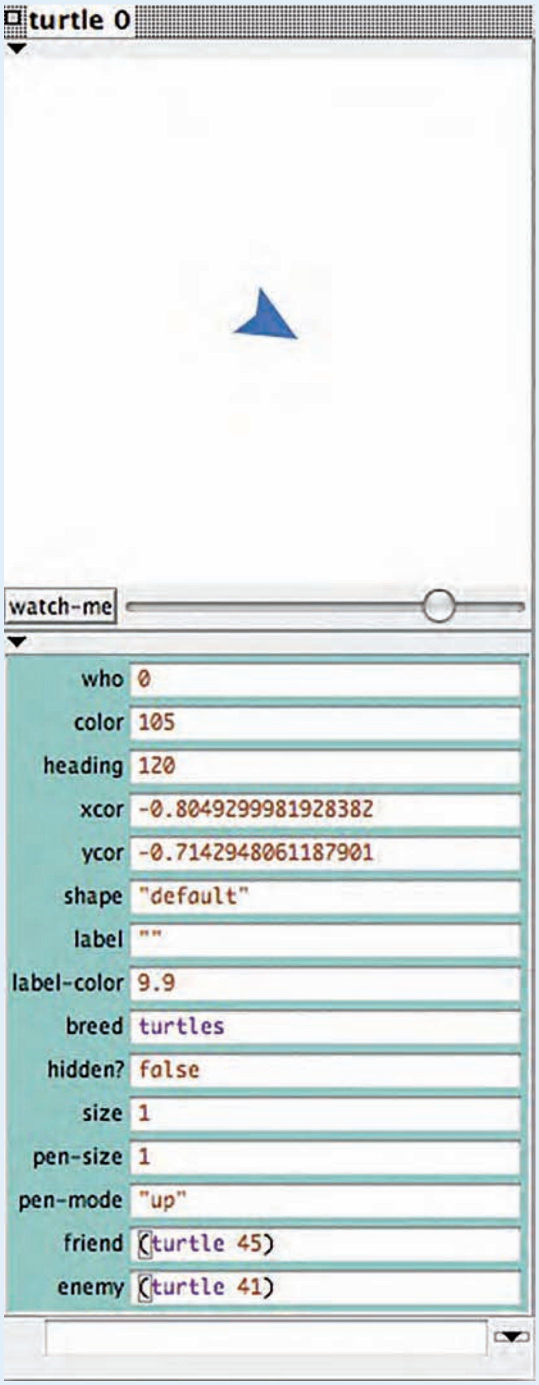
We will then see a green link connecting our focal turtle with its FRIEND.

This GO procedure simply asks the blue turtles to act bravely, the red turtles to act cowardly, and then advances the clock one tick. We will need to define the ACT-BRAVELY and ACT-COWARDLY procedures later, but first, let's take a look at the behavior of the model. If you are following along and writing this model as you go, then save your current version of the model and load the version of the Heroes and Cowards model that is in the chapter two subfolder of the IABM Textbook Folder of the NetLogo models library.

In the library model, we have added a GO button (a forever button) and set the PERSONALITIES chooser to COWARDS. We have also turned wrapping off both vertically and horizontally in the Model Settings dialog since in the real world a person cannot wander out of one door of a room and come in the other side. In the finished model with the settings in the next figure, if we press SETUP, the model creates 68 red turtles. When we press GO, we see the turtles fleeing to the boundaries of the view. They are all cowards, so they are trying to get behind their friends, which leads them to move outward (see figure 2.26).

Now, let's set the personalities chooser to HEROES. Can you predict what the agents will do? You may have correctly predicted the model's hero behavior based on what we

Box 2.6
(continued)



who	0
color	105
heading	120
xcor	-0.8049299981928382
ycor	-0.7142948061187901
shape	"default"
label	""
label-color	9.9
breed	turtles
hidden?	false
size	1
pen-size	1
pen-mode	"up"
friend	(turtle 45)
enemy	(turtle 41)

Figure 2.23
A turtle monitor of a heroic agent.

Box 2.7
Turtle Monitors and Links, Continued

Similarly, we can type:

```
create-link-with turtle 41 [set color red]
```

which is equivalent to:

```
create-link-with enemy [set color red]
```

and we will see a red link connecting our turtle with its ENEMY.

described at the beginning of this section. All the agents collapse into one small area (see figure 2.27).

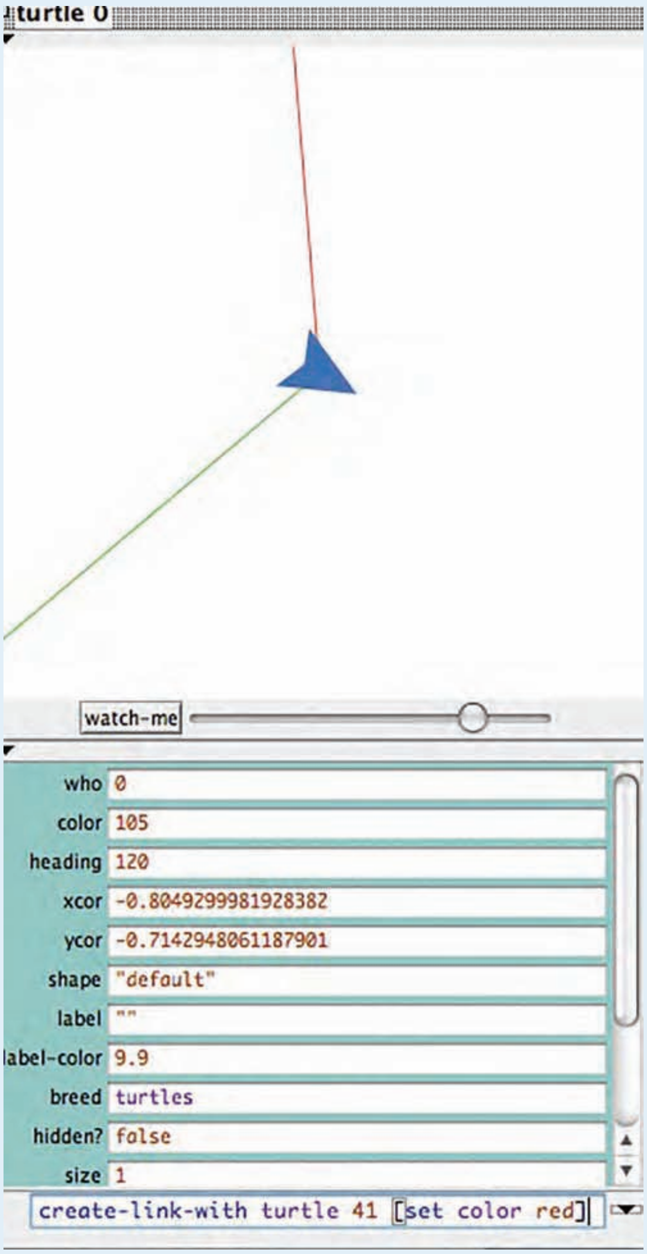
A more complex behavior emerges when we set the personalities chooser to "mixed." Can you predict what will happen? Most people are unable to predict the behavior of the "mixed" model. In mixed mode, the model exhibits a variety of different possible behaviors. Since the behavior of the agents is deterministic, the model's behavior is completely determined by the initial setup. As we will describe, this initial setup contains enough randomness to generate several different interesting behaviors.

To examine this, we developed a special version of the model in which the buttons below the SETUP and the GO button of the interface encode two initial setups that exhibit interesting behavior. Each button sets NetLogo's random number generator to a particular "seed" state. The random numbers spewed out by the generator determine the "random" locations of the initial agents. They also determine which agents are heroes and which cowards as well as which friends and enemies they choose. Each press of these two setup buttons will generate the same exact initial setup and hence the same model behavior from then on.

You will notice that the model exhibits a variety of qualitatively different "final behaviors." Once reached these final behaviors, similar to "attractor states" in dynamical systems theory, stay qualitatively stable.[12] We will illustrate four such final behaviors for this model. In the associated model file (Heroes and Cowards.nlogo), we have setup buttons for two of these final states, "frozen" and "slinky." There are many possible states, and we leave it as an exercise to the reader to find other interesting final states.

12. For a good introduction to attractors and dynamical systems, please see *Nonlinear Dynamics and Chaos* by Steven Strogatz published in 1994 by Westview Press.

Box 2.7
(continued)



who 0
color 105
heading 120
xcor -0.8049299981928382
ycor -0.7142948061187901
shape "default"
label ""
label-color 9.9
breed turtles
hidden? false
size 1

```
create-link-with turtle 41 [set color red]
```

Figure 2.24
Turtle monitor: focal agent linked to its friend and enemy.

Box 2.7
(continued)

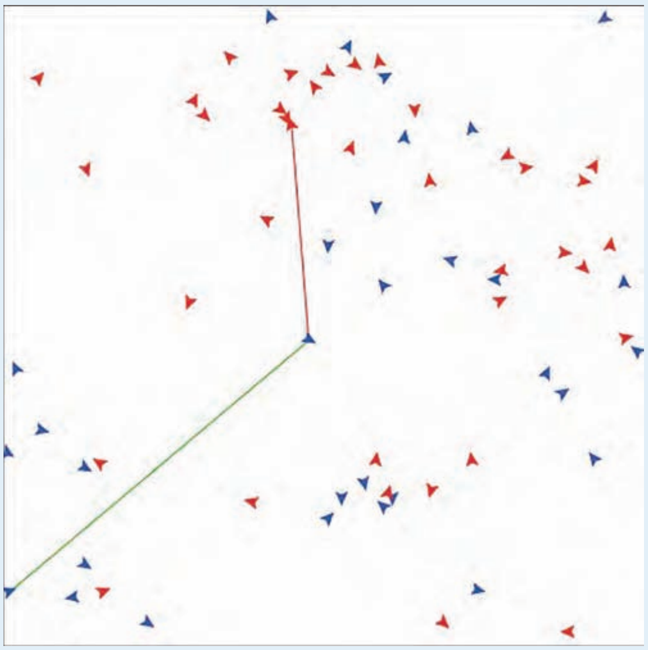


Figure 2.25
The NetLogo view can see the focal agent and its links.

1. *Frozen* The agents end up scattered on the screen, frozen at a certain location. They vibrate a little but do not stray from their location (see figure 2.28).
2. *Slinky* The agents form a line and the line slinks on the edges of the view and then jumps across the view (see figure 2.29).
3. *Dot* The agents coalesce into a small vibrating dot (see figure 2.30).
4. *Spiral* The agents coalesce into a loose rotating spiral and gradually get into a tighter spin (see figure 2.31).

Now that we have explored the behavior of the model, we can go back to looking at the model code. If you were working along with your own version before, then go ahead and reload that version. Make sure to create SETUP and GO buttons as we did before and the PERSONALITIES chooser and NUMBER slider if you have not already done so. Finally, don't forget to turn wrapping off.

Once all of that is done, we will begin creating the ACT-BRAVELY and ACT-COWARDLY procedures. Each of these makes the agent point in a direction and move a

Box 2.8

A Text-Based Pseudo-Code Format

When first working with agent-based models, it can often be useful to use a text-based form for explicitly describing the model rules. This pseudo-code format, should have two sections: "Initialize," which describes how the model's initial conditions, and "At each tick," which describes the behavior of the agents at every tick of the NetLogo clock. For agent-based models, it is recommended that you write the text from the point of view of the agents, especially in the "At each tick" section. Here is a text-based form for the Heroes and Cowards model:

```
Initialize:
    Create NUMBER turtles, where NUMBER is set by a slider in the interface
    Each turtle moves to a random location on the screen
    If the PERSONALITIES slider is set to “brave,” each turtle turns blue
    If the PERSONALITIES slider is set to “cowardly,” each turtle turns red
    If the PERSONALITIES slider is set to “mixed,” each turtle “flips a
    coin” and depending on the outcome, it turns red or blue
    Each turtle picks one other turtle as a friend
    Each turtle picks one other turtle as an enemy
    The NetLogo clock is started

At each tick:
    Each turtle asks itself “Am I blue?” If yes, then I will act bravely by
    moving a step towards a location between my friend and my enemy
    Each turtle asks itself “Am I red?” If yes, then I will act cowardly by
    moving a step towards a location that puts my friend between my enemy
    and me
```

small step in that direction. ACT-BRAVELY points the agent to a spot midway between its friend and enemy. ACT-COWARDLY points the agent to a spot behind its friend. These procedures use a little bit of vector mathematics, so we won't review them carefully here. Readers who have a mathematical background will recognize the mathematical form of the vectors. Others may prefer to examine the two figures below each procedure, which illustrate pictorially what the code does. Essentially, the ACT-BRAVELY code points the turtle toward the midpoint between the friend and enemy, while the ACT-COWARDLY code points the turtle toward a point that is as far away from its FRIEND as the FRIEND is from the ENEMY. (See figures 2.32 and 2.33.)

```
to act-bravely
    ;; move toward the midpoint of your friend and enemy
    facexy ([xcor] of friend + [xcor] of enemy) / 2
           ([ycor] of friend + [ycor] of enemy) / 2
    fd 0.1
end
```

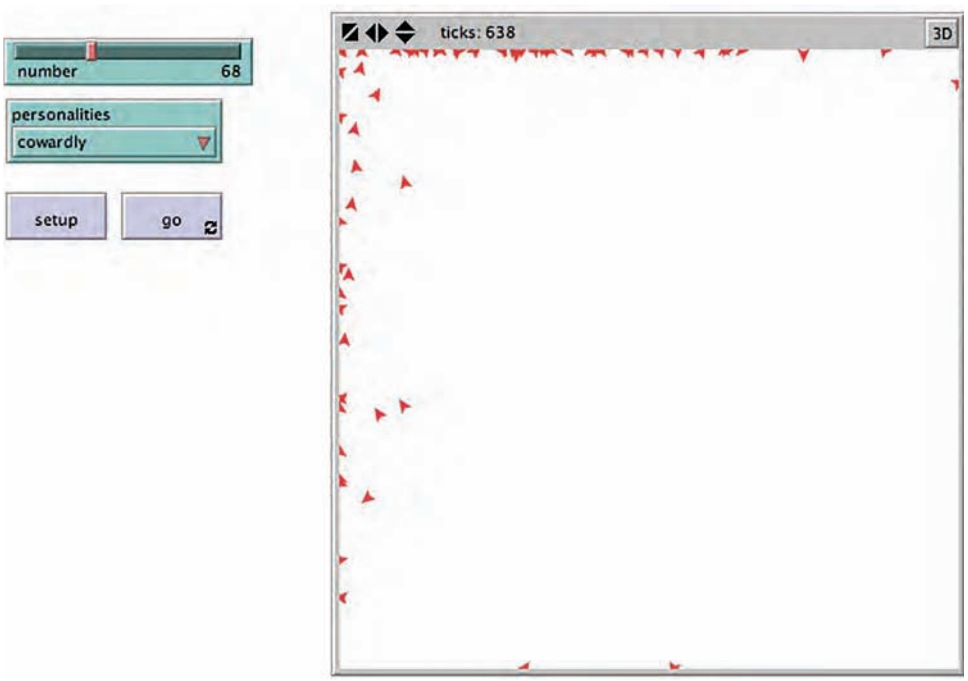


Figure 2.26
Heroes and Cowards cowardly behavior. All the agents move out to the edges.

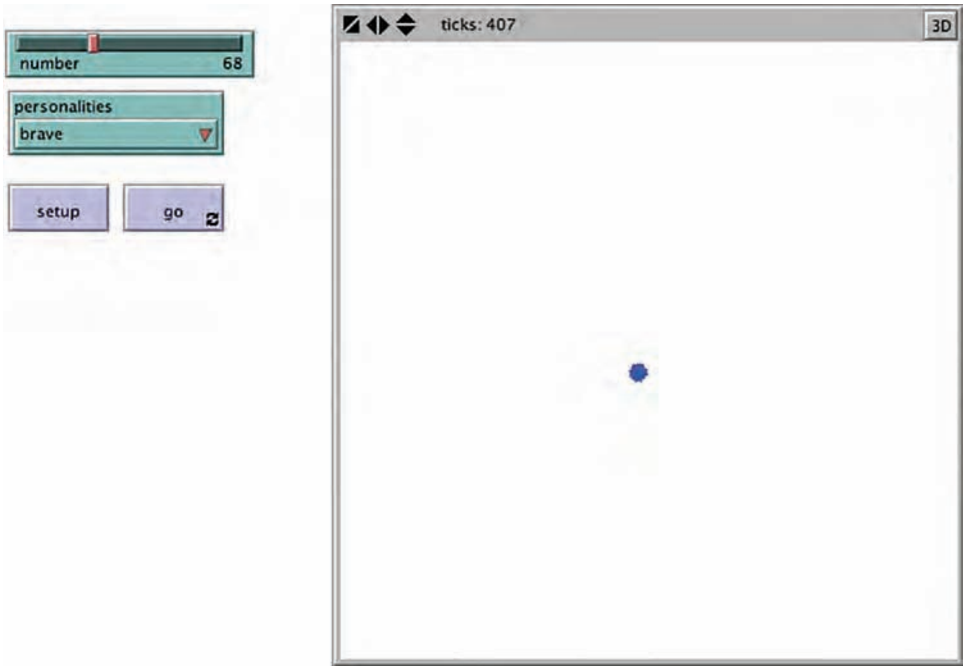


Figure 2.27
Heroes and Cowards brave behavior. All the agents move to the center.

Box 2.9
Random Number Generators

Agent-based models frequently need to make use of randomness; agents' behavior is often best modeled as a random process. In general, when programmers need to make use of randomness in computational environments, they use random numbers from a random number generator (RNG). A *random number generator* is a computational device (it can also be a physical device) designed to generate a sequence of numbers or symbols that lack any pattern and that seem to be random. However, the random numbers used by computer programs are actually "pseudo-random," so that while they appear random, they are generated deterministically. "Deterministic" means that if you start with the same random "seed" you will always get the same results. We'll explain in a minute what we mean by "seed." If you wanted to generate random numbers physically rather than pseudo-random, there are many different ways that have been developed, several of which have existed since ancient times, e.g., rolling of dice, flipping of coins, shuffling of playing cards, or using yarrow stalks in the I-Ching. As you can imagine, generating large amounts of random numbers using these techniques took a long time. As a result, books and tables of random numbers would sometimes be generated and then distributed for use by mathematicians. Nowadays, since RNGs generate numbers that are almost indistinguishable from true random numbers, they are used for everything from computer simulations to lotteries to slot machines.

In the context of scientific modeling, pseudo-random numbers are more desirable than true random numbers. This is because it's important that a scientific experiment be reproducible/replicable, so that anyone who runs the experiment will get the same result. (We will discuss this idea more in chapter 8.) Since NetLogo uses pseudo-random numbers, other researchers/scientists/modelers can reproduce the "experiments" that you conduct with it.

Here's how it works. NetLogo's random number generator can be started with a certain seed value, which is just an integer. Once the generator has been "seeded" with the random-seed command, it always generates the same sequence of random numbers from then on. For example, if you run these commands:

```
random-seed 137
show random 100
show random 100
show random 100
```

You will always get the numbers 79, 89, and 61, in that order. To create a number suitable for seeding the random number generator, use the NEW-SEED command. NEW-SEED creates a seed, evenly distributed over the space of all possible seeds, based on the current date and time. It never reports the same seed twice in a row.

If you do not set the random seed yourself, NetLogo sets it to a value based on the current date and time. There is no way to find out what random seed it chose, so if you want your model run to be reproducible, you must set the random seed yourself ahead of time.

See the Random Seed Example model from the Code Examples section of the NetLogo models library.

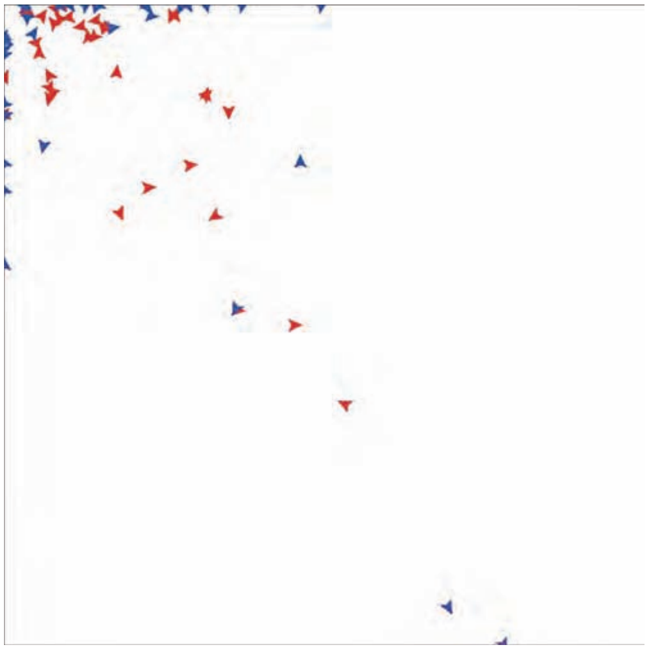


Figure 2.28
The Frozen final state of the Heroes and Cowards model.

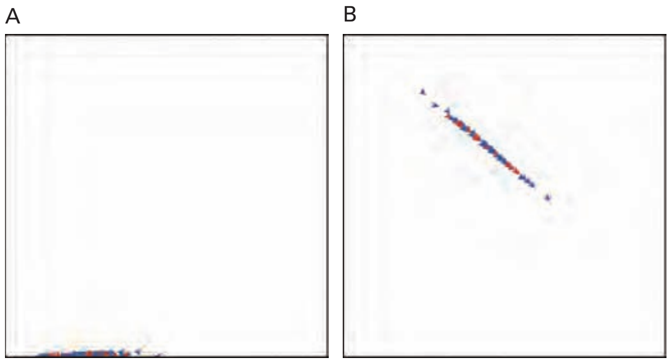


Figure 2.29
(A, B) Snapshots of the Slinky final state of the Heroes and Cowards model.

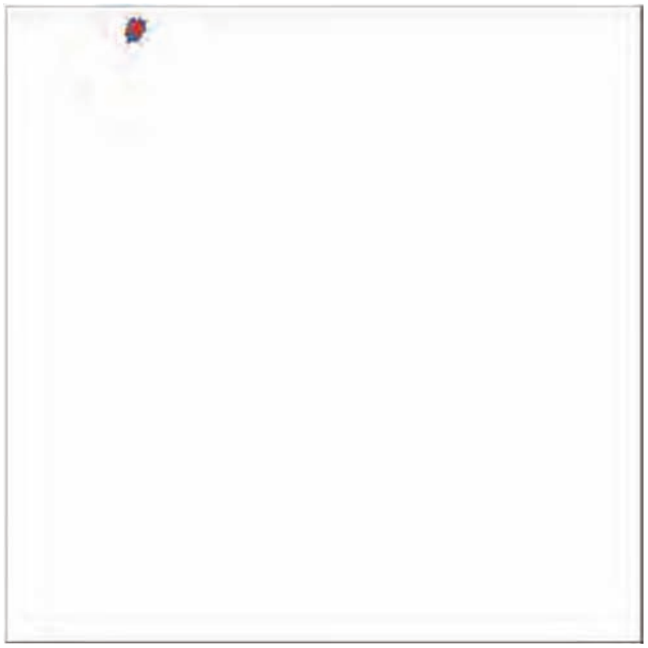


Figure 2.30
The Dot final state of the Heroes and Cowards model.

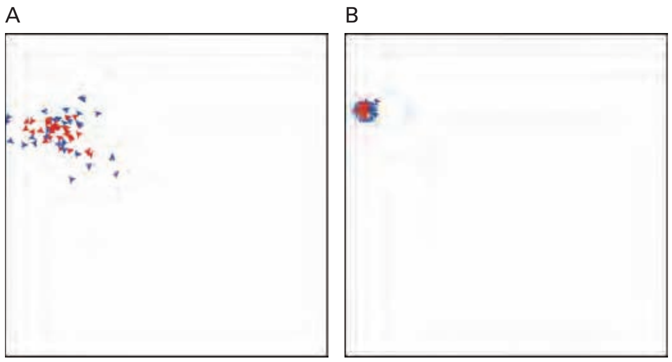


Figure 2.31
(A, B) Two snapshots of the Spiral final state of the Heroes and Cowards model.

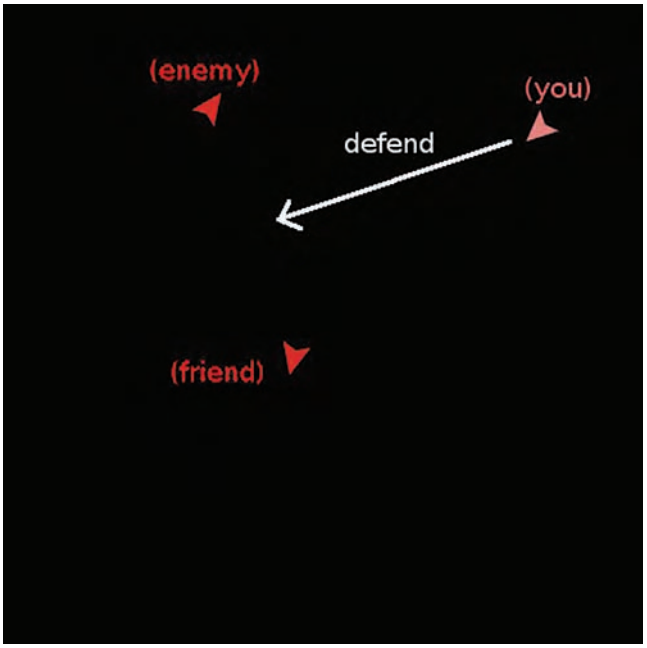


Figure 2.32
A brave agent sets it heading toward a spot midway between its friend and its enemy.

```
to act-cowardly
    ;; put your friend between you and your enemy
    facexy [xcor] of friend + ([xcor] of friend - [xcor] of enemy) / 2
           [ycor] of friend + ([ycor] of friend - [ycor] of enemy) / 2
    fd 0.1
end
```

Although the Heroes and Cowards game is in many ways a parlor game, there has been a considerable amount of follow-up research on the game, due to the surprising diversity of configurations and behaviors that can arise from its simple rules. In some ways, the game can be seen as a microworld for exploring how swarms can self-organize from distributed elements. Many researchers have been interested in finding new configurations with especially interesting behaviors. Researchers have also worked on the “reverse-engineering” of rules to make for a particular desired swarm behavior, but, as we have discussed, finding the micro-level rules that generate a macro-level pattern generally can be very difficult. Bonabeau proposes one solution to both of these problems: finding new swarms and rules that generate a particular swarm by evolving the rules for known swarms. However, this is difficult, as Bonabeau and his colleagues stated in 2003:

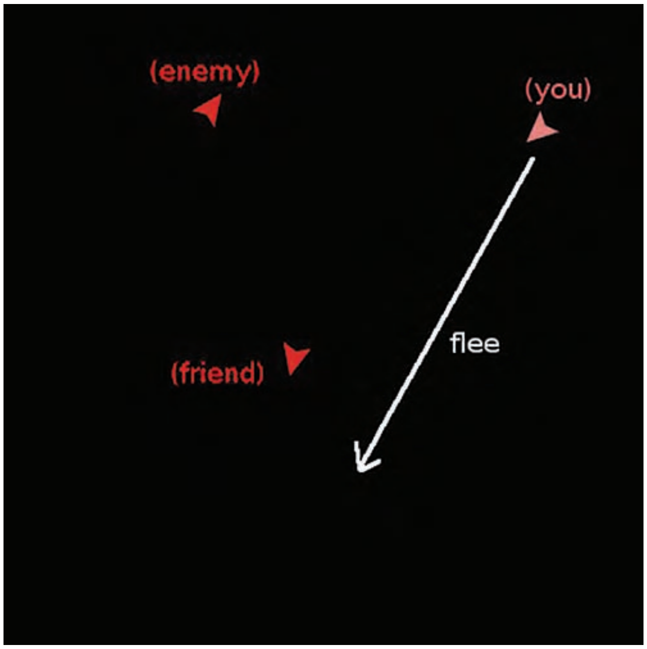


Figure 2.33
A cowardly agent sets its heading toward a spot behind its friend.

Designing the individual level rules of behavior and interaction that will produce a desired collective pattern in a group of human or non-human agents is difficult because the group's aggregate-level behavior may not be easy to predict or infer from the individuals' rules. While the forward mapping from micro-rules to macro-behavior in self-organizing systems can be reconstructed using computational modeling techniques, the inverse problem of finding micro-rules that produce interesting macro-behavior poses significant challenges, all the more as what constitutes "interesting" macro-behavior may not be known ahead of time.

Bonabeau et al. (2003) used an exploratory design method with an interactive evolutionary computation approach to find new configurations and behaviors. In this approach, users were provided with a set of original configurations. They could choose any one of these and mutate it slightly, or they could choose two and mate them. The resulting offspring are then evaluated for aesthetic interest and the best ones are kept as potential parents of the next generation.[13] In this way, users could use their aesthetic judgments to breed interesting configurations and behaviors (see figure 2.34 for some examples of offspring configurations). These final configurations can have complex dynamic behavior such as configuration (e), which Bonabeau describes as a "Chinese streamer," as in figure 2.35.

13. This is very similar to the biomorphs idea described by Richard Dawkins in *The Blind Watchmaker* (1986).

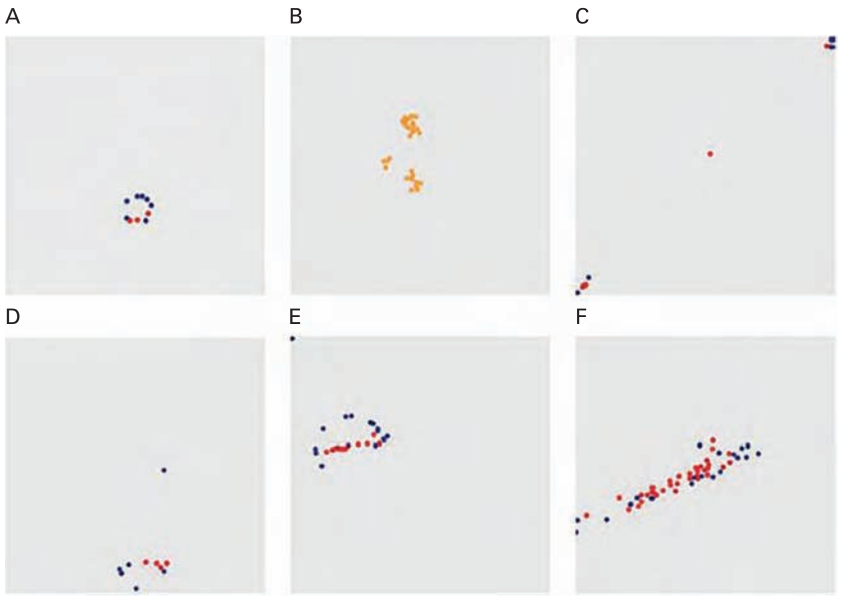


Figure 2.34
A few examples of evolved behaviors. (A) Circle: agents chase each other around in a circle. (B) Juggle: two blobs fuse and reemerge and sometimes toss a smaller blob at each other. (C) Corner-middle: two groups of agents go to opposite corners while one stays in the middle. (D) Pursuer-evaders: an agent follows a larger group that slows down, is reached by the pursuer, then escapes again. (E) Chinese streamer: a D shape that moves around. (F) Somersault: a thick line that makes a 360 degree turn, then stops, then turns back in the opposite direction. (From Bonabeau et al., 2003.)

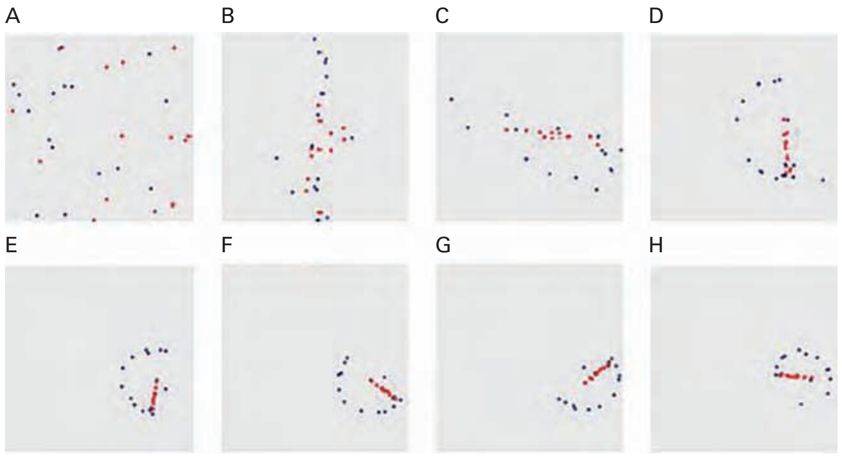


Figure 2.35
"Chinese streamer" pattern. From a random initial placement, a pattern quickly emerges (A–D) and starts turning, stabilizing in a shape with a handle and trailing ribbon, which rotates smoothly. The direction of rotation can be clockwise or counterclockwise (as here), presumably depending on the initial positions. (From Bonabeau et al., 2003.)

In this way, starting with one set of rules, we can discover new, “interesting” patterns of collective behavior when one does not know in advance what the system is capable of doing.

Simple Economy

In the past two decades, there has been increasing interest in and use of agent-based modeling in the social sciences. In fact, several of the models that we have already discussed, such as Ants, the Game of Life and Heroes, and Cowards have been applied to understanding social systems. Agent-based methods may be particularly valuable in the social sciences where agents are heterogeneous and mathematical descriptions often do not offer sufficient descriptive power. Several prominent communities have organized around using complex systems methods and agent-based modeling in the social sciences. Among these are the Complex Systems Social Sciences Association (CSSSA) and the World Congress on Social Simulation (WCSS), both of which also organize conferences. There are also a number of such events that are included as part of other meetings. In recent years, organizations such as the American Education Research Association (AERA), Marketing Science, Eastern Economics Association, and American Association of Geographers (AAG) have hosted sessions on the intersection of agent-based modeling and social science.

One area that has been receiving increasing attention from the ABM community is economics. As economies consist of heterogeneous actors such as buyers and sellers, there is a natural mapping of ABM methods to economics. In 1996, the economists, Josh Epstein and Robert Axtell published a book that depicted a world they called SugarScape, which was populated by economic agents.

In this section, we will build a very simple economics model that has some surprising results. Suppose you have a fixed number of people, say 500, each starting out with the same amount of money, say $100. At every tick, each person gives one of his or her dollars to any other person at random. What will happen to the distribution of money? An important constraint is that the total amount of money remains fixed, so no one can have less than zero money. If you run out of money, you cannot give any away until you get some back. Refining the question a little further, we ask: Is there a stable limiting distribution of the money? If so, what is it? For instance, will all the wealth be concentrated in a few hands or will it be equitably distributed?

Many people, when posed with this question, have an intuition that there is a limiting distribution and that it is relatively flat. The reasoning behind the intuition is that since no person starts off with an advantage and the selection of people to whom money is transferred is random, no person should have much of an advantage over any others. Thus, the resulting wealth distribution should be relatively flat: Everyone should wind up with roughly the same amount of money he or she started with. Other people have an intuition

Box 2.10
SugarScape and Agent-Based Economics

One of the first large-scale agent-based models was Epstein and Axtell's SugarScape from their book *Growing Artificial Societies: Social Science from the Bottom Up*. SugarScape consists of a series of models based on a population with limited vision and spatially distributed resources available (sugar and spice). Agents look around, find the closest cell filled with sugar, move, and metabolize. Agents can also leave pollution, die, reproduce, combat other agents, inherit resources, transfer information, trade or borrow sugar, or transmit diseases. Each of the models in the SugarScape series explores some of the conditions and dynamics. Sugar and Spice in the SugarScape models can be seen as a metaphor for resources in an artificial world through which we can study the effects of social dynamics such as evolution, marital status, and inheritance on populations. The SugarScape work inspired a host of generative social science models (Epstein, 2006) and invigorated the field of agent-based economics and social science in general (see also Tesfatsion & Judd 2006). Agent-based modeling is especially well suited as a methodology for behavior-based economics.

Behavior-based economists are unsatisfied with traditional approaches of economists—which have prioritized simplified approaches for the sake of soluble theoretical models over agreement with empirical data. In particular, traditional approaches have usually posited perfectly rational agents, whereas behavioral economists make use of "boundedly rational" (Simon, 1991) agents that do not have complete information and use shortcuts or heuristics to make decisions.

that the wealth should be normally distributed. To explore this question of wealth distribution, let us build the model we described to see if that intuition is correct.

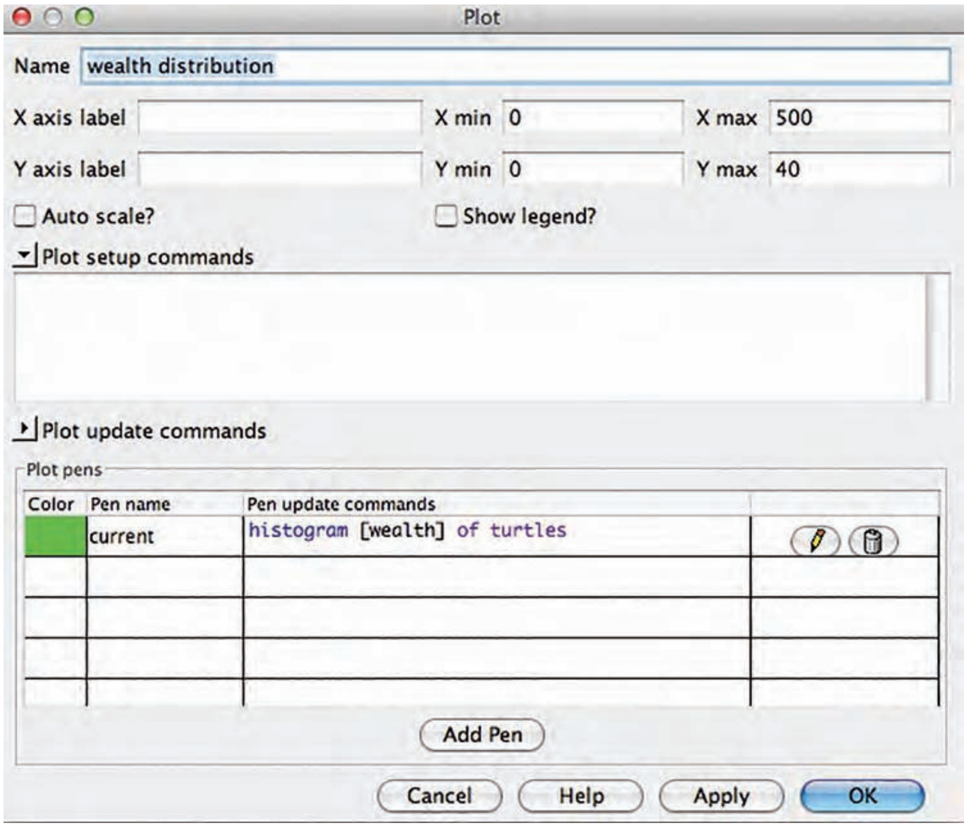
We start with our SETUP procedure. We will need to keep track of the wealth of each agent, so we need to give the agents a WEALTH variable:

```
turtles-own [wealth]
```

Since one of our main questions of interest is what the limiting distribution of money in this model is, we will also need to create a histogram of the wealth of the agents. To do that, we create a plot widget in the NetLogo interface and give it the plot command:[14]

```
histogram [wealth] of turtles
```

14. This is the easiest way to create a simple plot. In chapter 4, we will discuss a slightly more complex way that has some advantages, especially for complex plots.


Plot

Name: wealth distribution

X axis label: | X min: 0 | X max: 500
Y axis label: | Y min: 0 | Y max: 40

☐ Auto scale? ☐ Show legend?

Plot setup commands

Plot update commands

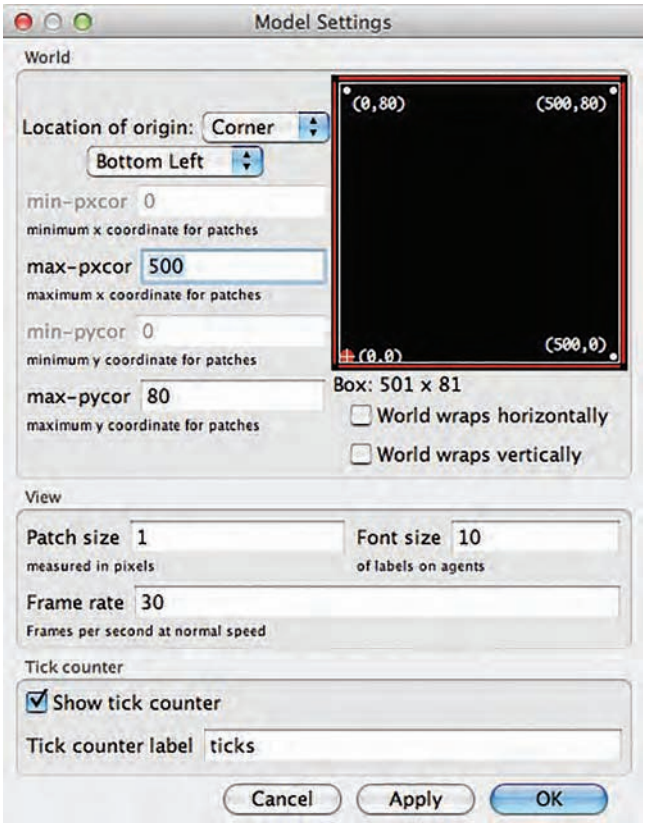
Plot pens

Color	Pen name	Pen update commands
🟩	current	`histogram [wealth] of turtles`

Figure 2.36
The plot widget that runs the histogram command at every clock tick.

At each clock tick the plot widget will run its plot command (see figure 2.36). To make our plot widget work with the parameters of our model, we also set the X MAX to 500 and the Y MAX to 40 and turn off AUTO SCALE? If you want to exactly match the images in the book you also need to change the pen color to green (by double-clicking on the color to the left of the pen name, and the pen mode to bar (by double-clicking on the pencil to the right of the pen update commands and selecting "bar" from the mode drop down menu).

Our SETUP procedure needs to clear the view, create 500 agents, give them some properties and then reset the tick counter. We visualize the agents as green circles and initialize every agent to start with $100. Though we have already set up a histogram to tell us about the wealth distribution, it is often useful to also visualize such distributions



Model Settings

World

Location of origin: Corner

Bottom Left

min-pxcor 0
minimum x coordinate for patches

max-pxcor 500
maximum x coordinate for patches

min-pycor 0
minimum y coordinate for patches

max-pycor 80
maximum y coordinate for patches

Box: 501 x 81

☐ World wraps horizontally

☐ World wraps vertically

View

Patch size 1
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

☑ Show tick counter

Tick counter label ticks

Figure 2.37
Configuring the view as a long rectangle.

in the view by changing the visualization of the agents in some way to reflect their wealth. One way to do that is to set the agent's XCOR to its wealth, so that the poorest turtles are on the left edge of the view, and as we move right, the turtles get wealthier (which some may claim may also have political consequences). To ensure that there will be plenty of space for the agents to move right, we shape the view as a long rectangle and make the patch size small.

In figure 2.37, we have set the view with an origin (coordinate 0,0) at the bottom left and a maximum x-coordinate of 500, so as to visualize agents with a wealth of up to 500 and a maximum y-coordinate of 80 so we can see a spread of agents. We also set a small patch size of 1, so the view fits well on a reasonably sized screen.

Because the patches are small, we increase the size of the agents to two (2) to better view them. Here is the code for the SETUP procedure.

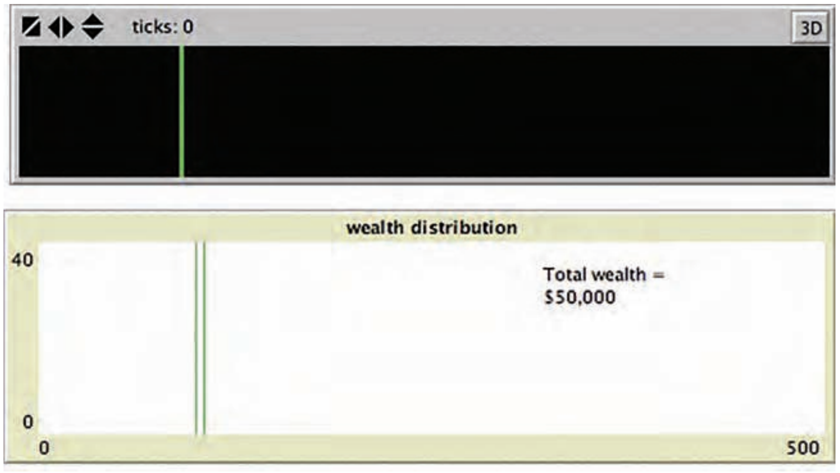


Figure 2.38
The view is placed above the histogram and aligned with it. All 500 agents start out with $100 apiece.

```
to setup
    clear-all
    create-turtles 500 [
       set wealth 100
       set shape "circle"
       set color green
       set size 2

    ;; visualize the turtles from left to right in ascending order of wealth
       setxy wealth random-ycor
    ]
    reset-ticks
end
```

After running SETUP, the view and histogram will look as it does in figure 2.38.

The GO procedure for the model is very simple. At every tick, each agent will need to transact (give one dollar) with one other agent. So the main line of code is:

```
ask turtles [transact]
```

We will then need to fill in the details of the TRANSACT procedure. This line of code, however, does not take into account the agents with no money left, as they cannot give away anything. To fix that "bug," we add a restricting WITH modifier, so only turtles with at least a dollar to their name can give away their money.[15]

15. An alternative way to code this is to keep the line as is, asking all agents to transact, and then code the transact procedure to exclude agents with no money.

Box 2.11
Agentsets and Lists

Agentsets and lists are two of the most used data structures in NetLogo.

An agentset is much like it sounds: a set of agents. Agentsets can have turtles, patches, or links, but they cannot mix agent types. You can ask an agentset to perform some commands. NetLogo comes with three special agentsets built in: “turtles,” “patches,” and “links,” which we have already asked to execute commands. What makes agentsets so powerful is that you can create your own agentsets—for example, an agentset of all the red turtles, or an agentset of all the patches in the upper right quadrant. Agentsets are always in random order. So if you ask an agentset, several different times, to execute some commands, each time the order in which the agents take turns executing the commands will be different.

Lists are ordered collections of data. You can have a list of numbers, a list of words, even a list of lists. You can also have a list of agents. Since they are in a list, these agents will have a particular order, so you can use the list to execute their commands in any order you might like.

We will discuss agentsets and lists more in chapter 5.

See also: http://ccl.northwestern.edu/netlogo/docs/programming.html#agentsets in the NetLogo programming guide of the NetLogo User Manual.

```
ask turtles with [wealth > 0] [transact]
```

The only other line in our GO procedure is to move the agents to an x-coordinate corresponding to their wealth. The code for the GO procedure is:

```
to go
    ;; transact and then update your location
    ask turtles with [wealth > 0] [transact]
    ask turtles [set xcor wealth ]
    tick
end
```

Now, we have to write the TRANSACT procedure. First, we decrease the agent’s wealth by one (1) dollar. Then we choose a random other agent and increase its wealth by one (1) dollar. The resulting code is:

```
to transact
    ;; give a dollar to another turtle
    set wealth wealth - 1
    ask one-of other turtles [set wealth wealth + 1]
end
```

That is the entire code for the Simple-Economy model. There is, however, one small bug in our GO procedure. We have asked the turtles to set their XCOR to their wealth. If their wealth gets very large, then the resulting XCOR will be beyond the world boundaries. To correct for that bug we add a check to ensure that we only relocate turtles within the world boundaries. The resulting code is:[16]

```
to go
    ;; transact and then update your location
    ask turtles with [wealth > 0] [transact]
    ask turtles [if wealth <= max-pxcor [set xcor wealth]]
    tick
end
```

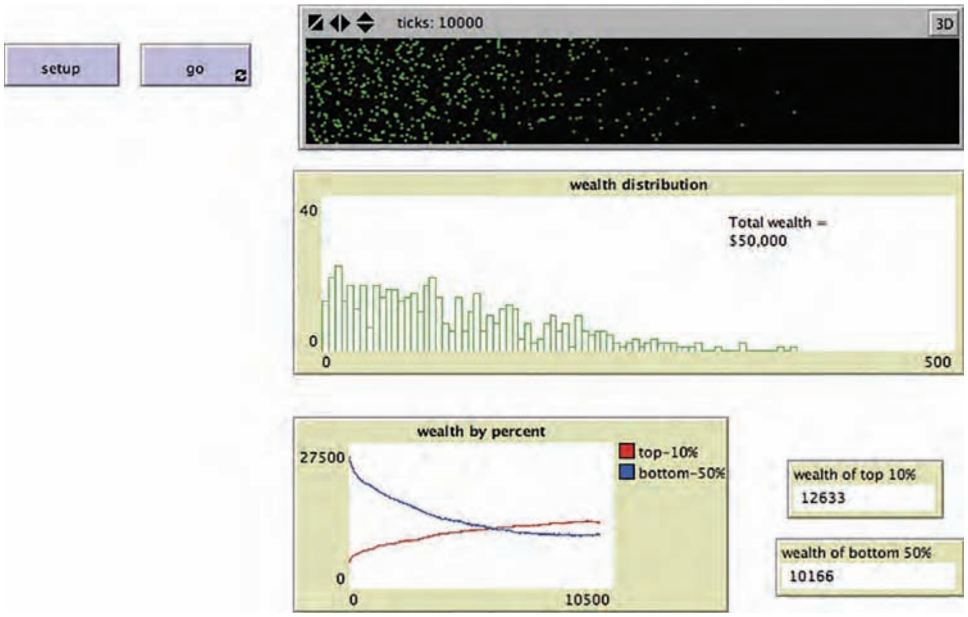
Now that we have revised the code for the GO procedure, let's pause for a moment to reflect on what we expect the behavior of the model to be. As mentioned earlier, most people have an intuition that the distribution will vary from tick to tick, but that overall, it will stay relatively flat. Surprisingly, it is not a flat distribution.

Running the model for 10,000 ticks yields the following picture (figure 2.39). We can see that the distribution is not at all flat. There are a few very wealthy individuals and many poor agents. At this time step in this model run, the wealth of the top 10 percent of the agents is a total of $12,633, or an average of $253 per person, whereas the total wealth of the entire bottom 50 percent is only $10,166, or an average of $41 per person. The crossover point at which the wealth of the top 10 percent of all agents exceeded the wealth of the bottom half, or 50 percent, of all agents was at 5,600 ticks.

If we run it for 25,000 more ticks, the gap continues to grow and now the top 10 percent have more than twice as much in total (and more than 10 times as much per individual) than the bottom 50 percent. (See figure 2.40.)

The distribution of money eventually converges to a stationary distribution. This distribution has been shown to be exponential, which means that there is great inequality in monetary wealth. The key condition that creates this stationary distribution is the conservation of money. The *conservation law of money* states that money is not allowed to be created or destroyed by any of the agents in our model, but can only be transferred between agents. Thus, the total amount of money in the system is always fixed. Indeed, any set of interacting agents that exchange a conserved quantity that cannot become negative will always result in such an exponential distribution. This is the result of a famous law from statistical mechanics, known as the *Boltzmann-Gibbs law* (see box 2.12).

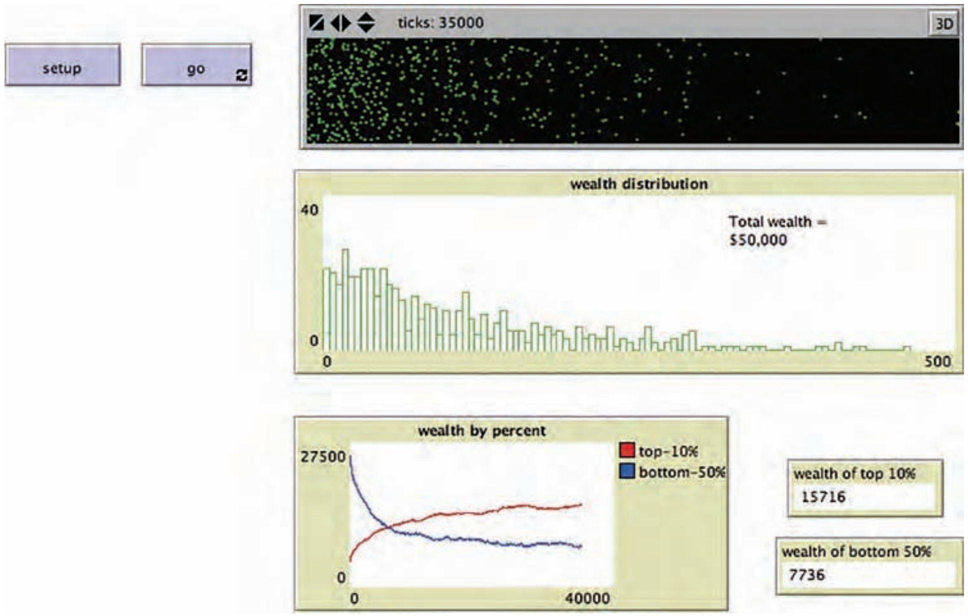
16. There are several alternative ways to code this check as well. We leave the exploration of these ways as an exercise for the reader.



wealth of top 10%
12633

wealth of bottom 50%
10166

Figure 2.39
The Simple Economy model after 10,000 ticks.



wealth of top 10%
15716

wealth of bottom 50%
7736

Figure 2.40
The Simple Economy model after 35,000 ticks.

Box 2.12
Boltzmann-Gibbs Law

The distribution we have just seen in the Simple Economy model is one instance of a broader class of models in which a conserved quantity is exchanged in a closed system. The great nineteenth-century Austrian physicist Ludwig Boltzmann first examined such models when he worked out the physics of gas molecules in a box. He imagined these molecules as "billiard balls" moving and colliding. When they collide, they change speed and direction, but because of the conservation of energy, the total energy after a collision remains the same. He derived an equation, now known as the *Boltzmann equation*, for the distribution of the energies of the billiard balls and solved the equation to show that their energies formed a stationary exponential distribution, known as Boltzmann distribution, the Boltzmann-Gibbs distribution, and the Maxwell-Boltzmann distribution.

The Boltzmann distribution is a central component of statistical mechanics. Statistical mechanics uses probability theory to understand the thermodynamic behavior of systems composed of a large number of particles. Statistical mechanics provides a framework for relating the microscopic properties of individual atoms and molecules to the macroscopic bulk properties of materials that can be observed in everyday life. In physics, the methods of statistical mechanics have been used with a range of "particles" from single atoms interacting to complex molecules, and they can be used to explain macroscopic concepts such as work, heat, and entropy. Since statistical mechanics gives scientists the ability to reason from microscopic particles to macroscopic patterns, it has been extremely useful in generating a comprehensive theory of physics. In many ways, agent-based modeling can be seen as extending this perspective to content domains beyond physics.

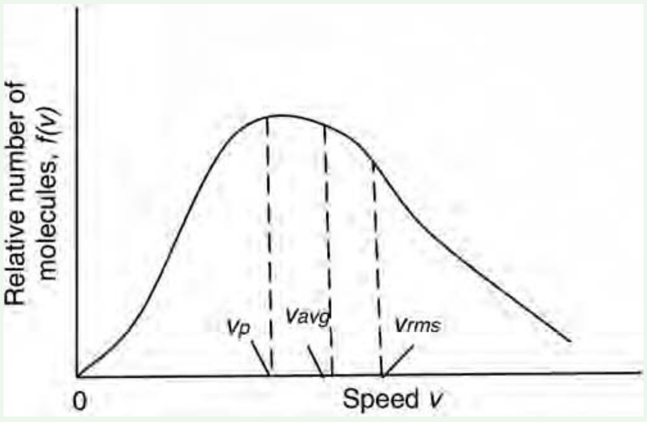


Figure 2.41
Maxwell-Boltzmann distribution of molecule speeds (illustration from Giancoli, 1984).

In recent years, theories and methods from physics have been applied to economics. The physicist Gene Stanley (1996) coined the term "econophysics" for this interdisciplinary research field.

In their 2000 paper, Dragulescu and Yakovenko describe several variants of the kind of monetary exchange we have explored in this model. They show that if agents are allowed to go into debt (subject to a fixed credit limit), there is still a stationary exponential distribution, but one that results in even greater inequality. We have referred to our agents as owning wealth. A more specific term for what we are modeling is *monetary wealth*. Dragulescu and Yakovenko caution:

> It is tempting to identify the money distribution with the distribution of wealth. However, money is only one part of wealth, the other part being material wealth. Material products have no conservation law: They can be manufactured, destroyed, consumed, etc. Moreover, the monetary value of a material product (the price) is not constant. The same applies to stocks, which economics textbooks explicitly exclude from the definition of money. So, in general, we do not expect the Boltzmann-Gibbs law for the distribution of wealth.

In fact, empirical studies have shown that wealth distribution in most countries is not distributed exponentially, but rather as a power law known as the Pareto distribution (Pareto, 1964). This distribution is even more unequal than an exponential distribution. Power law distributions arise in a surprising number of contexts and are typically associated with preferential attachment processes (Barabási, 2002), in which the amount that someone receives is proportional to the amount they already have (there is more on this process in chapter 5, and more on power laws when we discuss networks). If this process governs wealth distributions it would result in an ever widening wealth gap between the rich and the poor.

Summary

In this chapter, we have shown three simple models that are easily created as agent-based models. Even though they are easy to create, they exhibit quite complex behavior. In each case, these models demonstrate that predicting the emergent behavior of even the simplest models from knowledge of the rules that generate that behavior is very difficult to do. The ability to encode these conceptual models as ABMs enables us to easily explore a range of their behaviors and to dispel incorrect deductions we might easily make.

Beyond simple exploration of the model, we will often want to systematically analyze a model's behavior. Conducting such analyses can require considerably more effort. We will discuss analysis of agent-based models and conduct some analyses in chapters 6 through 8. In the next few chapters we will learn to modify existing agent-based models, describe a methodology for carefully building agent-based models, and review the many primitives and tools that are available for building ABMs.

Explorations

Chapter Model Explorations

1. Run the Life model several times. What is the minimum number of ticks before the model stabilizes? What is the maximum number of ticks you can find before it stabilizes? Is it possible for the model to never come to a stable state?
2. Write the rules for the Life Simple model in textual pseudo-code format.
3. In the Life model, we introduced the variable, live-neighbors. In the GO code, we initially calculate live neighbors and then branch on the value of live-neighbors as follows:

```
to go
    ask patches [
       set live-neighbors count neighbors with [pcolor = green]
       ]
    ask patches [
if live-neighbors = 3 ….
```

Why do we need to use the live-neighbors variable? Would the model behave any differently if we changed the code to count the neighbors “in-line” as follows:

```
to go
    ask patches [
       if  (count neighbors with [pcolor = green]) = 3
```

Can you explain why or why not?
4. In the Life model in the Computer Science section of the NetLogo models library, find a pattern where there are no more than 10 green cells to begin with but after 10 time steps there are at least 100 green cells. Also, start with at least 100 green cells and find a pattern that ends with no green cells after no more than 10 time steps.
5. Four well-known still-lifes are the loaf, the boat, the ship, and the beehive. Explain why each of these is a still-life. Can you find another still-life? Modify the Life model in the NetLogo models library to include a button that saves your new still-life configuration.
6. In the Game of Life, a blinker is a period 2 oscillator. Can you find another period 2 oscillator? How about a period 3 oscillator? A period 15 oscillator? Save your configurations as buttons in the Life model in the NetLogo models library.
7. In the Game of Life, can you construct a glider gun of glider emission period 15? Period 20? Save your configurations as buttons in the Life model in the NetLogo models library.
8. A methuselah is a small pattern that behaves chaotically for a large number of generations before settling down into a predictable pattern. Erik de Neve found a methuselah he named Edna. It runs on a 20 × 20 grid and is chaotic for 31,192 ticks. Can you find a methuselah on a 31 × 31 grid? How many ticks does it take to settle down?

9. In the Game of Life, we can define the "speed of light" (*c*, just as in physics) as the maximum attainable speed of any moving object, a propagation rate of one step (horizontally, vertically, or diagonally) per tick. This is both the maximum rate at which information can travel and the upper bound on the speed of any pattern. How fast (in terms of *c*) does the glider move? Can you find a spaceship that moves faster than a glider?
10. Besides the Life model that we have explored, there are several other types of Cellular Automata in the NetLogo models library. For instance, examine CA 1D Elementary (found in the Computer Science section of the library). There are two main differences between this model and the Life model. The first is that the model is 1D so the y-axis illustrates each time-step. Second, the model allows you to choose the rules that govern the behavior of the system rather than using prescribed rules such as we did in the Life model. Can you find a rule that starting with a single "live" cell will result in all of the cells becoming alive? Can you find a rule that starting with a single live cell will result in all of the cells becoming dead? Of course, you can always just turn all the rules on or all the rules off, but can you find solutions to this problem where exactly half the rules are on and half the rules are off?
11. Explore the Heroes and Cowards model. Can you find some other interesting final states beyond those captured in the model's buttons? Create some additional buttons to capture those behaviors and name them.
12. In Heroes and Cowards, what happens if you change the number of agents in the model? If you increase or decrease the number of agents is the pattern predictable?
13. There is a small bug in the Heroes and Cowards model. Can you find and fix it?
14. Another model that uses similar rules to Heroes and Cowards is the Follower model in the Art section of the NetLogo models library. How do the rules of this model differ from Heroes and Cowards? Why do we occasionally get similar behavior in both models? For instance, the Follower model sometimes seems similar to the dog-chases-tail pattern.
15. Write the rules for the Simple Economy model in textual pseudo-code format.
16. The Simple Economy model results in a surprising limiting exponential distribution. Describe in words why the limiting distribution is not relatively flat or normal.
17. What rule could you add to the Simple Economy model to increase the wealth inequality? What rule could you add that would decrease the inequality?
18. Open the SugarScape 1 Immediate Growback model from the Social Science section of the NetLogo models library. Explore the model. Try varying the initial POPULATION. What effect does the initial POPULATION have on the final stable population? Does it have an effect on the distribution of agent properties, such as vision and metabolism?
19. Open the SugarScape 2 Constant Growback model from the NetLogo models library. Explore the model. How dependent is the carrying capacity on the initial population size? Is there a direct relationship?

20. Open the SugarScape 3 Wealth Distribution model from the Social Science section of the NetLogo models library. Explore this model. How does the initial population affect the wealth distribution? How long does it take for the skewed distribution to emerge? How is the wealth distribution affected when you change the initial endowments of wealth? Do the results of this model seem similar to Simple Economy? Compare and contrast the results of the two models.

NetLogo Explorations

21. The NetLogo models library contains a folder of code examples, useful examples of how to code that aren't full-fledged models. Open the random walk example from the Code Examples folder in the models library. Inspect the code. What do you predict the turtle's path will look like? Run the model. Does the path look like you expected it would? Modify the model code so that the turtle's path is still random but is less "jagged," that is, is smoother and straighter.

22. Create a model that has two buttons. The first should create twenty-five turtles and scatter them around the world. The second button, when pressed, should ask each turtle that is to the left of the origin to print its WHO value to the command center output. If you press the "report WHO value button" repeatedly, what do you notice about the list that is displayed? What is the cause of this behavior? Why might this be desirable? After coding the model, please answer these questions by adding a new section of the Info tab (you can add a new section to the Info tab by clicking the "edit" button). (Hint: the OF and PRINT primitives might be useful for this activity.)