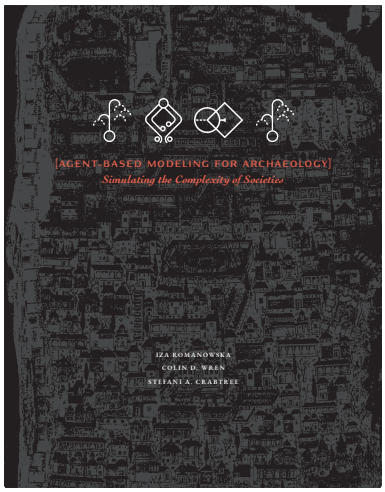
PLEASE NOTE:

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

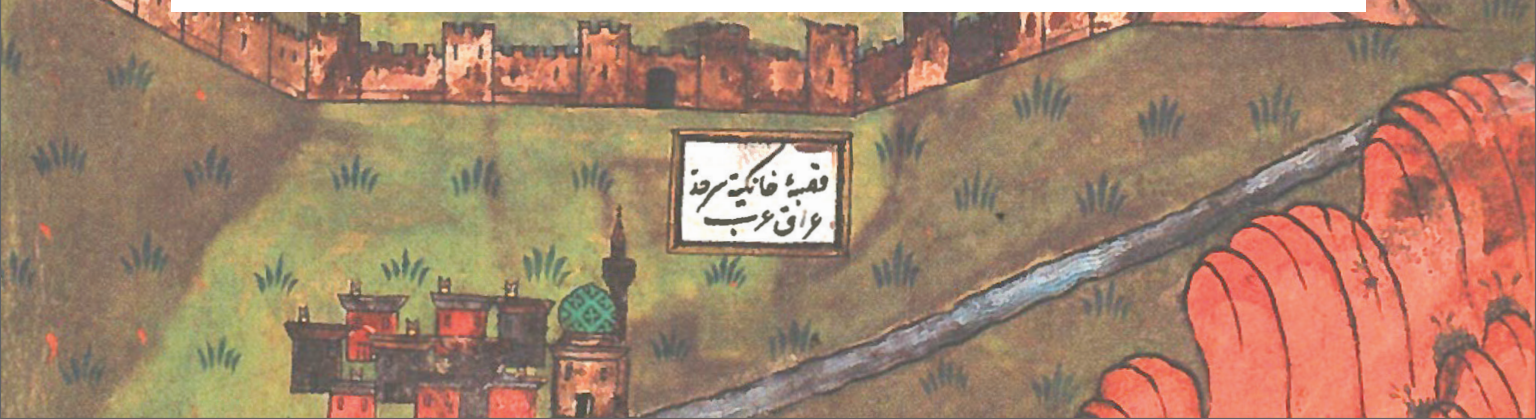


REGARDING COLOR:

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




شهر همدان

MODELING WITH SPATIAL DATA: BRINGING THE GIS WORLD TO ABM

7.0 Introduction

There are many intuitive reasons for making a model spatially explicit. Human activity occurs within space and landscapes, and more often than not the characteristics of those landscapes will impact their activities (see landscape archaeology: David and Thomas 2016; Rossignol and Wandsnider 1992; Verhagen 2012).

In his first law of geography, Tobler (1970, p. 236) states that "everything is related to everything else, but near things are more related than distant things." This basic principle, also known as spatial autocorrelation, generally applies to human behavior as well. People who are closer to each other are more likely to interact and, as a result, share characteristics (Ramachandran et al. 2005). Until the dawn of modern communication and transportation technologies, these interactions usually happened across small distances—in the local environment of individuals.

Space and the cost of traversing it are fundamental limiting factors to human activity. Physical geographic systems, whether in natural or constructed environments (e.g., cities), provide opportunities as well as constrain the movement and interactions of people within them (Banks et al. 2008). Accounting for space is often necessary if we want to understand how certain micro-level processes lead to large-scale patterns; for example, traffic jams can develop from individuals acting to optimize their own travel (Raney et al. 2003).

Similarly, while social dynamics usually play a central role in archaeological models (Binford 1962), the dynamics of human–environment interaction are commonly invoked to explain mechanisms governing past societies. Indeed, the entire field of *socionatural* or *natural–human systems* examines the impact of these types of dynamics. These cultural adaptations,

OVERVIEW

- ▷ Types of spatial data
- ▷ Handling GIS data in NetLogo
- ▷ Modeling spatial interactions
- ▷ Finding GIS data
- ▷ Artificial landscapes
- ▷ Code optimization with the Profiler extension

in turn, can have dramatic impacts on local ecosystems through the consumption of resources, burning, seed dispersal, and other ways in which human groups take advantage of various ecosystem services (Bird and Nimmo 2018; Crabtree, Bird, and Bird 2019), such as through land clearance for agriculture, horticulture, or other activities (Crabtree, Vaughn, and Crabtree 2017).

However, this does not necessarily mean that all models need to be, or should be, placed within an as-true-as-possible representation of a landscape. A thoughtful list of relevant factors should be selected and worked into the model to ensure that the final output is legible and useful. We have already come across the concept of **parsimony**, paring down the available cornucopia of information until only the bits that are most essential remain (see ch. 5). This rule is fundamental for building models, including the spatial representation in our simulation. As with all models, it is vital to understand the reference system you are studying in order to know how to incorporate spatial constraints adequately and whether to add realistic geographic information or limit the model to an abstract spatial representation.

The latest GIS in ABM textbook is Crooks, Heppenstall, and Malleson (2018).

It is, however, a good rule of thumb that realistic geographic information should only be added in the later phases of model building, and only after the model dynamics are very well understood. Otherwise, irregular behavior or interactions between variables can get lost in the heterogeneity that is introduced by realistic spatial data. Instead of jumping in with a detailed topography dataset from day one, it is a good practice to simplify its main characteristics and use that simplified representation to develop agents' dynamics. Perhaps the landscape can be represented as, for example, a strip of water to the north, a middle strip of agricultural land, and a strip of mountains in the south (see *AmphorABM* in ch. 6 or *SugarScape* in ch. 3). This will allow you to understand, for example, what happens on the "boundary" between regions and control for potential bugs in the code—something that would be much more difficult to notice with rugged and irregular spatial distributions that characterize real landscapes.

Creating an abstract landscape may be possible using NetLogo code (see O'Sullivan and Perry 2013) or may require a separate GIS package.

Equally, creating artificial landscapes where certain features are controlled for can be a much cleaner approach than uploading raster maps of real geographies. Imagine you are interested in how patchiness of resources impacts a group's foraging strategy. Simulating landscapes with

precise characteristics (e.g., 10 clusters but with a different degrees of clustering) is a powerful method in experiment design.

All of these questions—of how the micro-level processes of human–environment interaction can lead to macro-level signatures across space and time—link strongly with the theories of complex systems discussed in the introduction, and after working through these exercises you will see how they fold neatly into the agent-based modeling paradigm. Further, you will see how the coupling of geographic information science (GIS) and ABM can provide a powerful framework for answering new questions.

The "S" in the acronym GIS can stand for both "Systems," referring to the software, and "Science," referring to the broader approach to geographic data.

Throughout this chapter we will discuss a number of examples where an explicit modeling of space is particularly useful. These are divided into four categories on the basis of what is affecting what (table 7.0). We will use a simple model of a forager moving around the landscape picking up stone raw material to make tools as our GIS landscape-based model. It is a replication of *Neutral Procurement*, by Brantingham (2003). Davies et al. (2019) coded the replication and published it along with a step-by-step tutorial, which we follow and expand on here. We will also discuss the topic of code profiling and optimization, the first step toward the final stages of model development: experiment design and running (ch. 9).

Model: *Neutral Procurement* by Brantingham

Table 7.0. Matrix of possible interactions between agents and patches that are modeled in explicit space.

Actor / affected	Agent affected	Patch affected
Agent action	Agent interacts with another nearby agent (e.g., trade, reproduction, cooperation)	Agent changes patch characteristics (e.g., extracts resources, harvests, builds, or drops an artifact)
Patch "action"	Patch is a constraining or motivating factor of agent movement (e.g., resource distribution, natural barriers)	Neighboring patches interact (e.g., erosion moves sediment, something "spills" from a patch, patch variables combine into new useful index)

7.1 Bringing GIS Data into NetLogo

In chapter 1, we imported images to use as maps by converting their color into patch variables that agents can sense and react to. That was a simple yet effective way of dealing with spatial data. However, anyone with some spatial data experience likely recognizes its limitations. For example, we

avoided using a coordinate reference system, leading to a somewhat inaccurate representation of distance as well as distortion of the shapes of landmasses; further, it may have been difficult to pinpoint an exact location. In this section, we will improve upon the representation of spatial data in NetLogo models, and then learn to manipulate that spatial data in the next section. As the purpose of this book is to teach you to write agent-based models, we only cover the basics of geographic information science theory here; a dedicated GIS text will provide a more thorough background and in-depth discussion of GIS theory and methodology.

We also created artificial landscapes by leveraging geometrical operations to get well-defined spatial structures, such as hills or barriers (ch. 4).

We recommend Conolly and Lake (2006) and Gillings, Hacigüzeller, and Lock (2020) for archaeology-specific guides to GIS.

NetLogo's GIS capabilities are found within the GIS extension, which we can enable by adding at the top of the model code:

CODE BLOCK 7.0

```
extensions [gis]
```

Some extensions, like GIS and Rnd, are included with NetLogo. Others must be downloaded separately using the EXTENSION MANAGER in the TOOLS menu.

The two basic types of GIS data structures are raster and vector. Raster data is gridded spatial information that is roughly equivalent to the patches in NetLogo. Vector data is stored as lists of coordinates that define points, lines (i.e., a series of connected points), or polygons (i.e., a series of points connected to form a shape). NetLogo handles each type in a particular way.

RASTER DATA

NetLogo can only read the ASCII raster data format, so you may have to use GIS software, such as QGIS, to transform your data if it comes in any other format. The .asc file starts with a small preamble that defines the corner (here in decimal degrees), shows the number of columns and rows of data, the cell size, and what value represents no-data (fig. 7.0). If your data is missing any of this information, NetLogo will not read the file.

ASCII files can be viewed and edited in any text editor, (e.g., Notepad). If the file is missing any of the necessary information, simply type it in.

We can import a raster file, for example, containing elevation values of our landscape, `dem.asc`, by first creating a global variable to hold the data `elevation-dataset`, and then loading the file into it.[1]

[1] You can find all code and example datasets from this chapter in the ABMA Code Repo: https://github.com/SantaFeInstitute/ABMA/tree/master/ch7.

```
ncols         4320
nrows         4320
xllcorner     -118.650000000000
yllcorner     33.199999999998
cellsize      0.000092592593
NODATA_value  -9999
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 7.0. Format of an example raster data file formatted as an ASCII (.asc). Here the rows of 0s represent cells at sea level as this corner of the map is ocean.

```
globals [elevation-dataset]

to setup
  set elevation-dataset gis:load-dataset "dem.asc"
end
```

CODE BLOCK 7.1

ABMA **Code Repo:** *ch7_gis*

NetLogo now recognizes the raster data file but does not yet know how to translate it to a patch variable. First, we have to establish the coordinate reference system of our data, that is, its projection. This information has to be imported in WKT format (see table 7.1).

Table 7.1. Projections that are available in NetLogo, and a corresponding example of correct and incorrect projection formats.

Projections known to NetLogo	Example correct WKT .prj file
GEOGCS (unprojected) lbers_Conic_Equal_Area Lambert_Conformal_Conic_2SP Polyconic Lambert_Azimuthal_Equal_Area Mercator_1SP Robinson Azimuthal_Equidistant Miller Stereographic Cylindrical_Equal_Area Oblique_Mercator Transverse_Mercator Equidistant_Conic hotine_oblique_mercator Gnomonic Orthographic	GEOGCS["GCS_North_American_1983", DATUM["D_North_American_1983", SPHEROID["GRS_1980", 6378137,298.257222101]], PRIMEM["Greenwich",0], UNIT["Degree",0.017453292519943295]] **Incorrect format for NetLogo** ArcMap exports .prj files in an incompatible format like this: Projection GEOGRAPHIC Datum NAD83 Spheroid GRS80 Units DD Zunits NO Parameters

The best way to do this is to have NetLogo import a properly formatted .prj file to set the projection.[2] Add the following line before loading your data:

CODE BLOCK 7.2

```
gis:load-coordinate-system "dem.prj"
```

IMPORTING PATCH LANDSCAPES FROM RASTER DATA

In chapter 1, we manually adjusted the dimensions of the world to match the imported data. You should do that here as well by opening the dem.asc file in your text editor of choice to check its `ncols` and `nrows`,[3] then adjusting NetLogo's MODEL SETTINGS to match. However, if your raster's dimensions are too fine or too coarse using the GIS extension, we can also automatically stretch our raster data (fig 7.1) (e.g., if it was too fine a spatial resolution for our purposes) to our chosen patch landscape dimensions by setting the world envelope to the size of our raster dataset:

This is similar to GRASS GIS's approach of setting the spatial extent and cell size with `g.region`.

CODE BLOCK 7.3

```
gis:set-world-envelope gis:envelope-of
  elevation-dataset
```

Next, we want to copy the raster values of our elevation data over to a patch variable. Create a patch variable `patch-elevation`, then add the following to the `setup` code to (1) apply the raster data to the patch variable and (2) adjust `pcolor` according to their elevation value:

CODE BLOCK 7.4

```
patches-own [patch-elevation]

to setup
  ...
  gis:apply-raster elevation-dataset patch-elevation
```

You'll get a warning `GIS Extension Warning: datasets previously loaded...` if you click on `setup`. Just ignore it.

[2] The website spatialreference.org has WKT .prj files for every kind of projection, or see the following link on how to make ArcMap create a compatible .prj file: https://support.esri.com/en/technical-article/000012439.

[3] Notepad++ (for Windows) is a good open-source text editor that supports some code highlighting: https://notepad-plus-plus.org

CODE BLOCK 7.4 (cont.)

```
  let mx gis:maximum-of elevation-dataset
  ask patches [
    set patch-elevation ( gis:raster-sample
      elevation-dataset self )
    ifelse patch-elevation > 0 [
      set pcolor scale-color green patch-elevation 0 mx
    ][
      set pcolor blue
    ]
  ]
end
```

Repeat the code lines to load multiple rasters into different patch variables. Keep in mind that NetLogo's memory requirements will increase rapidly.

Rather than setting all patches to the same color, `scale-color` uses `patch-elevation` to read a `pcolor` from a gradient between 0 and the maximum elevation stored in the temporary variable `mx`.

DON'T FORGET
Add `patch-elevation` and later `patch-quarry` to `patches-own`.

If there was more than one raster dataset of different sizes and we wanted an envelope big enough to contain them both, we could use:

CODE BLOCK 7.5

```
gis:set-world-envelope gis:envelope-union-of
  gis:envelope-of elevation-dataset
  gis:envelope-of vegetation-dataset
```

However, this can create problems if the two raster datasets do not have the same cell resolution, if their grids do not line up perfectly, or if the spatial resolution of your raster dataset is too fine or too coarse for your chosen ABM spatial scale. In all these cases, you should choose your world size first, then the imported data can be resampled during import (fig. 7.1) using a resampling method set by `gis:set-sampling-method` before `gis:apply-raster`. For better-looking results, use `"nearest_neighbor"` for categorical data and `"bilinear"` for continuous data.

Bilinear will average several adjacent cells, whereas nearest neighbor will just take the closest cell value.

CODE BLOCK 7.6

```
gis:set-sampling-method vegetation-dataset
  "NEAREST_NEIGHBOR"
gis:set-sampling-method elevation-dataset "BILINEAR"
```

Skip this example code if you're only adding the elevation raster.

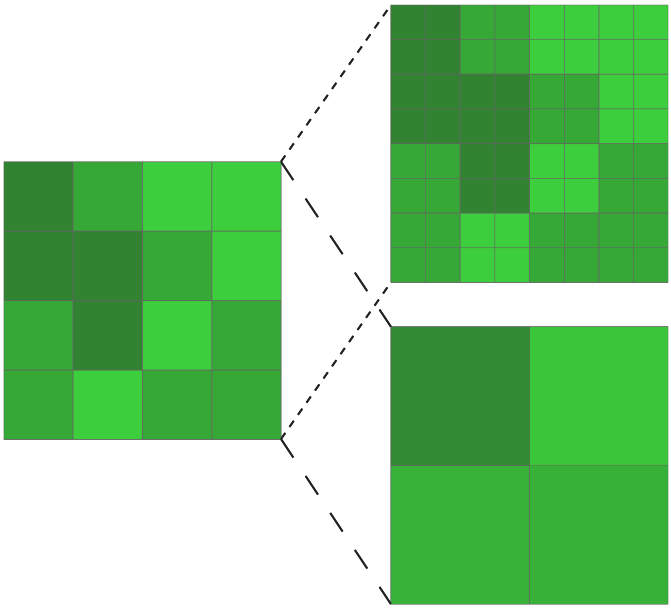


Figure 7.1. An example raster (left) could either be resampled to a finer-resolution landscape (top right, small dashes) or to a coarser-resolution patch landscape (bottom right, large dashes).

It is likely that NetLogo will run out of memory if you try to load a really large raster landscape into your model. In this case, you have two choices:

By default, NetLogo uses `1024m` or 1 gigabyte of memory. This is rarely enough for large or multiple rasters.

1. Resample the data to a coarser grid size as we just described (fig. 7.1); or
2. Increase the amount of memory your computer is allocating to NetLogo. This requires modifying the NetLogo.cfg file stored in the NetLogo program folder.[4]

IMPORTING PLACES WITH VECTOR DATA

The second basic type of spatial information is vector data. NetLogo will only read a vector shapefile format, so again you may have to use a GIS program to change formats first. Vector data is read into NetLogo similarly to raster data, but you have the option of turning those data into either

[4] See https://ccl.northwestern.edu/netlogo/docs/faq.html, and scroll to the question, "How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?"

agents or patch variables. The easiest method is to create a patch variable that records the presence or absence of a vector feature.

CODE BLOCK 7.7

```
globals [elevation-dataset quarries-dataset]
patches-own [patch-elevation patch-quarry]

to setup
  ...
  set quarries-dataset gis:load-dataset "quarries.shp"
  ask patches [
    if gis:intersects? quarries-dataset self [
      set patch-quarry true
      set pcolor red
    ]
  ]
end
```

This method is quite simple as `gis:intersects?` only asks if something in the quarries dataset, which is a `VectorDataset` type, is in the same spatial position as that patch. It works for any type of vector feature—points, lines, or polygons. However, if two quarries (points) fell on the same patch, the code would only mark that patch once. To correct this, we will use a second method that works directly with the vector data structure. To make this work, we have to read through the data, which are stored in a fairly complicated set of nested lists in the vector file, to find the actual coordinates of the points' locations (fig. 7.2).

See the GIS Extension manual, VectorDataset Primitives section, for other topological primitives.

The first step is still to use `gis:load-dataset` as before, but the process diverges after that. Table 7.2 shows the structure of the shapefile. Notice that the coordinates are buried deep within the `VectorFeature` (fig. 7.2).

See Chapter 7 in the ABMA Code Repo for step-by-step details on how to manually peel off the VectorFeature data structure's layers.

Table 7.2. A reconstituted spatial and attribute table of the quarries shapefile.

	Coordinate table (VectorFeature)		Attribute table	
Point #	X	Y	"ID"	"Name"
0	11.9	-6.5	"0.0"	"Cero"
1	21.1	-18.1	"1.0"	"Uno"

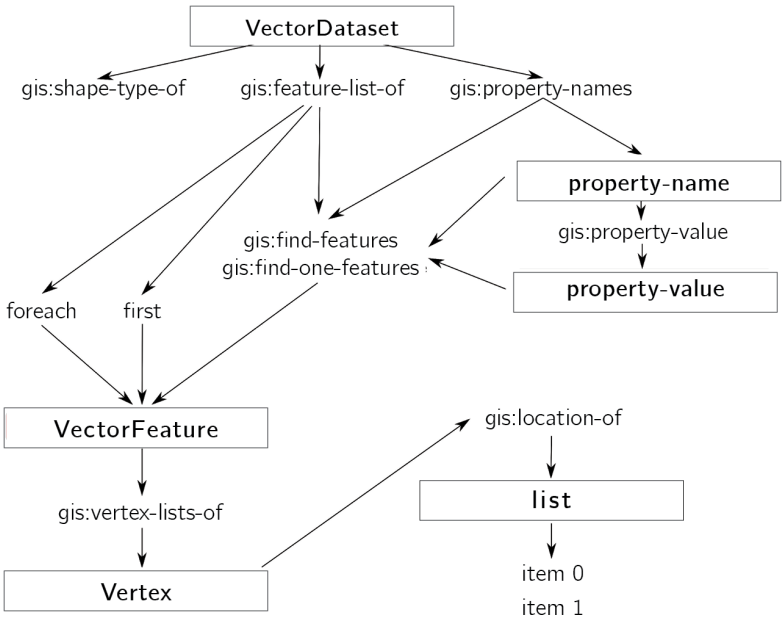


Figure 7.2. A visualized workflow for working with the VectorDataset's data structure. Boxes indicate the data types and other text represents NetLogo primitives. Arrows indicate that the data type or output is a parameter of that primitive.

Type in the console:

CODE BLOCK 7.8

```
gis:location-of (first first gis:vertex-lists-of (first
gis:find-features quarries-dataset "ID" "10.0") )
observer: [-57.70706490103701 32.12431606519094]
```

To turn this into some useful code to actually load at `setup`, we first create a new breed of agents called quarries as well as some quarry variables to hold the attribute data, and then put all of the steps above inside of a `foreach` loop.

CODE BLOCK 7.9

```
breed [ quarries quarry ]
quarries-own [ id name ]
```

CODE BLOCK 7.9 (cont.)

```
to setup
  ...
  let points gis:feature-list-of quarries-dataset
  foreach points [ quarry-point ->
    let location gis:location-of (first first
      gis:vertex-lists-of quarry-point)
    let temp_id gis:property-value quarry-point "ID"
    let temp_name gis:property-value quarry-point
    "Name"
    create-quarries 1 [
      setxy item 0 location item 1 location
      set size 2
      set shape "circle"
      set color red
      set id temp_id
      set name temp_name
    ]
  ]
  ...
end
```

The `foreach` primitive loops through each point in the list and uses the temporary variable `quarry-point` to hold each `VectorFeature` in turn (fig. 7.3). We also add two additional lines to extract the `property-values` of each point and assign them to the `quarries-own` variables of `id` and `name` (see also fig. 7.2).

PERCEIVING RASTER & VECTOR DATA

Once the data have been copied into NetLogo's architecture as patches or agents, you can then follow the methods you learned in previous chapters to have the agents move and interact with those data. First, initialize agents in the `setup` (fig 7.3. We need to create them as a different breed because we already have one breed of agents—the quarries.

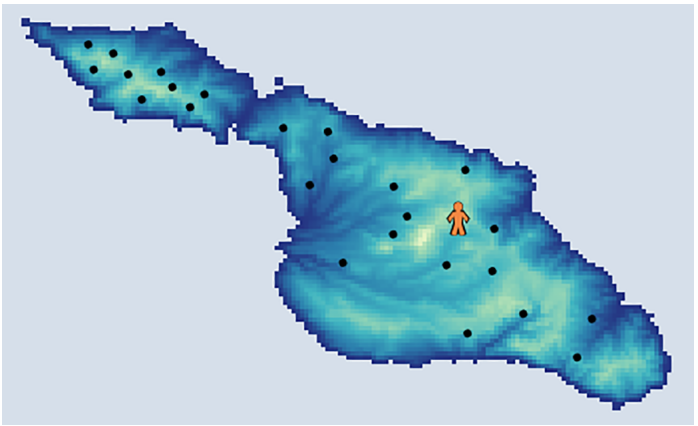


Figure 7.3. Imported GIS raster representing an island and vector data representing stone quarries (black dots).

CODE BLOCK 7.10

```
breed [foragers forager]
...
to setup
...
ask n-of 1 patches with [ patch-elevation > 0 ] [
    sprout-foragers 1 [
      set size 8
      set shape "person"
      set color yellow
      set toolkit [ ]
    ]
  ]
```

DON'T FORGET

`toolkit` is a turtle variable that needs to be initialized in `turtles-own`.

Now agents can interact with space in several ways. Although the following won't end up in our final model, for example, they could perceive the elevation in order to walk downhill using:

```
to go-down-hill
  ask foragers [
    let next min-one-of neighbors with
      [patch-elevation > 0] [patch-elevation]
    move-to next
  ]
end
```

CODE BLOCK 7.11

In Davies' model, agents were restricted from going above a certain `elevation-threshold`.

Or agents could look for the closest quarry patch (i.e., using the patch version of quarries, code block 7.7) to approach with a directed walk:

```
to walk-quarry
  ask foragers [
    let nearest-quarry min-one-of patches with
      [patch-quarry = true] [distance myself]
    face nearest-quarry
    fd 1
  ]
end
```

CODE BLOCK 7.12

Note that we have not included code to stop agents crossing water if they are at the coast. Can you modify the code to fix this?

As you can see, combining GIS data with the mobility algorithms we learned in chapter 4 can lead to a richer and more nuanced model capable of asking and answering different types of questions than its more abstract versions. By modeling space in a realistic way, our agents have very well-defined and specific opportunities and constraints, which may explain some of the features of the archaeological record. By embedding agents in a "real" landscape, we can examine specific contexts corresponding to real past landscapes, thus focusing on past people who used to live in a particular place.

In the two examples above, a patch variable influences agent movement either by being an agent's target or by constraining the agent's movement choices (patch–agent). In the following sections, we will discuss examples of GIS use in an ABM where other types of interactions occur: agent–patch, agent–agent, and patch–patch.

7.2 Agent–Patch Interactions

Agent–patch interaction is often a primary focus of archaeological agent-based models. There are two ways agents typically interact with patches beyond just reading a patch value: they either take something away from it or deposit something into it. The former may include collecting lithic resources, harvesting plants, or reducing the number of fish in a lake. We discussed these in detail in the subsistence chapters (chs. 3 and 6), so here we will focus on the second mode, which was only briefly explored (ch. 5): agents leaving material traces of their behavior by depositing objects on patches. Material traces left behind by past peoples, such as lithics, potsherds, or architecture form the basis of our knowledge of past societies. Thus, a common approach in archaeological ABMs is to model the agents leaving behind some record of their activity (e.g., Kohler and Varien 2012). We can then export a complete *in silico* artifact spatial distribution for comparison to a region's archaeological record. First, though, our agents need to acquire some materials to leave behind. Here we will switch our agents' movement to a random walk and then allow them to acquire resources from quarries that they encounter.

The random walk mimics the idea of embedded procurement, as agent mobility is not connected to quarry locations (Binford 1979; Brantingham 2003).

CODE BLOCK 7.13

Note that this is written to query the shapefile directly. Can you re-write this to use the quarry agents from code block 7.9?

```
to go
  ask foragers [
    move-to one-of neighbors with
      [ patch-elevation > 0 ]
    if patch-quarry = true [
      reprovision-toolkit
    ]
  ]
end

to reprovision-toolkit
  let t gis:property-value first (filter [q ->
    gis:contained-by? q patch-here]
    (gis:feature-list-of
    quarries-dataset)) "ID"
```

CODE BLOCK 7.13 (cont.)

```
  while [length toolkit < 100] [
    set toolkit lput t toolkit
  ]
end
```

TIP
Add a `turtles-own [ toolkit ]` to the top of the code and `set toolkit [ ]` to the `sprout` code in the `setup` to initialize that variable as a list.

The `gis:property-value` extracts the `"ID"` numbers of the quarries (see table 7.2) from the vector file. In this case, our quarries only have an ID number and a name, but they could also contain other attributes that could impact tool acquisition, such as average nodule size, cortex ratio, flakeability, or other indicators of quality (for a worked example, see Davies, Holdaway, and Fanning 2018).

Next, we want our agents to drop some of their tools to create a simulated archaeological record. This could occur during specific activities, but here we will have them drop a random tool at each step. Add the following to the `setup` code inside the `ask foragers [ ]`. Write the `discard-tools` below as a separate procedure:

CODE BLOCK 7.14

```
to go
  ...
  if length toolkit > 0 [
    discard-tools
  ]
  ...

to discard-tools
  let i random length toolkit
  ask patch-here [
    set assemblage lput (item i [toolkit] of myself)
      assemblage
  ]
  set toolkit remove-item i toolkit
end
```

The first line of `discard-tools` selects a random number up to `length toolkit` to act as the position along the toolkit list to be

dropped. The next part adds that tool's original quarry ID to the patch's assemblage variable. Last, we remove that same tool from the agent's toolkit. Here, the key to avoiding errors is using `i` first to define the tool, then copy it to the patch's assemblage, and finally delete the tool from its original location. It's worth running a good set of tests when manipulating multiple lists, as it is a common source of errors.

Next, we will visualize the patch assemblages by adding a `display-assemblages` procedure:

CODE BLOCK 7.15

```
to display-assemblages
  let mx max [length assemblage] of patches
  ask patches with [length assemblage > 0] [
    set pcolor scale-color red (length assemblage)
    0 mx
  ]
end
```

For more on testing techniques, see chapter 8.

Here we simply use the length of the assemblage list as the number of tools on the patch. Note that using `pcolor` to visualize this data will overwrite the elevation `pcolor`, so make a separate button for it or create a separate `display-elevation` procedure and a SWITCH `viz-assemblage?` to toggle between displays. Alternatively, you can create a slider, TIME-LIMIT, and use it to stop the simulation and trigger the visualization at the end of the run.

DON'T FORGET

Add `assemblage` and `assemblage-size` to `patches-own` and initialize them in `setup`. `assemblage` should be a list.

We can also write a procedure to export the patch assemblage sizes to a new raster dataset that can be then analyzed in any GIS software. Since we aren't including a file path, the exported file will be written to the model's directory.

CODE BLOCK 7.16

```
to write-assemblages
  ask patches [ set assemblage-size length assemblage ]
  let assemblage-dataset gis:patch-dataset
    assemblage-size
  gis:store-dataset assemblage-dataset
    "assemblages.asc"
end
```

Let's test whether targeted mobility changes the spatial distribution of the simulated archaeological record. In short, we want to know how the archaeological record would look if foragers made sure they always had tools in their toolkit rather than walking around with empty proverbial pockets much of the time. We can create a switch on the INTERFACE tab called `random-walk?` to test each mobility option (discussed in ch. 4) without having to rewrite the model or make multiple copies of it. Replace the `move-to` line in `go` with a conditional test:

CODE BLOCK 7.17

```
ask foragers [
  ifelse random-walk? [
    random-walk
  ][
    target-walk
  ]
  ...
```

Then we'll add code for the two alternative movement types, which should be familiar from chapter 4.

CODE BLOCK 7.18

```
to random-walk
  move-to one-of neighbors with [patch-elevation > 0]
end
to target-walk
  ifelse length toolkit < max-carry * 0.1 [
    let t min-one-of patches with [patch-quarry = true]
    [distance myself]
    face t
    ifelse [patch-elevation] of patch-ahead 1 > 0 [
      move-to patch-ahead 1
    ][
      random-walk
    ]
  ][
    random-walk
  ]
end
```

What would happen if we just told agents to move to the nearest quarry without specifying that their toolkit should be getting empty first?

The new `target-walk` is a bit longer since it requires some contingencies. First, we set a conditional with a 10% threshold of their toolkit `max-carry` length. If the agent's toolkit is almost empty, they will move toward the nearest quarry; otherwise they will walk randomly. Also, because we are working with an imported landscape—in this case, one with coastlines—there is a possibility that agents will get stuck facing the nearest quarry but be unable to get there because a body of water is in their way. Thus, we add another test `ifelse [patch-elevation] of patch-ahead 1 > 0` that allows the agents to make a random step and get themselves unstuck if needed (for more complex coastal landscapes, a stronger unstuck contingency may be required, such as longer random steps or following the coastline). This is an issue known as *edge effects*. Agents may behave differently (e.g., get stuck) when they reach the edge of the world, a coastline, or another barrier. For an abstract landscape, we may be able to wrap it into a **torus**, essentially removing the edge effect all together.

DON'T FORGET
Define the new slider `max-carry` in the INTERFACE tab and set its range to 0–100.

Adjusting code for contingencies often occurs as you discover your agents walking off the map or picking up more weight than they should be able to carry.

torus: a doughnut-shaped landscape topology where an agent going off the top will appear at the bottom, and when traveling out one side will reenter from the other.

However, in any realistic landscape or an island context, edge effects are built into the model—agents should not venture off the littoral into the deep water or disappear off the northern border of Utah to reappear on its southern border. If edges are not taken into account in the building of the model, then this can become problematic and skew the results. Designing specific algorithms, such as the one written above, can help to avoid these challenges.

In the next two chapters, we will learn more about how to conduct large numbers of experimental runs and compare the outputs, but for now we can just visually compare the two exported raster maps (fig. 7.4).

AGENT–AGENT & PATCH–PATCH INTERACTIONS

The two remaining types of interactions are agent–agent and patch–patch interactions. Agent–agent interactions are useful if agents are directly interacting with one another based on proximity or other spatial characteristics. These include cases of agents reproducing with nearby agents, only moving to unoccupied patches, and passing items to other agents that require using primitives such as `in-radius`, `any? turtles`, or `min-one-of turtles [distance myself]`. The accounting of spe-

See chapter 8 for use of network structures rather than Euclidean proximity for agent–agent interaction.

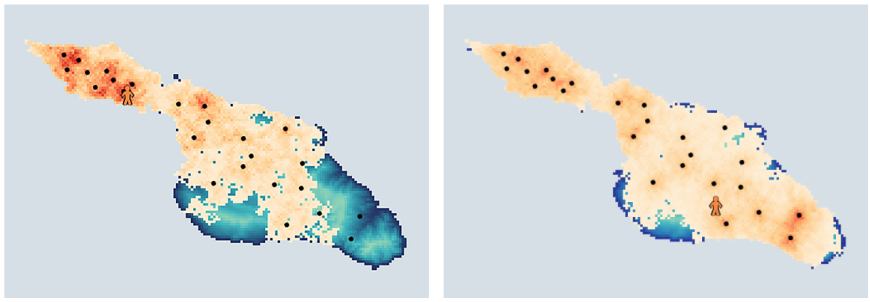


Figure 7.4. Sample results of the ABM with random-walk (left) and target-walk (right). Note that target-walk covers a larger area, showing how reprovisioning can lead to archaeological material being more distributed, even though the agent does random-walk most of the time in both scenarios.

cific items requires much of the same code in agent–agent as it does in agent–patch interactions:

CODE BLOCK 7.19

```
to pass-artefact
    let t min-one-of other foragers in-radius 3
      [distance myself]
    let i random length toolkit
    let passed? false

    if t != nobody[
      ask t [
        if length toolkit < max-carry [
          set toolkit lput (item i [ toolkit ] of
          myself) toolkit
          set passed? true
        ]
      ]
    ]

    if passed? [
      set toolkit remove-item i toolkit
    ]
end
```

You need to increase the number of foragers to minimum two for this procedure to work. Call it just before `discard-tools` in `go`. See ABMA Code Repo for the working example.

The temporary variable `passed?` helps control which parts of the code should run. Here, the agent should not remove a tool from their toolkit if they haven't passed it to another agent.

Lastly, there are many instances where patches interact with each other. For example, certain resources, such as a forest, may only grow next to patches with such resources already present, or an item dropped by an agent may spill onto neighboring patches.

See the *Ants* model in the NetLogo Models Library for a diffusion algorithm.

For example, Riris (2018) modeled the anthropogenic impact of swidden agriculture on tropical forest in Venezuela (see ch. 6). In his model, the rate of forest regrowth in a cell is directly related to the type of vegetation surrounding it. When the cell's neighbors consist of primary forest, it is more likely to be colonized by pioneering species than one encircled by other fallowed plots.

There are also pragmatic reasons to model interactions between patches. Combining patch variables for future use in a model, rather than repeatedly calculating them on the fly, is a good way to save computational time. For example, we might want patches to always know how far they are from the nearest source of a resource such as a quarry or water, so that agents do not have to repeatedly calculate the distance for themselves:

CODE BLOCK 7.20

Calling this once in `setup` would be much more computationally efficient than repeating the distance measure each time an agent needs the (unchanging) information.

```
to calc-water-distance
  ask patches with [patch-elevation > 0] [
    let p min-one-of patches with [patch-elevation <=
    0]
      [distance myself]
    set water-distance distance p
  ]
end
```

There are many other useful approaches to modeling patch interactions in NetLogo. For example, within NetLogo's Models Library, the GIS Gradient Example demonstrates how to use `gis:convolve` to generate new slope and aspect rasters (`gis:create-raster`) from a previously loaded elevation raster dataset—it uses a little math to write values to a new raster one cell at a time with `gis:set-raster-value`. This can be useful for calculating what is uphill and what is downhill within the patch instead of having an agent calculate it on the fly.

Review the other GIS examples in the NetLogo Models Library to see the breadth of applications.

7.3 GIS Data Sources & Types

We have spent this chapter discussing how to import and manipulate GIS data in NetLogo's patch and agent terminology. Now you might want to know how to get a hold of these data to begin with. As always, the types of data needed will depend on the research questions of your study. However, certain standard data types tend to be particularly useful for archaeological research questions.

A **digital elevation model (DEM)** will be necessary in any study looking at the energetics of travel (e.g., Gravel-Miguel and Wren 2018); it also applies if the modeled population avoids certain altitudes or steep slopes, or if you are addressing questions like vertical transhumance. Although many global datasets exist, for accuracy and high spatial resolution, the Shuttle Radar Topography Mission (SRTM) dataset is hard to surpass. It is available globally at ~30- and ~90-meter horizontal cell resolution for all but the very highest latitudes (making it less useful for research in Scandinavia, Siberia, and arctic Canada). The data are available in tiles and can be downloaded for free via the USGS Earth Explorer web tool.[5] Combine tiles in a GIS software or load multiple datasets and set the world envelope to the union of them (see sec. 7.1).

DEM: digital elevation model. A datafile with topography of the mapped area recorded as elevation values.

Some measure of vegetation cover, ecology, soils, or distribution of food resources is also commonly required in studies focusing on subsistence or settlement patterns. This is more difficult to acquire since, unlike elevation, ecology has varied dramatically over the course of the human past and must be reconstructed. Paleoclimatic and paleoecological reconstructions form a complex domain of science, often built off of computational simulations themselves. Many global climate models (GCMs) are being used to hindcast/retrodict climatic data, often using coupled ocean–atmospheric models run on high-performance computing architectures (e.g., supercomputers). The outputs of these simulations may then be run through additional models to simulate vegetation cover or, in a few cases, vegetation models are directly coupled to the ocean–atmosphere models. For reviews of these approaches, see Braconnot et al. (2012), Harrison, Bartlein, and Prentice (2016), and Haywood et al. (2019), which also include discussions of

[5] https://earthexplorer.usgs.gov

model validation using archaeological and paleoecological proxies (e.g., ice, marine, and lake cores).

Since this is all well beyond the expertise of even a dedicated computational archaeologist, the Synthesizing Knowledge of Past Environments (SKOPE) project has worked to make paleoenvironmental data more accessible.[6] As of this writing, it may be the most synthetic and user-friendly package available for creating past environmental data.

A number of large-scale archaeological research projects, including the foundational Stage Three Project (Andel and Davies 2003), have reconstructed past climates and human-relevant aspects of land cover using these methods. These data are usually freely available, assuming they exist for the specific time slice relevant to your research questions, but often require considerable GIS experience to be converted into NetLogo-compatible formats (e.g., from NetCDF).[7] As well as SKOPE, WorldClim[8] carries a range of data from global climate simulations (monthly measures of temperature, precipitation, and other bioclimatic variables) but currently only for the mid-Holocene (~6,000 years ago), the Last Glacial Maximum (MIS 2 ca. 22,000 years ago), and the last interglacial (MIS 5e ca. 125,000 years ago). These hindcast data are derived from the Palaeoclimate Modelling Intercomparison Projects,[9] the most up-to-date comprehensive dataset of the simulated climate variables in the last 120,000 years is Beyer, Krapp, and Manica (2020). Finally, the NOAA website has a large repository of open-access climatic data (not only spatial but also temporal),[10] and GEBCO is the most commonly used source of bathymetry data, which is useful if you need to adjust sea levels.[11] Alternatively, you could try to meet some colleagues in your neighborhood Atmospheric Science department and hope they have access to a high-performance computing (HPC) laboratory to simulate your particular time slice.

A common data format in NetCDF. It consists of separate temporal slices of spatial data. Use QGIS or other GIS software to parse the data you need.

[6] https://www.openskope.org

[7] https://simulatingcomplexity.wordpress.com/2014/11/17/working-with-netcdf-files-in-an-agent-based-model-skinning-the-model-input-data-cat/

[8] https://worldclim.org/data/v1.4/paleo1.4.html

[9] As of this writing, currently on its fourth iteration. See https://pmip.lsce.ipsl.fr.

[10] https://www.ncdc.noaa.gov/data-access/paleoclimatology-data

[11] https://www.gebco.net

7.4 Using Artificial Landscapes to Control Spatial Variables

There are pros and cons to using highly detailed, spatially explicit data in your agent-based model. At worst, the data will be far more detailed than is necessary for your research questions, which will slow the model's runtime and make analysis needlessly complicated. It may also be difficult to understand to what extent the outcome of the model is due to the specific characteristics of your "realistic" landscape compared to other factors such as the dynamics among the agents themselves. You might find yourself wondering if your result would be different if your landscape changed, or how the agents might react if your imported temperature raster was one degree cooler. This limits the generality of your results, possibly making it difficult for other archaeologists to assess how your conclusions would apply to another case study.

On the other hand, a realistic landscape may help with the model validation if spatial distribution is important.

The alternative is to create artificial landscapes with controllable attributes. For example, Lake (2001) used a fractal algorithm (GRASS GIS's r.surf.fractal) to create landscapes with different degrees of patchiness or heterogeneity for his MAGICAL simulation of Mesolithic foraging. In an innovative GIS study of flows of human movement through a landscape, Llobera, Fábrega-Álvarez, and Parcero-Oubiña (2011) created a series of artificial landscapes so that they could illustrate the functioning of their GIS model (see also Fábrega-Álvarez 2006). We replicated their artificial landscapes using a combination of GRASS GIS R, and then uploaded them to NetLogo. All of these programs use the .asc (ASCII) raster format described above, making it relatively straightforward to switch between them (fig. 7.5). Many of the algorithms needed for creating artificial spatial structures can be found in O'Sullivan and Perry (2013). These tightly controlled artificial landscapes can help to explain the dynamics of a model without the complexity of "real" GIS data, while also serving as a sanity check during model development to make sure that the model is doing what you expect.

It is very common to start the model on an artificial landscape and only later import GIS data.

In an agent-based model of Ache hunting patterns, Janssen and Hill (2014) first published a model using the most realistic ecological data they could find. They then wrote a follow-up paper artificially increasing and decreasing the patchiness of resources in order to demonstrate how that specific characteristic of their landscape would impact the model results. In doing so, they demonstrated that more "clumped" habitats with the same

It is critical for the modeler to fully explore the impacts of the underlying spatial data on the model results just as one does for each parameter.

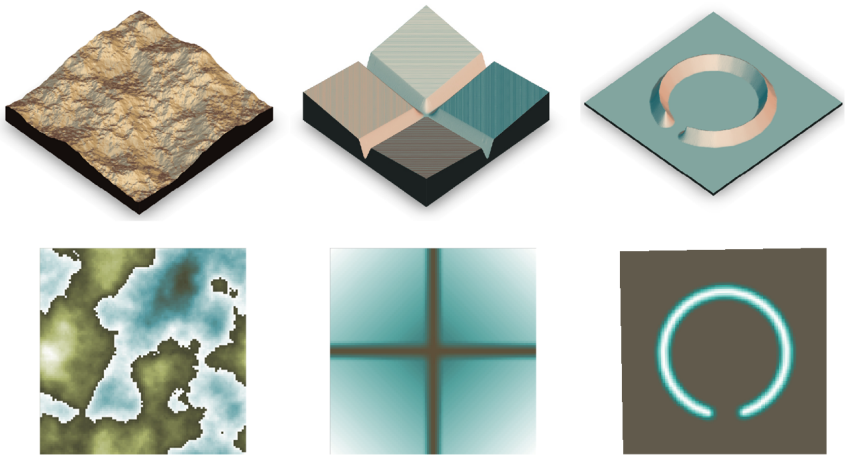


Figure 7.5. Replication of Llobera, Fábrega-Álvarez, and Parcero-Oubiña's (2011) artificial landscapes in GRASS (GRASS Development Team 2018), visualized using R's rayshader package (Morgan-Wall 2019). Top row, from left: simulated topography created with GRASS-GIS's r.surf.fractal algorithm; intersection with corners sloping toward the center; and a ring hill with a single entrance. The bottom row shows what these landscapes look like when imported into NetLogo.

overall area favored lower levels of mobility with higher foraging returns (Janssen and Hill 2016).

7.5 Profiling Your Code

Models that incorporate detailed GIS data tend to take more processing time to run, as they tend toward the "realism" end of the modeling spectrum (see ch. 8), often testing very specific hypotheses that relate to the condition of a specific landscape. This often requires a thorough parameter sweep implying hundreds of runs. Although a slight departure from the GIS theme of this chapter, this section will discuss model processing speed:

- What makes a model slow?
- How do you time the various components of a model?
- How can you speed up your model?

It is fair to say that all modelers want their models to run faster. A faster runtime means we can do more tests during development, run the model longer, run a broader suite of parameters during sensitivity analyses, and get results from NetLogo's BehaviorSpace within a day rather than a year

(see ch. 9 for details on the latter two). It is also often the case that archaeological models are "overdesigned and under run" (to quote Mark Lake), and so being able to run your model faster is a good way to run it more and make sure you thoroughly understand its dynamics.

ABMs are limited by the computer's processing speed. This is affected by the processor itself, but also the memory (i.e., RAM) and the hard disk speed depending on what you are asking the model to do. For the sake of this chapter, we will keep it simple and just think about this as the total time it takes to accomplish the tasks we program into the model. A simple count of all the commands that are called in the model gives a rough estimate. Most of the tasks and little math operations can be counted as roughly equivalent (for exceptions see below).

Before we get to NetLogo's Profiler extension, think about how many patches and how many agents there are in your model. Probably you have more patches than agents. So any time you ask all the patches to do something, it is probably going to take longer to process than asking all the agents to do something. For this reason, it is a good idea to try to get most things done by turtles, even if it is updating a patch variable.

Using `breeds` is another way of reducing the number of unnecessary calls. Similarly, defining a `patchset` ensures that a procedure is only called on a selection of relevant patches.

For example, this:

CODE BLOCK 7.21

```
ask patches [if any? turtles-here
  [set resources resources - ( count turtles-here
  * harvest-rate )]]
```

will take much longer than this:

CODE BLOCK 7.22

```
ask turtles [ask patch-here [set resources resources
  - harvest-rate]]
```

Even though both are accomplishing the same task (having the agents acquire some of their current patch's resources), in the first version, every patch in the landscape (and there are likely a lot of them) is first asked a simple question about whether an agent is present. In the lithic procurement model above, there are over 66,000 patches, so that is 66,000 questions, 99% of which will return a "No" (or `FALSE` in NetLogo terms). Fewer than 20,000 of those patches are actually land, so asking only `patches with [patch-elevation > 0]` would already be much faster. In the second

version, the number of patches asked to perform the task will be equal to the number of agents, likely fewer than 40,000. In addition, the `if any? turtles-here` question does not need to be asked at all, since it is always going to be true when it is the turtle asking, so we can skip that step, which also saves a bit of computational time.

To quantify a model's processing speed in a comprehensive manner, we need to find out which parts of the model are run, how many times, and how long each part takes. This is the purpose of NetLogo's Profiler extension.[12] To enable it, add the following to the top of your model's code:

CODE BLOCK 7.23

```
extensions [profiler]
```

Next, add a button to the INTERFACE of the model, name it PROFILER, and then in the COMMANDS box of the button enter:

CODE BLOCK 7.24

The code here is copied directly from NetLogo's Profiler Manual.

```
setup                  ;; set up the model
profiler:start         ;; start profiling
repeat 30 [go]         ;; run a procedure
profiler:stop          ;; stop profiling
print profiler:report  ;; view the results
profiler:reset         ;; clear the data
```

This is assuming you have set up your model in the standard way, as we have in this chapter, with the initialization procedure called `setup`, and the iterating part of the model called `go`. Now when we push the button, it will run the model through 30 ticks of the model and then output the processing time results into the COMMAND CENTER of NetLogo.

CODE BLOCK 7.25

Note that times are listed in milliseconds. Headers actually have `(ms)` but we trimmed them for space.

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                Calls Incl T  Excl T   Excl/calls
DISPLAY-ELEVATION   30  2000.473  2000.473     66.682
RANDOM-WALK         30     0.887     0.887      0.030
GO                  30  2002.080     0.720      0.024
```

[12] https://ccl.northwestern.edu/netlogo/docs/profiler.html

CODE BLOCK 7.25 (cont.)

```
Sorted by Inclusive Time
GO                  30  2002.080     0.720      0.024
DISPLAY-ELEVATION   30  2000.473  2000.473     66.682
RANDOM-WALK         30     0.887     0.887      0.030

Sorted by Number of Calls
GO                  30  2002.080     0.720      0.024
DISPLAY-ELEVATION   30  2000.473  2000.473     66.682
RANDOM-WALK         30     0.887     0.887      0.030
END PROFILING DUMP
```

The results are structured in three tables, which is really just one table re-sorted by each of the table's columns. Each row represents a model procedure that was called at least once. The first column is the number of times that procedure was *called* or asked to run. The second is the "Inclusive Time" in milliseconds for that procedure, or the total time the model spent within that procedure *and* the procedures it called (e.g., the inclusive time for `go` is the total runtime minus `setup`). The second column is the "Exclusive Time," which is the total time spent just on that procedure *without* any of the procedures it called. The last is the exclusive time per call (rather than summed over all the calls). The Exclusive Time column is the most useful in terms of finding out where the computational bottlenecks are, since it tells you exactly which part of the model has taken the most time overall.

We focus on profiling `go` since the `setup` procedure is run only once.

If you have designed your model such that everything happens within one big `go` procedure, you will not get very interesting results here. To make the most of PROFILER, you need to break up the model into separate procedures, as we have done above (i.e., `random-walk`, `target-walk`, `reprovision-toolkit`, and `discard-tools`). This is good practice for coding anyway, since it helps to see the overall structure of the model in `go`, almost like a table of contents (see ch. 2).

Writing out code as separate procedures is known as **modular code development** (see sec. 2.4).

For this model, we have only a single agent, and in 30 ticks it is possible our agent moved without encountering a quarry, so change the `repeat 30` in the PROFILER button code to a higher number, like 200, to make sure that all the procedures are run. This time, turn

`random-walk?` off and `viz-assemblages?` on, and click the PROFILER button again.

CODE BLOCK 7.26

```
Sorted by Exclusive Time
Name                 Calls   Incl T   Excl T  Excl/calls
DISPLAY-ASSEMBLAGES 200 802731.319 802731.319  4013.657
TARGET-WALK          200     15.293     12.510     0.063
GO                   200 802759.149      8.599     0.043
DISCARD-TOOLS        195      3.300      3.300     0.017
RANDOM-WALK          188      2.784      2.784     0.015
REPROVISION-TOOLKIT  6        0.638      0.638     0.106
```

You can easily see from this table (just the Exclusive Time) that `display-assemblages` is the part taking the most time, and it is called every tick. For the sake of this example, we wrote the `display-assemblages` procedure like this:

CODE BLOCK 7.27

```
to display-assemblages-slow
  ask patches with [ length assemblage > 0 ] [
    set pcolor scale-color red (length assemblage) 0
    (max [ length assemblage ] of patches)
  ]
end
```

Within this procedure we have asked every patch (`ask patches`) to assess every other patch (`max [length assemblage] of patches`). While it does work, this is a ridiculously computationally expensive way to do something as simple as updating the patch color. A better way is like this (which is what we did earlier in the chapter; see code block 7.15):

CODE BLOCK 7.28

```
to display-assemblages-faster
  let mx max [ length assemblage ] of patches
  ask patches with [ length assemblage > 0 ] [
    set pcolor scale-color red (length
    assemblage) 0 mx
```

```
  ]
end
DISPLAY-ASSEMBLAGES-FASTER
    200   1094.816   1094.816   5.474
```

CODE BLOCK 7.28 (cont.)

On our test computer this took more than 4,013 ms per call the first way and only 5.4 ms the second way. Even faster methods would be to make a patchset called `land` in `setup` to exclude the water and only `ask land` to update color (0.4 ms), or to have your agents update the `pcolor` of the patch when they drop a flake (0.2 ms). Both of these methods have the advantage of most patches not being asked to do anything for most ticks, which greatly reduces the processing time. It might have taken a bit of creativity and time to code, but your model is now 20,000 times faster.

We don't provide the code for these even faster methods—try to work it out on your own.

If you're unsure which of your coding methods is going to run faster, then make a new button to run them both. On the INTERFACE, make a new button, call it `profile-procedures`, and do the same as before but with `go` replaced by the name of your various procedures:

Depending on the model, you may need to comment out `setup` from the PROFILER `button` code, and run the model for a while before starting PROFILER. For example, maybe you're testing something that happens only after the agent population has grown to a certain size.

```
;setup
profiler:start
repeat 30 [ display-assemblages
  display-assemblages-faster ]
profiler:stop
print profiler:report
profiler:reset
```

CODE BLOCK 7.29

Note that we have commented out setup. So, use the SETUP button and then run the model forward in time to make an assemblage to display before pushing this button.

When you click the PROFILE-PROCEDURES button, it will give you specific timings of those two procedure options without the rest of your model getting in the way.

A few other common things to fix when trying to speed up your NetLogo model:

1. INTERFACE plots:
   While necessary, plots take a lot of computational time; they update when `tick` is called, so their computational time is included

If the procedures you're testing are turtle-specific—like `consume` or `move`—then you will need to add `ask turtles` into the button's code.

in the GO section of the PROFILER report. Add a `plots-on?` switch to the INTERFACE, and `if plots-on?` to the plot update code so that you can turn them off when they're not needed.

TIP
If your model takes more than an hour to run, be concerned. The next step, experiment design and running, is likely to be an uphill battle.

2. Output to COMMAND CENTER and text files:
   While `show`, `type`, and `print` are useful for debugging, make sure to comment out those lines when debugging is complete, as writing to COMMAND CENTER or output files slows things down a lot.
3. Doubled asking:
   If you inadvertently nest `ask turtles` in another `ask turtles`, then each turtle will ask every other turtle to do something. This is a not only time consuming but also likely to be wrong.

```
ask foragers [ target-walk ]
...
to target-walk
  ask foragers [
    ...
  ]
end
```

NetLogo will catch this error with `Only the observer can ASK the set of all turtles.` if you just use the default `turtle` breed, but not if you've created other breeds.

This last point is a horrible bug that you may make when breaking up code into separate procedures. The code looks right at first, but upon closer inspection you can see that every agent asks every other agent to perform the task because the `ask turtles` line was repeated in the `go` code and in the `target-walk` procedure it calls. In the PROFILER report, you will see that `target-walk` is called far more times than it should be.

Although these general guidelines are great to make your model run faster, sometimes none of them help; the algorithm just takes time to run. It may be worth checking if an alternative implementation of that algorithm might be quicker. For example, there are plenty of ways to code a random-walk (ch. 4) with many subtle differences in coding and in timing. What may have looked like inconsequential differences between different implementations may take double the computational time (or more). If you have thousands of agents and/or thousands of

time steps, then shaving off even milliseconds for each one will make a large cumulative difference. **Benchmarking** the timings of algorithm versions using `profile-procedures` makes it easy to determine which implementation is the fastest and can help you to choose accordingly.

benchmarking: a computer science term for measuring the computational speed of an algorithm. Often done before and after a change to the code or to compare different algorithms.

However, a famous saying in computer science is "Premature optimization is the root of all evil," attributed to Donald Knuth (1974). This means "Don't try to make your code fast too early." There is a standard sequence of actions for software development: write the code, make it work, test it, profile it, and only then try to optimize it. Trying to make the code as efficient as possible from an early stage of a project can be tiresome, can lead to unnecessarily complicated code, and is often completely unnecessary. For example, in our stone procurement model, `setup` is quite slow, but there is little point in optimizing the `setup` procedure because it is run only once. Equally, trying to guess what is slowing down your simulation without profiling your code is likely to be a waste of time.

7.6 Summary

When building an agent-based model, a tight or loose coupling of the GIS to the ABM may be called for, depending on the questions being asked (Crooks et al. 2019). Following Davies et al. (2019), this chapter used a loose coupling, where data are prepared in a GIS and loaded into NetLogo (e.g., the digital elevation model and quarry points). If the model output is spatial (e.g., distribution of pottery or locations of model's settlements), data can be uploaded again to a GIS for analysis. Since GIS software generally does not deal well with temporal data, the pipeline of switching between a GIS platform and NetLogo is often the best solution. A tight coupling would involve a close integration between the modeling platform and the GIS platform, as in the *MedLanD* project described in previous chapters (Robinson et al. 2018). This is certainly possible with NetLogo, thanks to the R extension, which enables NetLogo to be run from the R programming environment (R Core Team 2019), which also provides a powerful way to do GIS analysis, for instance, with the raster (Hijmans 2019) and SP (Bivand, Pebesma, and Gomez-Rubio 2013) packages. Similarly, you can run NetLogo using Python via

Binder and do your spatial data analysis in, for example, Jupyter Notebook (Graham 2018). Many new books on geographic information science (e.g., Huang 2017; Crooks et al. 2019) devote entire sections to coupling GIS and ABMs. While it is not necessary to have strongly coupled software to create a good model, appropriate methods are available if your research project requires them.

GIS data and ABM feel like a logical fit, but as with all modeling, we need to take care only to model what is necessary for our research questions. In this chapter we have covered how to import both raster and vector data types into NetLogo's patch and agent environment. Raster data are fairly simple, since NetLogo's patch landscape is essentially a raster data structure already. The vector data architecture is a little (okay, a lot) more complex, but once a shapefile of point data is converted into an agentset or a patch variable, we can return to our familiar modeling approaches.

The dynamics of human–environment interactions are a mainstay of archaeological research. Modeling these dynamics comes down to asking yourself two simple questions: What is acting upon what? What is the nature of that interaction? The answers depend on the system we model and our research questions. While gaining GIS skills is a whole new area of expertise, it pays off in the long term.

End-of-Chapter Exercises

1. Create an alternative visualization which calculates diversity as the number of unique elements in each patch's assemblage list, then export this as an ASCII raster.

2. Go to the Pattern and Process website and download the *Point Process* model.[13] You will need to install the Gradient extension.[14] Explore the model and learn how to generate different numbers of clusters with different degrees of clustering.

[13] https://patternandprocess.org/2-6-point-processes-with-clustering

[14] See instructions here: https://ccl.northwestern.edu/netlogo/docs/extensions.html.

3. Combine the code from the *Point Process* model with different movement algorithms from chapter 4. Explore which one gives the most divergent results depending on the clusters' properties.

4. Find a freely available topography dataset of an area of interest and import it into NetLogo. Integrate it with the *SugarScape* model developed in chapter 3 so that it runs on a realistic landscape.

5. In the NetLogo Models Library, find the *Tijuana Bordertowns* model. It is a great example of an artificially created urbanscape. Explore the model and its parameters.

Further Reading

- J. Conolly and M. W. Lake. 2006. *Geographical Information Systems in Archaeology*. Cambridge, UK: Cambridge University Press. doi:10.1017/CBO9780511807459

- A. T. Crooks et al. 2019. *Agent-Based Modelling and Geographical Information Systems.* London, UK: Sage Publishing. https://www.abmgis.org

- I. Romanowska et al. 2019. "Agent-Based Modeling for Archaeologists: Part 1 of 3." *Advances in Archaeological Practice* 7 (2): 178–184. doi:10.1017/aap.2019.6

- B. Davies et al. 2019. "Combining Geographic Information Systems and Agent-Based Models in Archaeology: Part 2 of 3." *Advances in Archaeological Practice* 7, no. 2 (May): 185–193. doi:10.1017/aap.2019.5

- B. Huang. 2017. "Agent-Based Modeling." In *Comprehensive Geographic Information Systems,* edited by T. J. Cova et al. Amsterdam, Netherlands: Elsevier

- D. O'Sullivan and G. L. W. Perry. 2013. *Spatial Simulation: Exploring Pattern and Process.* Chichester, UK: John Wiley & Sons, September. https://patternandprocess.org/