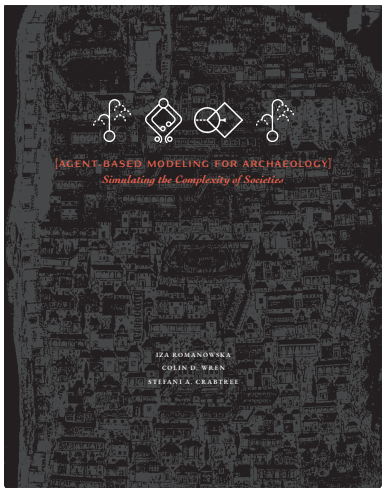
PLEASE NOTE:

The contents of this open-access PDF are excerpted from the following textbook, which is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License:

Romanowska, I., C. D. Wren, and S. A. Crabtree. 2021. *Agent-Based Modeling for Archaeology: Simulating the Complexity of Societies.* Santa Fe, NM: SFI Press.

This and other components, as well as a complete electronic copy of the book, can be freely downloaded at https://santafeinstitute.github.io/ABMA

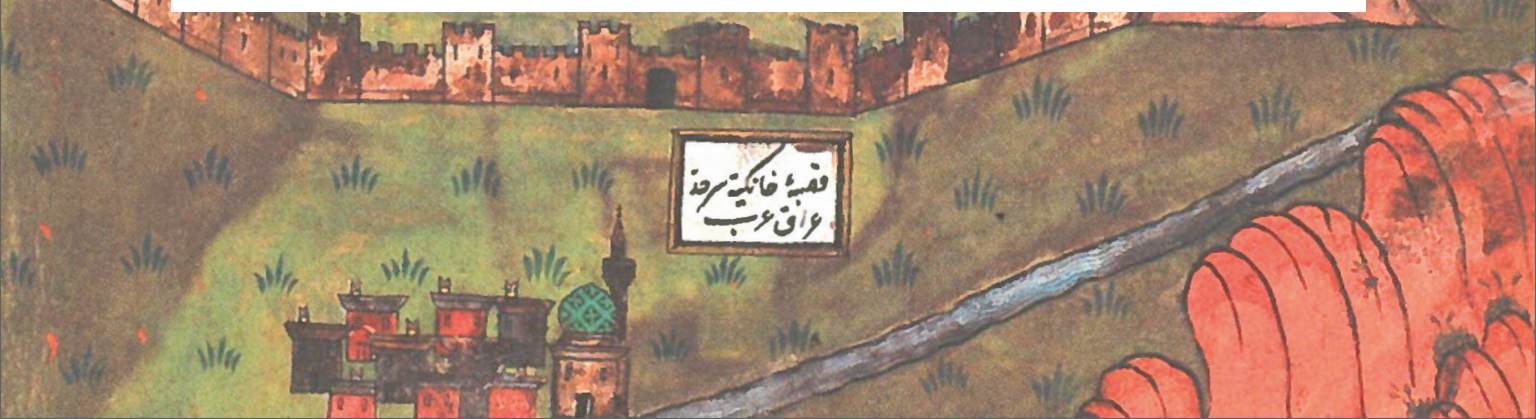


REGARDING COLOR:

The color figures in this open-access version of *Agent-Based Modeling for Archaeology* have been adapted to improve the accessibility of the book for readers with different types of color-blindness. This often results in more complex color-related aspects of the code than are outlined within the code blocks of the chapters. As such, the colors that appear on your screen will differ from those of our included figures. See the "Making Colorblind-Friendly ABMs" section of the Appendix to learn more about improving model accessibility.

SFI PRESS

THE SANTA FE INSTITUTE PRESS 1399 Hyde Park Road, Santa Fe, New Mexico 87501 | sfipress@santafe.edu




بالش سر گنبد

REAPING THE REWARDS: ADDRESSING ARCHAEOLOGICAL QUESTIONS

3.0 Introduction

In the previous chapter, you mastered the foundations of computer programming: lists, loops, variables, etc. Proficiency in coding is a challenging but necessary aspect of computational modeling, as it enables a swifter and easier manipulation of the models you write in NetLogo or any other programming language. The skills you develop in NetLogo syntax and the knowledge of primitives add a level of dynamism in representing agents and their environment. Similarly, writing elegant code that does not repeat commands or have unnecessary redundancies facilitates the subsequent steps of the modeling process. The skills you are building will enable you, as a modeler, to approach more difficult concepts and ask more complicated questions.

OVERVIEW
▷ Tutorial in advanced NetLogo: complex structures
▷ Definitions of parameterization, artificial data, validation, and equifinality
▷ Simulating archaeological record and validation against data
▷ Writing efficient code

Most of the agent-based models that approach complicated questions require these more advanced programming techniques. Modeling subsistence requires some of the concepts we dealt with in the first two chapters—movement to find a resource patch and exchange code to gather it—but also quite a lot of accounting code to sort through all the rates, amounts, and transactions. As usual, we will start with a basic model, and then build up to discuss the programming concepts in more detail. The goal is to replicate the *SugarScape* model, which you can then take further toward the full *Artificial Anasazi* simulation. As opposed to the first two chapters, here we will not introduce many new coding concepts. Instead we will make the code more complex, with loops within loops, lists within lists, etc. In addition, we will give you most of the necessary code but also leave some small coding decisions to you. By the end of this chapter, you should have all the fundamental building blocks of NetLogo code well in hand.

Since NetLogo is a language, the building blocks include the vocabulary (primitives), grammar (syntax), and conceptual logic.

We will also touch upon a critical modeling phase: **validation**. We looked at **verification** (debugging) in chapter 2. This focused on remov-

validation: establishing that the model is conceptually correct with respect to the system being modeled.

ing mistakes in the code. In contrast, the validation phase is designed to demonstrate that your model is a reasonably accurate representation of reality. You need to get this phase right so that the model can serve its intended purpose. There is much discussion within the modeling community about how to facilitate validation, avoid potential pitfalls, and determine the minimum requirements to ensure that the results produced by the model can be used to infer the dynamics of the real-world system (e.g., Arifin and Madey 2015). Without doubt you will recognize this challenge when building your own models.

3.1 The Model: SugarScape & Artificial Anasazi

Perhaps one of the longest-running and most hotly debated questions in North American archaeology centers around why the Ancestral Pueblo people chose to abandon their homelands in the late AD 1200s. Given the complexity of their societal structures, relations with their neighbors, and interactions with the constantly shifting environment, it clearly merited a formal model for investigation. Two families of models have examined the questions surrounding the depopulation of the Ancestral Puebloan settlements in the US Southwest: the *Artificial Anasazi* model led by Axtell (Axtell et al. 2002; Dean et al. 2000), later reproduced by multiple independent labs (Janssen 2009; Stonedahl and Wilensky 2010), and the *Village Ecodynamics Project* led by Kohler (Crabtree et al. 2017; Kohler and Varien 2012).

The Ancestral Puebloans were referred to as the Anasazi until the 1980s. However, "Anasazi" is considered disrespectful by living Pueblo people, so the term has been changed.

To briefly sum up the main points, the Ancestral Pueblo people migrated into the northern part of the American Southwest (areas of northern Arizona, southern Utah, and southwestern Colorado) in the early sixth and seventh centuries AD, settling as maize farmers in the region. After seven centuries of successful farming, they abandoned the region and moved to New Mexico in the late thirteenth century, leaving behind much of their material culture. Throughout their occupation, regular episodes of drought beset the farmers in these regions (Bocinsky and Kohler 2014; Kintigh and Ingram 2018) as well as periodic incidents of violence (Kohler et al. 2014; Kohler, Cole, and Ciupe 2009; Kuckelman 2010), yet they continued to inhabit the region. So what, in the end, made them all decide to leave?

Solving this question requires looking at the full historical trajectory of Ancestral Puebloans within the region, which is why the teams of Axtell and Kohler independently turned to agent-based modeling. In chapter 8, we will reimplement one of the published models from the *Village Ecodynamics Project*, originally written in Java and translated to NetLogo. In this chapter, however, we will focus on *Artificial Anasazi*, replicated in NetLogo by Janssen (2009). It will allow us to explore how building complexity on top of a simple base model can generate new understanding.

It is very common to build an ABM of a particular case study upon an existing more general model.

Artificial Anasazi is built on a simple resource extraction model: the *SugarScape* model (Epstein and Axtell 1996). It has agents moving about a landscape with resources distributed across patches and attempting to collect and metabolize as much "sugar" as they can. Dean et al. (2000) set out to test a hypothesis that environmental variability led to the abandonment of Long House Valley, Arizona. They wanted to assess how much of the historical trajectory of this society was due to the climatic changes that affected the valley. To investigate this, they combined a model of environmental variability with an expanded version of the *SugarScape model*. Agents in *Artificial Anasazi* represent maize-farming households that move across the landscape, constantly looking for the most productive locations to farm. Households grow due to reproductive capacity related to farming productivity, but if they cannot grow sufficient amounts of maize, they die. Climate fluctuation over the simulated period of 500 years changes the amount of resources that agents can gather.

Environmental change is often a first go-to narrative explanation for cultural change. However, human complex systems are rarely that simple, and other factors need to be considered.

If the model showed that Ancestral Puebloans struggled to make a living during the thirteenth century, then the environmental variability hypothesis would be supported as plausible. Otherwise, it would have to be rejected. And so it was. The simulation was instrumental in demonstrating that environmental variability alone cannot explain the depopulation of this region in the fourteenth century. At the same time, it showed that a computer model can capture the dynamics of an ancient population. With a bit of tuning of the model parameters, *Artificial Anasazi* produced an *in silico* archaeological record similar to the one created by the people who inhabited Long House Valley over half a millennium ago.

3.2 First Steps in Subsistence & Model Accounting

Model: Archaeological SugarScape

ABMA Code Repo: *ch2_sugar_basic*

The first step in any subsistence model is to decide on the characteristics of a resource landscape. We already touched on the issue of how agents assess patch variables and how they pass data between one another. In this chapter, we will focus on the nature of that agent–environment interaction in more detail. Previously, patches were passive entities holding variables and having them read by agents (e.g., agents checking `pcolor` in chapter 1). But a patch is really an agent itself, just an immovable one, and we can give patches *agency* when needed. So an agent consuming a resource from a patch is a specific case of an agent–agent trade similar to the one we developed in the previous chapter; a resource is passed from the patch to the agent, and the variables of each are updated accordingly. Subsistence, though, includes a combination of patch and agent procedures including regrowth rates, gathering rates, heterogeneous resource landscapes, search algorithms for finding the best source of resources, and even population dynamics that stem from having either insufficient or a surplus of food. The simplest way is to start with a homogeneous landscape, but all sorts of interesting things can happen when we introduce spatially variable resources including randomly scattered patches, gradients, and shaped "hills." Our basic subsistence model here will include procedures for gathering, plant regrowth, consumption of minimal stored resources, and searching for new plots. Agents will each be a small forager group, and each time step of the model will be a month.

DON'T FORGET
Remember to include `clear-all` and `reset-ticks` in `setup`, and `tick` in `go`.

We will take a closer look at landscapes in chapter 7.

Begin by creating a new model[1] and write code for the details listed below without looking ahead to the upcoming code block:

`random-pxcor` and `random-pycor` are shortcut primitives that look at the size of your landscape and return a randomized coordinate within that range. `move-to one-of patches` would also work.

- `go` and `setup` procedures and their buttons.
- Add a third button named `step`, where the command is also `go` but the FOREVER box is unchecked. This allows us to slowly *step* through the model tick by tick.
- Create a `forager` breed and a `number-foragers` slider on the INTERFACE.

[1] You can find all code written in this chapter in the ABMA Code Repo: https://github.com/SantaFeInstitute/ABMA/tree/master/ch3

- Write code in the `setup` to create those foragers, give them a shape, scatter them randomly across the landscape, and create a `storage` variable initialized to 0.
- Add a `resources` variable for patches in `patches-own`.
- Put code in `setup` to give patches a base `resources` value of 10 and `pcolor` them green.

Your code should resemble something like this:

CODE BLOCK 3.0

```
breed [foragers forager]
globals [ ]
patches-own [ resources ]
foragers-own [ storage ]

to setup
  ca
  ask patches [
    set resources 10
    set pcolor green
  ]

  create-foragers number-foragers [
    set shape "house"
    set storage 0
    setxy random-pxcor random-pycor
  ]
  reset-ticks
end

to go

  tick
end
```

If you get an error, make sure you created a storage variable for the forager agents (i.e., in `foragers-own [ ]`).

A few key primitives have short forms, e.g., `ca` is short for `clear-all`.

We want our forager agents to be able to gather resources from their land, and since these are semi-sedentary foragers, they will store a little

bit of their resources. With each time step, `ask foragers` to run a new `gather` procedure, and write out that procedure like this:

CODE BLOCK 3.1

```
to gather
  set storage storage + [resources] of patch-here
end
```

Complex models require a lot of testing. A method for this type of issue is to check whether the total amount of resources in the world remains the same.

While this is a good start, we have already made an error in our accounting. While the forager now has resources, we did not remove them from the environment. Think of this like an exchange from the patch to the forager's storage. We could just set the patch to zero, but then we might lose track of how many resources were gathered. Instead, we will make sure to account for the correct resource amount by using a local variable:

CODE BLOCK 3.2

```
to gather
  let current-gather [resources] of patch-here
  ask patch-here [ set resources resources -
  current-gather ]
  set storage storage + current-gather
end
```

The foragers do well in month one, but since they gather *all* the resources, there is nothing left for month two. In real life, however, there is a limit to how much a single person can deplete a patch of land. Foragers also need to eat, so after gathering they should consume some resources. Finally, with all of these dynamic changes it would be good to visualize the impact on the landscape, so we will develop a procedure to update our map's color scheme.

Add two sliders to the INTERFACE tab: one for `gather-rate` (values 0 to 10) and one for `consumption-rate` (values 0 to 10). These are stand-in values for our sliders that enable us to build a working simulation. Sometimes we have empirical values—for example, we know how much people consumed and gathered (i.e., we can parameterize the model). For now, though, we will use these stand-in values to enable swift code building and to verify that the model is functioning correctly. Move to the CODE tab, and add the code block below that makes the foragers `eat` a particular amount from their stores each month.

DON'T FORGET
Make sure to add the `eat` procedure to the `go`.

To keep our accounting in order, we also need to make sure agents do not eat more than they have in storage nor do they gather more than there is in the patch.

CODE BLOCK 3.3

```
to eat
  ifelse storage >= consumption-rate [
    set storage storage - consumption-rate
  ][
    set storage 0
  ]
end
```

Remember that the `ifelse` loop has a structure of:

PSEUDOCODE BLOCK 3.4

```
if condition is true [ do something ]
else [ do something else ]
```

In our case, if the `consumption-rate` (number of resource units consumed by the household per time step) is equal to or larger than the available resources `storage`, then that amount is consumed (i.e., it disappears from the storage). Otherwise the `storage` is set to zero because all that is left has been eaten. We will need to adjust the `gather` procedure in a similar fashion so that the foragers do not gather more resources than are available on the patch.

CODE BLOCK 3.5

```
to gather
  let current-gather 0
  ifelse [resources] of patch-here < gather-rate [
    set current-gather [resources] of patch-here
  ][
    set current-gather gather-rate
  ]
  ask patch-here [ set resources resources
    - current-gather ]
  set storage storage + current-gather
end
```

Here, we have defined a local variable `current-gather` and temporarily set its value to 0. If there are not enough resources, then agents can only get what is available (`[resources] of patch-here`); otherwise the forager can gather the maximum amount possible (i.e., the `gather-rate`).

Finally, let's ensure that we can observe the dynamics of the simulation: how foragers gather and consume resources. Before reading the code below, go to the NetLogo Dictionary and look at the syntax for the `scale-color` primitive. You can use it to color the patches depending on their resource level. Write a procedure, `to update-display` , in which patches set their color depending on their level of resources and foragers change color if their storage is empty. Your code should resemble this code block:

TIP
Get to the Dictionary quickly by placing your cursor on a primitive and pressing F1.

CODE BLOCK 3.6

```
to update-display
  let max-color max [resources] of patches * 2
  ask patches [ set pcolor scale-color green resources
  0 max-color ]
  ask foragers [
    ifelse storage > 0
    [
      set color blue
    ][
      set color red
    ]
  ]
end
```

Note that we adjusted the `max-color` (`* 2`) just for aesthetics so that full patches would be green instead of white. Look at the Colors section of the NetLogo Programming Guide.[2]

You can also find the color swatches in the TOOLS menu.

Add `update-display` to the end of `setup` right before `reset-ticks` , then add all the procedures we have just written inside `go` (including `update-display`). By now it should look like this:

[2] https://ccl.northwestern.edu/netlogo/docs/programming.html

CODE BLOCK 3.7

```
to go
  ask foragers [
    gather
    eat
  ]

  update-display
  tick
end
```

Now you can run your model with the sliders set to different amounts, and everything should work. If your `gather-rate` is equal to or lower than your `consumption-rate`, then your foragers will be in the red. Use the STEP button we created earlier to tick through a few time steps of the model and inspect one of the agents. After a few ticks, the foragers go red when their patch runs out of resources (fig. 3.0). At that point, the foragers should have a `move` procedure to transport them to a fresh patch. Also, patches should regrow their resources at a steady rate; otherwise the whole world will soon be depleted, and all the foragers will go red.

Let's begin with the `move` procedure. First, we want foragers to check whether their patch has enough resources to feed them. If not, they need to choose another patch that has no foragers on it and at least enough resources to fulfill the forager's consumption rate. Try writing the code yourself and then check with the following:

CODE BLOCK 3.8

```
to move
  if [resources] of patch-here < consumption-rate [
    let p one-of patches with [not any? foragers-here
    and resources >= consumption-rate]
    if p != nobody [move-to p]
  ]
end
```

Be careful with `if p != nobody` as it will not catch empty patchsets. NetLogo understands empty patchsets as `(agentset, 0 patches)` instead of `nobody`.

Notice the `if p != nobody`. The local variable `p` is a randomly chosen patch (as we used the primitive `one-of`) from among those that are unoccupied and have enough resources. The `if p != nobody` ensures that

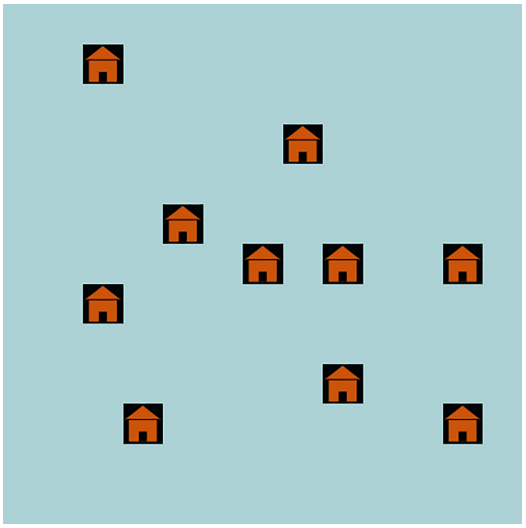


Figure 3.0. Forager agents have depleted their patch (shown here as black cells) compared to the surrounding unexploited patches. Using "update-display" will show the foragers that have depleted their stores in a darker shade.

such a patch exists (is not equal `!=` to `nobody`). Recall that in the previous chapter we used `if any? patches with` [...][do ...]. These two methods are nearly equivalent, but note that you need the `one-of` to make sure `p` is a single patch. Finally, write a simple procedure in which patches regrow their resources and add it to the `go` code.

You might start with something that looks like this:

CODE BLOCK 3.9

```
to regrow-patches
  set resources 10
end
```

Here, the patches undergo instant regrowth, meaning there is no incentive for the foragers to move because food is always available. Even though they consume resources, the patches replenish each month, so foragers stay where they are. This is not very realistic for most resources, so we will change the plant regrowth to be more gradual. Add a `growth-rate` `rate` slider (values 0 to 10) and change `regrow-patches` to this:

CODE BLOCK 3.10

```
to regrow-patches
  set resources resources + growth-rate
end
```

DON'T FORGET
Add `move` to the `ask turtles` and `regrow-patches` to `ask patches` in the `go` procedure.

Running this model, initially the foragers quickly exhaust the resources (displayed as darker shades of green) of their patches and move to another patch (use the STEP button, or set the model speed much slower than normal to see). However, over time the foragers will move less and less frequently. Why is that? Inspect one of the green patches by right-clicking on it. The value of the resources variable is much higher than expected, isn't it? In our `setup`, we gave each patch a resource starting value of 10, but nothing in the model is limiting how many resources there can be on a single patch. Again, we have forgotten to control our accounting. Create a `max-plants` slider (values 1–100) in the INTERFACE tab and change `regrow-patches` so that it only regrows the resources if they are below the maximum value, that is, the `max-plants`:

This is the normal iterative process of model building: add a component, check it, find bug, fix bug, move on.

CODE BLOCK 3.11

```
to regrow-patches
  if resources < max-plants [
    set resources resources + growth-rate
  ]
end
```

The `go` procedure should now look like:

CODE BLOCK 3.12

```
to go
  ask foragers [
    gather
    eat
    move
  ]

  ask patches [
    regrow-patches
  ]
```

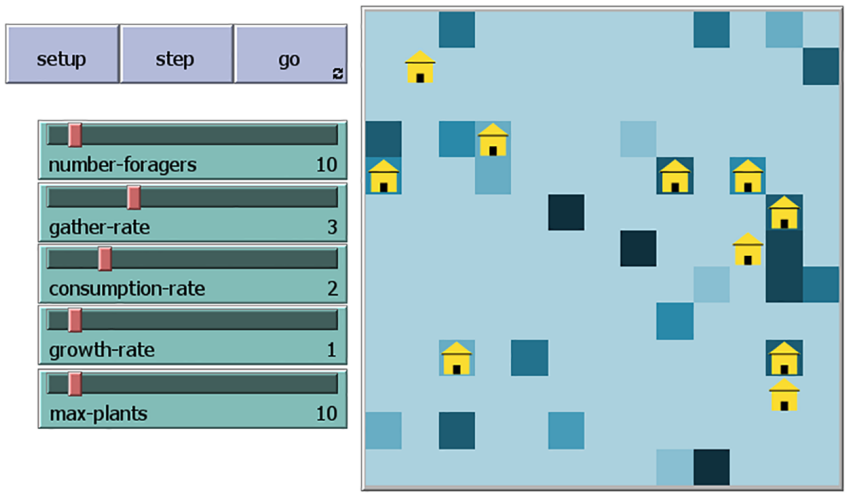


Figure 3.1. After a few steps, the agents have moved on to new patches several times, leaving the previously occupied patches at various stages of regrowth (darker patches). The sliders are set to sustainable levels and the foragers remain fed, indicated by houses in a lighter shade.

CODE BLOCK 3.12 (cont.)

```
  update-display
  tick
end
```

You may also adjust the color scheme by changing the first line in `update-display` to `let max-color max-plants * 2`.

Adjusting these parameter values and running the model repeatedly is a good way to verify that the model is logically sound and that you understand its dynamics.

At this point you can let the model run while manipulating the values in the sliders: `growth-rate`, `consumption-rate`, `gather-rate`, `max-plants`, and `number-foragers` (fig. 3.1). Intuitively, `consumption-rate` must be equal to or less than `gather-rate` for the foragers to not go hungry (red). It is perhaps less easy to predict the `growth-rate` value, relative to the other sliders, that will keep the landscape healthy or how many foragers are sustainable for a given value. You may also add a line of code so that if their storage value stays at 0 for a prescribed time, such as three or four months, the agent has to `die`.

Toy landscapes are simple landscapes with carefully controlled characteristics designed to help us understand a model's functioning (see ch. 7).

A TOY LANDSCAPE

For the last section of code we will redesign the resource landscape. Rather than import a map like in chapter 1, we are going to make a **toy landscape** with two resource "hills" using this code block:

CODE BLOCK 3.13

```
to make-hills
  let hills (patch-set patch (max-pxcor * 0.33)
  (max-pycor * 0.33) patch (max-pxcor * 0.67)
  (max-pycor * 0.67))
  ask patches [
    let dist distance min-one-of hills [distance
    myself]
    set resources max-plants - (dist / (max-pxcor *
    0.75) * max-plants)
    set max-resources resources
  ]
end
```

The first line defines the centers of two high resource hills. The second part asks patches to first measure their distance from the nearest peak and then to convert that distance measurement into a resource value (similar to ch. 2 when we set the `distance-level`). The last part, `set max-resources resources`, sets the regrowth maximum for each patch so that it cannot grow beyond that patch's capacity. Think of this as the maximum level of resources, similar to `max-plants`, but specific to each particular patch.

TIP
If you get an error, check your parentheses and square brackets. They are necessary for NetLogo to match input values with each primitive.

Now we have to adjust the rest of the code to this new function. First, add a switch to the INTERFACE called `hills?`. INTERFACE switches create global variables with only two possible values: `true` and `false`. Next, take the old patches `setup` procedure (`ask patches...`) from the `setup` and move it to a new procedure `to make-plain` and modify it to reflect our new `patches-own` variable, so that it looks like this:

DON'T FORGET
Got the "Nothing named . . ." error? Did you add `max-resources` to `patches-own`?

CODE BLOCK 3.14

```
to make-plain
  ask patches [
    set resources max-plants
    set max-resources max-plants
  ]
end
```

Change the location of the origin of the world to the bottom left corner in SETTINGS.

Then add the following code to the `setup`:

CODE BLOCK 3.15

```
ifelse hills? [ make-hills ][ make-plain ]
```

TIP
Naming switch variables with a question mark is common NetLogo practice to keep track of the logic.

You could keep the `make-plain` code in the `setup`, but by writing it as a separate procedure you make the code more readable, facilitate debugging, and can efficiently profile it later (i.e., check how fast it runs; we will work on this in ch. 7). Finally, change `regrow-patches` such that resources grow not to the global variable `max-plants` but to the patch variable `max-resources`. This ensures that the landscape is maintained throughout the run, and the `regrow-patches` procedure will not turn the hills back into a homogeneous plain.

Move the slider to see how different resource values change the landscape of resources in the model. If the `hills?` switch is off, the model should perform the same as before, but what's different about the agent behavior if `hills?` is on? How do the foragers respond depending on their `consumption-rate` and `gather-rate`?

3.3 Validation: Is my Model "Right"?

You'll find several SugarScape extensions in the NetLogo Models Library.

The model you just wrote constitutes the basic version of the *SugarScape* model (Epstein and Axtell 1996). This model is very simple, yet it was used to explore complex aspects of social evolution, such as the formation of wealth inequality in societies. Its ability to explain complex phenomena with a simple set of rules is shared by a number of classic agent-based models: the aforementioned *Boids* (Reynolds 1987) and the *Segregation* model by Schelling (1969) and Sakoda (1971) that demonstrated how moderate prejudice leads to strongly segregated neighborhoods. So how do simple models like these allow us to study highly complex social phenomena? And how much can we trust them? In the end, foragers don't live on "hills" of resources, and they engage in many activities other than foraging.

Notice that while designing and coding the simulation we made a number of decisions: we set the scale of the simulation, decided to use agents to represent forager groups, gave those groups the ability to gather resources, assumed no group would trade with any other, etc. Now, how can we assess whether these decisions are correct? Can we prove that our model represents the reality of a past society and be used to study its dynamics? As we

mentioned briefly in chapter 1, the simple answer is "No." It is not possible to formally prove that a given model is the best possible representation of the system—this is known as the **equifinality principle** (Premo 2010). In simple terms, it is never possible to assure beyond any doubt that another model would not be a better representation of the reality than the current one. This is not a limitation particular to agent-based modeling, or even simulation in general, but a result of the fundamental characteristics of inductive reasoning, also known as the **underdetermination of scientific theory** (Stanford 2017). Indeed, most archaeologists should be familiar with this principle from trying to interpret archaeological data.

equifinality: multiple past processes could equally have led to the same final archaeological pattern; note, though, that not all are equally likely. This also explains why "correlation does not imply causation."

Underdetermination of scientific theory is a complex idea, but the *Stanford Encyclopedia of Philosophy* explains well its many facets.

However, it is definitively possible to say that a given model is *not* correct (i.e., to reject a hypothesis), that it is possible (i.e., confirm its plausibility), or that it matches data better than another one (i.e., ranking hypotheses). To achieve that, we use a process known as **validation**. It is a fundamental element of almost all simulations, so we will spend some time explaining different validation techniques in chapter 4, as well as an arsenal of closely related methods such as calibration, sensitivity analysis, and model selection in chapter 9. For now it will suffice to define validation and work through an example of how it can be applied. Validation refers to the phase of the modeling process when the output of a model (so-called artificial data) is compared to real-world data. In our case, this usually refers to comparing the model's prediction to the archaeological record. If the two match, we can state that the model is a plausible hypothesis regarding the system we study (e.g., out-of-Africa dispersal, Roman trade, or foraging societies). If they do not match, however, we can reject the model as an impossible or unlikely scenario of how things were in the past. As you can imagine, this is usually not as straightforward as it seems, and decisions need to be made regarding such issues as: What do we compare to what? How similar do they need to be to one another to be deemed "well matching?" And does the model match the data within a realistic range of parameter values?

"Well matching" may be quite abstract while still being useful. In programming this chapter's model, we have been using **arbitrary values** like 10 or 100 for our parameters, rather than carefully selected values derived from, for example, present-day foraging societies or historical sources. This is like excavating by arbitrary 10 cm levels rather than by stratigraphic levels. Both allow you to move forward in a controlled fashion but must be

arbitrary values: setting a model's variables not to real-world values but to numbers that reflect ratios between the variables: e.g., X is double Y, $X = 10$ if $Y = 100$.

analyzed slightly differently. In agent-based modeling, choosing arbitrary values allows us to verify that the model design functions as intended without getting stuck validating every parameter value. Parameterization is usually a much later stage of the research project (Romanowska 2015b), and we will talk about it in detail in chapter 6. Using arbitrary values lets you experiment with how the relative values (i.e., ratios) between **parameters** affect your model's dynamics. How does our foraging model change if the resources' regrowth rate is higher or lower than the consumption rate? What if we run simulations with our population size set to 5%, 25%, or 50% of the patch count, thereby increasing our population density? Understanding the interactions and causality chains between model's entities is critical if you do not want to produce a "black box."

parameter: a constant used to describe the modeled world and a particular scenario, such as initial number of agents, population growth rate, or carrying capacity of a patch.

We will explore the trade-off between realism, precision, generalism, and tractability, i.e., the ability to describe the causality in the model, in ch. 8.

SugarScape was entirely built using arbitrary values, such as its 50×50 grid of cells with 400 agents, and a per-time-step metabolism of 1 to 4. However, by looking at the ratios, the modelers were able to demonstrate that the uneven wealth distribution of their agents matched well-known economics principles like the Lorenz curve and Gini ratio (Epstein and Axtell 1996).

In the next section, we will create an output from our model so that we can validate it against the archaeological record; this is where we build upon the simple *SugarScape* model to recreate some key parts of *Artificial Anasazi*. In the process we will make the model more complex and experiment with modeling concepts such as lists and reporters.

3.4 Simulating the Creation of an Archaeological Record

ABMA Code Repo: *ch2_sugar_archrec*

As we discussed above, to validate the model against the archaeological record, we need to create some model outputs that can be compared to it. In chapter 1, we compared the order of arrival of the wave of dispersal in different continents against the dates of archaeological sites, while in chapter 2 we looked at the change in the frequency distribution of pottery at different archaeological sites. In this chapter, we focus on the interaction between humans and their landscape, making the spatial distributions the most appropriate dimension to compare to a real archaeological record. We will record the occupation of a patch as the number of time steps a forager stays on it; one time step is one unit. What that unit represents does not matter right now, since we are interested in a relative frequency of occupa-

tion of all sites over the duration of the simulation. Although we are only counting the foraging group's presence or absence, we are assuming that the occupation is correlated with duration and therefore the density of objects left behind (mimicking the creation of the archaeological record; see Varien and Mills 1997).

Create a new `record-occupation` procedure to update a new variable `occupation-frequency`, which will act as a rolling counter of where agents have spent their time. Your code should look like this:

Should `occupation-frequency` be added to (`globals`), (`patches-own`), or (`foragers-own`)? Not sure? Try each and see what happens.

CODE BLOCK 3.16

```
to record-occupation
  set occupation-frequency occupation-frequency + 1
end
```

You may be surprised that it is the foragers that set `occupation-frequency` even though it is a patch variable. This works because NetLogo understands that agents are only talking to `patch-here`, that is, the patch they are currently standing on. Likewise, in the code sections above where we wrote `[resources] of patch-here`, we could have just written `resources`. This would be cleaner code, but we wrote it in longer form to make sure the accounting was easy to follow.

Call `record-occupation` from the `ask foragers` part of `go`, after the `move` procedure.

We will also duplicate and then modify the `update-display` code to visualize our results.

CODE BLOCK 3.17

```
to update-display-occupation
  let max-color max [occupation-frequency] of patches
  ask patches [set pcolor scale-color red
  occupation-frequency 0 max-color]
  ask foragers [set hidden? true]
end
```

TIP

It's common to duplicate and rename code to use it for a slightly different task. What would happen if you didn't change the name of the duplicate procedure?

Since we are interested in the long-term results of the simulation (its archaeological record) and not individual foragers, we told the foragers to "disappear" from the view using the built-in turtle primitive `hidden?` (fig. 3.2). We can call this procedure at the end of the run or create a button in the INTERFACE. How does the occupation map compare to the two different resource landscapes on the short term versus after a long run of the simulation? Would it be different if instead the occupation was record-

Figure 3.2. Simulated archaeological record on the hills landscape with more occupation as darker shades close to the peaks of the hills. Note that forager agents are set to "hidden? true."

ed only during certain activities? There are many GIS techniques we could use to compare this occupation frequency map to the spatial distribution of archaeological material in our study area, some of which we will describe in chapter 7.

DYNAMIC LISTS & THEIR MANIPULATION

TIP
Lists are computationally "cheap," making your simulation faster than if you used a lot of variables.

Thus far, we have avoided using lists in this simulation, though recall we introduced lists in the previous chapter. One way of avoiding lists is to make more variables. For example, if our foragers wanted to gather three edible species instead of just one, we could have resourceA, resourceB, and resourceC as patch variables. But what if we are not sure how many resources we need, or if the number of resources will vary during the simulation? It would not scale either if there were 1,000 resource types. When we need to hold only one piece of information, simple variables are the best option. But if we need to keep track of a lot of information, especially if of unknown quantity, lists are a more flexible approach.

data structure: the format and organization of variables and parameters. Common data structures in NetLogo are integers, floats (decimal numbers), strings, Booleans (true/false), lists, agents, and agentsets.

Let's give each forager the option of gathering multiple resources. What we are doing here is switching **data structures** from a single numerical value to a list, and that tends to cause a cascade of necessary changes to our code. It is common to change the underlying architecture of the model

as you develop the code, but it is also a major source of code errors, so it is worth paying attention.

We can actually reuse the variable names `patches-own [resources max-resources]` of the existing model because in the declaration of a new variable (e.g., in the `turtles-own`), NetLogo does not care what data structure it will end up being (e.g., integer, list, etc.). NetLogo will care that the procedures that act upon the variable receive the correct data type. Since NetLogo has a good syntax-checking debugger, we can run the model and let the debugger find all the errors that we introduced. Find the first place we assign a value to that variable, change that, and then try to run the model again. Repeat until all errors have been found. The declaration of the `resources` and `max-resources` occurs first in `setup`'s `make-plain` procedure:

Admittedly, this is a bit of a hacky way of developing our code, and ideally all this would have been thought out in advance. But you should not be afraid of the yellow error bar, so hack away and fix bugs as you go.

CODE BLOCK 3.18

```
set resources max-plants
set max-resources resources
```

We will replace `max-plants` with code initiating a list. In chapter 2, we did this once by just writing out `set mean-goods [ 0 0 0 0 0 0 ]` to make a six-item list, and then again using `replace-item` to dynamically replace items as needed. Here we will first make a `num-resources` slider (values 1–10) in the INTERFACE tab, and then use the NetLogo primitive `n-values`:

CODE BLOCK 3.19

```
set resources n-values num-resources [ max-plants ]
set max-resources max resources
```

`n-values` creates a list containing the value of `[max-plants]` repeated the number of times given by `num-resources`. Notice also that the second line has `max` added, since now `resources` is a list and we still want a single number to be attached to `max-resources` (i.e., the maximum value in that list). In the INTERFACE, try to set up and run your model. Creating the agents goes fine, but `update-display` gets an error because it is trying to `scale-color` against a list instead of the expected single number.

TIP
NetLogo Dictionary specifies which data structure the primitive accepts, e.g., `hatch <number>`.

We will shift our display to represent the total among all resource types by using `sum resources` to add the values of the list together. We will also change `max-color` to scale properly with multiple resources:

CODE BLOCK 3.20

```
let max-color max-plants * 2 * num-resources
ask patches [ set pcolor scale-color green sum
resources 0 max-color ]
```

If you try to pass the wrong data structure to a primitive, you'll get the error "primitive x expected input to be . . ."

`setup` should work without error now, assuming `hills?` is false. However, when we run the model we quickly find that `gather`, `eat`, and `move` all need to be updated. For `gather`, we need to think about the logic of having multiple resources available to a forager. Can you gather them all at once? Is there a cost to switching? Do we choose some to gather before others? To keep it simple, we'll assume that our list is in order of preference, and you can gather one resource type per time step. If there is not enough of the first type, the forager will gather the second, and so on. Update the `gather` procedure so that it looks like this:

CODE BLOCK 3.21

```
to gather
  let current-gather 0
  ifelse max [ resources ] of patch-here < gather-rate
  [
    set current-gather max [resources] of patch-here
    let i position current-gather [resources] of
    patch-here
    ask patch-here [ set resources replace-item i
    resources 0 ]
  ][
    let resources-available map [ x -> x >=
    gather-rate ] resources
    let i position true resources-available
    set current-gather gather-rate
    ask patch-here [ set resources replace-item i
    resources (item i resources - current-gather) ]
  ]
  set storage storage + current-gather
end
```

Recall that the NetLogo list index starts from 0.

Most of this code should be understandable. For example, `max` in front of `[resources]` returns a single number, i.e., the highest value in

the list, which makes the comparison to `gather-rate` possible. In the `ifelse` loop, the "if true" part uses the primitive `position`, which searches for a value, in this case `current-gather`, and returns its position in the list (the index). We then ask the forager's patch to update its resources list at that position, to show it has been gathered, using `replace-item`. We are assuming an agent can only gather one resource type at a time. The below shows the steps of an agent gathering from a hypothetical patch with five resource types when `gather-rate` is set to 7:

PSEUDOCODE BLOCK 3.22

```
;initial state of a partially consumed patch with 5
resource types
[resources] of patch-here: [1 6 6 6 6]
;set current-gather to most that can be gathered here
current-gather: 6
;position of first current-gather value in list
i: 1
;final state after gathering
[resources] of patch-here: [1 0 6 6 6]
```

The "else" part of the `ifelse` statement is a bit more complex. For this, we already know that one of the resources in our list is enough to satisfy our `gather-rate`, using `max`, but we are not sure in which position in the list that resource is, so we are going to check each position on the list using the `map` primitive, where true means "yes, there is enough" and false means "no, there is not enough." The command `map` includes what NetLogo calls the **anonymous procedure** `->`. Anonymous procedures apply a function to each item in the list one by one while collecting the results into a new list. Here, we are going through the list `resources` and asking for each item `x`, "Is x greater than or equal to `gather-rate`?" The answer is either "true" or "false" for each item. The resulting list `resources-available` is a sequence of true and false values, for example, `[true true false...]`. In short, you can read this line as "create a list called resources-available that has true for each item x from a resources list greater than gather-rate or false otherwise."

For Python coders: This is the same as a lambda function.

TIP
If you feel lost, print out the values at each step, as if you were debugging (see ch. 2).

Next, we record in a temporary variable `i` the `position` along the `resources-available` list where the first `true` appears. We then ask `patch-here` to subtract that amount of gather from the item of the list with index `i`.

Remember that lists are immutable, so we have to set a new list made up of the old one after we adjust it using `replace-item` like in the lists in chapter 2. Again, we show the output of each step for an agent gathering a hypothetical patch, though this time there must be enough resources to satisfy the `gather-rate` of 7:

PSEUDOCODE BLOCK 3.23

```
;initial state of a partially consumed patch with 5
resource types
[resources] of patch-here: [5 4 4 10 10]
;after map compares to gather-rate of 7
resources-available: [false false false true true]
;position of first true in list
i: 3
;after taking gather-rate from position i
[resources] of patch-here: [5 4 4 3 10]
```

The next changes will be much easier. Run the model again so that the debugger shows you where the next problem is. You can use the `max` primitive to adjust `move`. Simply write `max` before each instance of `resources`. To update `regrow-patches` we can use `map` again, though we will also put the `ifelse` into the function part of `map`:

CODE BLOCK 3.24

```
to regrow-patches
  set resources map [ x -> ifelse-value (x <
  max-resources) [x + growth-rate] [x] ] resources
end
```

Notice that `ifelse-value` doesn't require you to add `set` as you would in a regular `ifelse` command block.

Here we update the resource list by mapping a function to each value. But contrary to the previous code block, we use the `ifelse-value` primitive. This works just like the regular `ifelse` except that the two parameters after the conditional statement are reporters and not commands like with a traditional `ifelse`. In short, you change the `x` value to `x + growth-rate` if the condition `x < max-resources` is true

and `x` if it is not. In our case, *each* item in the list has to be checked one by one to see whether it needs updating and, if so, to apply the `growth-rate` to just that item.

The last thing we need to change is a little update to `make-hills`:

CODE BLOCK 3.25

```
to make-hills
  let hills (patch-set patch (max-pxcor * .33)
  (max-pycor * .33) patch (max-pxcor * .67) (max-pycor
  * .67))

  ask patches [
    let dist distance min-one-of hills [distance
    myself]
    set resources n-values num-resources [round
    (max-plants - (dist / (max-pxcor * .75) *
    max-plants))]
    set max-resources max resources
  ]
end
```

In this update, we have left the two resource landscapes the same shape for all of the resources. If needed, we could have separate `growth-rates`, `max-plants`, and spatial distributions for each resource. We will learn more about how to work with spatially heterogeneous landscapes in chapter 7.

As we have just seen, `map` is useful when applying a function across a list, particularly when you want the outcome to also be a list, because the function you give as a parameter is structured as a **reporter**. A similar primitive, `foreach`, also uses anonymous procedures (i.e., the `->`) to work through a list but does not produce a list as a result. Instead, `foreach` can be used to run a code block of commands repeatedly, where an argument for that code is each one of the list items in turn. NetLogo's Dictionary clarifies the difference in both the syntax and the parameters needed:

reporter: a type of procedure that calculates a value and *reports* it back.

CODE BLOCK 3.26

```
map reporter list
foreach list command
```

For example, in the `regrow-patches` code block above, `map` needed the reporter `x + growth-rate`, whereas if we programmed something similar with `foreach`, we would need to write it as `set x x + growth-rate` because it takes a **command** as the parameter. With `foreach`, you would also need to write extra code to collect the result of those commands into a list, for example, using `lput`. We will see a few examples of that in the subsequent chapters (e.g., ch. 5).

3.5 Building Artificial Anasazi

Complexity science excels in simple, abstract models, which can be used across very different systems.

In this chapter we built a simplified model of foragers gathering resources from a landscape and moving about to find more plentiful resources. This is a simple model, but it formed the backbone to the *Artificial Anasazi* model, in which patches continuously change their productivity in a simulation of environmental variability. This required only a slight modification of *SugarScape*, yet the authors were able to examine an archaeological hypothesis. Although *Artificial Anasazi* models a semi-sedentary farming society rather than hunter–gatherer foraging, you can see this does not really affect the model's ontology; the farmer agents "forage" for arable land to plant. There is a certain universality in simple models. Often they can represent very different real-world systems (Page 2018, ch. 3).

Comparing the model output against multiple independent data sources is the ideal way to do validation.

Although it is an excellent model in many respects, it is worth focusing here on the authors' validation strategy. Axtell and colleagues used two independent sources of archaeological data to compare to their model results: first, they had the spatial-temporal distribution of Ancestral Puebloan settlements, and second, they counted the number of households over time as a demographic proxy (fig. 3.3). Thus, the model results were compared to two independent data patterns, giving them confidence that this correspondence did not happen by chance.

The *Artificial Anasazi* model was replicated by Janssen (2009) in NetLogo and is available on CoMSES. See Further Reading at the end of this chapter.

In subsequent chapters, we will build up to some other aspects of *Artificial Anasazi*, like its targeted mobility (ch. 4) and population dynamics (ch. 6). You will also see many more of these "baseline" models that we use to approach our particular case studies.

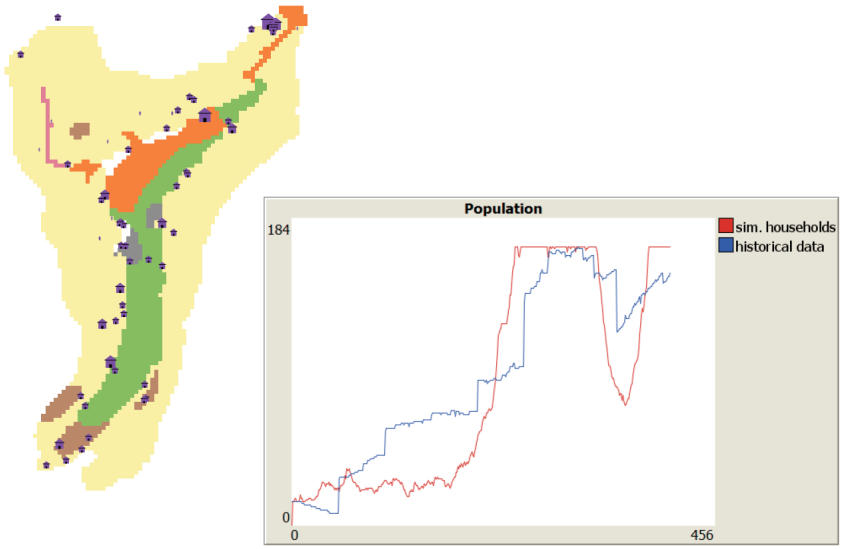


Figure 3.3. Map and population plot produced from a single run of Janssen's NetLogo replication of *Artificial Anasazi*. One can compare the evolution of the artificial population with historical records, but also the spatial distribution of simulated settlements, to the one recorded archaeologically.

3.6 Summary

Modeling subsistence is all about tracking the flow of resources, energy, and population. Manipulating lists is a difficult task for both seasoned practitioners and new modelers, but they serve this accounting task well. We intentionally rounded out the last coding exercises of part I, **Learning to Walk**, with genuinely challenging tasks to push you to think about how these simple NetLogo primitives can be combined in more complex ways. Combining and expanding on the techniques you have learned so far will make some truly flexible code that does exactly what you want it to do. We do not expect that you will grasp every aspect on your first read-through, but hopefully through trying it out on your own, and adding in `print`, `type`, and `show` lines to examine the state of the model and its variables at each step, you will be able to figure it out. Don't forget that whenever you get stuck, you can simply search for answers on the internet. Usually this will lead you to Stack Overflow,[3] which is where coders of all programming

TIP
You can apply these strategies whenever you need to dive into the code to understand what it is doing.

[3] https://stackoverflow.com

languages go to check whether someone else has already solved a similar issue.[4]

The first three chapters consisted of replications of published models. Replication is as critically important in agent-based modeling as it is in science in general. We always learn something new in the process of reproducing an existing model, sometimes just reinforcing that the previously published research was sound. Often, finding new ways to code and extend an old model leads to valuable and novel research directions (Marwick 2017). Replication is easier when the model's code is accessible, and agent-based modelers are usually very good about sharing their code.

See chapter 9 for more details on how, where, and why you should make your models available to others. Taking a highly abstract, generalized model and applying it to your specific case study is a common way to develop models.

SugarScape is a highly abstract model, while *Artificial Anasazi* is a highly empirical one. The key difference is in the conceptualization and parameterization of the model, that is, what you decide the unit being passed from patch to agent represents and what its characteristics are (value, quantity, abundance, species, etc.). By looking at these models in detail, one realizes that these building blocks—algorithms—are highly flexible and adaptable to many different research questions. Part II, **Learning to Run**, focuses on expanding your knowledge of these building blocks so that you will be able to start building your own models by combining and extending these common algorithms in novel ways.

End-of-Chapter Exercises

1. Add a line plot to the INTERFACE tab to track the average amount of resources in storage of the agents over time. Next, create a histogram to show the variance in storage amount (i.e., wealth inequality) over time. What can these models tell us about wealth equality/inequality?

2. Experiment with different population sizes. Does this have an effect on the "wealth" of agents? How?

3. Change the code to design several new landscapes with: (a) four hills, each with a different amount of sugar; (b) sugar distributed randomly in patches throughout the landscape; and (c) one or more barriers

[4] It is so prevalent that a satire news site circulated an article titled "Computer Programming to Be Officially Renamed 'Googling Stack Overflow.'"

that partially restrict movement between the sugar hills. How does this lead to areas of wealth and areas of poverty in your model?

4. Add in a patch `pollution` variable. Have agents leave pollution when they eat sugar and restrict agents from entering cells with a pollution amount that is too high (use a slider to play with what is "high" here). Pollution should dissipate over time as well, so write a countdown procedure for the patches' pollution. This is one of the original *SugarScape* experiments.

5. Download Janssen's replication of *Artificial Anasazi* from CoMSES. Find the `harvestconsumption` procedure and work through his list code to figure out how it works. Add `print` statements to print out the list contents at different points within the procedure. A STEP button might also be helpful so you can go slow enough.

Further Reading

- A. Angourakis et al. 2017. "The Nice Musical Chairs Model: Exploring the Role of Competition and Cooperation Between Farming and Herding in the Formation of Land Use Patterns in Arid Afro-Eurasia." *Journal of Archaeological Method and Theory* 24, no. 4 (December): 1177–1202. doi:10.1007/s10816-016-9309-8

- T. Baum et al. 2020. "How Many, How Far? Quantitative Models of Neolithic Land Use for Six Wetland Sites on the Northern Alpine Forelands between 4300 and 3700 BC." *Vegetation History and Archaeobotany* 29, no. 6 (November): 621–639. doi:10.1007/s00334-019-00768-9

- S. A. Crabtree et al. 2017. "How to Make a Polity (in the Central Mesa Verde Region)." *American Antiquity* 82, no. 1 (January): 71–95. doi:10.1017/aaq.2016.18

- J. S. Dean et al. 2000. "Understanding Anasazi Culture Change through Agent-Based Modeling." In *Dynamics in Human and Primate Societies: Agent-Based Modeling of Social and Spatial Processes,*

edited by T. A. Kohler and G. J. Gumerman, 179–205. Oxford, UK: Oxford University Press, July.

▷ J. M. Epstein and R. Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* Complex Adaptive Systems. Washington, DC: Brookings Institution Press.

See NetLogo Model Library for SugarScape models.

▷ M. A. Janssen. 2009. "Understanding Artificial Anasazi." *Journal of Artificial Societies and Social Simulation* 12 (4): 13. https://jasss.soc.surrey.ac.uk/12/4/13.html, NetLogo Code: https://doi.org/10.25937/krp4-g724

▷ T. A. Kohler, R. K. Bocinsky, et al. 2012. "Modelling Prehispanic Pueblo Societies in their Ecosystems." *Ecological Modelling* 241 (August): 30–41. doi:10.1016/j.ecolmodel.2012.01.002

NOTES