From Animations to Science 5

5.1 Introduction and Objectives

Beginners often believe that modeling is mainly about formulating and implementing models. This is not the case: the real work starts *after* a model has first been implemented. Then we use the model to find answers and solutions to the questions and problems we started our modeling project with, which almost always requires modifying the model formulation and software. This iterative process of model analysis and refinement—the modeling cycle—usually is not documented in the publications produced at the end. Instead, models are typically presented as static entities that were just produced and used. In fact, every model description is only a snapshot of a process.

The Models Library of NetLogo has a similar problem: it presents the models and gives (on the Information tab) some hints of how to analyze them, but it cannot demonstrate how to do science with them. These models are very good at animation: letting us see what happens as their assumptions and equations are executed. But they do not show you how to explore ideas and concepts, develop and test hypotheses, and look for parsimonious and general explanations of observed phenomena.

In this chapter we illustrate how to make a NetLogo program into a scientific model instead of just a simulator, by taking the Butterfly model and adding the things needed to do science. Remember that the purpose of this model was to explore the emergence of "virtual corridors": places where butterflies move in high concentrations even though there is nothing there that attracts butterflies. Our model so far simulates butterfly movement, but does not tell us anything about corridors and when and how strongly they emerge. Therefore, we will now produce quantitative output that can be analyzed, instead of just the visual display of butterfly movement. We will also replace our very simple and artificial landscape with a real one read in from a topography file. In the exercises we suggest some scientific analyses for you to conduct on the Butterfly model.

We assume that by now you have learned enough about NetLogo to write code yourself while frequently looking things up in the User Manual and getting help from others. Thus, we no longer provide complete code as we walk you through changes in the model, except when particularly complicated primitives or approaches are involved. Those of you working through this book without the help of instructors and classmates can consult our web site for assistance if you get stuck. And be aware that part II of this book should reinforce and deepen your understanding of all the programming techniques used here. If you feel like you have just been thrown into deep water without knowing how to swim, do your best and don't worry: things will become easier soon.

Learning objectives for this chapter are to:

- Learn the importance of version control: saving documented versions of your programs whenever you start making substantial changes.
- Understand the concept that using an ABM for science requires producing quantitative output and conducting simulation experiments; and execute your first simulation experiment.
- Learn to define and initialize a global variable by creating a slider or switch on the Interface.
- Develop an understanding of what reporters are and how to write them.
- Start learning to find and fix mistakes in your code.
- Learn to create output by writing to an output window, creating a time-series plot, and exporting plot results to a file.
- Try a simple way to import data into NetLogo, creating a version of the Butterfly model that uses real topographic data.

5.2 Observation of Corridors

Our Butterfly model is not ready to address its scientific purpose in part because it lacks a way to quantitatively observe the extent to which virtual corridors emerge. But how would we characterize a corridor? Obviously, if all individuals followed the same path (as when they all start at the same place and *q* is 1.0) the corridor would be very narrow; or if movement were completely random we would not expect to identify any corridor-like feature. But we need to quantify how the width of movement paths changes as we vary things such as *q* or the landscape topography. Let's do something about this problem.

First, because we are about to make a major change in the program:

Create and save a new version of your butterfly software.

Programming note: Version control

Each time you make a new version of a model, make a new copy of the program by saving it with a descriptive name, perhaps in a separate directory. (Do this first, before you change anything!) Then, as you change the model, use comments in the Procedures tab to document the changes you made to the program, and update the Information tab to describe the new version. Otherwise you will be very sorry! It is very easy to accidentally overwrite an older model version that you will need later, or to forget how one version is different from the others. It is even a good idea to keep a table showing exactly what changes were made in each copy of a model.

Even when you just play around with a model, make sure you save it first as a temporary version so you don't mess up the permanent file.

Programmers use the term "version control" for this practice of keeping track of which changes were made in which copy of the software. Throughout this course we will be making many modifications to many programs; get into the habit of careful version control from the start and spare yourself from lots of trouble and wasted time.

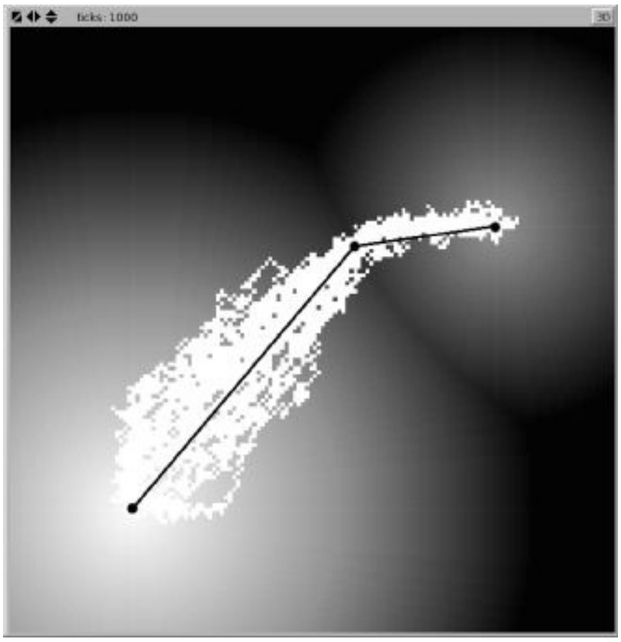

Figure 5.1

The measure of corridor width for a group of uphill-migrating butterflies (here, 50 butterflies that started at location 85,95 with q = 0.4). Corridor width is (a) the number of patches used by any butterflies (the white patches here), divided by (b) the average straight-line distance (in units of patch lengths) between butterflies' starting patch and ending hilltop. Here, most butterflies went up the large hill to the lower left, but some went up the smaller hill to the right. The number of white patches is 1956 and the mean distance between butterfly starting and ending points (the black dots) is 79.2 patches; hence, corridor width is 24.7 patches.

Pe'er *et al.* (2005) quantified corridor width by dividing the number of patches visited by all individuals during 1000 time steps by the distance between the start patch and the hill's summit. In our version of the model, different butterflies can start and end in different places, so we slightly modify this measure. First, we will assume each butterfly stops when it reaches a local hilltop (a patch higher than all its eight neighbor patches). Then we will quantify the width of the "corridor" used by all butterflies as (a) the number of patches that are visited by any butterflies divided by (b) the mean distance between starting and ending locations, over all butterflies (figure 5.1). This corridor width measure should be low (approaching 1.0) when all butterflies follow the same, straight path uphill, but should increase as the individual butterflies follow different, increasingly crooked, paths.

To analyze the model we are thus going to produce a plot of corridor width vs. *q*. This is important: when analyzing a model, we need to have a clear idea of what kind of plot we want to produce from what output, because this tells us what kind of simulation experiments we have to perform and what outputs we need the program to produce (see also chapter 22).

Now that we have determined what we need to change in the model to observe and analyze corridor width, let's implement the changes.

Because it is now obvious that we need to conduct experiments by varying q and seeing its effect, create a slider for it on the Interface tab. (The Interface Guide of NetLogo's User Manual will tell you how.) Set the slider so that q varies from 0.0 to 1.0 in increments of 0.01.

Change the setup procedure so that 50 individuals are created and start from the same position. Then vary q via its slider and observe how the area covered by all the butterfly paths changes.

Programming note: Moving variables to the Interface
As we start doing experiments on a model, we often need to vary the value of some parameter many times. NetLogo provides sliders, switches (for boolean variables), and choosers on the Interface tab to make this convenient: once we know which global variable we want to change, we move it to the Interface by creating one of these Interface elements to control the variable. But moving a variable to the Interface is often a source of frustration to beginners. It is important to understand that these Interface controls both *define* and *initialize* a variable; here are some tips.

First, only global variables can be controlled by sliders, switches, and choosers; not turtle or patch variables. Second, once one of these Interface controls has been created for a global variable, that variable cannot still appear in the globals statement at the top of the Procedures. We recommend commenting the variable out of the globals statement so when you read the procedures you are reminded that the variable exists.

The biggest potential problem is forgetting to remove statements setting the variable's value from the setup procedure. If, for example, you put the Butterfly model's q parameter in a slider and set the slider to 0.5, but forget to remove the statement set q 0.2 from the setup procedure, setup will reset the slider to 0.2 every model run. NetLogo will not tell you about this mistake. So whenever you move a variable to the Interface, comment out its initialization statement in setup.

Modify the move procedure so that butterflies no longer move if they are at a hilltop. One way to do this is to test whether the turtle is on a patch with elevation greater than that of all its neighbors and, if so, to skip the rest of the move procedure. The code for this test is a little more complex than other statements you have used so far, so we explain it. Put this statement at the start of the move procedure:

```
if elevation >=
     [elevation] of max-one-of neighbors [elevation]
  [stop]
```

This statement includes a typical NetLogo compound expression. Starting on the left, it uses the if primitive, which first checks a true-false condition and then executes the commands in a set of brackets. The statement ends with [stop], so we know that (a) the true-false condition is all the text after if and before [stop], and (b) if that condition is true, the stop primitive is executed.

You may be confused by this statement because it is clearly in turtle context—the move procedure is executed by turtles—but it uses the patch variable elevation. The reason it works is that in NetLogo turtles always have access to the variables of their current patch and can, as they do here, read and write them just as if they were turtle variables.

To understand the right side of the inequality condition (after ">="), you need to understand the reporter max-one-of, which takes an agentset—here, the set of patches returned by the reporter neighbors—and the name of a reporter. (Remember that a reporter is a procedure that returns or "reports" something—a number, an agent, an agentset, etc.—back to the procedure that called it.) The reporter here is just [elevation], which reports the elevation of each neighbor patch. It is very important to understand that you can get the value of an agent's variables (e.g., the elevation of a neighbor patch) by treating them as reporters in this way.

You also need to understand the of reporter. Here, in combination with max-one-of, it reports the elevation of the neighbor patch that has highest elevation. Therefore, the if statement's condition is "if the elevation of my current patch is greater than or equal to the elevation of the highest surrounding patch."

Compound expressions like this may look somewhat confusing at the beginning, which is not surprising: this one-line statement uses five primitives. But you will quickly learn to understand and write such expressions and to appreciate how easy they make many programming tasks.

Now we need to calculate and record the corridor width of the butterfly population. How? Remembering our definition of corridor width (figure 5.1), we need (a) the number of patches visited and (b) the mean distance between the butterflies' starting and stopping patches. We can count the number of patches used by butterflies by giving each patch a variable that keeps track of whether a turtle has ever been there. The turtles, when they stop moving, can calculate the distance from their starting patch if they have a variable that records their starting patch. Thus, we are going to introduce one new state variable for both patches and turtles. These variables are not used by the patches and turtles themselves but by us to calculate corridor width and analyze the model.

Add a new variable to the patches-own statement: used?. This is a *boolean* (true-false) variable, so we follow the NetLogo convention of ending the variable name with a question mark. This variable will be true if any butterfly has landed in the patch.

Add a variable called start-patch to the turtles-own statement.

In the setup procedure, add statements to initialize these new variables. At the end of the statement that initializes the patches (ask patches […]), add set used? false. At the end of the turtles' initialization in the crt statement, set start-patch to the patch on which the turtle is located.

The primitive patch-here reports the patch that a turtle is current on. When we assign this patch to the turtle's variable start-patch, the turtle's initial patch becomes one of its state variables. (State variables can be not just numbers but also agents and lists and agentsets.)

Programming note: Initializing variables

In NetLogo, new variables have a value of zero until assigned another value by the program. So: *if* it is OK for a variable to start with a value of zero, it is not essential to initialize it—*except* that it is very important to form the habit of always assigning an initial value to all variables. The main reason is to avoid the errors that result from forgetting to initialize variables (like global parameters) that need nonzero values; such errors are easy to make but can be very hard to notice and find.

Now we need the patches to keep track of whether or not a butterfly has ever landed on them. We can let the butterflies do this:

Add a statement to the move procedure that causes the butterfly, once it has moved to a new patch, to set the patch's variable used? to "true." (Remember that turtles can change their patch's variables just as if they were the turtle's variables.)

Finally, we can calculate the corridor width at the end of the simulation.

In the go procedure's statement that stops the program after 1000 ticks, insert a statement that executes once before execution stops. This statement uses the very important primitive let to create a local variable final-corridor-width and give it the value produced by a new reporter corridor-width.

Write the skeleton of a reporter procedure corridor-width, that reports the mean path width of the turtles. Look up the keyword to-report in the NetLogo Dictionary and read about reporters in the “Procedures” section of the Programming Guide.

In the new corridor-width procedure:

Create a new local variable and set it to the number of patches that have been visited at least once. (Hint: use the primitive count.)

Also create a new local variable that is the mean, over all turtles, of the distance from the turtle's current patch and its starting patch. (Look at the primitives mean and distance.)

From the above two new local variables, calculate corridor width and report its value as the result of the procedure.

In the go procedure, after setting the value of final-corridor-width by calling corridor-width, print its value in an output window that you add to the Interface. Do not only print the value of final-corridor-width itself, but also the text “Corridor width:”. (See the input/output and string primitives in the NetLogo Dictionary.)

If you did everything right, you should obtain a corridor width of about 25 when q is 0.4. The output window should now look like figure 5.2.

Figure 5.2
An output window displaying corridor width.

Programming note: Troubleshooting tips

Now you are writing code largely on your own so, as a beginner, you will get stuck sometimes. Getting stuck is OK because good strategies for troubleshooting are some of the most important things for you to learn now. Important approaches are:

- Proceeding slowly by adding only little pieces of code and testing before you proceed;
- Frequently consulting the User Manual;
- Looking up code in the Models Library and Code Examples;
- Using Agent Monitors and temporarily putting "show" statements (use the `show` primitive to output the current value of one or several variables) in your program to see what's going on step by step;
- Looking for discussions of your problem in the NetLogo User's Forum; and
- Using both logic ("It should work this way because…") and heuristics ("Let me try this and see what happens…").

And be aware that the next chapter is dedicated entirely to finding, fixing, and avoiding mistakes.

5.3 Analyzing the Model

Now we can analyze the Butterfly model to see how the corridor width output is affected by the parameter *q*. If you use the slider to vary *q* over a wide range, write down the resulting corridor widths, and then plot corridor width versus *q*, you should get a graph similar to figure 5.3. (NetLogo has a tool called BehaviorSpace that helps with this kind of experiment; you will learn to use it in chapter 8.) we will talk about this tool in class explicitly!

These results are not very surprising because they mainly show that with less randomness in the butterflies' movement decisions, movement is straighter and therefore corridor width smaller. It seems likely that a less-artificial landscape would produce more interesting results, but our simplified landscape allowed us to test whether the observation output behaves as we expected.

There is one little surprise in the results: corridor width is well below 1.0 (about 0.76) when *q* is 1.0. How is it possible that the number of patches visited is 24% less than the distance between starting and ending patches? Do the butterflies escape the confines of Euclidean geometry and find a path that's shorter than the straight-line distance? Is there something wrong with the distance primitive? We leave finding an explanation to the exercises.

5.4 Time-Series Results: Adding Plots and File Output

Now let's try something we often need to do: examine results over time as the simulation proceeds instead of only at the end of the model run. Plots are extremely useful for observing results as a model executes. However, we still need results written down so we can analyze them, and we certainly do not want to write down results for every time step from a plot or the output window. Instead, we need NetLogo to write results out in a file that we can analyze.

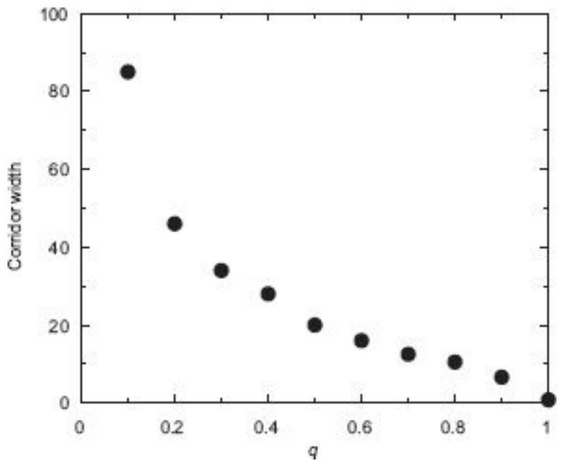

Figure 5.3
Corridor width of the Butterfly model, using the topography and start patch in figure 5.1, as a function of the parameter *q*. Points are the results of one simulation for each value of *q*.

But before we can add any of these time-series outputs, we have to change the Butterfly model program so it produces results every time step. Currently we calculate corridor width only once, at the end of a simulation; now we need to change the model's schedule slightly so that butterflies calculate their path width each time step. The go procedure can look like this:

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

The new plot statement does two things: it calls the corridor-width reporter to get the current value of the corridor width, and then it sends that value to be the next point on a plot on the Interface tab. (Starting with version 5.0, NetLogo allows you to write the code to update plots directly in the plot object on the Interface. We prefer you keep the plotting statements in your code, where it is easier to update and less likely to be forgotten. In the dialog that opens when you add a plot to the Interface, there may be a window labeled "Update commands" that contains text such as plot count turtles. Just erase that text.)

Change the go procedure as illustrated above.

Read the Programming Guide section on "Plotting," and add a plot to the Interface. Name the plot "Corridor width." Erase any text in the "Update commands" part of the plot dialog.

Test-run the model with several values of *q*.

Looking at this plot gives us an idea of how corridor width changes with *q* and over time, but to do any real analysis we need these results written out to a file. One way to get file output (we will look at other ways in chapter 9) is with the primitive export-plot, which writes out all the points in a plot at once to a file.

Read the NetLogo Dictionary entries for export-plot and word, and then modify the statement that stops the program at 1000 ticks so it first writes the plot's contents to a file. The name of the file will include the value of q, so you will get different files each time you change q. Use this statement:

```
export-plot "Corridor width"
    word "Corridor-output-for-q-" q
```

Now, with several model runs and a little spreadsheet work, you should be able to do analyses such as figure 5.4, which shows how corridor width changes over time for several values of *q*.

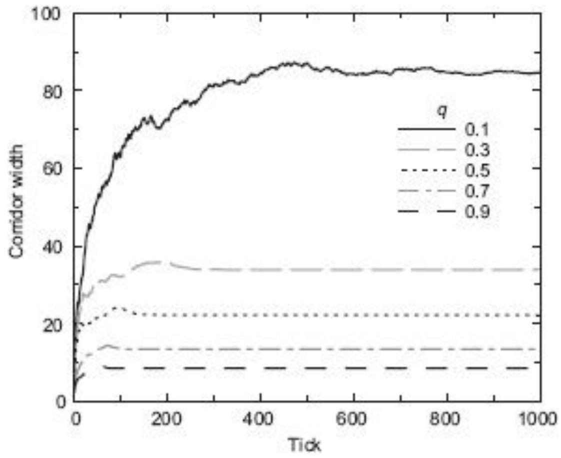

Figure 5.4
How corridor width changes during simulations, for five values of *q*. Each line is output from one model run.

Programming note: CSV files

When NetLogo "exports" a file via export-plot and other export-primitives, it writes them in a format called *comma-separated values or .csv.* This file format is commonly used to store and transfer data, and .csv files are readily opened by common data software such as spreadsheet applications. If you are not familiar with .csv format, see, for instance, the Wikipedia article on comma-separated values, or just open a .csv file in a word processor or text editor to see what it looks like. The format is very simple: each line in the file corresponds to a data record (e.g., a spreadsheet row); the values or fields are separated by commas, and values can, optionally, be enclosed in double quotes. In chapter 6 you will learn to write your own output files in .csv format.

The .csv files produced by export-plot and other primitives can cause confusion on computers that use commas as the decimal marker and semicolons as the field delimiter in .csv files (as many European ones do). The .csv files produced by NetLogo always use commas to delimit fields, and periods as the decimal marker. If you have this problem, you can solve it by editing the output to replace "," with ";" and then replace "." with ",". Or you can temporarily change your computer to use English number formats while you import your NetLogo output file to a spreadsheet or other analysis software.

5.5 A Real Landscape

Now we will see what happens to butterfly behavior and virtual corridors when we use a real, complex landscape. Importing topographies of real landscapes (or any other spatial data) into a NetLogo program is straightforward—once the data are prepared for NetLogo. On this book's web site, we provide a plain text file that you can import to NetLogo and use as the landscape. This file (called ElevationData.txt) is from one of the sites where Guy Pe'er carried out field

this elevation file is on Moodle!

studies and model analyses of butterfly movements (Pe'er 2003; Pe'er *et al.* 2004, 2005). The file contains the mean elevation of 25-meter patches.

Programming note: A simple way to import spatial data

Here is a way to import grid-based spatial data as patch variables. By “grid-based,” we mean data records that include an x-coordinate, a y-coordinate, and a variable value (which could be elevation or anything else that varies over space) for points or cells evenly spaced in both x and y directions as NetLogo patches are.

The challenging part is transforming the data so they conform to the requirements of NetLogo's World. Real data are typically in coordinate systems (e.g., UTM) that cannot be used directly by NetLogo because NetLogo requires that patch coordinates be integers and that the World includes the location 0,0. The transformations could be made via NetLogo code as the data are read in, but it is safest to do it in software such as a spreadsheet that lets you see and test the data before they go into NetLogo. These steps will prepare an input file that provides the value of a spatial variable to each patch. (It is easily modified to read in several patch variables at once.)

The spatial input must include one data line for each grid point or cell in a rectangular space. The data line should include the x-coordinate, the y-coordinate, and the value of the spatial variable for that location.

NetLogo requires the point 0,0 to be in the World. It is easiest to put 0,0 at the lower left corner by identifying the minimum x-and y-coordinates in the original data, and then subtracting their values from the x-and y-coordinates of all data points.

NetLogo patches are one distance unit apart, so the data must be transformed so points are exactly 1 unit apart. Divide the x-and y-coordinates of each point by the spatial resolution (grid size, the distance between points), in the same units that the data are in. For example, if the points represent squares 5 meters on a side and the coordinates are in meters, then divide the x-and y-coordinates by 5. As a result, all coordinates should now be integers, and grid coordinates should start at 0 and increase by 1. Save the data as a plain text file.

In NetLogo's Model Settings window, set the dimensions to match those of your data set (or, in the `setup` procedure, use the primitive `resize-world`). Turn world wrapping off (at least temporarily) so that NetLogo will tell you if one of the points in the input file is outside the World's extent.

Place the spatial data file (here, ElevationData.txt) in the same directory as your NetLogo program. (See the primitive user-file for an alternative.)

Use code like this (which assumes you are reading in elevation data) in your setup procedure:

```
file-open "ElevationData.txt"
while [not file-at-end?]
[
 let next-X file-read
 let next-Y file-read
 let next-elevation file-read
 ask patch next-X next-Y [set elevation next-elevation]
]
file-close
```

The ask patch statement simply tells the patch corresponding to the location just read from the file (see primitive patch) to set its elevation to the elevation value just read from the file.

(If you expect to use spatial data extensively, you should eventually become familiar with NetLogo's GIS extension; see section 24.5.)

Create a new version of your model and change its setup procedure so it reads in elevations from the file ElevationData.txt. (Why should you *not* place the new code to read the file and set patch elevations from it inside the old ask patches [] statement?)

Look at the elevation data file to determine what dimensions your World should have.

Change the statement that shades the patches by their elevation so the color is scaled between the minimum and maximum elevations. (See the primitives max and min.)

When we try the program now with the real landscape, we realize that the rule causing butterflies to stop moving when they reach a hilltop was perhaps good for quantifying corridor width in an artificial landscape, but in this real landscape it makes butterflies quickly get stuck on local hilltops. It is more interesting now to use the original model of Pe'er *et al.* (2005) and let the individuals keep moving for 1000 ticks even after they reach a hilltop.

Comment out the statement causing butterflies to stop if they are at a local hilltop.

We also might want to let our butterflies start not from a single patch, but a small region.

In the `crt` block, assign `xcor` and `ycor` random values that are within a small square of 10 by 10 patches, located somewhere you choose.

The model world should now look like figure 5.5, in which 50 individuals start in a small square and then move for 1000 time steps with *q* set to 0.4.

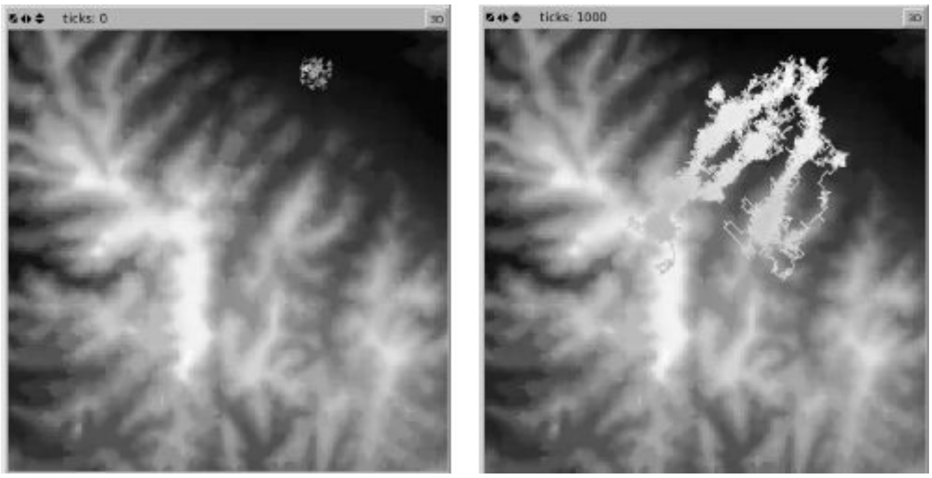

Figure 5.5
Topography of a real landscape showing (left) butterfly starting locations (initial x ranges 100–109; y is 130–139) and (right) virtual corridors that emerged from movement behavior and topography with q = 0.4.

5.6 Summary and Conclusions

The title of this chapter promises to take you from using NetLogo only for simple animations to doing agent-based science, but of course we made only a few first steps: modeling a system of multiple agents, adding quantitative observations that we can analyze, starting to use simulation experiments, and using real spatial data. We will, though, continue to build such skills for doing science with ABMs for the rest of the book.

The Butterfly model is extremely simple, but it already requires some difficult decisions—there are, for example, several different ways you could define path widths and corridor sizes, which likely would produce different results. It is thus important that you learn how to analyze simple models before you turn to more complex ones.

One reason the Butterfly model is good to start with is that people in all disciplines can understand it, even though it has been used in serious ecological research (Pe'er *et al.* 2004, 2005, 2006). But remember that ABMs and NetLogo are not restricted to organisms moving through landscapes. Almost exactly the same techniques and code can be used to model very different kinds of agents (people, organizations, ideas…) moving through many kinds of "space" (political or economic landscapes, social networks, etc.), and NetLogo can model agents that interact without moving at all.

5.7 Exercises

1. Implement the new versions of the Butterfly model that we introduce in this chapter.

2. The traces left by turtles in their "pen-down" mode are useful for seeing their path, but make it difficult to see the turtles' final location. Make a new button on the Interface of one of your butterfly models that erases these traces. (Hint: for moving and drawing, there is very often a primitive that does exactly what you want.)

3. Does your version of the model from section 5.2 produce the result that corridor width is less than 1.0 when *q* is 1.0? How can this be? (Hint: there are two reasons. Look at how the butterflies move, think about how space is represented, and make sure you understand how the primitive distance works.)

4. Try running the Butterfly model with the butterflies all starting in one corner of the space. Remember that we have world wrapping turned off, so the model would stop and report an error if a turtle tried to cross the edge of the World. Why do the butterflies not cross the edge and out of the World, causing an error, no matter how low a value of *q* you use?

5. Using the model version with a real landscape, vary the parameter *q* and see how it affects the interaction between landscape and butterfly movement. Start the butterflies in various places. From just looking at the View, what value of *q* seems best for finding mates on or near hilltops?

6. Analyze the question in exercise 5 in a more rigorous way. Have 50

butterflies start within an area of 10 × 10 patches near the edge of the landscape and move for 1000 ticks. Develop a measure of how closely clumped the butterflies are at the end of the run, to serve as an indicator of how likely they are to find a mate. One example is the average, over all butterflies, of the number of other butterflies within some radius. Another example is the number of patches containing butterflies (the lower this number, the more clumped the butterflies). Conduct simulation experiments to see how this indicator of mating success varies with *q*.

7. Continuing from exercise 6, repeat the experiment with the butterflies starting in different parts of the landscape. Do the results depend on the local topography?

8. Try some alternative models of how butterflies move. For example, butterflies might not choose the highest neighbor patch, but instead randomly select among all the neighbor patches that are higher. What effect do the different traits have in the two landscapes (e.g., on the relation between *q* and corridor width)?

9. With your version of the Butterfly model, can you perform all the analyses that Pe'er *et al.* (2006) did with their version?

10. On this book's web site is another topographic input file for a rectangular area. Download it, make the necessary transformations, and use it in the Butterfly model.

11. Another ABM as simple as the hilltopping Butterfly model is that of Jovani and Grimm (2008). Obtain this publication (it is available through this book's web site), read its ODD model description (a version of which is also in section 23.4.1 of this book), and program the model in NetLogo. (The publication omits the value of parameter *SD*, which should be 1.0.) See if you can reproduce the general results reported in this paper.