Implementing a First Agent-Based Model

4

4.1 Introduction and Objectives

In this chapter we continue your lessons in NetLogo, but from now on the focus will be on programming—and using—real ABMs that address real scientific questions. (The Mushroom Hunt model of chapter 2 was neither very agent-based nor scientific, in ways we discuss in this chapter.) And even though this chapter is mostly still about NetLogo programming, it starts addressing other modeling issues. It should prime you to start actually using an ABM to produce and analyze meaningful output and address scientific questions, which is what we do in chapter 5.

Learning objectives for chapter 4 are to:

- Understand how to translate a model from its written description in ODD format into NetLogo code.
- Understand how to define global, turtle, and patch variables.
- Become familiar with NetLogo's most important primitives, such as ask, set, let, create-turtles, ifelse, and one-of.
- Start learning good programming practices, such as making very small changes and constantly checking them, and writing comments in your code.
- Produce your own software for the Butterfly model described in chapter 3.

4.2 ODD and NetLogo

In chapter 3 we introduced the ODD protocol for describing an ABM, and as an example provided the ODD formulation of a butterfly hilltopping model. What do we do when it is time to make a model described in ODD actually run in NetLogo? It turns out to be quite straightforward because the organizations of ODD and NetLogo correspond closely. The major elements of an ODD formulation have corresponding elements in NetLogo.

Purpose

From now on, we will include the ODD descriptions of our ABMs on NetLogo's

We will talk a lot more about ODD later in this module!

Information tab. These descriptions will then start with a short statement of the model's overall purpose.

Entities, State Variables, and Scales

Basic entities for ABMs are built into NetLogo: the World of square patches, turtles as mobile agents, and the observer. The state variables of the turtles and patches (and perhaps other types of agents) are defined via the turtles-own [ ] and patches-own [ ] statements, and the variables characterizing the global environment are defined in the globals [] statement. In NetLogo, as in ODD, these variables are defined right at the start.

Process Overview and Scheduling

This, exactly, is represented in the go procedure. Because a well-designed go procedure simply calls other procedures that implement all the submodels, it provides an overview (but not the detailed implementation) of all processes, and specifies their schedule, that is, the sequence in which they are executed each tick.

Design Concepts

These concepts describe the decisions made in designing a model and so do not appear directly in the NetLogo code. However, NetLogo provides many primitives and interface tools to support these concepts; part II of this book explores how design concepts are implemented in NetLogo.

Initialization

This corresponds to an element of every NetLogo program, the setup procedure. Pushing the setup button should do everything described in the Initialization element of ODD.

Input Data

If the model uses a time series of data to describe the environment, the program can use NetLogo's input primitives to read the data from a file.

Submodels

The submodels of ODD correspond closely but not exactly to procedures in NetLogo. Each of a model's submodels should be coded in a separate NetLogo

procedure that is then called from the go procedure. (Sometimes, though, it is convenient to break a complex submodel into several smaller procedures.)

These correspondences between ODD and NetLogo make writing a program from a model's ODD formulation easy and straightforward. The correspondence between ODD and NetLogo's design is of course not accidental: both ODD and NetLogo were designed to capture and organize the important characteristics of ABMs.

4.3 Butterfly Hilltopping: From ODD to NetLogo

Now, let us for the first time write a NetLogo program by translating an ODD formulation, for the Butterfly model. The way we are going to do this is hierarchical and step-by-step, with many interim tests. We strongly recommend you always develop programs in this way:

- Program the overall structure of a model first, before starting any of the details. This keeps you from getting lost in the details early. Once the overall structure is in place, add the details one at a time.
- Before adding each new element (a procedure, a variable, an algorithm requiring complex code), conduct some basic tests of the existing code and save the file. This way, you always proceed from "firm ground": if a problem suddenly arises, it very likely (although not always) was caused by the last little change you made.

First, let us create a new NetLogo program, save it, and include the ODD description of the model on the Information tab.

- Start NetLogo and use File/New to create a new NetLogo program. Use File/Save to save the program under the name "Butterfly-1.nlogo" in an appropriate folder.
- Get the file containing the ODD description of the Butterfly model from the chapter 4 section of the web site for this book. this is on Moodle!
- Go the Information tab in NetLogo, click the Edit button, and paste in the model description at the top.
- Click the Edit button again.

You now see the ODD description of the Butterfly model at the beginning of the documentation.

Now, let us start programming this model with the second part of its ODD

description, the state variables and scales. First, define the model's state variables—the variables that patches, turtles, and the observer own.

- Go to the Procedure tab and insert

```
globals [ ]
patches-own [ ]
turtles-own [ ]
```

- Click the Check button.

There should be no error message, so the code syntax is correct so far. Now, from the ODD description we see that turtles have no state variables other than their location; because NetLogo already has built-in turtle variables for location (xcor, ycor), we need to define no new turtle variables. But patches have a variable for elevation, which is not a built-in variable, so we must define it.

- In the program, insert elevation as a state variables of patches:

```
patches-own [ elevation ]
```

If you are familiar with other programming languages, you might wonder where we tell NetLogo what type the variable “elevation” is. The answer is: NetLogo figures out the type from the first value assigned to the variable via the set primitive.

Now we need to specify the model's spatial extent: how many patches it has.

- Go to the Interface tab, click the Settings button, and change “Location of origin” to Corner and Bottom Left; change the number of columns (max-pxcor) and rows (max-pycor) to 149. Now we have a world, or landscape, of 150 × 150 patches, with patch 0,0 at the lower left corner. Turn off the two world wrap tick boxes, so that our model world has closed boundaries. Click OK to save your changes.

You probably will see that the World display (the “View”) is now extremely large, too big to see all at once. You can fix this:

- Click the Settings button again and change “Patch size” to 3 or so, until the View is a nice size. (You can also change patch size by right-clicking on the View to select it, then dragging one of its corners.)

The next natural thing to do is to program the setup procedure, where all

entities and state variables are created and initialized. Our guide of course is the Initialization part of the ODD description. Back in the Procedures tab, let us again start by writing a "skeleton."

- At the end of the existing program, insert this:

```
to setup
  ca
  ask patches
  [

  ]
  reset-ticks
end
```

Click the Check button again to make sure the syntax of this code is correct. There is already some code in this setup procedure: ca to delete everything, which is almost always first in the setup procedure, and reset-ticks, which is almost always last. The ask patches statement will be needed to initialize the patches by giving them all a value for their elevation variable. The code to do so will go within the brackets of this statement.

Assigning elevations to the patches will create a topographical landscape for the butterflies to move in. What should the landscape look like? The ODD description is incomplete: it simply says we start with a simple artificial topography. We obviously need some hills because the model is about how butterflies find hilltops. We could represent a real landscape (and we will, in the next chapter), but to start it is a good idea to create scenarios so simple that we can easily predict what should happen. Creating just one hill would probably not be interesting enough, but two hills will do.

- Add the following code to the ask patches command:

```
ask patches
[
  let elev1 100 - distancexy 30 30
  let elev2 50 - distancexy 120 100

  ifelse elev1 > elev2
    [set elevation elev1]
    [set elevation elev2]

  set pcolor scale-color green elevation 0 100
]
```

The idea is that there are two conical hills, with peaks at locations (x,y-coordinates) 30,30 and 120,100. The first hill has an elevation of 100 units

(let's assume these elevations are in meters) and the second an elevation of 50 meters. First, we calculate two elevations *(elev1* and *elev2)* by assuming the patch's elevation decreases by one meter for every unit of horizontal distance between the patch and the hill's peak (remind yourself what the primitive distancexy does by putting your cursor over it and hitting the F1 key). Then, for each patch we assume the elevation is the greater of two potential elevations, *elev1* and *elev2:* using the ifelse primitive, each patch's elevation is set to the higher of the two elevations.

Finally, patch elevation is displayed on the View by setting patch color pcolor to a shade of the color green that is shaded by patch elevation over the range 0 and 100 (look up the extremely useful primitive scale-color).

Remember that the ask patches command establishes a patch context: all the code inside its brackets must be executable by patches. Hence, this code uses the state variable elevation that we defined for patches, and the built-in patch variable pcolor. (We could *not* have used the built-in variable color because color is a turtle, not a patch, variable.)

Now, following our philosophy of testing everything as early as possible, we want to see this model landscape to make sure it looks right. To do so, we need a way to execute our new setup procedure.

- On the Interface, press the Button button, select “Button,” and place the cursor on the Interface left of the View window. Click the mouse button.
- A new button appears on the Interface and a dialog for its settings opens. In the Commands field, type “setup” and click OK.

Now we have linked this button to the setup procedure that we just programmed. If you press the button, the setup procedure is executed and the Interface should look like figure 4.1.

There is a landscape with two hills! Congratulations! (If your View does not look like this, figure out why by carefully going back through *all* the instructions. Remember that any mistakes you fix will not go away until you hit the setup button again.)

NetLogo brainteaser

When you click the new setup button, why does NetLogo seem to color the patches in spots all over the View, instead of simply starting at the top and working down? (You may have to turn the speed slider down to see this clearly.) Is this a fancy graphics rendering technique used by NetLogo? (Hint: start reading the Programming Guide section on Agentsets very carefully.)

Now we need to create some agents or, to use NetLogo terminology, turtles. We do this by using the create-turtles primitive (abbreviated crt). To create one turtle, we use crt 1 [ ]. To better see the turtle, we might wish to increase the size of its symbol, and we also want to specify its initial position.

- Enter this code in the setup procedure, after the ask patches statement:

```
crt 1
[
```

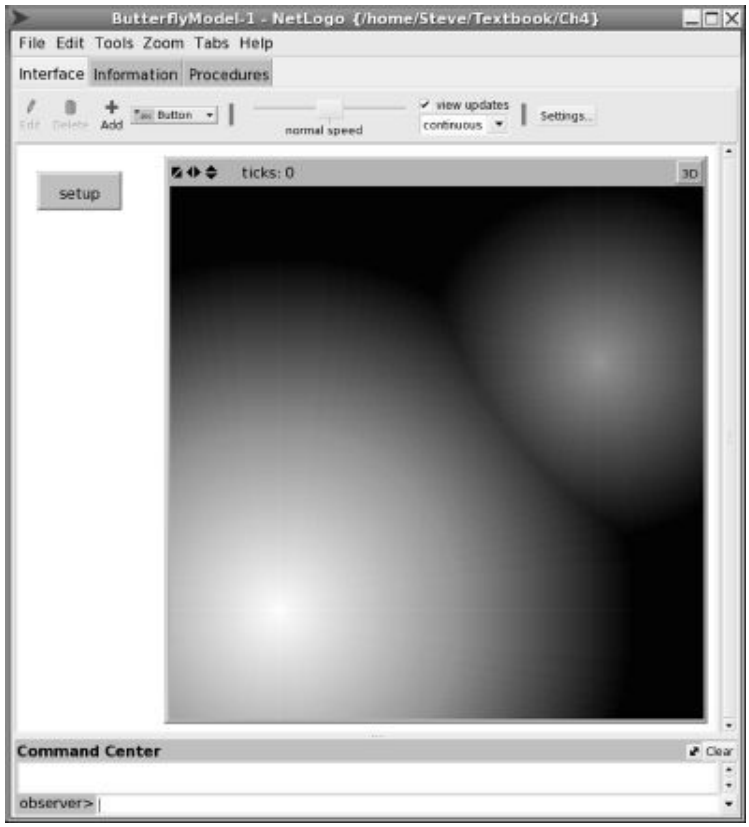

Figure 4.1
The Butterfly model's artificial landscape.

```
  set size 2
  setxy 85 95
]
```

- Click the Check button to check syntax, then press the setup button on the Interface.

Now we have initialized the World (patches) and agents (turtles) sufficiently for a first minimal version of the model.

The next thing we need to program is the model's schedule, the go procedure. The ODD description tells us that this model's schedule includes only one action, performed by the turtles: movement. This is easy to implement.

- Add the following procedure:

```
to go
  ask turtles [move]
end
```

- At the end of the program, add the skeleton of the procedure move:

```
to move

end
```

Now you should be able to check syntax successfully.

There is one important piece of go still missing. This is where, in NetLogo, we control a model's "temporal extent": the number of time steps it executes before stopping. In the ODD description of model state variables and scales, we see that Butterfly model simulations last for 1000 time steps. In chapter 2 we learned how to use the built-in tick counter via the primitives reset-ticks, tick, and ticks. Read the documentation for the primitive stop and modify the go procedure to stop the model after 1000 ticks:

```
to go
  ask turtles [move]
  tick
  if ticks >= 1000 [stop]
end
```

- Add a go button to the Interface. Do this just as you added the setup button, except: enter "go" as the command that the button executes, and activate the "Forever" tick box so hitting the button makes the observer repeat the go procedure over and over.

Programming note: Modifying Interface elements

To move, resize, or edit elements on the Interface (buttons, sliders, plots, etc.), you must first select them. Right-click the element and chose "Select" or "Edit." You can also select an element by dragging the mouse from next to the element to over it. Deselect an element by simply clicking somewhere else on the Interface.

If you click go, nothing happens except that the tick counter goes up to 1000—NetLogo is executing our go procedure 1000 times but the move procedure that it calls is still just a skeleton. Nevertheless, we verified that the overall structure of our program is still correct. And we added a technique very often used in the go procedure: using the tick primitive and ticks variable to keep track of simulation time and make a model stop when we want it to.

(Have you been saving your new NetLogo file frequently?)

Now we can step ahead again by implementing the move procedure. Look up the ODD element "Submodels" on the Information tab, where you find the details of how butterflies move. You will see that a butterfly should move, with a probability *q*, to the neighbor cell with the highest elevation. Otherwise (i.e., with probability 1 – *q*) it moves to a randomly chosen neighbor cell. The butterflies keep following these rules even when they have arrived at a local hilltop.

- To implement this movement process, add this code to the move procedure:

```
to move
  ifelse random-float 1 < q

    [ uphill elevation ]
    [ move-to one-of neighbors ]
end
```

The ifelse statement should look familiar: we used a similar statement in the search procedure of the Mushroom Hunt model of chapter 2. If *q* has a value of 0.2, then in about 20% of cases the random number created by random-float will be smaller than *q* and the ifelse condition true; in the other 80% of cases the condition will be false. Consequently, the statement in the first pair of brackets (uphill elevation) is carried out with probability *q* and the alternative statement (move-to one-of neighbors) is done with probability 1– *q*.

The primitive uphill is typical of NetLogo: following a gradient of a certain variable (here, *elevation)* is a task required in many ABMs, so uphill has been provided as a convenient shortcut. Having such primitives makes the code more compact and easier to understand, but it also makes the list of NetLogo primitives look quite scary at first.

The remaining two primitives in the move procedure (move-to and one-of) are self-explanatory, but it would be smart to look them up so you understand clearly what they do.

- Try to go to the Interface and click go.

  NetLogo stops us from leaving the Procedures tab with an error message telling us that the variable q is unknown to the computer. (If you click on the Interface tab again, NetLogo relents and lets you go there, but turns the Procedures tab label an angry red to remind you that things are not all OK there.)

  Because *q* is a parameter that all turtles will need to know, we define it as a global variable by modifying the globals statement at the top of the code:

```
globals [ q ]
```

  But defining the variable *q* is not enough: we also must give it a value, so we initialize it in the setup procedure. We can add this line at the end of setup, just before end:

```
set q 0.4
```

  The turtles will now move deterministically to the highest neighbor patch with a probability of 0.4, and to a randomly chosen neighbor patch with a probability of 0.6.

  Now, the great moment comes: you can run and check your first complete ABM!

- Go to the Interface and click setup and then go.

  Yes, the turtle does what we wanted it to do: it finds a hilltop (most often, the higher hill, but sometimes the smaller one), and its way is not straight but somewhat erratic.

  Just as with the Mushroom Hunt model, it would help to put the turtle's "pen" down so we can see its entire path.

- In the crt block of the setup procedure (the code statements inside the brackets after crt), include a new line at the end with this command: pen-down.

  You can now start playing with the model and its program, for example by modifying the model's only parameter, q. Just edit the setup procedure to change the value of *q* (or the initial location of the turtle, or the hill locations), then pop back to the Interface tab and hit the setup and go buttons. A typical model run for $q = 0.2$, so butterflies make 80% of their movement decisions randomly, is shown in figure 4.2.

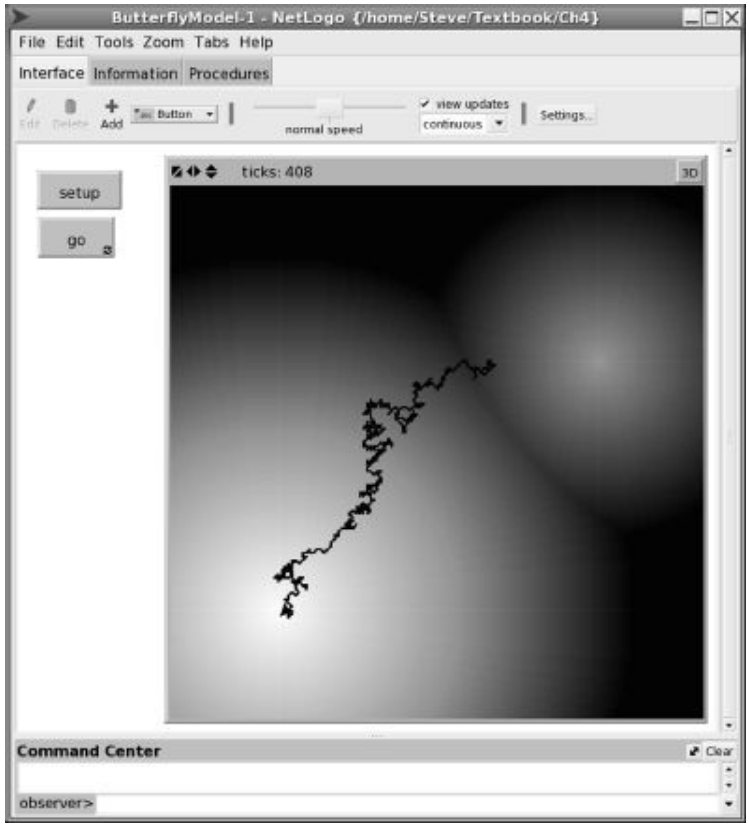

Figure 4.2
Interface of the Butterfly model's first version, with $q = 0.2$.

4.4 Comments and the Full Program

We have now implemented the core of the Butterfly model, but four important things are still missing: comments, observations, a realistic landscape, and analysis of the model. We will address comments here but postpone the other three things until chapter 5.

Comments are any text following a semicolon (;) on the same line in the code; such text is ignored by NetLogo and instead is for people. Comments are needed to make code easier for others to understand, but they are also very useful to ourselves: after a few days or weeks go by, you might not remember why you wrote some part of your program as you did instead of in some other way. (This surprisingly common problem usually happens where the code was hardest to write and easiest to mess up.) Putting a comment at the start of each procedure saying whether the procedure is in turtle, patch, or observer context helps you write the procedures by making you think about their context, and it makes revisions easier.

- As a good example comment, change the first line of your move procedure to this, so you always remember that the code in this procedure is in turtle context:

```
to move ; A turtle procedure
```

The code examples in this book use few comments (to keep the text short, and to force readers to understand the code). The programs that you can download via the book's web site include comments, and you must make it a habit to comment your own code. In particular, use comments to:

- Briefly describe what each procedure or nontrivial piece of the code is supposed to do;
- Explain the meaning of variables;
- Document the context of each procedure;
- Keep track of what code block or procedure is ended by “]” or end (see the example below); and
- In long programs, visually separate procedures from each other by using comment lines like this:

```
;————————————————
```

On the other hand, detailed and lengthy comments are no substitute for code that is clearly written and easy to read! Especially in NetLogo, you should strive to write your code so you do not need many comments to understand it. Use names for variables and procedures that are descriptive and make your code statements read like human language. Use tabs and blank lines to show the code's organization. We demonstrate this “self-commenting” code approach throughout the book.

Another important use of comments is to temporarily deactivate code statements. If you replace a few statements but think you might want the old code back later, just “comment out” the lines by putting a semicolon at the start of each. For example, we often add temporary test output statements, like this one that could be put at the end of the ask patches statement in the setup procedure to check the calculations setting elevation:

```
  show (word elev1 " " elev2 " " elevation)
```

If you try this once, you will see that it gives you the information to verify that elevation is set correctly, but you will also find out that you do not want to repeat

this check every time you use setup! So when you do not need this statement anymore, just comment it out by putting a semicolon in front of it. Leaving the statement in the code but commented out provides documentation of how you checked elevations, and lets you easily repeat the check when you want.

Here is our Butterfly model program, including comments.

```
globals [ q ] ; q is the probability that butterfly moves
              ; directly to the highest surrounding patch
patches-own [ elevation ]
turtles-own [ ]

to setup
  ca

  ; Assign an elevation to patches and color them by it
  ask patches
  [
    ; Elevation decreases linearly with distance from the
    ; center of hills. Hills are at (30, 30) and
    ; (120, 100). The first hill is 100 units high.

    ; The second hill is 50 units high.

    let elev1 100 - distancexy 30 30
    let elev2 50 - distancexy 120 100

    ifelse elev1 > elev2
        [set elevation elev1]
        [set elevation elev2]

    set pcolor scale-color green elevation 0 100
  ] ; end of "ask patches"

  ;  Create just 1 butterfly for now
  crt 1
  [
    set size 2
    ; Set initial location of butterflies
    setxy 85 95
    pen-down
  ]

  ; Initialize the "q" parameter
  set q 0.4

  reset-ticks

end ; of setup procedure

to go ;  This is the master schedule
  ask turtles [move]
  tick
  if ticks >= 1000 [stop]
end

to move ; The butterfly move procedure, in turtle context
        ; Decide whether to move to the highest
        ; surrounding patch with probability q
  ifelse random-float 1 < q
    [ uphill elevation ] ; Move deterministically uphill
    [ move-to one-of neighbors ] ; Or move randomly
end ; of move procedure
```

- Include comments in your program and save it.

we'll deal with all this specifically in later lessons!

The second thing that is missing from the model now is observation. So far, the model only produces visual output, which lets us look for obvious mistakes and see how the butterfly behaves. But to use the model for its scientific purpose—understanding the emergence of "virtual corridors"—we need additional outputs that quantify the width of the corridor used by a large number of butterflies.

Another element that we need to make more scientific is the model landscape. It is good to start programming and model testing and analysis with artificial scenarios as we did here, because it makes it easier to understand the interaction between landscape and agent behavior and to identify errors, but we certainly do not want to restrict our analysis to artificial landscapes like the one we now have.

Finally, we have not yet done any analysis on this model, for example to see how the parameter *q* affects butterfly movement and the appearance of virtual corridors. We will work on observation, a realistic landscape, and analysis in the next chapter, but now you should be eager to play with the model. "Playing"—creative tests and modification of the model and program ("What would happen if…")—is important in modeling. Often, we do not start with a clear idea of what observations we should use to analyze a model. We usually are also not sure whether our submodels are appropriate. NetLogo makes playing very easy. You now have the chance to heuristically analyze (that is: play with!) the model.

4.5 Summary and Conclusions

If this book were only about programming NetLogo, this chapter would simply have moved ahead with teaching you how to write procedures and use the Interface. But because this book is foremost about agent-based modeling in science, we began the chapter by revisiting the ODD protocol for describing ABMs and showing how we can quite directly translate an ODD description into a NetLogo program. In scientific modeling, we start by thinking about and writing down the model design; ODD provides a productive, standard way to do this. Then, when we think we have enough of a design to implement on the computer, we translate it into code so we can start testing and revising the model.

The ODD protocol was developed independently from NetLogo, but they have many similarities and correspond quite closely. This is not a coincidence, because both ODD and NetLogo were developed by looking for the key

characteristics of ABMs in general and the basic ways that they are different from other kinds of model (and, therefore, ways that their software must be unique). These key characteristics were used to organize both ODD and NetLogo, so it is natural that they correspond with each other. In section 4.2 we show how each element of ODD corresponds to a specific part of a NetLogo program.

This chapter illustrates several techniques for getting started modeling easily and productively. First, start modeling with the simplest version of the model conceivable—ignore many, if not most, of the components and processes you expect to include later.

The second technique is to develop programs in a hierarchical way: start with the skeletons of structures (procedures, ask commands, ifelse switches, etc.); test these skeletons for syntax errors; and only then, step by step, add “flesh to the bones” of the skeleton. ODD starts with the model's most general characteristics; once these have been programmed, you have a strong “skeleton”—the model's purpose is documented in the Information tab, the View's dimensions are set, state variables are defined, and you know what procedures should be called in the go procedure. Then you can fill in detail, first by turning the ODD Initialization element into the setup procedure, and then by programming each procedure in full. Each step of the way, you can add skeletons such as empty procedures and ask statements, then check for errors before moving on.

Third, if your model will eventually include a complex or realistic environment, start with a simplified artificial one. This simplification can help you get started sooner and, more importantly, will make it easier for you identify errors (either in the programming or in the logic) in how the agents interact with their environment.

Finally, formatting your code nicely and providing appropriate comments is well worth the tiny bit of time it takes. Your comments should have two goals: to make it easier to understand and navigate the code with little reminders about what context is being used, what code blocks and procedures are ended by an] or end, and so on; and to document how and why you programmed nontrivial things as you did. But remember that your code should be easy to read and understand even with minimal comments. These are not just aesthetic concerns! If your model is successful at all, you and other people will be reading and revising the code many times, so take a few seconds as you write it to make future work easy.

Now, let's step back from programming and consider what we have achieved so far. We wanted to develop a model that helps us understand how and where in a

landscape “virtual corridors” of butterfly movement appear. The hypothesis is that corridors are not necessarily linked to landscape features that are especially suitable for migration, but can also just emerge from the interaction between topography and the movement decisions of the butterflies. We represented these decisions in a most simple way: by telling the butterflies to move uphill, but with their variability in movement represented by the parameter *q*. The first results from a highly artificial landscape indicate that indeed our movement rule has the potential to produce virtual corridors, but we obviously have more to do. In the next chapter we will add things needed to make a real ABM for the virtual corridor problem: output that quantifies the “corridors” that model butterflies use, a realistic landscape for them to fly through, and a way to systematically analyze corridors.

4.6 Exercises

1. Change the number of butterflies to 50. Now, try some different values of the parameter *q*. Predict what will happen when *q* is 0.0 and 1.0 and then run the model; was your prediction correct?

2. Modify your model so that the turtles do not all have the same initial location but instead have their initial location set randomly. (Hint: look for primitives that provide random X and Y locations.)

3. From the first two exercises, you should have seen that the paths of the butterflies seem unexpectedly artificial. Do you have an explanation for this? How could you determine what causes it? Try analyzing a smaller World and add labels to the patches that indicate, numerically, their elevation. (Hint: patches have a built-in variable plabel. And see exercise 4.)

4. Try adding “noise” to the landscape, by adding a random number to patch elevation. You can add this statement to setup, just after patch elevation is set: set elevation elevation + random 10. How does this affect movement? Did this experiment help you answer the questions in exercise 3?