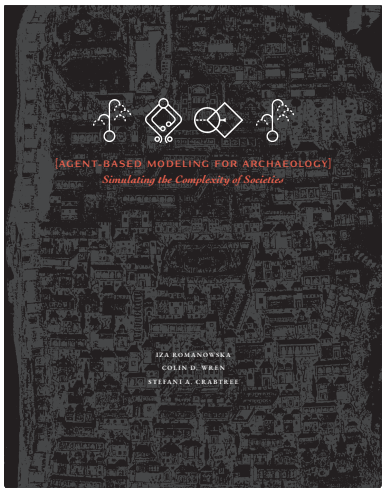
PLEASE NOTE:

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

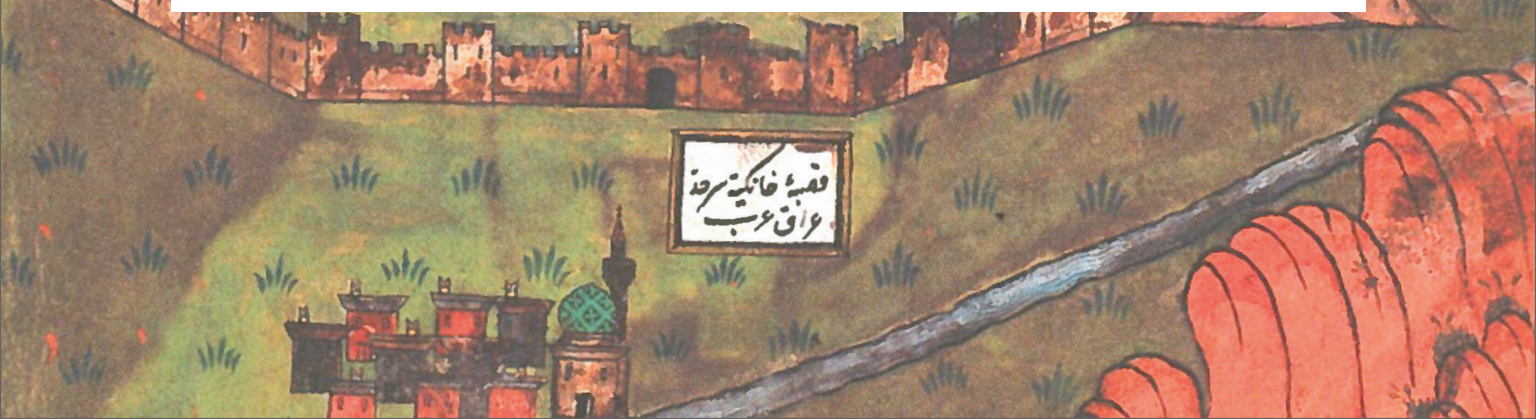


REGARDING COLOR:

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




اوجان
خان لیکس
تورکن کندی

THE FOUNDATIONAL STEPS OF BUILDING AN AGENT-BASED MODEL

1.0 Introduction

In this and the subsequent two chapters, we will use examples of simulations typical of archaeological agent-based modeling. Our aims are to:

- familiarize you with the agent-based modeling technique and some vocabulary specific to complexity science and simulation;
- introduce the basic principles of coding; and
- explain the epistemological position of ABM in scientific practice and its role in the wider field of complexity science.

OVERVIEW

▷ Intro tutorial in NetLogo software: INTERFACE and CODE tabs, agents, and procedures
▷ Definitions of modeling, simulation, and algorithm
▷ What is pseudocode?
▷ Types and purposes of models

Thus, by the end of **Part I: Learning to Walk**,[1] you will have a good grasp of what ABM is all about, what it's for, and how can it benefit archaeological inquiry. You will also be able to build a basic simulation and understand the challenges involved. Contrary to many textbooks that begin with background, each chapter will start with a hands-on practical element—implementing a model in NetLogo—and only later provide theoretical explanation. In our experience, our students gain a much better understanding of the concepts related to simulation and complexity science once they have tried them out for themselves first. If, while working through these chapters, you feel as if you do not fully understand what you are doing, reading ahead a few pages will likely clarify the situation, as each chapter is organized as a self-contained unit; trust that all will fall into place by the end of the part. Important note: When in doubt about any aspect of coding, head straight to the NetLogo documentation. In particular, the NetLogo Dictionary[2] and the Programming Guide[3] will be

[1] You can find all code written in this chapter in the ABMA Code Repo: https://github.com/SantaFeInstitute/ABMA/tree/master/ch1

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

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

of great help whenever you feel lost. They may initially look confusing but you'll quickly get to know them well.

To begin our tutorials, we will start with a simple model of human dispersal. Models of human movement lend themselves particularly well to simulation, especially ABM. They concern an inherently spatial and dynamic process, characteristics that make ABM particularly useful in comparison with other modeling techniques. To illustrate the process of developing a simulation, we will use a well-known model by Young and Bettinger (1995). It was designed to investigate the first dispersal of humans out of Africa. This model is a good example of the so-called "models from first principles" or "toy models"—that is, simple, abstract models investigating basic dynamics of a system, which became very popular in archaeology in the late 1990s (Lake 2014). We will implement the Young and Bettinger model in a NetLogo tutorial, taking you through the full process of model development. Finally, we will look back and use the experience gained to define such terms as *model* and *simulation*. We will discuss how these tools fit within the scientific process and how they can aid archaeological research.

Model:
Young & Bettinger (Y&B) Dispersal

ABMA Code Repo:
ch1_Y&Bdispersal

1.1 The Model: Young & Bettinger's Simulation of Dispersal

In their 1995 paper, Young and Bettinger (*Y&B* henceforth) used a simple diffusion model to investigate the out-of-Africa dispersal of modern humans (Gamble 2013). The authors asked whether the patterns in the archaeological record could be explained using the most basic principles of population growth and spread. Their simulation is not an agent-based model; instead, the authors used an equation known as the Fisher–Skellam–KPP model. Using equation-based models as the foundation for building an agent-based model is common, as they often provide a very clear baseline.

You can find a detailed description of the Fisher-Skellam-KPP model in chapter 4.

The main *Y&B* algorithm creates a simple diffusion wave in all directions from the point of origin (East Africa) following a gradient, from more densely inhabited areas to those that are less populous. In a mechanism similar to that of a spreading wildfire, the population moves away from the point of origin in all directions and without turning back on itself.

If this was the whole model, we could easily predict the results—areas closer to the point of origin in a strict geographical distance will be inhabited earlier than those farther away from it. However, the archaeological

data indicate that humans arrived in some areas (e.g., Europe) later than in others (e.g., Southeast Asia) irrespective of their geographical proximity. Clearly, a factor other than geographical distance must have influenced migration, slowing down or speeding up the dispersal in certain regions. *Y&B* hypothesized that the most plausible factor is the environment: species spread more rapidly across familiar environments than those to which they have not yet adapted. To account for that, the population growth rate and mobility values were set higher in the tropics and semitropical conditions similar to the starting area of the dispersal and lower in the northern, less familiar regions.

Once the model was run, the pattern of dispersal was compared to archaeological data consisting of locations of archaeological sites and the dates of earliest traces of *Homo sapiens* in each region. The two patterns (often referred to as *artificial data* and *empirical data*) matched very well. But what does it mean in terms of our understanding of the out-of-Africa dispersal? Had the *Y&B* model captured the dynamics of that process? We will talk about the knowledge generation process in simulation studies at the end of this chapter.

Let's look at the model in more detail now. We will write an algorithmic description of it to make the subsequent implementation in code easier. An **algorithm** is simply a set of instructions like a cooking recipe: "Take A, mix with B, add C, and you get D" (see ch. 4 for a more thorough definition). We often write algorithms in **pseudocode**, that is, in a way that resembles the syntax of computer code but is readable to humans. You can then use it as a step-by-step summary of what the model does.

algorithm: a sequence of instructions given to a computer.

pseudocode: a simplified notation of the structure of the code.

Almost all ABM simulations consist of two phases. First is the **initialization phase**, where you define the world and its inhabitants, effectively setting up the state of the world at time step zero. Let's write out our model's setup in pseudocode:

PSEUDOCODE BLOCK 1.0

```
Set parameters [ population-growth = popG
                 initial-pop-size = n
                 initial-location = (x,y) ]
```

PSEUDOCODE BLOCK 1.0 (cont.)

```
Create n agents
For each agent, A,
    Place A in (x, y)
```

Second, we define the **run phase** as the clock starts ticking and at each time step the world undergoes a series of events, such as climate change and agents' actions. In pseudocode:

PSEUDOCODE BLOCK 1.1

```
At each time step, T,
  For each agent, A,
    Draw random number, N, between 0-1
    If N < popG AND
    If there is at least one adjacent cell, C,
    (x±1, y±1) that is empty
       Create new agent, B,
       Place B on cell C
```

The pseudocode says that at each time step, each agent will create a new agent if two conditions are met. The first condition is expressed probabilistically. Let's imagine we define population growth as 0.1, aiming at a population that grows by approximately 10% over the course of one time step. Each agent draws a random number between 0 and 1; if this number is between 0.0 and 0.1, then the condition is met. The probability that a random number between 0 and 1 is lower than 0.1 is about 10%, so on average about 10% of agents could produce an offspring. However, we only allow them to do so if there is an empty cell available in their immediate neighborhood that can be colonized. Read carefully through the pseudocode, ensuring that you understand all abbreviations and can point out where they are defined. You have probably noticed that the pseudocode above is a simplified version of the *Y&B* model that does not yet have the environmental factor built in. We will start with this simple version and then expand it as we go. This is a common approach to building models: you start simply and gradually layer on complexity (see ch. 5).

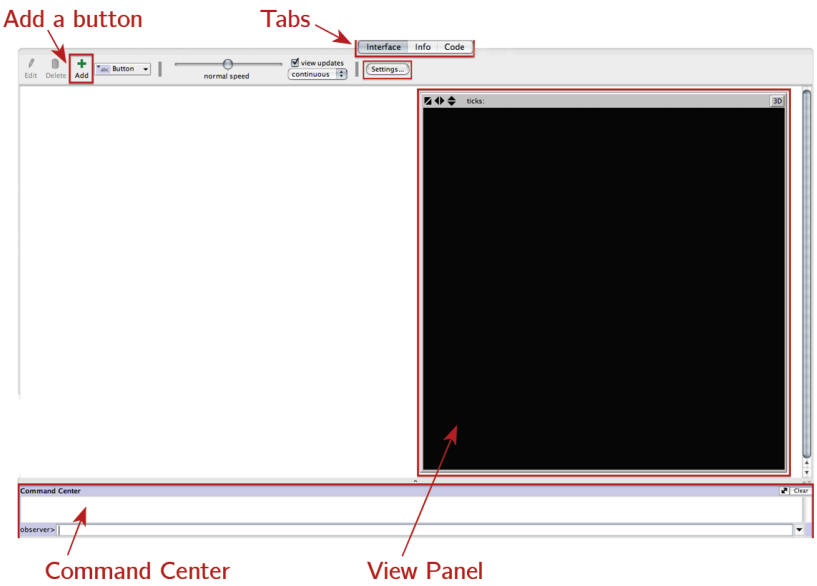


Figure 1.0. Main elements of NetLogo's graphical user INTERFACE tab.

NETLOGO INTERFACE

NetLogo is used widely across virtually all scientific disciplines. It is often the first choice when creating proof-of-concept models, but it can also be used for modeling systems to their full complexity. It is particularly prevalent among social and life scientists (Hauke, Lorscheid, and Meyer 2017) and is by far the most popular agent-based modeling framework among archaeologists (Davies and Romanowska 2018). Logo—the ancestral language of NetLogo—was developed as an educational tool, so it resembles a natural language, making it easy to read and write. Nevertheless, if this is your first attempt at computer programming, you may initially feel intimidated. Fear not: you will rapidly grow accustomed to writing code. We will go particularly slowly with the coding challenges in this chapter to ease you in.

This heritage is why NetLogo refers to agents as **turtles** and to grid cells as **patches**.

Let's start with the NetLogo program itself (fig. 1.0). When you open it, you will immediately notice three tabs: INTERFACE, INFO, and CODE. In the next few pages, we will look at these in turn.

The INTERFACE tab consists of:

- The VIEW panel for watching the simulation;

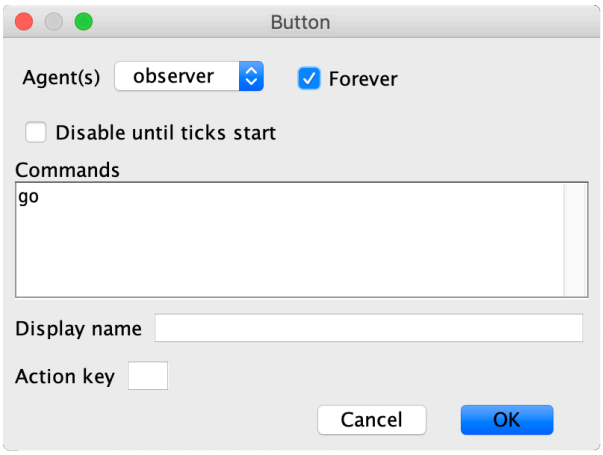


Figure 1.1. The dialog box for adding a GO button. Note that the FOREVER box is checked.

- A few buttons, choosers, and a slider along the top of the window; and
- The COMMAND CENTER toward the bottom of the window.

This INTERFACE tab is where you will run and observe the dynamics of your model while the CODE tab is where you will write out the code of the model itself. You will flip back and forth between these tabs constantly as we continue. Finally, when you're finished you should describe the model and how to use it in the INFO tab (there are headings to get you started).

SETUP PROCEDURE

The two phases we have seen in the pseudocode (initialization and run) are usually called `setup` and `go` in the NetLogo lingo. In the `setup` procedure, we will create the starting population of agents and build their environment. The `go` procedure is the main simulation loop with all processes that the agents and the environment undergo at each time step.

TIP

Right-click anywhere on the white space in the INTERFACE to create a button.

Click the FOREVER box if you want the code to repeat indefinitely.

We will activate (call) those procedures using buttons. Right-click anywhere on the white space and choose BUTTON. A dialog box will pop up (fig. 1.1). Type `setup` in the COMMANDS box and click OK. Follow the above step to create a second button and write `go` in the COMMANDS box. This time also tick the FOREVER box.

Ticking FOREVER means that this action will repeat until the simulation ends or the user toggles the button off again by pressing the GO button a

second time. You can see that the text on both buttons has instantly turned red, indicating an error. The NetLogo interpreter does not recognize the code because `go` and `setup` have not been defined yet. Let's move to the CODE tab to fix it.

The CODE tab is dominated by white space. This is where we write code directing the flow of the simulation. To do so we use **commands** and **reporters**. Commands define the actions of an agent, while reporters calculate and report a value. We will come back to reporters in chapters 2 and 3. There are two types of commands and reporters in NetLogo: the user-defined **procedures** and built-in ones called **primitives**. The NetLogo Dictionary provides a description of all existing primitives and includes sample code.[4]

A **procedure** consists of all code enclosed between `to` and `end`.

primitive: built-in procedure or variable name.

TIP
To check a primitive's documentation, move your cursor so that it is within the word and press F1.

First, to define a procedure we use the keywords `to` and `end`. Like brackets they mark the beginning and the end of a procedure. In addition, we can already specify one action that the simulation has to perform at each time step: move the clock forward after each step.

The `tick` primitive is often used to mark the end of all procedures in the current time step and to start a new one. It moves the time forward and triggers many commands that are called once per time step; for example, it updates plots. Type the following in the CODE tab:

CODE BLOCK 1.2

```
to setup
end

to go
  tick
end
```

Click CHECK every time you write new code or change the existing one.

If you now click the PROCEDURES list at the top of the screen, you'll find that `setup` and `go` are listed there. Next to PROCEDURES is the debugger button, CHECK, which when clicked will check if the code's basic syntax is correct. Click it now. If it throws an error, look at the line it highlights and make sure your spelling is correct.

DON'T FORGET
If you do not `clear-all`, your model will include turtles and patches from previous runs.

Now, let's build the initial state of our model in the `setup` procedure. This procedure usually starts with the `clear-all` primitive, which

[4] https://ccl.northwestern.edu/netlogo/docs/dictionary.html

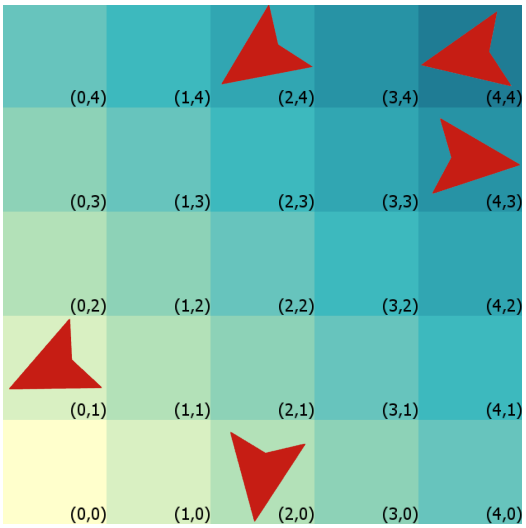


Figure 1.2. A small 5×5 patch landscape with randomly located agents and shaded patch color. We have added numbers that show the (X,Y) coordinates of each patch.

removes any remnants of the previous run, including clearing plots and monitors, and finishes with `reset-ticks`, which resets the clock to zero. Write them in separate lines after `setup` but before `end`.

Now that all the housekeeping has been taken care of, we can move to defining the elements of the simulation. If you look back at the pseudocode, you'll see we need to create a number of agents. In NetLogo, agents are referred to as **turtles**, and the grid cells on which they live are called **patches**. This is a holdover from when NetLogo was developed to teach programming to children. We will use the `create-turtles` primitive to bring our agents to life. We also need to position the agents somewhere in the world using the `setxy` primitive. Let's place them on a random cell between cells $(0, 0)$ and $(5, 5)$. Refer to figure 1.2, where we labeled the shaded patches to give you an idea of the area encompassed by `random 5 random 5`. Inside the `setup` procedure, type:

You can play with the setup in the *ch1_patches* model in the ABMA Code Repo.

CODE BLOCK 1.3

```
to setup
  clear-all
  create-turtles 20 [
    setxy random 5 random 5
  ]
  reset-ticks
end
```

`random 5` chooses one value at random among 0, 1, 2, 3, and 4.

Go back to the INTERFACE tab and hit SETUP. Do your agents appear? The brackets after `create-turtles` can also enclose features (variables) of the agent other than its location, for example, `color`, `size`, or `shape`. All of these built-in variables have default values, but you can also customize them, as in this example:

You can check the documentation of `create-turtles` in the NetLogo Dictionary.

CODE BLOCK 1.4

```
to setup
  clear-all
  create-turtles 20 [
    set color random 140
    set size 2
    set shape "turtle"
    setxy random 5 random 5
  ]
  reset-ticks
end
```

TIP
Brackets are square `[...]`, parentheses are round `(...)`, and curly brackets are . . . curly `{...}`.

When you're done, hit the CHECK button to make sure there are no errors and move back to the INTERFACE tab. Click the SETUP button again. Your agents are clustered to the upper right from the middle point of the screen, because by default the patch at point $(0, 0)$ is located in the center. Instead, we would like the agents to start from the corner. Click the SETTINGS button in the top right corner of the INTERFACE tab. A window will pop up. Change the LOCATION OF ORIGIN to the bottom left corner and untick the WORLD WRAPS HORIZONTALLY and WORLD WRAPS VERTICALLY boxes. This will make our map a flat surface. If you hit SETUP now, all the agents should cluster in the bottom left corner.

The map's `Point of Origin (0,0)` can be moved to any one of the corners or to the center of the VIEW.

GO PROCEDURE

`ask turtles [...]` will ask each turtle (in random order) to perform the actions specified within the square brackets.

The body of all simulations, the `go` procedure, is a loop of commands that repeat at each time step. All commands aimed at agents in NetLogo are initiated by the word `ask`, followed by the entity that is to perform the tasks. The code block that defines these tasks is enclosed in square brackets `[ ]`. In this piece of code, we'll ask all turtles (`ask turtles`) to turn right (`rt`) by a random number of degrees between 0 and a full circle (`random 360`) and go forward one step (`fd 1`). Write this code inside the `go` procedure, before `tick`, so that it looks like code block 1.5:

CODE BLOCK 1.5

```
to go
  ask turtles [
    rt random 360
    fd 1
  ]
  tick
end
```

Many procedures require input values, e.g., `fd 1` or `create-turtles 10`. Consult the NetLogo Dictionary to see what type of input is required.

The number `1` after `fd` indicates that the turtle should move forward by the length of one patch. Hit CHECK, move to the INTERFACE tab and hit GO. You can use the SPEED slider at the top of the screen to make the simulation slower so you can see the turtles' movement pattern more clearly.

In the *Y&B* model, two factors drive the wave of advance. One is the random mobility that we have just coded, and the other is population growth. Without growth, the population will not be able to cover a larger region; instead, they will become like butter scraped over too much bread. So next we will code in population growth through reproduction, and then put the two together.

INTRODUCING THE IF STATEMENT

In the previous step, the movement happened with every tick of the model. We could do the same with reproduction, but we would rather be able to control how often reproduction occurs. This is where an if statement comes in. An if statement consists of two elements: a **condition** and a **code block**. The if statement's code block is only run if the condition is fulfilled (true). If the condition is false, the code is simply not executed.

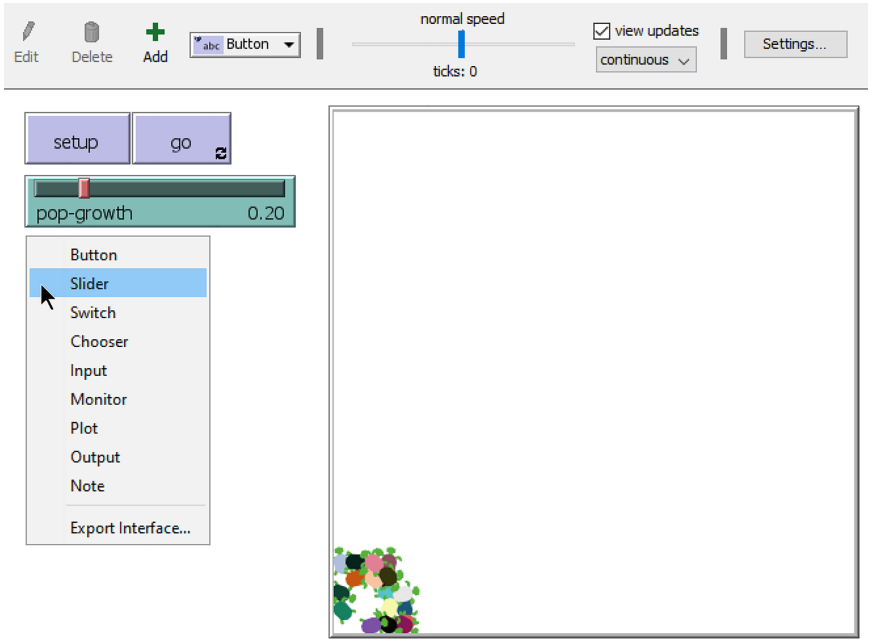


Figure 1.3. To add interface elements to your model (buttons, switches, sliders, etc.) right-click where you want to place them (within the white area) and a drop-down menu will appear. Alternatively, click the ADD button and select the element from the drop-down list, before left-clicking the desired location.

While we work on the reproduction part of the code, we will comment out the turn and move commands. We will tell NetLogo not to execute these lines of code by placing a semicolon (`;`) at the beginning of the line. Commenting out code is usually safer than just deleting it, as we may need it later. You should also use the semicolon to write short notes next to the code that explain what it does.

Comment out the first two lines after `ask turtles` by adding a semicolon at the start of each line before the movement commands. Then add the if-conditional statement, so that your procedure looks like this:

DON'T FORGET
Remember to comment out the `rt random 360 fd 1` code! The commented code will turn light gray.

CODE BLOCK 1.6

```
ask turtles [
  ;rt random 360
  ;fd 1
  if random-float 1 <= pop-growth [ reproduce ]
]
```

`random-float 1` will give out any value larger than 0 but smaller than 1, e.g., 0.54321.

The primitive `random-float` selects a random decimal number, or floating point number, up to the number you specify. This primitive is commonly used to represent probability where, for example, 0.20 indicates a 20% chance. If you now click CHECK, NetLogo will give you an error. We will solve it by making a slider in the INTERFACE tab. Switch to the INTERFACE tab, right-click anywhere on the white background, and choose SLIDER from the drop-down menu (fig. 1.3). Type `pop-growth` into the GLOBAL VARIABLE box. We want `pop-growth` to be a percentage chance of reproduction, so we set minimum to 0, maximum to 1, and increment to 0.01. Set the default value to 0.20 for now. We will experiment with it later.

TIP
"Nothing named pop-growth has been defined" indicates NetLogo cannot find anything called `pop-growth`. First check your spelling, then see if you have actually defined the procedure.

Next we switch back to the CODE tab to write our reproduction procedure. Because `reproduce` is not a NetLogo primitive, we need to define it ourselves. We will use the `hatch` primitive. A hatched turtle is a clone of its parent, and so will inherit all of its parent's characteristics, called **state variables**, like `color`, `size`, and `shape`. We will change the newly hatched agent's `color` a little at birth just for visual effect. NetLogo's TOOLS menu has a COLOR SWATCHES option, which shows how the numbers match up to the colors. Add the following code after the `end` of the `go` procedure:

Built-in commands show in color. User-defined commands are black.

state variable: attributes of a model's entities, such as age, energy, color, etc.

CODE BLOCK 1.7

```
to reproduce
  hatch 1 [
    set color color + 0.1
  ]
end
```

The repeated `color` is not a typo; it needs to be there twice. You can read it as "`set` my `color` to my current `color` plus a little brightness (`+ 0.1`)."

Check the code and move to the INTERFACE to click SETUP and then GO. It won't look like much is happening because we commented out the movement code before. Instead, the new turtles inherit their parent's `xy` position, creating turtle piles. If you want to see the turtles moving around, uncomment the code describing movement by deleting the semicolons (don't forget to comment it out again). Young and Bettinger's model actually used a population gradient to decide where new groups would

Right-click on a turtle and choose INSPECT. You'll see there are multiple turtles on each patch.

move, but our version will be a close approximation in which turtles hatch onto empty cells at birth. Update the `reproduce` procedure:

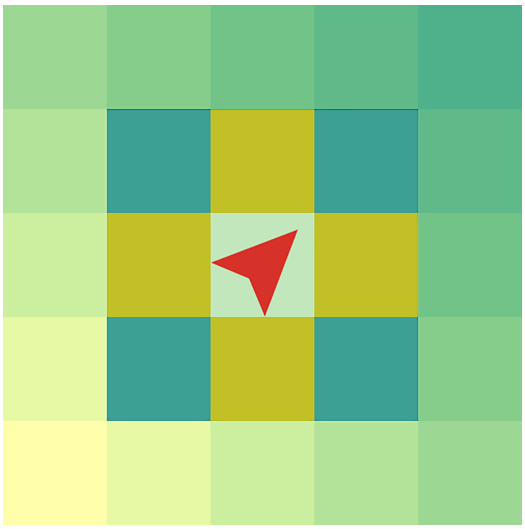
CODE BLOCK 1.8

```
to reproduce
  if any? neighbors with [count turtles-here = 0] [
    let empty-patch one-of neighbors with
      [count turtles-here = 0]
    hatch 1 [
      set color color + 0.1
      move-to empty-patch
    ]
  ]
end
```

Let's look at the code a bit more closely. You have probably already recognized that we introduced an if statement. First, it checks for an agentset, that is, if there are `any?` neighboring cells (`neighbors`) with no turtles on them (`with [count turtles-here = 0]`). It returns a value of `true` (yes, there are), or `false` (no, there are not). If `true`, it assigns (`let`) one of those cells (`one-of neighbors`) to a temporary variable called `empty-patch`, and then asks the newly hatched turtle to `move-to` that `empty-patch`. The NetLogo primitive `neighbors` refers to the eight cells that are touching the current cell (so-called **Moore neighborhood**; black and dark gray patches in fig. 1.4). There is also a `neighbors4` version that looks at the four cells in cardinal directions (so-called **von Neumann neighborhood**; only the black patches in fig. 1.4). Including the command `with` followed by a condition in square brackets further reduces the number of potential *birth cells* by only looking at those without turtles on them. Note here that because of the `move-to` primitive, the turtles always sit at the very center of a patch, while the turning (`rt`) and forward (`fd`) method we coded earlier results in turtles moving all over the patches. We will discuss the differences more in chapter 4; for now, just note that the two approaches are equally valid but produce slightly different results.

The **von Neumann neighborhood** consists of four cells and the **Moore neighborhood** eight surrounding a central cell. In GIS, these are sometimes referred to as rook and queen neighborhoods after the moves of chess pieces.

Next, go back to the INTERFACE tab and run the model a few times (i.e., use the SETUP and GO buttons). What patterns do you see?



ABMA Code Repo:
ch1_patches

Figure 1.4. Two terms for defining neighboring patches in ABM are von Neumann neighborhood (the four cardinal cells) and Moore neighborhood (the eight black and dark gray cells bordering the center). Note that the four dark gray cells are actually slightly farther away if measured from patch-center to patch-center (~1.41 map units rather than 1.00 to the black patches).

- Do all turtles contribute equally to the dispersal wave? Why or why not?
- What happens if a turtle tries to reproduce but there are no empty patches in the neighborhood?
- What happens when you change the population growth rate? Do the higher values always translate into faster dispersal? (Hint: You can compare the number of ticks before the whole world is occupied).

1.2 Theory of Simulation & Modeling

model: a simplified, often abstract, representation of a system.

Thus far, we have used the terms *model* and *simulation* almost interchangeably. However, they are not the same thing. A **model** is a simplified representation of a real system. A toy plane is a model, a map is a model, and a Paleolithic handaxe made by a modern experimenter is a model. All of these things aim to preserve some, but not all, aspects of the real thing. For example, the toy airplane aims to look like a real plane but is much smaller, does not have a real engine with fuel inside, and probably cannot actually fly. This does not prevent the end user (e.g., a child) from considering it an aircraft. This is the very nature of models: they are shorthands for the real

thing in which only certain aspects of reality are represented. There are no hard-and-fast rules about which aspects should be included in the model. Rather, the choice is driven by the particular needs of its user. A child will want their plane to have painted windows; an adult model plane enthusiast may insist on accurate side markings; and an engineer optimizing the wing will not give the color a passing thought.

Let's look again at the model we just built. We have ignored the vast majority of human behavior. Although we do not doubt that people in the past hunted prey, chatted around a fire, and taught their children how to knap stone tools, none of these behaviors has been included in the model. Instead, their lives have been simplified to just two aspects: reproduction and movement. Why? Because the original research question was whether the basic demographic rules governing any living population could have resulted in the observed data pattern—the dispersal of hominins across the globe—so we have modeled just these two elements.

simulation: a dynamic model of a system, that is, a model that includes dynamics which change the state of the model over time.

A **simulation** is a specific type of model with an extra dimension of time. We use simulations to observe the changes and the evolution of the model as well as its final outcome. Thus, the toy airplane is not a simulation, but one can imagine using an upscale version of it in a wind tunnel to simulate the drag generated under different wind conditions. Why do we use models and simulations? The three main functions of simulation are **hypothesis testing and prediction**, **theory building**, and **data exploration** (Premo 2010), although most models fall somewhere in between these three objectives.

Models are often built for inaccessible systems that cannot be observed or experimented on directly.

First, simulation is often used in a **hypothesis-testing** capacity. In many cases, observing the real system is impossible, a problem very familiar to archaeologists. However, we are not the only ones who do not have direct access to the system we study. Long-term societal change, the evolution of species, microscopic interactions occurring in human cells, black holes, or the evacuation of a building during fire emergency are all processes that are difficult to observe directly or experiment on. In these cases, models represent our hypotheses about the system that may or may not be correct, and thus require testing. Instead of experimenting on a real system to test such hypotheses, we construct digital, computational, models.

Second, simulation supports **theory building** when interactions between the system elements are not fully understood. For example, although the stock exchange consists of agents who follow a simple be-

havioral rule (i.e., "buy low, sell high"), we cannot easily predict financial crashes and fluctuations, revealing our ignorance about some elements of the system. In these cases, we build simulations to better understand the system itself and its underlying dynamics. This process is known as theory building because it establishes the logic of a hypothesis rather than its relevance to the real world.

Finally, in **data exploration** the goal is to uncover the underlying processes behind patterns identified in data. Data analysis gives us tools to establish co-occurrence or the existence of a relationship between different variables. However, statistical and analytical techniques are neither designed nor able to establish causality or to determine what mechanism connects different phenomena. You might have heard the statisticians' mantra "correlation is not causation." Simulation acts as a virtual lab to reveal causation, that is, processes that lead to particular data patterns. You get correlation from analyzing data, while causation can only be inferred from modeling.

Simulation is a principal tool for evaluating causal relationships.

You may have already identified that the *Y&B* model falls somewhere between the hypothesis-testing and the theory-building functions of simulation; it tests a particular theory about the fundamental processes driving the system in question.

Young and Bettinger's simulation is a **model from first principles**, where *first principle* describes the simplest and most plausible of assumptions, also referred to as the **first-order approximation** (meaning at the most basic level of description). The model does not *prove* that the observed empirical pattern (archaeological data) is a result of the modeled process; rather, it confirms its plausibility and general agreement with the data, demonstrating that the modeled process *could have* produced a data pattern similar to the observed one. It does not exclude the possibility that other processes would prove to be equally or even more successful in replicating the data pattern. This is known as the **equifinality** problem common to a lot of archaeological inference. To assess the plausibility of other scenarios, respective models need to be built and their results compared to the one that currently best matches the data. The *Y&B* model has the advantage of being exceedingly simple, a characteristic that is highly valued in terms of explanation, even if it is not necessarily decisive.

Models from first principles simulate how the world would look under the most basic of assumptions.

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."

The beauty of formal models such as this one is that it is easy to build upon them, thus continuously gaining new insights. If, as is usually the case, the answers you have uncovered have only spurred more questions, you can extend and modify the model to address them. This iterative process of model building, testing, and rejecting forms the cornerstone of all scientific inquiry in archaeology, as in any other branch of science. But we need not stop there. In the next section we will finally look at *Y&B*'s hypothesis that environmental zones shaped the out-of-Africa dispersal.

1.3 Modeling a Landscape

We advanced far with Young and Bettinger's model, but our implementation is missing an important part: the environment. The distribution of habitats or environmental zones (biomes) is not just about giving a visual background to our dispersing population; there are also some significant barriers to dispersal, such as oceans, lakes, and ice sheets, for which we should account. Dispersing populations will need to go around these areas and funnel through some geographic bottlenecks.

First, we need data showing the world as it was during the time period, that is, with sea level 85 meters lower than today. The map we are using was derived from a global elevation and bathymetry dataset (Becker et al. 2009; Smith and Sandwell 1997), which you can download from our model repository.[5] Once you download the file, make sure you place the map in the same folder where you are saving your NetLogo model (*Y&B_dispersal*); otherwise, you will get an error that NetLogo cannot find the file.

DON'T FORGET
The map must be in the same folder as the model for NetLogo to find it. Otherwise you have to write the full or relative file path.

Once we load a map into our NetLogo model, we must achieve the following:

1. We want the map image to be imported as actual data, not just as a background image.
2. We need to prevent agents from moving into uninhabitable patches, such as water.

[5] https://github.com/SantaFeInstitute/ABMA/tree/master/ch1.1/

NetLogo can handle both raster and vector data using the GIS extension (see ch. 9).

If you have ever worked with geographic information systems (GIS), you already have some experience with gridded raster data. If not, all you need to understand here is that the map is a grid of cells, and that we can assign each cell to a `patch` by taking a number corresponding to its color value. We need to import our map into NetLogo so that numeric values can be assigned to each `patch`'s `pcolor` variable. NetLogo has a command to do this easily. Within the `setup` procedure, before we create any turtles but after the `clear-all` command, add the following line so that it looks like this:

CODE BLOCK 1.9

```
to setup
  clear-all
  import-pcolors "ch1_map.png"
  ...
```

We use ellipses (`...`) in code blocks to show that this is added to existing code. Don't type these ellipses into your code.

Click SETUP on the INTERFACE tab. We can see the map now, but it doesn't fit our VIEW panel very well. The map image we imported is 600 × 351 pixels, and we would like each pixel to be assigned to one patch. Click the SETTINGS button and set the `max-pxcor` to 599 and the `max-pycor` to 350 (fig. 1.5). Why not 600 and 351? NetLogo space, like all Cartesian spaces, starts at a point (0, 0). Thus, the first column number is 0, the second is 1, etc. Whether it is the primitive `random` or the screen dimensions, NetLogo (as well as our book) always starts counting from 0.

Check the file properties
Windows: right-click > Properties > Details
Mac: right-click > Get Info > More Info > Dimensions

Since the earth is a globe, also check the WORLD WRAPS HORIZONTALLY box (but not WORLD WRAPS VERTICALLY—this would turn our world into a torus-ring doughnut shape). Check the text under the black square in the MODEL SETTINGS panel to verify that everything is correct. It should state "Vertical Cylinder: 600 × 351." Next, click OK. If your VIEW panel is far too large now, reopen MODEL SETTINGS and set the PATCH SIZE to a smaller value.

The PATCH SIZE setting regulates how the VIEW panel looks on your screen; it does not change the number of patches.

TIP

In NetLogo colors are represented by numbers from 0 to 140, but you can also refer to them using keywords like `white`, `green`, `sky`, or `violet`.

Click SETUP once more and verify that the map fits well. Run the model again, and you'll notice a couple of things: first, it takes a lot longer to finish the run, because there are a lot more agents and patches now, and second, the agents are completely ignoring our map.

Y&B's dispersal model keeps things simple by coding patches as either 1 for land or 0 for water or ice sheets. All we need to know is the

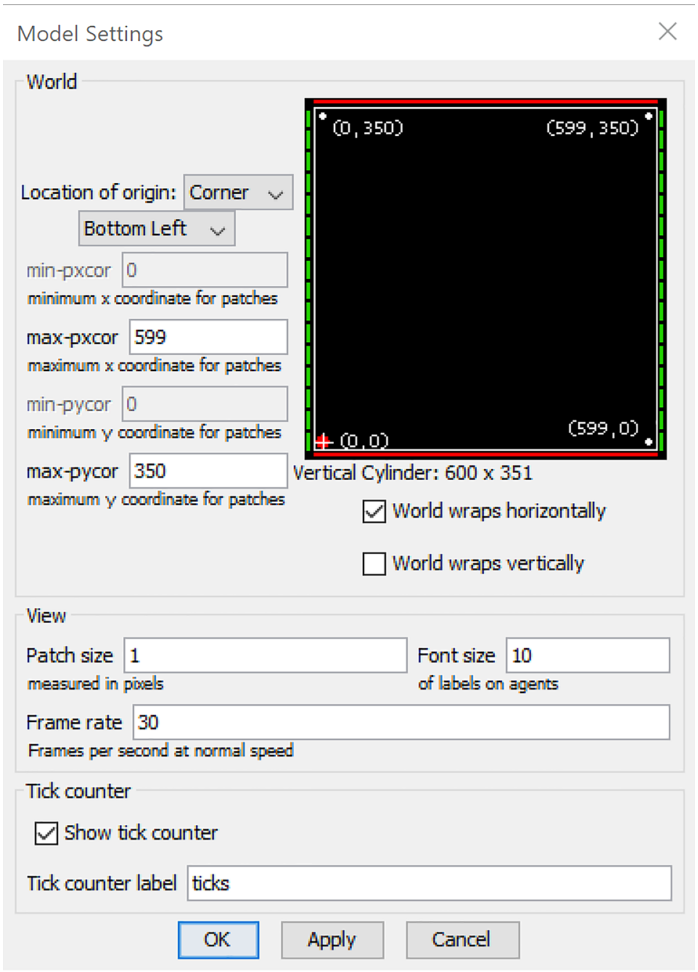


Figure 1.5. The MODEL SETTINGS box controls the size of the patches, position of the grid origin (one of the corners, center, etc.), and edge-wrapping options.

`pcolor` assigned to water on the imported map. Right-click on any water patch and choose INSPECT PATCH X Y from the drop-down menu, where x and y correspond to the coordinates of the water patch you clicked on. An INSPECTION panel will come up that provides variable values of that patch, as well as a close-up view of the patch. In our case, the `pcolor` value is 9.9, corresponding to `white`. We will use that value to tell turtles where not to go.

We have two things left to do: move the turtles' start location to eastern Africa, and then tell the turtles that they can only move on land. In the VIEW panel, right-click on any cell in East Africa and note

DON'T FORGET

Parentheses are often not necessary, but they help to keep the code readable—an important consideration for future readers of your code (including yourself six months from now).

its x, y coordinates. In `setup`, change the x, y coordinates inside the `create-turtles` command to (`your_X_coordinate + random 5`) like this:

CODE BLOCK 1.10

```
setxy ( 360 + random 5 ) ( 170 + random 5 )
```

Next, go to the `reproduce` procedure and change the following lines to further limit the set of patches the turtles are evaluating for their offspring to only include land patches:

CODE BLOCK 1.11

```
to reproduce
  if any? neighbors4 with
      [ count turtles-here = 0 and pcolor != white ] [
  let empty-patch one-of neighbors4 with [
    count turtles-here = 0 and pcolor != white ]
  ...
```

The `!=` means "does not equal," which tells the agents *not* to consider any `white` cells for their offspring. The `and`, known as a logical operator in computer programming, is a way of combining criteria within the condition where both have to be true. To prevent agents from jumping diagonally across bodies of water, we will also switch to the `neighbors4` primitive, which means agents only move in cardinal directions. Check your code, then switch to the INTERFACE tab and run the simulation a few times (fig. 1.6). What do you notice about the dispersal wave as it passes through geographic bottlenecks like the Sinai Peninsula or south of the Black Sea? What pattern do you notice in the agent colors, and what could this represent?

DON'T FORGET
Change `neighbors` to `neighbors4` to avoid unintended moves over diagonal boundaries.

Note that we wrote the `reproduce` procedure to be agent-specific, thus only turtles can perform the procedure. We know this because it contains the turtle-specific primitive `hatch`. At each time step, each turtle (in random order) assesses their own situation (e.g., `if any? neighbors4`) and performs their actions accordingly. If you stop to think about it, though, who is doing the asking? NetLogo refers to this as the **observer**. Many commands can only be called by the observer, such as `clear-all`, `tick`, and our `setup` and `go` procedures. We'll

If you open `hatch` in the NetLogo Dictionary, you'll spot a turtle icon.

observer: an *agent* that builds and controls all other simulation objects.

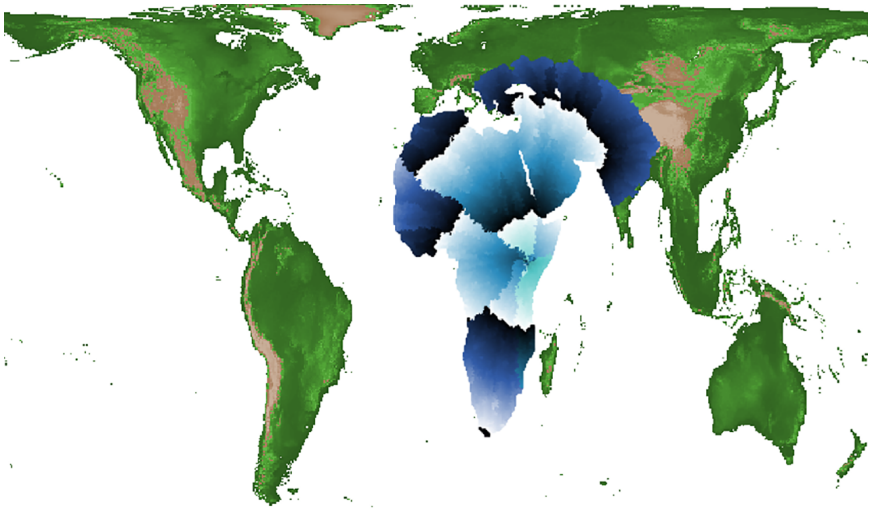


Figure 1.6. The population of agents disperses from eastern Africa and begins to spill out into Eurasia. Note that the agents are small enough that they almost look like a continuous surface.

discuss how to control which variables belong to agents, patches, or the observer in the next chapter.

Finally, reflect on how you wrote your code. Just as you put thought and effort into creating a clear and easy-to-follow methods section in a scientific paper, you must also create code that can easily be followed by an outsider. NetLogo doesn't care about white spaces, so you could write this whole model in one very long line of code. This is a bad idea, though. Like the methods section of your paper, code tends to get more and more complicated as you progress, so keeping it well laid out will spare you a lot of frustration. We have not included comments in the code blocks here, but if you check the model's code in the ABMA Code Repo you'll see inline documentation.

1.4 Summary

The simulation we built in this chapter is a model of human dispersal. It is a model because it is a simplified representation of the behavior of past human groups including their mobility and reproduction. Many possible factors influencing dispersal were deliberately left out of our model because we were using it to test the hypothesis that the pattern and speed of human dispersal across the world was primarily driven by population

growth and undirected movement. To test this hypothesis, we must parameterize the model by assigning reasonable values to our population growth rate and movement distance relative to the time step. You can consult Young and Bettinger's paper for those values. If the simulated population arrives in Israel, Italy, China, Indonesia, Alaska, Chile, and other locations around the date of the known first occupation of those regions, then we will have demonstrated the plausibility of the hypothesis.

Our simulation results may also have implications for theory building concerning early human dispersal. If the model exposes particular patterns in the spreading dispersal wave that disagree with archaeological data, this spurs the question of what factors might have been in play. For example, your agents might arrive in France before China, despite ample evidence that the contrary occurred in the Pleistocene. This could be because of differences in population growth rates or carrying capacity, as *Y&B* suggested, or it could be due to competition from Neanderthals. These and other options can be tested and compared against each other by extending the simulation further. These kinds of considerations can also nudge the researcher to return to the archaeological record and explore its possible biases and gaps. Do some geographical areas show particularly poor fit to the simulation? Have they been extensively studied, or is it likely that traces of the dispersal are underreported there?

"All models are wrong, but some are useful."
– G. Box (1979)

Models are not static end products in the research process but part of the cumulative understanding we are building about the past at both theoretical and empirical levels (Hesse 1978; Dunnell 1982; Neiman 1995). Models will not prove your hypothesis to be "true" with perfect certainty. A common saying in modeling, attributed to statistician George Box (1979), is that "all models are wrong, but some are useful." In other words, while you're unlikely to replicate the exact timings of dispersal across the world, the model will provide insight into what is and is not a likely factor driving dispersal. Even better if a model, built according to a common narrative explanation, is completely wrong and we must change our own understanding.

Finally, it is worth reflecting on the technical side of things. If, prior to reading this chapter, you had never seen a line of code and the whole programming enterprise felt daunting, take a moment to contemplate the

fact that, after only a couple hours of basic training, you were able to replicate a published scientific study. It truly is not rocket science (even if rocket scientists build models, too).

End-of-Chapter Exercises

1. Change the other basic agent characteristics: `size` and `shape`.

2. Add a second population of turtles spreading from eastern Asia. Give each population its own fixed `color`.

3. Add an `initial-pop-size` slider. Modify the `setup` code to use this to vary the size of the original population. Also adjust the code so that these initial agents must be created on empty patches only.

4. First, add more population dynamics by adding age and death after a certain age. Second, add probability of death (which is similar to reproduction-probability). Define this probability as a function of age, so that the older a turtle gets, the higher its probability of dying.

5. *Y&B* use a variable *carrying capacity* to affect the dispersal characteristics. First, give all patches a variable called `carrying-capacity` (check `patches-own` in the NetLogo Dictionary). For land patches, set it to 1, and for water patches set it to 0. Modify the reproduction procedure to only allow turtles to hatch onto patches where `carrying-capacity` is greater than the count of turtles on that patch. Second, follow *Y&B* in setting `carrying-capacity` higher for patches below 30° of latitude, which corresponds on our map to `pycor 307`. Does this make any difference to the dispersal rate?

Further Reading

- A. J. Ammerman and L. L. Cavalli-Sforza. 1973. "A Population Model for the Diffusion of Early Farming in Europe." In *The Explanation of Culture Change: Models in Prehistory,* edited by C. Renfrew, 343–358. London, UK: Duckworth.

- J. Fort. 2018. "The Neolithic Transition: Diffusion of People or Diffusion of Culture?" In *Diffusive Spreading in Nature, Technology and*

Society, 313–331. Cham, Switzerland: Springer. doi:10.1007/978-3-319-67798-9_16

- S. J. Mithen and M. Reed. 2002. "Stepping Out: A Computer Simulation of Hominid Dispersal from Africa." *Journal of Human Evolution* 43 (4): 433–462. doi:10.1006/jhev.2002.0584

- R. L. Bettinger and D. A. Young. 2004. "Hunter-Gatherer Population Expansion in North Asia and the New World." In *Entering America: Northeast Asia and Beringia Before the Last Glacial Maximum,* edited by D. B. Madsen. Salt Lake City, UT: University of Utah Press, September.

- P. Riris. 2018. "Assessing the Impact and Legacy of Swidden Farming in Neotropical Interfluvial Environments through Exploratory Modelling of Post-Contact Piaroa Land Use (Upper Orinoco, Venezuela)." *The Holocene* 28, no. 6 (June): 945–954. doi:10.1177/0959683617752857

- I. Romanowska. 2015b. "So You Think You Can Model? A Guide to Building and Evaluating Archaeological Simulation Models of Dispersals." *Human Biology* 87 (3): 169–192. doi:10.13110/humanbiology.87.3.0169

- J. Steele. 2009. "Human Dispersals: Mathematical Models and the Archaeological Record." *Human Biology* 81 (2-3): 121–140. doi:10.3378/027.081.0302.

- U. Wilensky and W. Rand. 2015. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo.* Cambridge, MA: MIT Press, April.

- D. A. Young and R. L. Bettinger. 1992. "The Numic Spread: A Computer Simulation." *American Antiquity* 57 (1): 85–99. doi:10.2307/2694836.

- D. A. Young and R. L. Bettinger. 1995. "Simulating the Global Human Expansion in the Late Pleistocene." *Journal of Archaeological Science* 22 (1): 89–92. doi:10.1016/S0305-4403(95)80165-0

NOTES