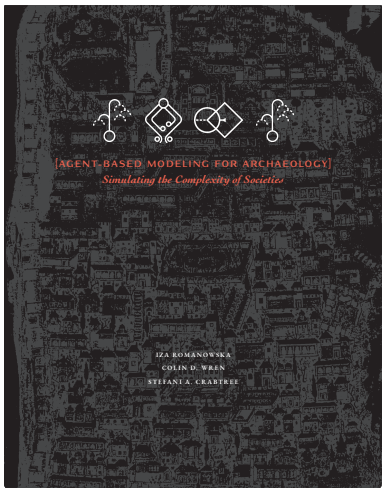
PLEASE NOTE:

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

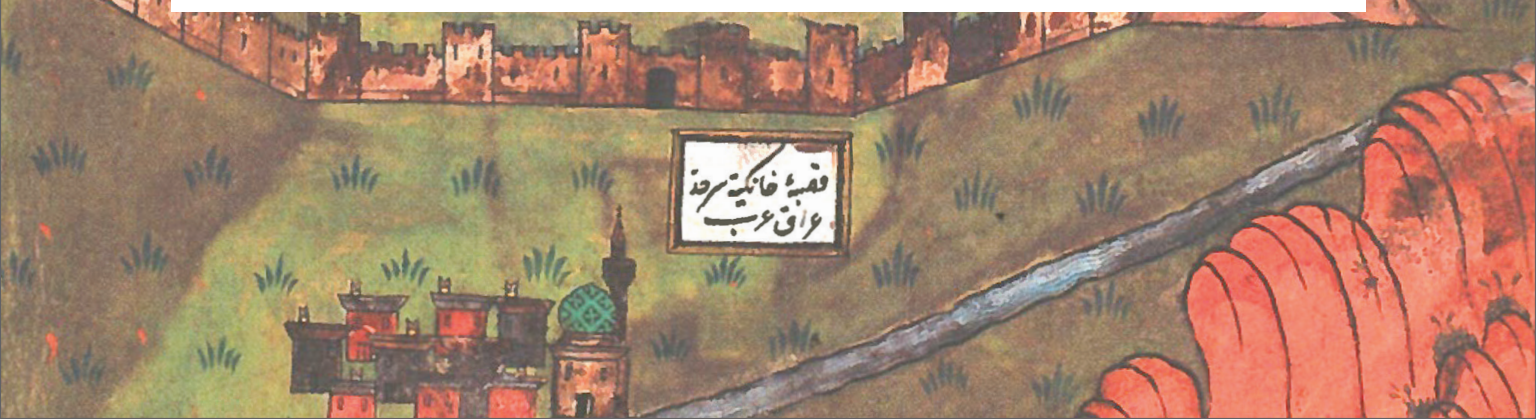


REGARDING COLOR:

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



جهان وصاحب قران زمانك درديدن سلطان آفتاب دمي اول تاجدار سپهسالار وامراء
بختيار لره عموماً خلعت فاخره يله رعايت وسايرذوي الاحترام آدملرينه التفات
سوار ويادده وبنده وازاده قولرينه خصوصاً انعام واحسان ايله مرحمت اولنوب انواع
بهجت حبور وغبطت وسرور ايله محل مذكوره ده درت كون اقامت اولنوب اندن صكره
ترتيب صاحب قراني وآيين جهان باني اوزره خداوندكار خضر بكري سمند سهند مانند
ورخش تيزگام همت بلند سوار اولوب دخي شقاره شقاره جلادت وخون آشام ضرغام
شجاعت اولان كوه شكوه كروه جيول لجمه ورايات ظفر پرتو جم بدن كلشن عراق
عجمه اولان سلطانيه جانبنه متوجه اولندي اولده اولسه ذكر اولان سلطانيه وارنجه
واقع اولان منازل بونلرده ذكر اولنور الواقع ١٧ ربيع الاول سنه

TRADING UP TO COMPLEX MODELS OF ECONOMIC INTERACTIONS

2.0 Introduction

Production, distribution, exchange, and consumption of goods are all topics of immense interest to those studying past societies. It is also a branch of scientific inquiry so vast it comprises hundreds of models, theoretical frameworks, and analytical methods. In this chapter we will show how to approach a modeling study departing from a data pattern and how to formulate research questions and hypotheses that we will then investigate using simulation. We will also consider the role of simple abstract models in giving us general intuition as to what dynamics might drive the system we study. Often we do not need to include all possible details into a model to gain a better insight into certain processes. This is something to keep in mind particularly when approaching economic models that are usually developed in data-rich environments, such as present-day economies—circumstances that we simply cannot mimic within the archaeological record.

Nevertheless, in this chapter we will take a more data-centered approach compared with chapter 1 and look at how simulation can help us to understand common types of data patterns—here, the frequency curves of different types of artifacts. In particular, we will investigate how the distance from a Roman production center, the trading capacity of intermediate markets, and the number of goods being produced influence the introduction time of a product and its uptake curve at different markets. These trend lines can be directly compared with the changes in the distribution of different types of archaeological material at archaeological sites. We will further tackle the topic of exchange (which includes, but is not limited to, trade) in chapter 5, and we will focus on how to formally represent relationships using network science techniques in chapter 8.

OVERVIEW

▷ Tutorial in intermediate NetLogo: variables, loops, breeds, reporters, lists, and plots
▷ Building up to complex models
▷ Definitions of scale, entity, stochasticity
▷ Principles of code development and debugging techniques

In the first chapter, we used a simple model of hominins spreading across the world to introduce the process of building a simulation and to begin a few basic coding tasks. In this chapter, we will make further progress on coding skills. Similar to learning a natural language, such as French or Hindi, it takes some time and effort to fully grasp the grammar and the vocabulary of a programming language—so be aware that you may feel lost at times. This is absolutely normal and disappears quickly as you gain confidence in coding and solving errors. Similarly, thinking through a problem by means of formal representation, such as a simulation, may take some time to get used to, especially if this is your first experience with modeling.

syntax (in computer programming): the set of rules, or *grammar*, governing the programming language.

This chapter will introduce most of the key components of NetLogo **syntax**, which constitute its core functionality:

- breeds and agents' variables, global and local variables;
- if, while, and for loops;
- lists; and
- common primitives, such as `of`, `with`, `any?`, `one-of`, and `myself`.

We will continue following in parallel two important lines of the development of your modeling capacity: we will work on coding fluency by introducing new structures and methods, but we will also work through the process of building a simulation to better understand what kind of decisions the modeler needs to make and how to best approach them.

2.1 The Model: Simple Roman Trade

The origin of many models lies in the quest to explain a data pattern revealed in the archaeological record.

When approaching a new modeling project, it is worth spending some time to evaluate what we know about the system, what kind of data can help us and what format it comes in, and what exactly we want to learn. In this chapter's case study we start with a simple observation that at most archaeological sites one can trace the frequency of different types of artifacts over time (think of the famous battleship-shaped seriation curves). Imagine you plot the number of different pottery sherd types from your site on a time axis—all of them will likely start with a low number and then increase. However, these temporal trends have spe-

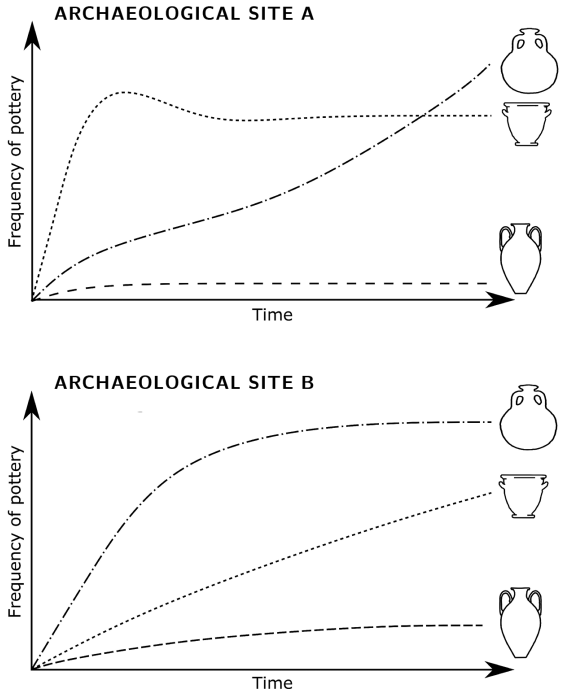


Figure 2.0. Hypothetical pottery frequency curves at two archaeological sites. This type of data pattern is common in the archaeological record.

cific shapes—some grow in their popularity more gradually, others see a meteoric rise with a very rapid uptake curve (fig. 2.0).

For example, you may observe that at your site certain types of Roman pottery (say, amphora type Dressel 20) arrive later than others (say, amphora type Pascual 1) but become ubiquitous quickly or, on the contrary, take years to get established. Why is that? Isn't it true that "a pot is a pot is a pot"? Perhaps some were better than others, or perhaps some were produced in higher quantities, or maybe some types were easier to transport. What processes cause differences among products has been a source of much debate and usually focuses on the price, accessibility, and functionality of goods, though can also amount to a preference for one type over another (Brughmans and Poblome 2016; Crabtree 2016). We cannot test all possibilities at once, so in this chapter we will focus on factors related to accessibility, while exploring the general dynamics

Each simulation scenario is like a single experiment; as in any experiment, you can only test one factor at a time.

of a simple trade model following the approach set out in Romanowska (2018).

The aim of this study was to establish a baseline of how the frequency curve of different Roman amphora types may appear under different circumstances: low production, limited flow of goods between settlements, or restricted trading capacity of intermediate markets. This abstract model could then be useful for comparing against real archaeological data; its generality is its strength in that it can be applied to any category of archaeological finds that were produced and traded. We will build a simplified model of a trading system in which goods move from a production center to different markets following simple exchange rules. We will then record the frequency of goods at each market over time, thus essentially creating an artificial archaeological record. Because the format of the simulated assemblages will be the same as frequency curves recorded at real archaeological sites, it will give us an idea as to what factors could have caused these patterns.

As we saw in chapter 1, it's common to begin a model from the simplest approximation by capturing the key dynamics of a system in the most general sense, even if this implies making some gross simplifications. It's worth revisiting the definition of model at the end of chapter 1: a *simplified* representation of a real system consisting of elements deemed relevant to the research questions. The first simplification we will make in the model is that we will not specify what is being traded. This could be foodstuffs (e.g., grain or livestock), craft products (e.g., pottery), or specialized goods (e.g., luxury items). It may come as a surprise that we would model each of these as if they were the same, since in reality they would display particularities in how they were handled by different commercial actors. However, on reflection one quickly notes that all products go through the same phases of their economic life: production, transport, distribution, consumption, and discard. By focusing on the general process rather than the specific details relevant to each product, we can create a model that is applicable to a much wider range of cases (we will discuss in detail the trade-offs it involves in ch. 8).

entity: any object in a model.

The second simplification we make is collapsing a market with all its vendors and consumers into one **entity**. If we think of the buzzing marketplaces across the world's small and big cities, this may seem unwise. Yet

considering the **scale** of the model, it may be appropriate to represent the entirety of commercial activity within one urban center as one entity. We can visualize this as a model in which towns and cities are traders, rather than particular individuals. When approaching a new model, it is important to identify the scale at which the system is represented and consistently apply that scale. This is particularly important in the first iteration of a model—one can always add more details and nuances later. Models are never a static last word but rather small steps forward in the incremental scientific process. In fact, if you want to explicitly model the system at multiple scales, a whole family of **multilevel models** exists to help model dynamic phenomena interacting across scales. (Hjorth et al. 2020).

scale: the size of spatial units and the duration of temporal units within the modeled system.

For **multilevel models**, check out the *LevelSpace* NetLogo extension.

2.2 Setting up the Trade Model

For our trade model[1] we will need to differentiate between producers, who create products, and vendors, who distribute them to different markets. Making specific groups of agents that behave differently from one another is fairly common. You could achieve this simply by giving them a variable (an attribute) and then calling all agents with a given value of that variable. For example (no need to type it in):

CODE BLOCK 2.0

```
ask turtles with [color = pink] [
    ; do something that only pink agents do
    ]
```

In this code snippet, we created an `agentset` on the fly. Here, turtles distinguished by the variable (`color = pink`) inside a reporter block can be asked to perform particular actions. You can use any value or a range of values, e.g., `turtles with [age > 30]` or `turtles with [energy <= 1]`. An important rule to remember is that whatever comes in the reporter block, that is, after `with`, needs to be enclosed in square brackets `[...]`.

agentset (NetLogo): an unordered set of agents of one type (e.g., turtles or patches).

However, if you know that an agentset will be consistently used throughout the entire simulation, you can predefine it at the start of your

[1] You can find all code written in this chapter in the ABMA Code Repo: https://github.com/SantaFeInstitute/ABMA/tree/master/ch2

Figure 2.1. NetLogo breeds. Although all entities are agents (turtles), teachers will have different variables (e.g., years of teaching experience) and ranges of values (e.g., age) than students, who are also more numerous.

breed (NetLogo): a predefined agentset. Can be used with several primitives instead of `-turtles`.

code. A predefined agentset is called a **breed**. Breeds have access to several useful primitives and are in general more flexible than agentsets defined on the fly. Since a breed is just a collection of agents, its size and characteristics may change over the course of a simulation run. Similarly, different breeds may have different sizes and different variables. You could, for example, create a simulation of a school with two breeds: teachers and students (fig. 2.1). There would naturally be fewer teachers than students, and they would have different variables (e.g., `science-grade, yrs-teaching-experience`) or ranges of values (e.g., `age`). Multiple breeds can be created in a model, and they provide a means of handling different groups of agents with ease throughout the simulation.

Model: Trade Distance

ABMA Code Repo: *ch2_trade_distance*

Defining and using breeds is exceedingly simple. Open a fresh NetLogo file. Give it a meaningful name and save it in a sensible location. Once this housekeeping is completed, head straight to the CODE tab. Here, we will create a population of producers and a population of vendors who will trade with one another. Type the following at the very beginning of the code window:

CODE BLOCK 2.1

```
breed [ producers producer ]
breed [ vendors vendor ]
vendors-own [ ]
producers-own [ ]
```

To create a breed you need to specify its plural and singular name (`producers producer`). You will use it the same way you use `turtles`: `ask turtles...` and `ask turtle 1...`. From now on if you `ask turtles` you will call all agents—producers *and* vendors—but if you `ask producers` only those turtles that are assigned producers will perform the action. Assigning variables to breeds works the same as with turtles or patches, using the keywords at the beginning of the script, `producers-own` and `vendors-own`. You can use many of the primitives with breeds in the same way as you would with turtles, for example, `create-turtles` or `turtles-here` can be replaced with `create-producers` and `producers-here`.

Each agentset (turtles, patches, breeds, etc.) can have a list of user-defined variables at the top of the code, for example, `turtles-own [ ]`.

Let's create a production center with vendors distributed around it. You learned how to write the `setup` procedure in the first chapter; now it's time to try it out for yourself. This one will be more complicated, so first we will write the most basic version of the code, and then gradually add complexity. Try to write the code yourself as you work through the steps below, then check your code against the answer.

DON'T FORGET
Make sure to do the NetLogo housekeeping with `clear-all` and `reset-ticks`.

1. First, create one turtle at the very center of the world. Set its shape to "house" and give it a color of your choice. Check whether the code works (you will need a SETUP button in the INTERFACE).
2. Now, change the `create-` primitive so that it specifically generates "producers" instead of generic turtles. Include a variable called `goods` in the `producers-own [ ]` list and initialize it at 0 in the `create-producers` procedure (`set goods 0`).
3. We also want to create four vendors directly around the production center. Write a second `create` procedure, but this time ask patches to create a vendor. Check the NetLogo Dictionary for the primitive `sprout` and how it's used with breeds. Next, `ask patches` to `sprout` vendors.

TIP
By default variables initialize at 0, but it's good practice to include the `set <var> 0` code in the `setup` anyway.

4. Set the vendors' shape to "person" and give them a variable `goods` the same way you did for the producer.
5. See what happens when you run the code. Likely, you created vendors on every patch on the world. However, we only need them around the production center on the patches directly to the north, south, east, and west. Go to the NetLogo Dictionary and look for the primitive `at-points`. You can get the coordinates of patches that should "sprout" vendors by right-clicking on those patches in the INTERFACE tab and checking their `pxcor` and `pycor`.
6. When you're done, cross-check your code with the answer below to ensure you got it right.

If you don't include `goods` in `vendors-own [ ]`, you'll get the `Nothing named...` error.

TIP
Use the CHECK button often to verify your syntax is correct.

CODE BLOCK 2.2

```
to setup
  ca
  create-producers 1 [
    set color red
    set shape "house"
    set goods 0

  ]
  ask patches at-points [[0 -1][0 1][-1 0][1 0]] [
    sprout-vendors 1 [
      set color grey
      set shape "person"
      set goods 0
    ]
  ]
  reset-ticks
end
```

TIP
Brackets are a major headache for all starting to code. If you open one and forget to close it you will get an error. The trick is to always write both the opening and closing brackets `[ ]` immediately and then type the code inside them.

If everything went fine, you should see a production center with four vendors surrounding it. To keep things simple, we'll ask vendors to create their neighbors in a centripetal fashion on all neighboring cells that are not occupied, giving them a variable `distance-level` that will label how far they are from the production center. The end result will look like figure 2.2.

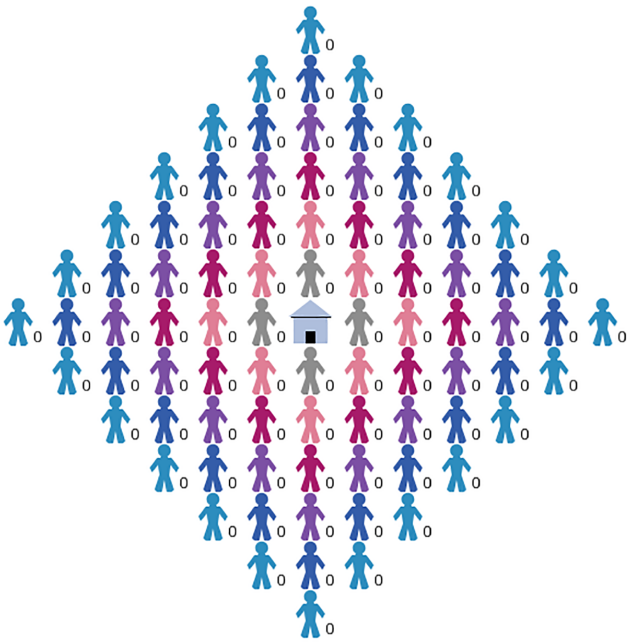


Figure 2.2. The final output of the model's setup. The central agent is the producer, who distributes goods to the nearest four markets. Each color denotes markets at a different distance.

To create a new band of vendors around the existing ones, we'll ask each existing vendor to count the number of empty spaces around them, and then create a new vendor for each one. This is a new procedure, so make sure it is outside of the `setup`.

CODE BLOCK 2.3

```
to populate
  ask vendors [
    let not-occupied-count count neighbors4 with
    [ not any? turtles-here ]
    hatch not-occupied-count [
      set color color - 10
      set distance-level [distance-level + 1] of myself
```

CODE BLOCK 2.3 (cont.)

```
      set size 1
      set goods 0
      move-to one-of neighbors4 with
      [ not any? turtles-here ]
    ]
  ]
end
```

Let's go through this code step by step. We first determine how many empty patches a vendor has and store that number in a local variable `not-occupied-count`.

TIP
You may want to take a second to explore the NetLogo Dictionary for the differences between `hatch`, `sprout`, and `create-turtles`.

Why do we use `turtles-here` instead of `vendors-here`? If we did the latter, we would place a new vendor on top of the production center.

We then hatch the required number of new agents and give them all the necessary variables. Note the use of `myself`, for example, in:

CODE BLOCK 2.4

```
set distance-level [distance-level  + 1] of myself
```

You can read this line of code as *set (my variable) distance-level* to be equal to the *distance-level* of the turtle who asked (`myself`, which here refers to my parent turtle) plus one. As a result, the first tier of vendors will have distance 1, the second will have 2, and so on. Did you get the "nothing named distance-level has been defined" error? Remember to add `distance-level` to the list of vendors and producers variables (`vendors-own [ ] producers-own [ ]`). Also, it is important to initialize it when creating the producer (`set distance-level 0`) and the first four gray vendors (`set distance-level 1`) in `setup`.

Right-click on the producer and one of the closest vendors and inspect whether they have the right distance level.

DON'T FORGET
Did you get the `Nothing named LEVEL...` error? Don't forget to make the slider. Also make sure to set the max to 6 or it will cause an error later.

Now we need to repeat the process with the number of levels of vendors we want. Start by creating a slider in the INTERFACE tab. Call it `level`, set the range from 1 to 6, and set the default value to 6. To deploy the `populate` procedure multiple times, we'll use the `repeat` primitive. Look up its definition in the NetLogo Dictionary. It takes a command block (a procedure or a primitive) and runs it the number of times specified. At the end of the `setup` procedure, write:

CODE BLOCK 2.5

```
repeat (level - 1) [populate]
```

Why `level - 1`? Because we have already created one `distance-level` of vendors: the first four gray agents from the `setup`. Finally, we will visualize the number of goods for each vendor using a label. Add the following line at the end of the `setup` procedure:

CODE BLOCK 2.6

```
ask vendors [set label goods]
```

The full `setup` procedure should look like this:

CODE BLOCK 2.7

```
breed [ producers producer ]
breed [ vendors vendor ]
vendors-own [ distance-level goods ]
producers-own [ distance-level goods ]

to setup
   clear-all
   create-producers 1 [
      set color red
      set shape "house"
      set distance-level 0
      set goods 0
   ]
   ask patches at-points [[0 -1][0 1][-1 0][1 0]] [
      sprout-vendors 1 [
         set color grey
         set shape "person"
         set size 1
         set distance-level 1
         set goods 0
      ]
   ]

   repeat ( level - 1 ) [ populate ]
   ask vendors  [ set label goods ]
   reset-ticks
end
```

TIP

We will talk more about testing in chapter 8. For now, make sure everything is correct every time you write a code block.

You have now finished a significant chunk of code, so it's a good time to test it. Head to the INTERFACE tab and use the SETUP button (create one if you haven't already) to make sure you get a colorful diamond of vendors with a number of goods equal to 0 (fig. 2.2). Right-click on one of the agents at each distance band and check that their `distance-level` is correct (including the producer at distance 0).

MANIPULATING DIFFERENT TYPES OF VARIABLES

variable: attribute of an agent, patch, or the world that can change over the course of a simulation run, such as age, fitness, grass, or number of agents alive.

A **variable** can be thought of as a changeable attribute or characteristic of any of the model's elements. It can denote any feature of an agent, patch, or the model in general. Age, gender, energy level, amount of money, and association with a given group are all examples of **agent variables**. Some of these characteristics are built-in within NetLogo, for example, the location (`xcor` and `ycor`) or color (`color`). Built-in patch variables start with "p" (e.g., `pcolor` and `pxcor`) to differentiate them from turtle variables. The modeler may define any number of custom variables, just like we did with the `goods` variable of the producer and vendors.

DON'T FORGET
In NetLogo, any variable (global, agent, or patch) may be read and modified by any other agent using `set` or `ask`.

All INTERFACE tools, such as sliders, choosers, switches, or input boxes, create global variables. The DISTANCE-LEVEL slider sets the **value** of a global variable (i.e., the observer) but is also accessible to all entities (turtles, patches, etc.). Another way to define a global variable is through the variable lists at the beginning of the code introduced in `globals [ ]`. If the variable is a characteristic of only certain agents, then use `turtles-own`, `patches-own`, or `<breeds>-own` (corner brackets here indicate that "breed" should be substituted with "producers" or "vendors," for instance). This allows you to attach a separate value for each turtle or each patch, even if they are all referred to by the same variable name. Although all turtles have a `color` variable, we may assign different values to each turtle: `[color] of turtle 0` may be different than `[color] of turtle 1`. You can use the COMMAND CENTER at the bottom of the INTERFACE tab to check this for yourself by typing `show [color] of turtle 1` into the bottom box and hitting ENTER. Try again using other turtles' ID numbers (their `who` variable), or swapping the `color` variable for another such as `xcor`.

Go through the code to identify variables and classify them into global, agent, patch, and breed-specific.

The COMMAND CENTER is a console on the INTERFACE tab. You can execute code in it without writing in the CODE tab.

All of the variables just described are permanent attributes of your model. The agents will continue to have them throughout the execution of the simulation. Their values may change—for example, a turtle's `age` will increase, but the same turtle will not find itself without an `age` at any point. On the other hand, agents often need to make a decision or perform an action based on current circumstances. In those cases we use **local variables**, which can be thought of as temporary attributes that are created on the fly. Have another look at the code we have written so far and try to find a local variable. Inside the `populate` procedure we have used the `let` primitive to define a temporary variable of a turtle, `not-occupied-count`, that counts (`count`) the number of empty (`with [ not any? turtles-here ]`) patches in its immediate neighborhood (`neighbors4`).

Only turtles have a `who` variable; the patch's equivalent is its coordinates (e.g., `patch 0 0` refers to the patch at the point of origin).

This variable is only accessible inside the `ask vendors` command within the `populate` procedure, that is, between the brackets we opened after `ask vendors` and closed at the end of the procedure. If you try to call it from the `setup` or from any other place, NetLogo will prompt you that "nothing named not-occupied-count has been defined." We will define local variables a few more times throughout the code, so look out for `let` and track where those variables are accessible (or not).

Do you remember how we checked whether the patch was empty in chapter 1? Often the same thing can be coded in multiple ways.

What happens if you write `print not-occupied-count` in the `setup` or at the beginning of the `go`?

2.3 Running the Trade Model

Now that the producer and vendors have their variables defined, we need to program the main loop of the simulation. This is typically a procedure named `go`, which defines the sequence of actions that happen at each time step. The first two tasks are to generate the goods and to distribute them to the four closest markets. Write inside the `go` procedure:

CODE BLOCK 2.8

```
ask producers [ set goods production-level ]
ask producers [ trade ]
```

The `production-level` variable is a global variable, and we would like it to be user-defined. Create a slider `production-level` and set it to a range between 4 and 50. Next, we need to write the `trade` pro-

If you decide to use a fixed value for production level, how would you define it instead of making a slider?

cedure. Trading consists of two actions: finding a suitable trading partner and performing a transaction. First, each agent needs to choose a vendor that is farther than them from the production center (to move the goods outward) and is eager to trade (their storage cannot be full). If they find an appropriate candidate, they can trade; that is, increase the other's number of goods and decrease their own. We will be doing this until we reach the number of items we want to sell. In this case we define this number of goods on sale as the percentage of items `storage-threshold` we currently have in the `storage`.

Well-laid-out code is easier to read. Use the TAB button to automatically lay out in NetLogo.

CODE BLOCK 2.9

```
to trade
  let next-tier-neighbors (vendors-on neighbors4) with
  [ distance-level = [ distance-level + 1 ] of myself ]

  while [goods > storage * storage-threshold and any?
  next-tier-neighbors with [ goods < storage ]] [
      set goods goods - 1
      ask one-of next-tier-neighbors with [ goods
      < storage ] [
        set goods goods + 1
      ]
    ]
end
```

If you hit the CHECK button, you'll get a familiar sight—the yellow warning that nothing named `storage` has been defined. Create a new slider and set the values to between 0 and 100 with 50 as the default. You will also need to do this for `storage-threshold`. This time, set the slider values to between 0 and 1 with an increment of 0.01 and a default value of 0.5. Altogether, this new code may look a bit intimidating at first, so let's unpack it line by line. First, we define who are we going to be trading with:

DON'T FORGET
You should have four sliders by now on the INTERFACE tab: LEVEL, PRODUCTION-LEVEL, STORAGE, and STORAGE-THRESHOLD.

CODE BLOCK 2.10

```
let next-tier-neighbors (vendors-on neighbors4) with
[distance-level = [distance-level  + 1] of myself]
```

Here, we have created a **local variable** `next-tier-neighbors`, which stores all the vendors on the neighboring patches (`vendors-on neighbors4`) whose distance level is one step higher than the calling agent (`with [distance-level = [distance-level + 1] of myself]`). Note that the parentheses around `(vendors-on neighbors4)` are necessary for the order of operations to be correct: `with [...]` refers to the variables of the vendors rather than the patches designated by `neighbors4`. The next few lines are the heart of this procedure. Let's express it in pseudocode:

TIP
If you click on a square bracket, NetLogo highlights its counterpart.

PSEUDOCODE BLOCK 2.11

```
While you have any goods left for sale AND if any
trading partner has space left in storage:
        ask one of them to increase the number of
        goods by one
        and decrease your own number of goods by one
```

Now that the flow of the logic is clear, let's look at a few potential traps in the code. First, you might have noticed the repetition of the following line:

CODE BLOCK 2.12

```
next-tier-neighbors with [ goods < storage ]
```

You may wonder why we haven't just asked one of the other traders to trade with us, like this:

CODE BLOCK 2.13

```
ask one-of next-tier-neighbors [ set goods goods + 1 ]
```

This would work most of the time. But sooner or later, one of the agents would find itself in a situation where none of their neighbors fulfilled the criteria (fewer goods than storage). What then? This is when NetLogo halts and throws a rather cryptic error message:

CODE BLOCK 2.14

```
ASK expected input to be an agent or agentset but got
NOBODY instead.
```

This algorithmic structure of first establishing that there are one or more turtles fulfilling a condition, and then asking one of them to do something is a common code structure in NetLogo.

What this means is that in order to use the primitive `ask`, there must be at least one agent (i.e., not "nobody") that can do it. If there is not one, the simulation does not know what to do and halts. To avoid this, we first check whether there is an agent who fulfills the criteria `any? next-`

`tier-neighbors with [ goods < storage ]` and only then ask one of them to perform an action. We need to specify again that it has to be a `next-tier-neighbors with [ goods < storage ]`; otherwise we will just trade with anyone. You will see this apparent repetition in NetLogo models again and again.

LOOPING CODE SEGMENTS

Congratulations! This is your first time using a while loop, one of three main control flow structures used in NetLogo. Let's take a closer look at these control flow structures.

Here, again, we will turn to the handy tool of pseudocode. The structure of an **if loop** is the following:

PSEUDOCODE BLOCK 2.15

```
If condition is true:
    [ do x ]
```

The structure of a **while loop** is the following:

PSEUDOCODE BLOCK 2.16

```
While condition is true:
    [ keep on doing x ]
```

The structure of an **ifelse loop** is the following:

PSEUDOCODE BLOCK 2.17

```
ifelse condition is true:
    [ do x ]
otherwise, if false:
    [ do y ]
```

We will reuse the trade procedure to model exchange between vendors. Add the following line to the `go` procedure:

CODE BLOCK 2.18

```
ask vendors with [ goods > 0 ] [ trade ]
```

Avoiding unnecessary operations speeds things up. For example, at the beginning of the simulation only a few agents have any goods.

Here we ask traders who have goods to go and sell their goods. We could have just asked all vendors `ask vendors [ trade ]`, but this way we avoid having empty-handed vendors perform all the trading operations for nothing.

To finish off the `go` procedure, add a line asking vendors to display the number of goods they have using the `label` variable (similar to the

one in the `setup`). Because the number of goods changes with each tick, we need to update the label accordingly. Also, write a line telling vendors that those who have at least one good in their storage lose an item at the end of a time step to account for accidental loss or breakage of goods; if we do not add this, the amount of goods never decreases. The full `go` procedure should look like this:

DON'T FORGET
Add `tick` at the end of the `go` procedure.

CODE BLOCK 2.19

```
to go
   ask turtles [ set label goods ]
   ask producers [ set goods production-level ]
   ask producers [ trade ]
   ask vendors with [ goods > 0 ] [ trade ]
   ask vendors with [ goods >= 1 ]
       [ set goods goods - 1 ]
   tick
end
```

Explore the way in which the frequency curves change depending on the production level, the storage, and trading capacity (storage-threshold). Does it matter how far from a production center the market is located?

LISTS

The simulation is mostly ready, but as you might have noticed when exploring the model with the sliders, the results are hard to interpret directly. We need some kind of output measure to see how the trends of goods change with the different scenarios. What interests us are the different amounts of market goods at various distances from the production center. The **stochastic** nature of the simulation means that the amount will differ from one vendor to another, but we can aggregate them to get a mean value for each `distance-level`. We will do this using NetLogo lists, an exceedingly useful and computationally efficient data structure. Lists help enormously when creating more complex simulations, so it really is worth spending some time to understand them.

stochasticity: degree of randomness in a model's dynamics and outcomes; involving randomly generated numbers.

list: an ordered sequence of values. They can be of any type (integer, string, another list, etc.) and can repeat multiple times.

Let's start with adding a new global variable `mean-goods` at the very beginning of the code. This will be the global list:

CODE BLOCK 2.20

```
globals [ mean-goods ]
```

Then we need to initialize our list in the `setup` procedure:

CODE BLOCK 2.21

```
set mean-goods [ 0 0 0 0 0 0 ]
```

If you go to the INTERFACE tab, click SETUP, and write `mean-goods` in the Command Center, it will print your new list: `[0 0 0 0 0 0]`. Each value stores the mean number of goods at each particular `distance-level`. So the first value will be used to store volumes from the `distance-level 1`, the second from the `distance-level 2`, and so on. At each time step we will calculate and add a new value to the appropriate position. Note that NetLogo lists are indexed such that the first entry is `item 0`. This becomes particularly important when you try to read the value from a specific position in a list.

The alternative way of initializing a list is: `set <listname> (list x y z)`

You can visualize a list as a kind of spreadsheet row in which each cell contains a value.

distance 1	distance 2	distance 3	distance 4	distance 5
index 0	index 1	index 2	index 3	index 4
34.7	28.98	25.07	15.12	5.1

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

Let's create a new type of procedure—a **reporter** procedure. It will calculate the volume of goods of all vendors in one `distance-level`, and then `report` a number back. Add the following code:

CODE BLOCK 2.22

```
to-report calculate-volume [ current-distance-level ]
  let vendors-tier vendors with [ distance-level =
  current-distance-level ]
  report mean [ goods ] of vendors-tier
end
```

You can easily spot a reporter because it starts with `to-report ...`, instead of `to ...` like in a standard procedure. You might have also noticed that we pass a variable here, `current-distance-level`, in the brackets after the procedure name. We use it to specify which distance-level we are calculating (`vendors with [distance-level = current-distance level]`). So when the reporter function is called to calculate the mean

number of goods for distance-level 1, it will pass `current-distance-level` equal to 1; when calculating distance-level 2, `current-distance-level` will be set to 2; and so on. We will now construct a procedure that will collect this **artificial data** for each distance-level by passing `current-distance-level` and appending the results of `calculate-volume` to the `mean-goods` list. Don't forget to add the `iterate-list-of-mean-goods` procedure to `go`.

artificial data: model output that can be compared to real archaeological data.

CODE BLOCK 2.23

```
to iterate-list-of-mean-goods
  let i 1
  while [ i <= level ] [
    let goods-at-distance calculate-volume i
    set mean-goods replace-item (i - 1) mean-goods
    goods-at-distance
    set i i + 1
  ]
end
```

This is the first time we are using lists, so let's go through one iteration of the while loop to see how it works in practice.

1. Initially, $i = 1$ (`let i 1`), so the condition `while i <= level` is true ($1 < 6$). This means that the rest of the code will be executed.
2. Next, `let goods-at-distance calculate-volume i`. Here, we pass 1 (current value of `i`) to `calculate-volume` to calculate the mean volume of goods of all agents at `distance-level 1`. We attach the result (say, 1.97) to a local variable `goods-at-distance`, which is now equal to 1.97.
3. Then `set mean-goods replace-item (i - 1) mean-goods goods-at-distance`. We replace the first item of the list (`mean-goods`) with the current value of `goods-at-distance`. Remember that the first item has the index 0, so `i - 1` returns `item 0`.
4. Finally, we move the counter (`set i i + 1`) so now `i = 2`.
5. The loop goes back to the beginning and executes all the previous steps again but with `i = 2`.

TIP
Make sure the while loop is certain to reach a `false` at some point, otherwise it will run forever. Use HALT from the TOOLS menu to force a stop if this happens.

6. This repeats until i + 1 = 7 and the `while [ i <= level ]` is no longer true (7 > 6). The program exits the while loop and continues execution with the next line of code.

Lists are highly computationally efficient. You can speed up your model considerably by making good use of them.

You may be wondering why we had to write `set mean-goods` instead of just updating it (i.e., using `replace-item`). Lists are immutable, which means that when we want to modify one, we need to create a new list that is an updated version of the original, and then use `set` to overwrite the old version. In the next chapter you will learn how to iterate more efficiently over a list to build a new updated list.

Going through a sequence of consecutive numbers and updating the current status is a very common process in programming, and you will most definitely use it in your models. In fact, this is what happens with `ask turtles`—the program goes through a shuffled list of turtles and makes each execute its actions until the last one is finished. Similarly, you can traverse through any list with the primitives `foreach` or `map` to run some code on each value. Chapter 3 will have a few examples of this.

PLOTTING MODEL DATA

If you change the number of levels, you'll also need to change the number of pens.

Now that we have the mean value of goods at each distance level stored in a list, we can make a graph that displays this data in an easy-to-read visual format. Go back to the INTERFACE tab, right-click anywhere on the white background, and choose PLOT.

There is already a pen built for us by default: `plot count turtles`. Remove it, then click ADD PEN, and a new line appears. Click the pen icon and in the PEN UPDATE COMMAND box, write:

CODE BLOCK 2.24

```
plot item 0 mean-goods
```

This code should be understandable now: we grab the first item from the `mean-goods` list and ask NetLogo to plot it. To plot the volume of goods at distance-level 2, 3, 4 . . . , we need to create separate pens. Repeat the steps above using the ADD PEN button and modifying the plotting command for each (fig. 2.3).

Reduce the pace of the model using the speed slider to see the initial phases of the frequency curves.

Keep going until you have pens for all the levels. You can add a legend by ticking the SHOW LEGEND? box. You can change the color of the pens by

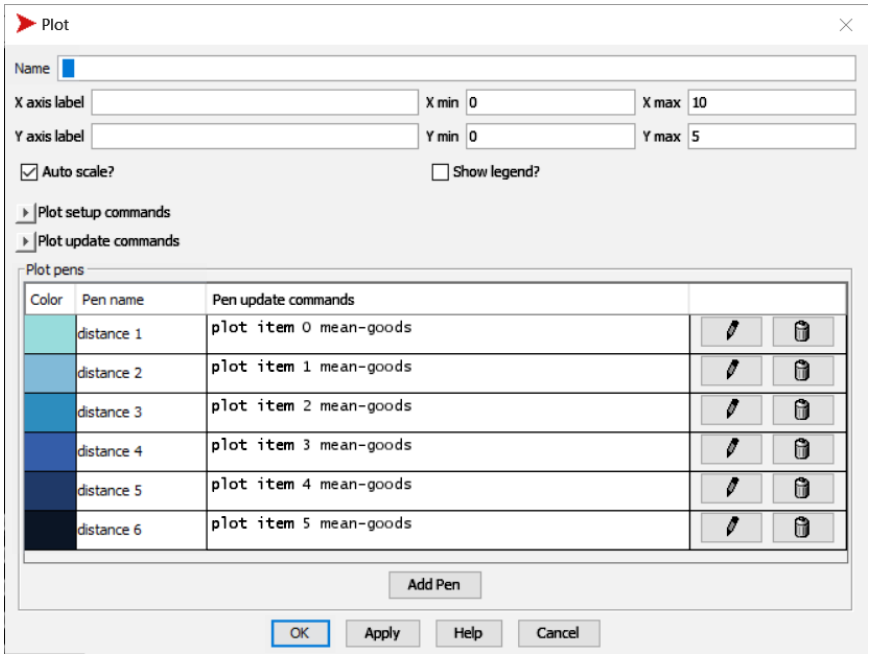


Figure 2.3. The PLOT window with input for six distance levels.

clicking on the color field. If you run the model now, you should get a plot similar to figure 2.4.

You now have the means to explore the model. When you run the simulation, you should see the goods spreading across the world from the production center to the most peripheral agents. Play with the sliders to see how the market curves change depending on the storage capacity of traders and the volume of production.

2.4 Debugging

It is certain that while writing the code you had a few moments where NetLogo treated you to the yellow ribbon of despair—that is, gave you an error message. Or maybe the simulation didn't work as intended (e.g., the goods did not move). It is a common saying among programmers that you spend 20% of the time writing code and 80% trying to figure out why it isn't working. It is very easy to introduce small errors into the code, and only some of them trigger the debugger.

verification: establishing that the code, as written, works correctly, i.e., as the modeler intended.

Verification is a process in which you ensure that the code works as intended, that is, that there are no code errors (so-called "bugs"). We discuss

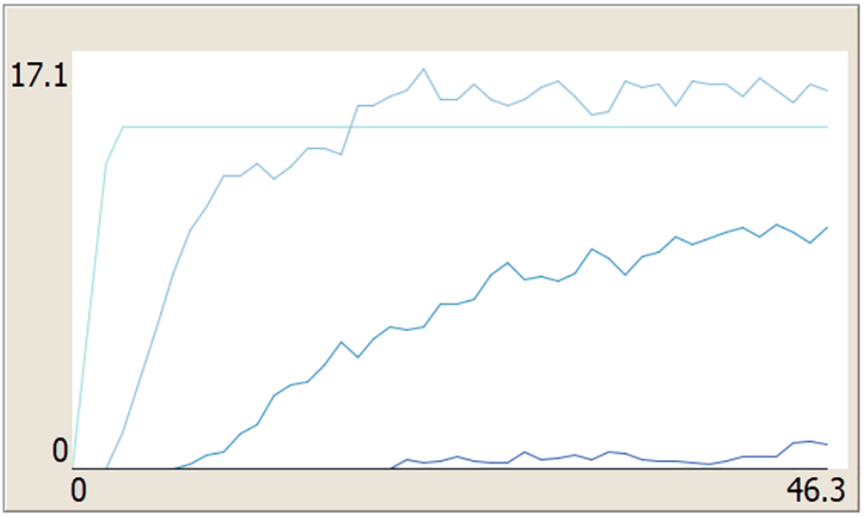


Figure 2.4. The difference in the uptake of goods at each distance level in figure 2.3. Markets closer to the production center get goods earlier and their uptake curves are steeper. The straight horizontal line represents the four markets closest to the production center which always have their storage full.

debugging and code testing in detail in chapter 8, but for now it is a good idea to think about how to avoid introducing errors in the first place. There are three main strategies you should follow to make code development less frustrating:

modular code development: an approach to coding where code is written incrementally and within many independent procedures.

- modular code development;
- pseudocode; and
- quick debugging with `show` and inspecting agents.

First, **modular code development** refers to the simple fact that it's easier to track down where errors hide if you continually check whether what you wrote works as intended. Instead of trying to write out the full simulation at once, write a single procedure and check it before you move on to the next one. Writing multiple small procedures instead of a long Frankenstein-like `go` makes it easier to spot problems as you debug each procedure individually. Similarly, always start from a simple implementation before you make a model more complex. For example, remember how in the `setup` we initially created traders on all patches before we restricted them to the four central ones? This is an example of starting with the simplest implementation before refining it and adding complexity.

TIP
It's better to develop many small procedures while checking them continuously than to write out everything in one go and not be sure where the bug is.

Second, your best friend in keeping control over your code is **pseudocode**. You should see it as the table of contents of your model. It's a good idea to look at your code periodically to ensure that what you are coding is really necessary, as it's easy to fall down a rabbit hole of making a particular aspect of the model exceedingly complex before developing other parts.

Pseudocode is as useful to you—the modeler—as it is to those who will read your code later.

Finally, and most frustratingly, the code sometimes just does not work. The traders do not appear, the goods do not spread through the world, etc. In those cases you need to first identify which part of the code is faulty. It's important to remember that code always "works": it does what you *told it to do*, even if it may not necessarily be the same as what you *want it to do*. In those cases, right-click on a turtle or patch to inspect its variables. If its values are not what you expected, look at the code where these variables are set. You should also familiarize yourself with the `print`, `type`, and `show` primitives—they enable you to check whether a section of code is actually being executed. They print out, in the COMMAND CENTER at the bottom of the INTERFACE tab, a short text (such as "Yes, procedure *ABC* is executing!") or the current values of variables, both of which will help you identify what went wrong. A common strategy is to add a series of `print "made it to X"` lines at various stages of a procedure so that you can identify the last successful step of the code, and by extension the one that does not work, such as this one:

TIP
Limit the number of agents when debugging. Otherwise you may end up with 100 turtles printing out their variables.

CODE BLOCK 2.25

```
to trade
...
  print "trading has been called"
  while [goods > storage * storage-threshold and any?
  next-tier-neighbors with [ goods < storage ]] [
      print "someone has goods and wants to trade"
      print next-tier-neighbors

      set goods goods - 1
      show goods
      ask one-of next-tier-neighbors with
      [ goods < storage ] [
```

Note the two strategies of (a) using quoted text strings and (b) printing variable values to make useful debugging messages.

CODE BLOCK 2.25 (cont.)

```
            show goods
          set goods goods + 1
            show goods
        ]
        print "trade happened!"
      ]
end
```

Remember to remove all print statements when you start running experiments—they take a lot of computational power and will slow down your run speed considerably.

ontology: the full representation of the model's world, including names and definitions of entities and their categories and properties, the relationships between entities, and the rules of behavior.

Let's look at the full pseudocode of the model as it is right now. It constitutes the **ontology** of the model. This is the artificial world we consider as representative of a past economic system we study. All conclusions we can draw on the basis of this model come with the caveat "assuming our representation is truthful and captures key dynamics of the real system . . ." We'll discuss in the next chapter how to gain confidence as to whether the model is, in fact, the right representation. For now, it's important to notice that this representation is formal and fully defined. There is no ambiguity as to what it means to trade or how many commercial partners each vendor has. In contrast to verbally defined models, there is only one way in which each concept can be understood. Compare the pseudocode to your model to see how you translated each concept into executable code.

PSEUDOCODE BLOCK 2.26

```
to setup
    create producers
    create vendors at neighboring patches
    populate the world with a number of concentric
    levels of vendors
```

PSEUDOCODE BLOCK 2.26 (cont.)

```
to go
    ask producers -> produce goods
    ask producers -> distribute goods (trade)
    ask vendors -> distribute goods over storage
    threshold (trade-vendors)
    ask vendors -> consume one item of its goods

to populate
    hatch agents at neighboring cells
    give them a distance level

to trade
    if any goods in your storage
         if any neighbors want to buy (their goods
         < storage)
             give an item to one of the neighbors
             remove one item from own storage
```

This is a highly abstract model designed to investigate the dynamics of a simple trade system. Although the majority of models aim to test hypotheses related to particular patterns in the archaeological record, occasionally we build more abstract, theory-driven models. Sometimes called *toy models*, they are used to play with the system to better understand, for example, how a change in one aspect of the system (e.g., level of production) affects another aspect (e.g., how far the goods travel). In this case, the exact model you have just replicated (Romanowska 2018) was used to investigate the role of intermediaries in Roman trade. It was possible to show that the limited capacity of a redistribution center can play a decisive role in terms of a distribution of a particular good, even in cases where production and demand for that good are high. We will further discuss the spectrum of models from highly abstract theory-driven simulations to more realistic, data-driven ones in chapter 8.

This model also illustrates one useful way to compare abstract dynamics to highly specific archaeological data.

2.5 Summary

In chapter 1, we introduced the most basic elements of NetLogo. Here, in chapter 2, we dove into the nitty-gritty of coding in NetLogo. We learned how using variables in a local context can be more dynamic, but that global variables are necessary when their values need to be accessed by all agents in the model. We explored loops, in particular, conditional loops, which are the main vehicle for model logic. We used breeds, which enabled us to categorize groups of agents efficiently. Finally, we introduced one of the most difficult yet important methods in NetLogo—lists—which enable a much swifter and easier manipulation of code. In the process, we used several common primitives and further cemented our familiarity with the NetLogo dictionary.

This may have been a lot to take in, but, on the plus side, we covered all the most important elements of NetLogo syntax. The foundations have been laid, and in theory you now have the ability to create any model, even if in practice this may require honing these new skills a little bit more. Moreover, loops, lists, variables, etc. are coding elements common to all programming languages, and you will come across them regardless of whether you continue with NetLogo or move on to a general-purpose language such as Python, R, or Java. In the next chapter, you will see these common coding elements combined in more complex structures, but in principle you will have seen nearly all of it before.

See chapter 9 for an introduction to programming in Python and R.

In this chapter we employed pseudocode, showing its many uses during model development; it's also a useful tool to guide you in communicating models to your peers. As you develop your code, you will want to ask friends and colleagues to check your work. Pseudocode can be instrumental in facilitating this. Finally, we explored different strategies used to squash bugs in the code and get the code to work the way we want it to. Practice makes perfect, so the more opportunities you have to write code, the easier it will become.

End-of-Chapter Exercises

1. If you set the level slider at more than 6, you will get an error. Can you figure out why and change the code so that it incorporates any

number of distance levels? Check out the `n-values` primitive, which creates a list composed of a specified number of elements.

2. Add a second production center. This may be by changing one of the existing vendors to a producer or by repeating the `setup` procedures to create two producers and two overlapping concentric sets of markets.

3. At the moment, markets remove one good at each time step to account for a general loss from breakage and the like. We can use this code to represent consumption instead. Change the number of consumed goods into a user-defined value and explore how the results change. Then, in a second iteration, make the number of consumed goods market-specific, so that some markets representing large cities consume more goods than others.

4. Can you think of other ways in which we could measure the economic activity in the model? The total number of goods, number of transactions, amount of archaeological material being generated at each step, etc. are just a few options. Think about what would be a meaningful measure and how to implement it.

Further Reading

- T. Brughmans and J. Poblome. 2016. "Roman Bazaar or Market Economy? Explaining Tableware Distributions in the Roman East through Computational Modelling." *Antiquity* 90 (350): 393–408. doi:10.15184/aqy.2016.35

- A. Chliaoutakis and G. Chalkiadakis. 2020. "An Agent-Based Model for Simulating Inter-Settlement Trade in Past Societies." *Journal of Artificial Societies and Social Simulation* 23 (3): 10. doi:10.18564/jasss.4341

- S. Graham and S. Weingart. 2015. "The Equifinality of Archaeological Networks: An Agent-Based Exploratory Lab Approach." *Journal of Archaeological Method and Theory* 22, no. 1 (December): 248–274. doi:10.1007/s10816-014-9230-y

▷ L. Hamill and N. Gilbert. 2016. *Agent-Based Modelling in Economics.* Chichester, UK: Wiley. doi:10.1002/9781118945520

▷ I. Romanowska et al. 2021. "A Study of the Centuries-Long Reliance on Local Ceramics in Jerash Through Full Quantification and Simulation." *Journal of Archaeological Method and Theory* (February). doi:10.1007/s10816-021-09510-0

▷ I. Romanowska. 2018. "Using Agent-Based Modelling to Infer Economic Processes in the Past." In *Quantifying Ancient Economies. Problems and Methodologies,* edited by J. R. Rodriguez, V. R. Calvo, and J. M. Bermudez Lorenzo, 107–118. *Instrumenta* 60. Barcelona, Spain: University of Barcelona.

NOTES