4 Implementing a first agent-based model

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 models that address real scientific questions. And even though this chapter is mostly still about NetLogo programming, it starts addressing other modeling issues.

First, we show that model descriptions in the ODD protocol are easily translated into NetLogo programs. Starting with the ODD description of the hilltopping butterfly model introduced in Chapter 3, we demonstrate how to program a model from scratch, using the ODD description as a blueprint. We go through the process of setting up the interface, writing and testing procedures, and visually inspecting model output. After finishing this chapter, you should be familiar with the format and structure of a NetLogo program, common programming syntax, and many of NetLogo's key primitives. And you should be primed to start actually using the model to produce and analyze meaningful output and address scientific questions, which is where we go in Chapter 5.

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 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 in a NetLogo program 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 the other procedures that implement all the model's actions, it provides an overview (but not the detailed implementation) of all processes (submodels) and specifies their schedule, i.e. 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 include a procedure that uses 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. But sometimes it is convenient to break a complex submodel into several smaller procedures.

Because of these correspondences, writing a NetLogo program from a model's ODD formulation will be 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 program first, before starting any of the details. This keeps you from getting lost in the detail early. Once the overall structure is in place, add the details one piece 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 (but 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 website for this book.
- Go the Information tab in NetLogo, click the Edit button, and paste the model description in at the top.
- Press the Edit button again.

You now see the ODD description of the Butterfly model at the beginning of the documentation. We keep the Information tab categories provided by NetLogo at the end of the Information tab so we can still use them.

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; NetLogo has built-in variables (xcor, ycor) for location. 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 x 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
   [

```

```
    ]
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. 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 brown elevation 0 100
 ]
```

The idea is that there are two 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 button). 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 `brown` 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 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 to 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". 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.

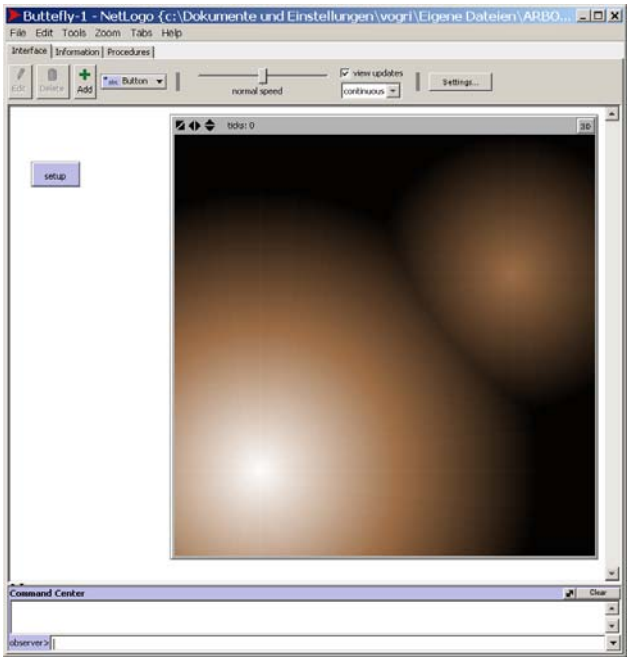

Figure 4-1. The Butterfly model's artificial landscape.

There is a landscape with two hills! Congratulations! (If your View does not look just 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 brain teaser

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.

- In the `setup` procedure, enter this code after `ask patches [ ]`:

```
crt 1
[
   set size 2
   setxy 95 80
]
```

- Press 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. After inspecting the Information tab, we realize that the schedule contains only one process, 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   ; a turtle procedure

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 `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. Un-select an element by simply clicking somewhere else on the Interface.

If you click go, nothing happens except that the tick counter goes up to 1,000—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    ; a turtle procedure
  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 short-cut. 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 declare 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
```

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

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. What if we want to see the turtle's entire path? Fortunately, NetLogo makes this easy—we just need to change the turtle's built-in variable pen-mode from its default value of "up" to "down":

- 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 start playing now with the program or model, respectively, for example by modifying the only model parameter, *q*. A typical model run for *q* = 0.2, so butterflies make 80% of their movement decisions randomly, is shown in Figure 4-2.

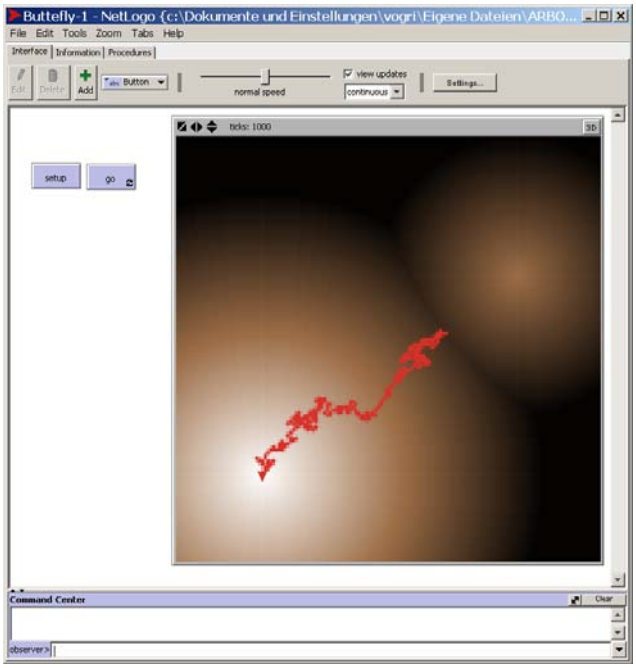

Figure 4-2. Interface of the hilltopping Butterfly model's first version, with *q* = 0.2.

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.

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.

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 from the book's website 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 non-trivial 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 de-activate code statements. If you replace a few statements but think you might want the old code back, 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 any more, just comment it out by