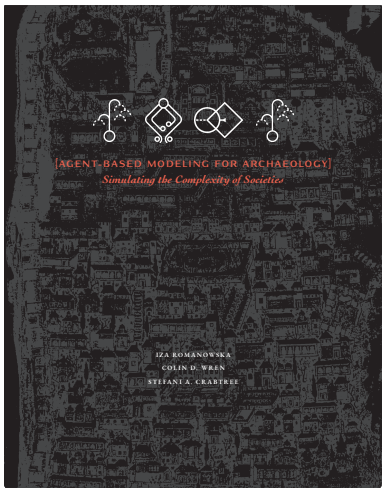
PLEASE NOTE:

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

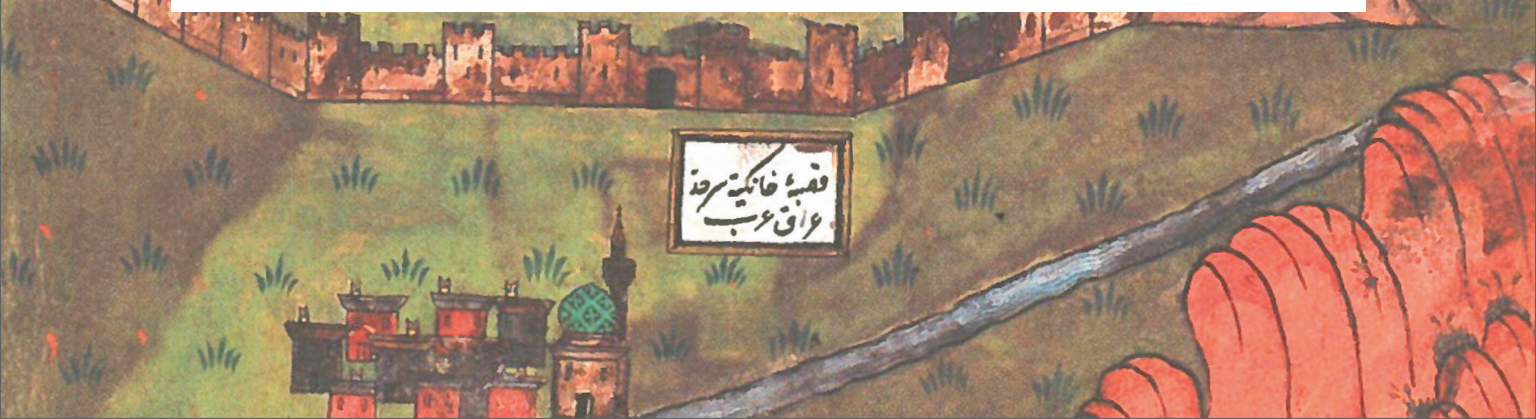


REGARDING COLOR:

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




دوجا در بندن آشوب
قمر دره اغزی
سکن اوه
آشوب غوغی

DATA ANALYSIS: LEVERAGING DATA SCIENCE TO EXPLORE ABM RESULTS

9.0 Introduction

Throughout this book we have explored how modelers use data. We validated a model's dynamics by comparing it to archaeological datasets (chs. 3 and 4); we made sure that the model's mechanisms were reasonable and justifiable given the knowledge of human behavior based on data (ch. 5); and we brought in both spatial data (ch. 7) and relational data (ch. 8) to incorporate into our models as input. In this chapter, we will talk about a different type of data—data that describes not the real world but the one we created in our simulation. Every model goes through the phase known as **experiment design**, which generates this artificial data. We will examine how to generate and analyze model outputs that will turn this whole endeavor into publishable science.

OVERVIEW
▷ Principles of experiment design
▷ Tutorial in NetLogo BehaviorSpace
▷ Calibration, sensitivity analysis, and parameter sweep
▷ Analyzing output data in Excel, R, and Python
▷ What is emergence?
▷ Documentation and dissemination of ABMs

experiment design: the final phase of the modeling process in which the modeler experiments on the modeled world by running the simulation in different configurations (scenarios).

Let's say you just watched 10,000 agents interact with each other and with their environment for 10,000 time steps, and you have a feeling that you know what is happening. But can you demonstrate it? Most of all, how can you convince your readers, who are most likely not going to run your model themselves, that your intuition is correct?

The task now is to demonstrate *empirically* that a given pattern occurs reliably under certain conditions; that specific factors, encoded as parameters, result in distinct outputs; and that you can explain *why* it happens. We will go through the process step by step using the set of questions in table 9.0 as our guide. We will:

1. Look at the types of data that a model can collect and export and at methods for understanding model dynamics;
2. Examine how to put a model through its paces to generate the range of data needed for analysis; and
3. Explore how to use NetLogo's BehaviorSpace tool to undertake parameter sweeps in an efficient fashion.

Computational approaches, including GIS and statistical methods, are all used to analyze artificial data in the same way they would be applied to real-world datasets.

Table 9.0. The questions in experiment design and analysis addressed in this chapter.

What kind of artificial data do I need to collect? How is it generated?	section: 9.1
How long should I run my model for? When and how should it stop?	section: 9.1
How to collect and export data? Should it be done at every step, at regular intervals, or at the end of the simulation run?	section: 9.1
Do I need to run two alternative scenarios (scenario comparison) or a range of slightly different scenarios (parameter sweep)? How will I vary my parameters? What ranges do I want to test?	section: 9.2
How many times do I need to run each scenario?	section: 9.2
Do I need to run all parameters, or can I hold some constant?	section: 9.2
How can I analyze the relevant data? And how quickly can I learn R or Python?	section: 9.3
Do I need to know why the results are the way they are? How can I explain the process behind the recorded trends?	section: 9.4
Does my model exhibit emergence?	section: 9.5
How do I document the code and what do I need to include in the publication?	section: 9.6

We will be talking about some important methods in the experiment-design phase of simulation development, such as sensitivity analysis, calibration, and full parameter sweep, also known as the exploration of the phase space. A set of well-designed experiments can make or break a model, so it is important to get it right if we want to present robust results to a scientific audience (Lorscheid, Heine, and Meyer 2012).

scenario: a particular combination of parameter values used to set up the model.

run: one iteration of a scenario.

sweep: shorthand for "parameter sweep."

Before we address these questions, let's quickly run through some of the terms we use in this chapter. A **scenario** is a singular setup of the model with set values for each of the parameters—think of it as a precisely defined world. With different values of parameters, this world will be slightly different. A **run** is one iteration of a scenario. If your model is stochastic, you will be conducting multiple runs of each scenario, each with its own random seed (fig. 9.0). Finally, a **sweep** is a set of multiple

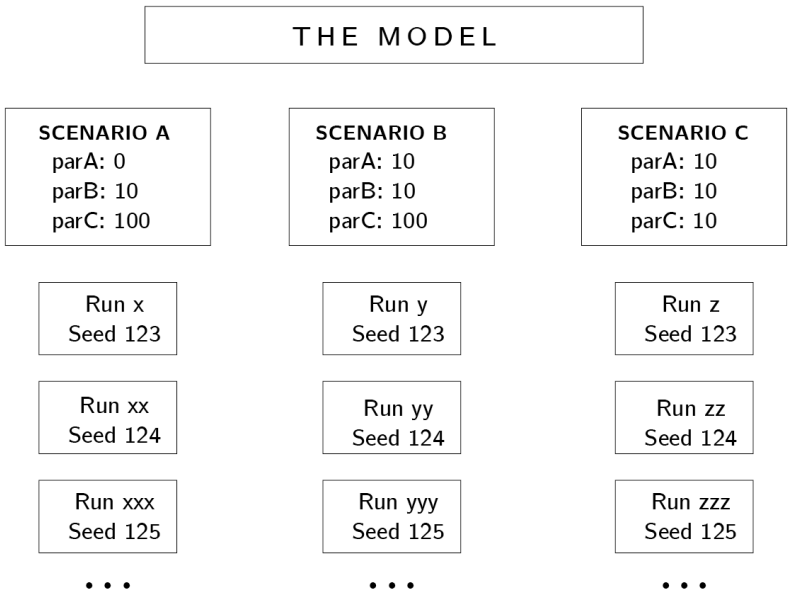


Figure 9.0. The relationship between a scenario, a run, and random seeds. One scenario represents the artificial world governed by specific parameter values. For each scenario we perform multiple runs, which use different random seeds, to simulate multiple alternative but equally possible trajectories of the model.

scenarios, each with slightly different parameter values, repeated over multiple runs to account for a model's stochasticity.

We will also demonstrate some basic steps in data visualization using three separate data analysis frameworks, so that we can generate publishable figures. Finally, we will talk about the fundamentals of *open science* and why it matters to be as transparent as possible in one's research process.

Note that some authors (somewhat confusingly) use "a simulation" to mean "one run of the simulation."

One of the most important aspects to writing a model is the peer review and publishing of your code so that it can be reused. In this spirit, throughout the book we have used agent-based models published by our colleagues in archaeology and beyond. Publishing one's code is important, as it helps in model refinement and development and in demonstrating the ways that models (of any kind) can and should be reviewed by other scientists. With each iteration of this process, we build on previous work, advancing our understanding of the past and helping to develop better methods—ultimately leading to better science.

9.1 Running Models

One author (CW) once spent a month designing and experimenting with many different toy landscapes and output measures to tease apart a confusing part of his model's dynamics. The experiment-design phase can take up to 40% of the total time of a modeling project.

ABMA Code Repo: *ch9_AmphorABM_analysis*

At a certain stage in the research process, there comes a point when you need to stop the primary code development and actually run the model, and run it a lot. The goal at this stage is to fully understand the dynamics of the model by manipulating the input parameter values and observing how the model reacts under a wide variety of scenarios. We discussed parameterization, where you select specific values or ranges for each input parameter, in chapter 6. Before we examine how to undertake this process more systematically, we will examine how to collect and export different types of data from your model.

The Gaulish wine model that we introduced previously, *AmphorABM* (Crabtree 2016),[1] is a good test case because it began with a simple archaeological observation: namely, that in late Bronze Age southern France, Etruscan wine amphorae were replaced by Greek amphorae. The model tests what aspects of trade, population dynamics, and/or wine preference could have influenced that transition in pottery types. To test this, the author generated a range of different types of model outputs and compared them to the archaeologically observed patterns.

WHAT DATA TO COLLECT?

Our first step in examining *AmphorABM* will be establishing what kind of data we want to collect from the model. Do we want to count the number of people, the amount of wine they produce, or the number of amphorae they store it in? Are we interested in the spatial distribution of Greek versus Etruscan winemakers? Or perhaps we need to record the rate of change? Unfortunately, there are no right and wrong answers here. However, in most cases, the two guiding criteria for collecting artificial data are research questions and the archaeological data available to us.

DON'T FORGET
You'll need to generate some artificial data for validation purposes as well.

If the initial question is, *What was the maximum amount of wine that could be produced in this area under certain conditions?*, then we need to calculate how much wine our agents produce. If we asked, *How quickly would Greek wine replace Etruscan wine if people had different levels of preference for the former?*, then recording the proportion between the two

[1] You can find all code written in this chapter in the ABMA Code Repo: https://github.com/SantaFeInstitute/ABMA/tree/master/ch9

over time, or the time step at which there is more Greek than Etruscan wine available, is the right way forward. Bringing the archaeological record into play may help us here. If we need to know *Under what conditions is the density of viticulture in different parts of the studied area consistent with the archaeological record?*, we will need to record the presence/absence of winemakers on each patch and its changes overtime.

This is also a good moment to consider the quality of the archaeological record and the biases it inevitably involves. Can we depend on the spatial distribution of the archaeological record we are comparing to? Or is that distribution mostly a result of a recent road construction? Is the number of amphorae a good proxy of the amount of wine produced in the region? Or are we missing all of the local wine because it was stored in perishable containers? Sometimes it is possible to overcome some of these issues by focusing on relative rather than absolute numbers. If there is no reason to suspect that the proportion of local vs. exported wine changed over time, then amphorae can be treated as a relative proxy for the total production. Often, the only data patterns that are robust enough are presence/absence or general trends such as increase, decrease, sudden change, or a plateau. This is obviously not ideal, but ultimately we need to work with the data that is available to us at the current moment rather than trying to guess what future discoveries may bring. If the patterns of the archaeological record change significantly in the future, so will our interpretations regarding the processes that led to its creation, leading to more opportunities to build agent-based models.

Sometimes you can use one data pattern to normalize another. For example, contrasting the number of households vs. the number of amphorae could give us a rough wine consumption per capita.

HOW LONG SHOULD YOU RUN THE MODEL?

The first important point of experiment design is: How do you know when to stop the model? Most models are stopped either after an arbitrary number of ticks (e.g., after 500 years), or when the model has reached a point of equilibrium. Arbitrary stop conditions are usually justified with the research context. For example, in *AmphorABM*, the Romans arrive after 500 years, changing the dynamics, so it makes 500 ticks a reasonable stopping place.

point of equilibrium: a moment in the simulation run when the model stabilizes and nothing new can be learned from it.

A **point of equilibrium** is reached when the model comes to a fixation, that is, does not exhibit any new behavior. A rule of thumb is to stop the model when we can no longer learn anything new from it. It might be quite

TIP
Population dynamics often cause the death of all agents under a wide range of parameters. Don't be alarmed when this happens in your model.

clear when this happens —because, for example, all agents have died—but often it is a bit subjective. In the *Wolf–Sheep Predation* model from chapter 6, for example, equilibrium is reached when the oscillations between predator and prey numbers stay within a certain magnitude; they do not ossify at one number but fluctuate relative to each other in a stable state. This model may never stop, because under this condition it is always *changing*, but we aren't learning new information from it.

This state—of a model fluctuating consistently around a stable mean value or increasing until infinity—is fairly common. Quantitative analysis can help determine how long you need to run a model in these cases. Rather than just eyeballing it, ten Broeke, van Voorn, and Ligtenberg (2016) suggest calculating the **coefficient of variation** to determine when a model run has gone long enough that the full variance has been captured. To calculate coefficient of variation (CV) you divide the standard deviation by the mean of a specific output measure. However, to assess variance and mean of each population, we will need a long-term record of their population sizes. In the Wolf–Sheep Predation model we could create a new `sheep-pop` global variable, initialize it as a list in `setup`, and add the value of `count sheep` to this list each tick using `lput`. The plotting command would thus be:

coefficient of variation: the variance divided by the mean of a variable.

CODE BLOCK 9.0

```
plot standard-deviation sheep-pop / mean sheep-pop
```

The sheep population dynamics oscillate widely and somewhat unpredictably with default settings, and vary per run due to the stochastic elements of the model. However, the CV of `sheep-pop` is nearly always stable by the 2,000th tick of the model. Again, this feels a little subjective since we are still eyeballing the point when the CV of `sheep-pop` becomes stable, but we can add another 500 ticks to be sure that the model is no longer changing beyond this point. In most models where the resource landscape is a limiting factor, that is, there is a carrying capacity, the population size will eventually find an equilibrium point.

DON'T FORGET
While NetLogo will always export a PNG image, we have to write in the .png extension as a part of the filename.

HOW & WHEN TO EXPORT THE DATA?

Now that we have designed the run and decided on the artificial data, we need to extract that data from the model. There are several methods to export output data from a NetLogo model. You can export the VIEW or any plot by right-clicking on it. For repeated export, NetLogo offers a variety

of primitives that usually start with `export-` and require a filename as a parameter. To export an image of the map, just use the primitive `export-view`:

CODE BLOCK 9.1

```
export-view "my_map.png"
```

Using the primitive `word`, we can also string together a combination of text and parameter values to come up with a more useful and meaningful filename:

CODE BLOCK 9.2

```
to export-my-data
  export-view (word "harvest" harvest-amount
  "_winepreference" weighted-trade-choice "_"
  date-and-time ".png")
end
```

When `word` has more than two parameters, you must wrap the whole string in parentheses. The parts without quotes are parameters, but the value of those parameters will become part of the filename, so `"harvest" harvest-amount` would become, for example, `harvest20`. If you're exporting multiple plots from the same scenario that have the same parameter values and therefore the same filename, you need to come up with a unique name; otherwise, they will overwrite each other and you'll be left with the image of only the last run. The `date-and-time` variable provides uniqueness since each run finishes at a slightly different time.

You can also use the primitive `random` to generate unique filenames.

Any plot that is a part of your NetLogo interface may be exported with `export-plot "plotname" "filename.csv"`, where `"plotname"` comes from the NAME box you filled in when you created that plot. Note that plots are exported as actual data in comma-separated values (CSV) format, not as an image of NetLogo's plot. We will come back to how to visualize and analyze these data later in the chapter. For now, just make sure that your filename ends in .csv.

To change the plot name, right-click on it and choose EDIT. Write the new name in the NAME box.

You can export data during the run or once the simulation is finished. Asking the model to create outputs takes quite a lot of computational power, as we saw using PROFILER in chapter 7, so limit your output to the end of a run whenever possible.

You could call the `export-my-data` procedure:

- at any point with a button on the INTERFACE;
- every tick by putting it at the end of the `go` procedure;
- every n number of ticks using `remainder` in `go`; or
- at the end of the simulation within a stop condition.

Perhaps you want to get a sense of how something like the agents' spatial distribution changes over the course of a run, but every tick would be excessive. In this case, add the following to the end of `go` to export the view only every 100 ticks:

CODE BLOCK 9.3

```
if remainder ticks 100 = 0 [export-my-data]
```

Change 100 to 5, 10, 1,000, or whatever frequency of output works best for your model.

If you only need to export data at the end of the simulation, do it within a stop condition in the `go` procedure. Make sure that the export commands come before `stop` or the model will stop before exporting anything.

CODE BLOCK 9.4

```
if ticks = 500 [
  export-my-data
  stop
]
```

To generate output directly from the code, simply use the `file-open`, `file-print`, and `file-close` primitives to create your own exportable data table.

To have the best of both worlds, you can record a value throughout the simulation and then export that data at the end of the run. For example, in chapter 3 we coded in the creation of the archaeological record over the course of the simulation. There was no need to export anything per tick; we just waited for the occupations to accumulate and then examined one map at the end. In *AmphorABM*, we might be interested in the total amount of amphorae created during a run. For this we just need to add a new `total-amphora` to `globals` and, within each procedure where farmers produce their wine, add a line to increase the `total-amphora` count accordingly (i.e., `harvestWineEtruscans`, after `set EtruscanWine...`, `add set total-amphora total-amphora + 10`, then repeat for the

Note that this is equivalent to exporting a line plot of `plot total-amphora`.

Greeks. This counter will then accumulate over the course of the run, and we can export that variable's value once at the end of the simulation. Alternatively, you could initialize a list `globals [total-amphora-list]` and `fput` the total number of amphorae to it at every tick by calling the following in `go`:

CODE BLOCK 9.5

```
to collect-data
  set total-amphora-list fput total-amphora
  total-amphora-list
```

At the end of the run, you'll have a record of how many amphorae were used at each time step.

Be aware that the setup of a model often creates an initial **burn-in** (or **warm-up**) phase that is quite unlike the middle of a run, and therefore may not be representative of your model's dynamics. For example, the landscape might be completely untouched or all the agents might be at age 1. This may unrealistically increase harvest rates right at the start of a run or negatively affect reproduction dynamics since no agents are of age to reproduce yet. In *AmphorABM*, you may notice a rapid death of Etruscans at the 84th tick (assuming default settings). This is caused by the Etruscans all being born on the day of their arrival and having the same life expectancy (Etruscan-arrival + life-expectancy = 84).

warm-up/burn-in period: the opening phase of the model's run when the dynamics can be affected by the initialization.

We often discard data from the burn-in period when we collect the long-term output variables. This is easy to do with a simple conditional `if ticks > 1000 [collect-data]`. Again, how long of a burn-in period to use will require some experimentation and a thorough understanding of your model's dynamics.

TIP
Initializing variables with random values (e.g., random age between 0 and 60) helps to limit the impact of initial conditions.

A rolling list, using a combination of `fput` and `but-last`, will also discard the burn-in data.

9.2 Running Experiments with BehaviorSpace

We talked about parameterization in chapter 6, and how to use the literature to select an appropriate range of values for each parameter. But once you have several parameters, each with their own range of values, you need a systematic way of testing them all. This is called a **parameter sweep** and is usually run by holding most of your parameters constant while incrementing the value of one parameter at a time, which you then repeat for all the other parameters. To do this manually would be time-consuming and la-

parameter sweep: systematically testing a range of values for several parameters, i.e., scenarios to determine the range of outcomes.

borious, so we will use NetLogo's BehaviorSpace tool, found in the TOOLS menu. It allows you to systematically run your model through a parameter sweep and collects the input and output values into a table for export. A sweep of your model's parameters is necessary to fully understand the dynamics of the model and to evaluate how the inputs are connected to the model outputs. The multidimensional space of all possible states of the model is known as the **phase space**. It is rarely possible to exhaustively sweep the parameters to reveal the full phase space of a model; instead researchers usually focus on the most promising area. We will first describe how to perform a parameter sweep and how to produce plots from the output, then we will return to describing the process of analyzing and interpreting the results of your simulation in sections 9.3 and 9.4.

phase space: often multidimensional space of all possible states of the model.

The first step to using BehaviorSpace is to make sure that the parameters you're interested in are not **hard-coded** in the code but are part of the INTERFACE as sliders, choosers, and switches. For example, instead of calling `crt 10`, you can have a slider `init-agents` and use `crt init-agents`. BehaviorSpace only uses the parameters from the INTERFACE to run experiments, so it needs that slider. If BehaviorSpace tries to adjust a variable that is hard-coded using `set` within the CODE tab, NetLogo will overwrite it or report an error. For example, in *AmphorABM*, the proportion of patches that are mines is currently hard-coded in the `setup` and needs to be replaced with a slider.

hard-coded: entering a parameter value directly into the model's code rather than as an adjustable parameter.

TIP

The experiment name will end up in the default filename for the output files, so be more creative than "experiment1."

In *AmphorABM*, open BehaviorSpace (fig. 9.1) from the TOOLS menu, and create a new experiment. BehaviorSpace automatically lists all the INTERFACE parameters with their current value in the form of, for example, `["weighted-trade-choice" 50]`. A guide below the parameters box shows you how, for each parameter, you can do any of the following:

- Enter a single value: `["weighted-trade-choice" 50]`.
- Run for specific chosen values: `["weighted-trade-choice" 25 50 75]`.
- Increment up through a range by a set amount: `["weighted-trade-choice" [0 10 100]]`, where the additional brackets include <start value>, <increment>, <stop value>.

DON'T FORGET

Every additional parameter multiplies the runs needed. Ten iterations of eleven runs is 110 runs and 1,210 runs with a second parameter added. This number tends to go up really quickly.

Recall that we use <...> to indicate that you should select an appropriate value to enter here.

Assuming all other parameters only have one value selected, the first scenario in the above list will only run the model once, the second will

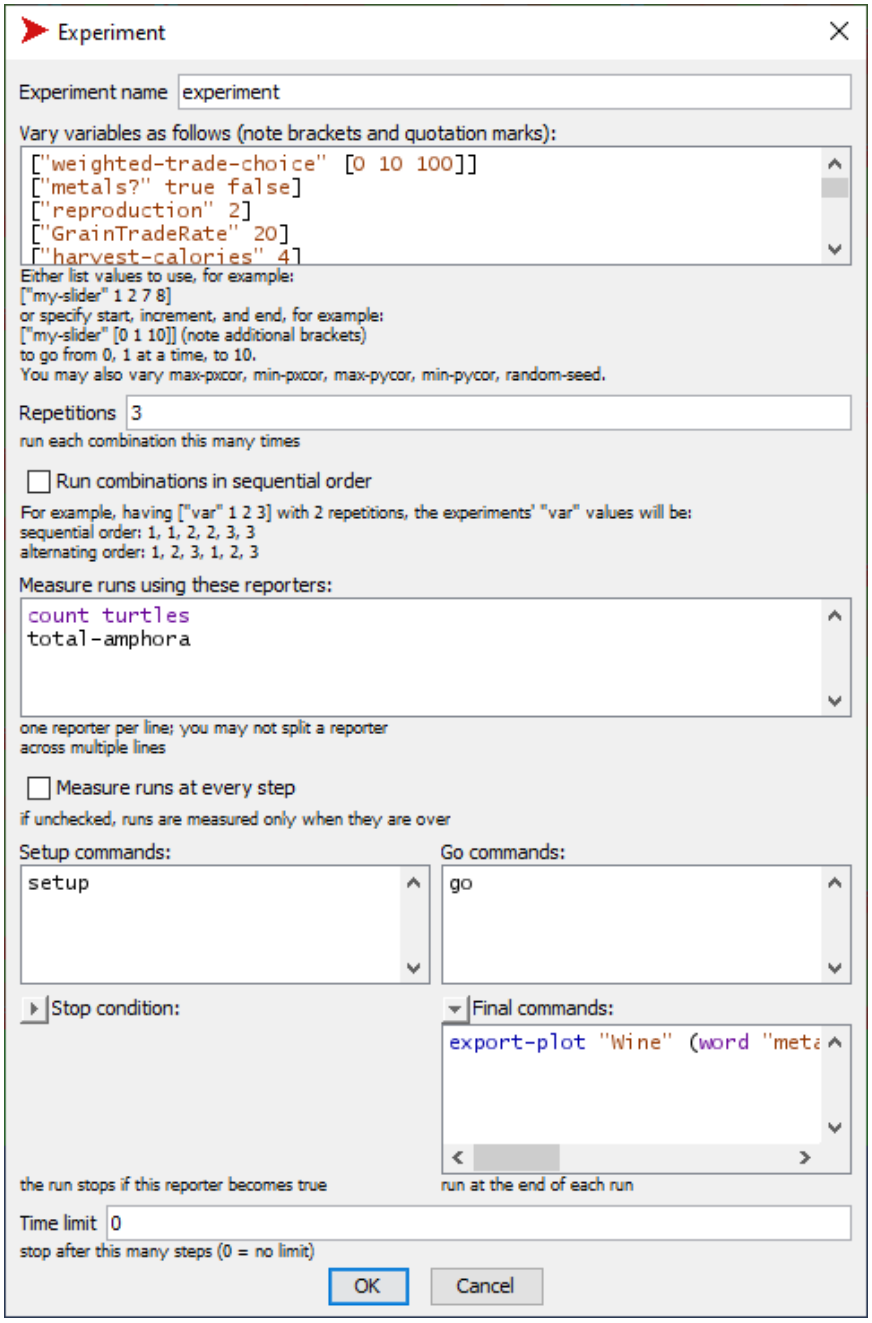


Figure 9.1. Example BehaviorSpace EXPERIMENT dialog box for AmphorABM. The top two variables show different ways of setting parameters for different runs of the model. "Metals?" lists the variable values, while "weighted-trade-choice" has extra square brackets to set up an incrementally increasing variable. Note also the boxes for REPETITIONS, MEASURE RUNS..., and FINAL COMMANDS to export a plot to a CSV file.

run it three times, and the third will run it eleven times (i.e., from 0 to 100, increasing the value by 10 each time). If you then choose to vary a second parameter, it will run through both with every combination for each. If your model is stochastic, you will also have to run it multiple times to establish the central tendency and variance in the outputs. Enter the number of iterations, such as 10, in the REPETITIONS box. Make sure SETUP COMMANDS has `setup` and GO COMMANDS has `go`. For reproducibility, you should include a **seed** for the random numbers generator as one of the parameters. This can be easily done with the `random-seed` primitive and a slider to give the generator the initial input.

seed: an initial number input to a (pseudo)random number generator used to allow for reproducibility in stochastic models.

Check out the **Random Seed Example** in the NetLogo Models Library.

NetLogo leaves the output measures entirely up to you, but it includes `count turtles` as an example. The output measures can be anything that is structured as a reporter, each placed on a separate line. For example, you could just enter `total-amphora` on the line below `count turtles`. Since we might modify the length of the run and will still want to compare the `total-amphora`, we may choose to switch that to `total-amphora / ticks` to get an average number of amphorae created per tick. You can also define a reporter in the main code as a `to-report` procedure and then call it from BehaviorSpace.

TIP
If you define a reporter, the column header in the output data file will be labeled using the reporter itself.

AmphoraABM has hard-coded stop conditions, but if it didn't, those could be entered in the STOP CONDITION box as `not any? turtles` or by entering `500` in the TIME LIMIT box at the bottom. Likewise, if you did not code export commands per tick or every 100 ticks, you could put commands like `export-view` into the FINAL COMMANDS box, again each on a separate line. Make sure to use `word` with your parameter values to create unique filenames for your exported files. If you have multiple runs of each scenario, add `behaviorspace-run-number` to the filename so that each iteration does not overwrite the previous ones.

`behaviorspace-experiment-name` is another primitive useful for filenames, if you plan to use the same directory for the output files of several experiments. However, the REPETITION NUMBER box does not have an associated variable.

Click OK to save your experiment, save your model as well, and then run your experiment. Before it starts, you have the option to turn off the view, plots, and monitors, which will speed up your experiment's runtime. We recommend TABLE OUTPUT, which we'll use later in the chapter for a brief analysis, though both TABLE and SPREADSHEET OUTPUT contain the same data.

For really large numbers of runs, use multiple computers to run BehaviorSpace many times simultaneously. Merge the outputs afterward, but be careful not to overwrite files.

Lastly, NetLogo will auto-detect the number of processing cores in your computer and recommend that you run your experiment's runs in parallel across all of these cores to minimize the time needed to conduct the experiment.

HOW MANY TIMES SHOULD YOU RUN A SCENARIO?

Earlier, we simply put 10 in the REPETITIONS box, but how many iterations of a scenario are necessary depends on how much variance your model produces. You may have to do a few trial sweeps to assess that. In general, it's better to run the model more times rather than fewer. In most cases, you will be looking at the order of 100 rather than ten runs per scenario.

TIP
If you want to use your computer while your experiment is running, leave one core open so you still have processing capacity left.

The coefficient of variation procedure described in section 9.1 may also be used to determine how many repetitions of each scenario are needed to capture the range of variation in the model's dynamics. In this case, the analysis would have to be conducted outside of NetLogo, since we cannot compare the results of multiple runs within a run. However, we can use BehaviorSpace to run a large number of runs that export a final value, then calculate the coefficient of variance of these multiple runs to see when adding more runs no longer modifies the CV, as done by Gravel-Miguel et al. (2021, check the supplementary information for details).

MODEL CALIBRATION & SELECTION

A full parameter sweep may be computationally impossible depending on the number and range of your parameters, the complexity, or the runtime of your model. The first strategy to deal with this problem is to go through a time of reckoning and reduce the parameter space to those that are actually relevant. Each additional parameter you include increases the dimensionality of the model, making the analysis a magnitude more complex. A second, and complementary, approach is **calibration**. This method involves using data to evaluate whether certain parameter values produce reasonable output values (Romanowska 2015b). Most archaeologists should be familiar with how tree ring sequences are used to create calibration curves for radiocarbon dating. Again, we know that the Gauls did not all die, so any parameter range that leads to their extinction can safely be ruled out, as can

calibration: using an external data source to reduce parameter ranges to a narrower and more plausible set.

reproduction rates that lead to implausible population densities. Very robust parameter estimation and calibration techniques can be found in modeling literature, and we recommend getting to know at least a few of them (Thiele, Kurth, and Grimm 2014).

While we recommend always running a parameter sweep to make sure you understand the model's dynamics, your research questions may also require running alternative scenarios representing different hypotheses. For example, you can run a scenario in which different types of wine merchants trade with each other, and another one in which they only trade with their own kin. Scenario comparison can be done in BehaviorSpace with, for example `["metals?" true false]`, or by creating a few specific experiments.

Use the DUPLICATE button to copy the original experiments, then change only the required value to ensure that you have correctly set up the new experiments. Introducing errors while manually entering parameter values is terribly easy.

Before moving on to the next section, run *AmphorABM* with `["weighted-trade-choice" [0 10 100]]`, `["metals?" true false]`, and three repetitions per parameter combination in BehaviorSpace. Make sure to uncheck the MEASURE RUNS AT EVERY STEP box since we only need to collect results at the end of the simulation. For the output reporters, include `count turtles` and `total-amphora`. Also export the Wine plot at the end of each run using a filename that includes the varying parameter values and `behaviorspace-run-number` so that the plots do not overwrite each other (fig. 9.1).

9.3 Working with BehaviorSpace Output Data

Both BehaviorSpace export formats—spreadsheet and table—are saved in the portable, open, and plain-text comma-separated values (CSV) file format. The table format is structured in two parts. The top six lines provide a header with the name of the model and experiment, the world dimensions, and the date and time of the run. The main table is organized such that each column is an input parameter or output variable (input comes first) and each row represents one run of your model.

The spreadsheet format is known as *long* data format, while the table one is *wide*.

A full treatment of data visualization and analysis techniques is beyond the scope of this book. However, there are a few standard data analysis packages and several common plot types that you'll likely come back to again and again. Here we will cover Microsoft Excel, R, and Python, showing how to create an x—y scatter plot, a bar plot, and a line graph.

For a more extensive treatment of exploratory data analysis, we recommend Wickham and Grolemund (2017) and McKinney (2017).

MICROSOFT EXCEL

Among quantitative social scientists, Microsoft Excel (and the derivatives such as Google Sheets or LibreOffice's Calc) receives a lot of derision, but it remains nearly ubiquitously available to researchers and students alike and is particularly good at quickly tossing together a plot to display some data. It is unlikely to be an efficient solution once the simulation output multiplies into numerous folders full of separate data files, so most modelers move relatively quickly toward scripting languages (Davies and Romanowska 2018).

At the 2019 CAA (Computer Applications in Archaeology) conference, a debate broke out about whether Excel should be taught to archaeology students at all, as it encourages bad habits that go against open-science principles.

Excel can usually open a CSV file correctly from the FILE menu, or sometimes just if you double-click on the file. However, if there are extra commas (e.g., inside quotes or parameter names) in any of the columns, the resulting spreadsheet will be all jumbled. If this happens, clean out the extra commas in a text editor before opening your CSV in Excel.

An Excel update changed how the DATA menu's FROM TEXT import feature works. This new method now formats the data columns oddly, which makes it more difficult to plot your data. R and Python are generally more concerned with maintaining backward compatibility than Excel, but make sure you can adapt to these software changes.

We have varied `weighted-trade-choice` and `metals?` as the experiment's input parameters, so we now want to plot one of those against population size to see what the effect may have been. Use your mouse to drag select just the header and data in the `weighted-trade-choice` column, then hold CONTROL (COMMAND on Mac) to also drag select the COUNT TURTLES column the same way. Once your columns are selected, find the INSERT SCATTER (X, Y) button (SCATTER on Mac) in the INSERT menu, CHARTS section. Pick the option with no lines and Excel should create your plot with `weighted-trade-choice` as the x-axis and your final population size on the y-axis (fig. 9.2). Interestingly, this plot shows that as the preference for Greek wine increases, there is a positive effect on the total population size (`count turtles` includes Gauls, Etruscans, and Greeks). It is unclear why that might be, but further simulation and exploration of this effect would certainly be warranted.

Recall that since each row is the result of a single run, each point represents the final state of the model for one run. If there is little stochasticity in your model, you may notice replicate runs clustering together.

We have just plotted both the `weighted-trade-choice` scenarios together, but we might be interested in whether metal mining had an impact on population size. For this we will create a paired bar plot, which is a little more difficult in Excel. First, select the all the data columns and rows (i.e., without the six header rows), and then use the SORT button on

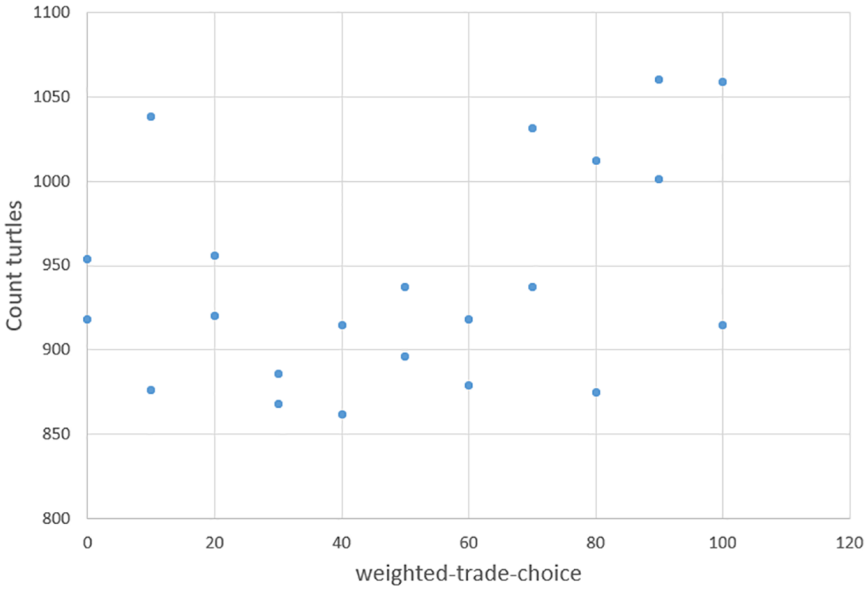


Figure 9.2. Scatter plot made in Excel looking at the effect of "weighted-trade-choice" on agent population size. Each run is shown as a single point. *y*-axis range has been adjusted to better show the data pattern.

the DATA menu to SORT BY the `metals?` column. Next, select just the values of `count turtles` that have `metals? = false`. In the INSERT menu, choose CLUSTERED COLUMN BAR CHART. On Mac this is CLUSTERED COLUMN in the 2D COLUMN chart category. In the CHART DESIGN menu, click SELECT DATA, and add a second data series using the ADD (+ on Mac) button. Click the icon beside SERIES VALUES (Y VALUES next to the box with SERIES on Mac), and drag to highlight the `count turtles` values associated with `metals? = true`. Click OK and you should see your bar chart. If you like, go back into SELECT DATA and edit the HORIZONTAL AXIS LABELS by selecting the `weighted-trade-choice` values. Your paired bar chart has one color for true and another for false, so that you can evaluate if `metals?` had any effect on the resulting population size (fig. 9.3). Although `metals? = false` seems to result in a larger final population size for the higher `weighted-trade-choice` runs, the effect is quite small—more runs and more interrogation of that effect would be needed before we can decide if it is significant.

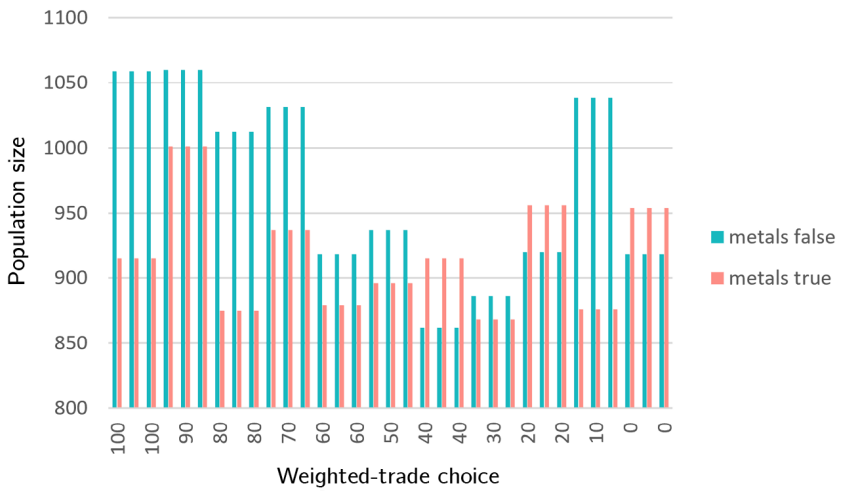


Figure 9.3. Excel's paired bar chart. Each pair of runs is shown as two bars with color indicating whether "metals?" is set to true or false. Replicate runs are each included as a separate bar. Note the adjusted y-axis range to highlight the difference in bar heights.

Last, we will take a look at the wine storage curves that were such a central subject of Crabtree (2016). Open any one of the CSV files that were exported from the wine plots. Again, there is a header section that contains the model name, settings, and the plot and pen settings. Below are the x and y values for each pen that the plot used to record data. Repeat the procedure from above to create a SCATTER (X, Y) plot of the Etruscan wine, this time as a line-style plot, and then add the Greek wine as the second series. All this has done is replicate the plot that NetLogo created during the run but in a format that is a little easier to manipulate for publication.

open science: a philosophy of research transparency in science. Advocates for open-access publishing, open-data sharing, and transparent methods.

open source/open code: a pillar of the open-science movement that encourages the publication of all scientific code, including analysis scripts.

R & PYTHON

Of this book's authors, two use R and one uses Python for their data analysis. More and more archaeologists are adopting one of these programming languages, particularly as more workshops are held at major archaeology conferences, introductory articles are written, and the **open-science** movement gains steam and encourages **open-source** approaches (Marwick et al. 2017). The scripted approach to data analysis using R or Python allows for improved transparency and replicability of published

research. This is contrasted with point-and-click software like Excel or IBM's SPSS, which may perform similar analyses or produce similar figures, but the methods are not easily replicable or transparent. It is also difficult to imagine Excel operating efficiently over multiple files generated by the simulation or creating tens or even hundreds of graphs necessary to fully assess the results. A simple rule of thumb is that if you estimate that the data analysis will take you a week of copy-pasting and figure-making in Excel, you'll be better off spending that time learning how to conduct your data analysis in R or Python.

ABMA Code Repo:
ch9_Python.ipynb
ch9_R.R

Here we will show you a simple pipeline for the analysis of NetLogo output with R and Python commands so that you can choose your preferred language. To replicate the analysis we just performed in Excel, first we need to install R or Python. If using R, download and launch RStudio.[2] If you prefer Python, or want to try, both download the Anaconda Distribution,[3] launch Anaconda Navigator from your START menu, and choose either RSTUDIO for R or SPYDER for Python (you can also use JUPYTER NOTEBOOK for both). Here we will just give the code to perform our analyses, but there are numerous textbooks to help you deepen your skills, some of which we list in the Further Reading section at the end of the chapter. Make sure your data file ("AmphorABM experiment-table.csv") is saved in the same folder as the script. For the code blocks below, first we show the R code, then the Python code for the same analysis steps.

Jupyter Noteboook is a simple yet powerful tool that enables you to blend code with description and figures. It takes mere minutes to get to know it.

R CODE BLOCK 9.6

Most programming languages have a comment character. R and Python use #, while NetLogo uses a semicolon (;).

```
# If you do not have the packages listed, uncomment
# the next line and run it once
# install.packages("ggplot2","dplyr","readr")
library(readr)
library(ggplot2)
library(dplyr)
simdata <- read_csv("AmphorABM experiment-table.csv",
    skip = 6)
```

[2] https://www.rstudio.com/products/rstudio/download/

[3] https://www.anaconda.com/distribution/

And in Python:

PYTHON CODE BLOCK 9.7

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
sns.set()
simdata = pd.read_csv('AmphorABM experiment-table.csv',
skiprows = 6)
```

The first few lines upload a few libraries and set up the environment. We then read in the data as `simdata` while telling R/Python to skip the first six lines with the header. Then we make our first plot in R:

R CODE BLOCK 9.8

```
ggplot(simdata,aes(`weighted-trade-choice`,`count
  turtles`)) + geom_point()
```

And in Python:

PYTHON CODE BLOCK 9.9

```
sns.scatterplot(x = "weighted-trade-choice",
y = "count turtles", data = simdata)
```

The first line calls the `ggplot` command, tells it which data to use, and sets up the *aesthetics*, or organization, in an `aes(X,Y)` format. Added to this, on the next line, is the geometry of the plot telling it to be an x–y point scatter plot. The Python script calls the scatter function to do the same, and takes the name of the columns to be plotted on the x and y axis, plus the data. Assuming all your code was written correctly, you should see that the resulting graph is the same as the one we generated with Excel.

Note that here R uses the grave accent character (`) when column names have spaces or special characters, not single or double quotes.

Next, we create the paired bar plot. In Excel, replicates were plotted separately and it was a bit fiddly to add the extra data series, but R/Python handles all of this for us:

R CODE BLOCK 9.10

```
ggplot(simdata,aes(`weighted-trade-choice`,
`count turtles`,fill = `metals?`)) +
geom_col(position = "dodge")
```

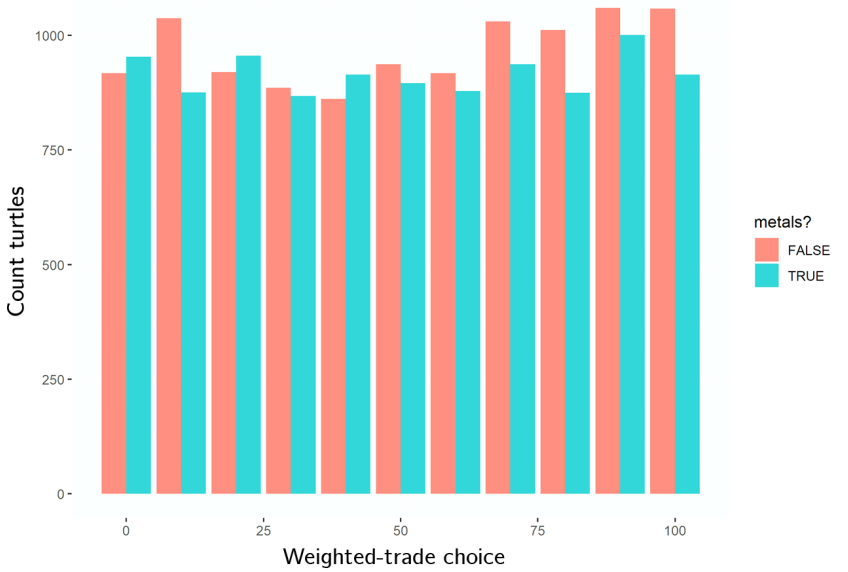


Figure 9.4. Bar plot made in R looking at the effect of "weighted-trade-choice" and "metals?" on agent population size. In contrast to Excel, R and Python automatically average the replicate runs.

And in Python:

PYTHON CODE BLOCK 9.11

```
sns.barplot(x = "weighted-trade-choice", y =
"count turtles", hue = 'metals?', data = simdata)
```

In R `Stacked` is used when the combined bars are parts of a whole and `dodge` when you need to compare scenarios.

Again, we call the plotting command `ggplot` and specify our data. The aesthetics are mostly the same, but we have added `fill` to group the bars by the additional data column `metals?`. Next we specify the geometry as a bar plot using `geom_col(position = "dodge")`, where "dodge" is used to draw the bars side-by-side rather than the default stacked. The Python code uses the barplot function specifying the columns to be used for the x- and y-axes and the parameter `hue` which is equivalent to the fill aesthetic in R (fig. 9.4).

Lastly, we will redraw the model's Wine plot using one of the exported CSV files. In this case, NetLogo's export format is not great for us, so we'll have to rename some duplicated column names to clarify which x and y belong to which population, and the plot's aesthetics will be set separately for each line (fig. 9.5) in R:

R CODE BLOCK 9.12

```
curve_data <- read_csv("metals_ true_repro3_13.csv",
skip = 17) %>%
   rename(Etruscan = y, Greek = y_1)
ggplot(curve_data) +
  geom_line(aes(x,Etruscan), color = "blue") +
  geom_line(aes(x_1,Greek), color = "orange")
```

And in Python:

PYTHON CODE BLOCK 9.13

```
curve_data = pd.read_csv("metals_ true_repro3_13.csv",
  skiprows=17)
curve_data = curve_data.rename(columns ={"y":
  "Etruscan", "y.1":"Greek" })

sns.lineplot(y = "Etruscan", x = "x", data =
  curve_data)
sns.lineplot(y = "Greek", x = "x.1", data = curve_data)
```

You can change the axis labels with `plt.xlabel(text)` and also add a legend with `plt.legend()`.

While scripting is not the easiest skill to learn, the combination of data manipulation and plotting libraries that can handle multiple input parameters and output variables, even across multiple files, is extremely powerful. The good news is that skills acquired when learning NetLogo will significantly ease the learning curve of R or Python. Once you get the basics working with your models, try using more options such as colors, labels and titles, legends or subplots (ggplot's `facet_grid()` and seaborn's `FacetGrid`), and other plot types to examine even more combinations of parameters simultaneously.

TIP
The stackoverflow.com community is helpful for solving scripting problems big and small.

9.4 Interpreting ABM Data

Creating visualizations of model outputs is the first step. Next we need to actually interpret that data to answer our research questions. In this section, we will review how parameter sweeps can reveal a lot more detail and help to uncover your model's dynamics. To simplify the task here, it is worth returning to the research questions and doing separate analyses

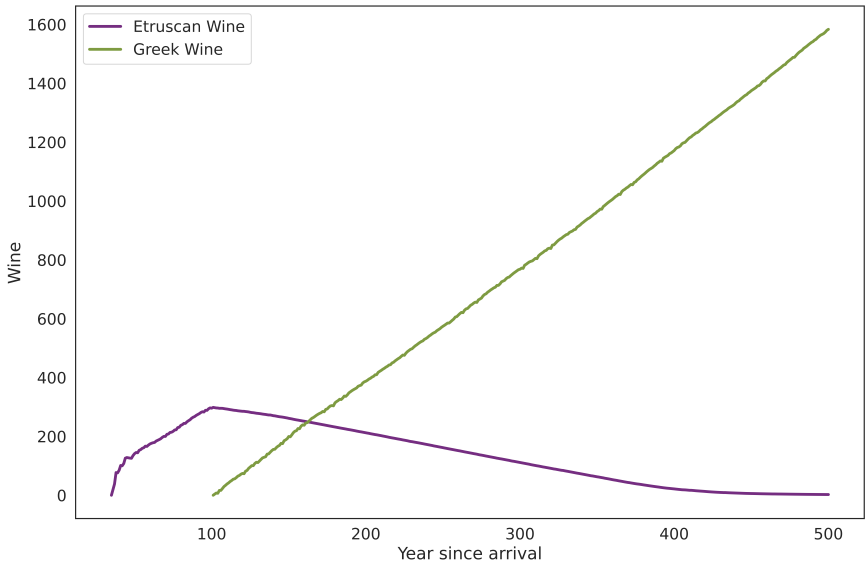


Figure 9.5. Line plot made in Python looking at the change in wine type frequency over a single run. Here "weighted-trade-choice" is set to 100, so Greek wine takes over the market.

with clearly defined parameters and output measures (sec. 9.1) for each of them consecutively.

DETECTING CAUSALITY, SENSITIVITY & UNCERTAINTY

We mentioned briefly earlier that there are distinct stages to the analysis of a model, at least in theory. Ideally, after the model development stage follows the experimental stage, then finally the "official runs" for a study. In practice, we often oscillate between these phases. The experiment phase involves a thorough evaluation of the model's dynamics, which often leads to revisiting the code, algorithms, and even the initial assumptions. To pick up on any issues in the model dynamics, though, we need to run through a wide variety of parameter settings in a systematic way.

sensitivity analysis: running a model repeatedly to determine how and to what degree varying input parameters changes output measures.

A full parameter sweep and **sensitivity analysis** are used to help understand model dynamics, to calibrate the model, and to select a final run set to form the backbone of your study. A wide parameter sweep determines the range of possible outcomes of the model, including any emergent effects (see below). Even though we may see agents creating interesting patterns during a run, exporting and analyzing our data is really the only way to robustly understand emergent phenomena. A thor-

ough understanding of *why the results are the way they are* is necessary for the model to truly have an explanatory value. There are multiple methods to explore the causality (*what is causing what and why?*) of a simulation, but we will focus on the most common ones: sensitivity analysis and parameter-space exploration (Kanters, Brughmans, and Romanowska 2021).

There are several distinct purposes to sensitivity analysis, which are frequently conflated (ten Broeke, van Voorn, and Ligtenberg 2016; Campolongo, Cariboni, and Saltelli 2007). A classic sensitivity analysis is conducted to find out how sensitive a model output is to changes in specific input parameters (Ligmann-Zielinska et al. 2020). That is, how variable are the outcomes of the model overall and which input parameters contribute most to that range of variable outcomes? For example, does increasing the reproduction rate in *AmphorABM* significantly alter the total number of amphorae traded, or does it affect it only a little? It may also be that a parameter has no effect on one output measure, such as the total number of amphorae, but a huge impact on another, such as the population size.

An **uncertainty analysis** is a closely related procedure in which a range of plausible values for a given parameter are evaluated to see to what extent they will affect the outcome of a model if we are not sure which values are valid in the first place. This process is still testing the sensitivity of the model, but for a slightly different purpose: a sensitivity analysis tests the model to see how much a model's output will be affected by incremental changes to various plausible parameter values; an uncertainty analysis tests whether different values that you do not know are true or false will have any impact. For example, we may have ethnographic data suggesting that a variety of agricultural societies tend to have total fertility rates ranging between 6 and 10, but, since we don't know where in this range the Gaulish fertility rates were, we want to know how sensitive our model output is to this range of uncertainty in our input data. If we are lucky, the model output will not be particularly sensitive to parameters we have little certainty about. If it is, we either need to find more evidence to narrow that range further, or we need to factor that range of uncertainty into our interpretation of the model results.

uncertainty analysis: running a model through the uncertainty range of the input parameters to determine the robustness of the model outcomes with respect to that uncertainty level.

The simplest and most common way to perform a sensitivity analysis is by varying parameter values one factor at a time (OFAT) while holding the others constant (ten Broeke, van Voorn, and Ligtenberg 2016). So, for example, we would keep all parameters constant but increase the `weighted-trade-choice` to 25%, 50%, 75%, and 100% using BehaviorSpace. For a complex model with a lot of parameters, this may result in a massive number of runs, but it enables the modeler to peg some of the parameters to constant values and remove them from the experiment design.

The OFAT parameter sweep allows you to determine whether there are linear, nonlinear, or even threshold effects between a given parameter and an output value. As we saw in *AmphorABM*, increasing `weighted-trade-choice` seemed to cause a linear increase in total population size (when `metals?` was false), whereas increasing the reproduction parameter would cause an exponential increase in population size. Crabtree (2016) identified a threshold effect where below a `weighted-trade-choice` of 30 (meaning the Gauls preferred Etruscan wine 70% of the time), the Greek population would not survive at all. These examples highlight that the same input parameter may have completely different effects on each output variable depending where on the scale it is (e.g., no effect at 25 but a massive effect at 31).

Such nonlinear relationships are primary results of modeling studies because they can only be uncovered in a formal modeling environment.

While the need for a thorough parameter sweep should be clear at this point, it is a time-consuming exercise to perform and analyze for complex models with many parameters. In those cases, we turn to parameter space exploration. Stonedahl and Wilensky (2010) used a custom-built tool called BehaviorSearch[4] to help explore the parameter space of NetLogo models. They examined the *Artificial Anasazi* NetLogo replication by Janssen (2009) and used a genetic algorithm approach to explore the effects of parameter changes instead of systematically conducting a full parameter sweep. A genetic algorithm uses an evolutionary analogy of parent and offspring parameter sets with a certain probability of a parameter mutation per generation. The "fitness" of run outputs is determined based on how closely a specific model output matches a predetermined value, for example, archaeological data. In this case, they compared the historically recorded population size to the simulated pop-

[4] https://www.behaviorsearch.org

ulation size. The closer the parameter set was to the real data, the higher its probability to be selected for the next generation of testing. In this way, Stonedahl and Wilensky (2010) focused on the areas within the parameter space that gave a better fit to the data, thus saving a lot of computational time.

Genetic algorithms must be repeated multiple times with different starting places within the parameter space, as there may be multiple local fitness optima within the parameter space.

There are several other techniques based on the same premise of random sampling of the parameter space to determine more or less promising areas, for example, Approximate Bayesian Computation (ABC) (Carrignon, Brughmans, and Romanowska 2020; Kovacevic et al. 2015). However, these methods should be used with caution to avoid overfitting, that is, zeroing in on a specific set of parameter values only because they are a tiny bit closer to the benchmark data while missing out on other sets that might be almost equally informative. A way to avoid overfitting is to use at least two independent datasets/types (e.g., change over time and spatial distribution like in ch.3) as the benchmark.

See ten Broeke, van Voorn, and Ligtenberg (2016) for other ways to explore parameter space and to more efficiently examine the sensitivity of models.

9.5 Emergence

In the introduction, we mentioned **emergence** but did not describe it in depth. **Emergent phenomena** refers to the quality of many complex dynamic models for which the aggregated interactions between individual entities yield surprising results (Epstein 1999). A somewhat related concept is the **self-organized criticality**, which Costopoulos (2018) simply describes as "unusual changes do not require unusual causes." Complex systems can unexpectedly cross a threshold, which results in a complete transformation of the system despite the fact that nothing extraordinary happened to it and the agents engaged in the same mundane activities as before. ABM straddles different scales of analysis—for example, we often model fine-scale individual behaviors, then record their population-level patterns—making it a primary tool for detecting emergence in many branches of science.

emergence: a property of model dynamics where an unexpected effect arises from the aggregated interactions of the individual entities, i.e., when the whole becomes more than the sum of its parts.

self-organized criticality: a phenomenon in which a system undergoes a transition from one semi-equilibrium state to another without any external influence.

In reality, most agent-based models result in a set of outcomes that emerge from dynamic agent–agent and agent–environment interactions that may be difficult to predict or assume without a formal model. This is similar to the patterns we observe in the archaeological record that are an aggregate effect of many individual people making choices in their lives.

There is a lot of academic debate over the exact definition of "emergence" (see Miller and Page 2009, ch. 4).

Epstein (1999) argued that "emergence" stands for temporary ignorance as we had failed to predict the consequences of the modeled dynamics prior to running the model. Once we understand the chain of causality that led to the population-level behavior, we may label that explanation as "obvious" or "trivial," but it's worth remembering that we did not see it when we started. For example, in chapter 7, we saw emergence in the way that a random walk can lead to a diverse lithic toolkit. Similarly, the *AmphorABM* model in this chapter demonstrated how individual preferences resulted in a society-wide pattern of material culture replacement. This ability to predict the trajectories of complex systems and understand their dynamics sets them apart from chaotic systems. Chaos, in complexity parlance, indicates that the system is extremely sensitive to initial conditions; that is, it can fall into dramatically different trajectories depending on a minute detail at the start. It is sometimes said that complex systems reside at the "edge of chaos" (Lewin 1999).

chaotic system: a type of dynamic system that is highly sensitive to the initial conditions. Chaotic systems, for example weather, are unpredictable beyond the horizon of predictability determined by our ability to measure the initial conditions.

Most of the models in this book have some form of emergent phenomena. However, this is not necessary for all agent-based models. Even if a model has no or little emergence, or the results turn out to be negative (as in they show that your hypothesis does not explain the data well), this is still a step forward. The exercise of building a model formalizes our ideas about it and lets us identify the main gaps in the current knowledge. Discarding hypotheses is valuable in itself, as it allows us to move on to new (and better) ones. Communicating all results, even the ones that are not that surprising, is an important step in creating a scientific record and advancing our understanding about the world (Crabtree et al. 2019).

9.6 How and What to Publish

Supplementary information files in ABM papers are often much longer than the publication itself.

After writing a comprehensive model, running a parameter sweep, and creating multiple plots, the time has come to turn the model into a publishable research package and to communicate its results to various scientific audiences. As with any research study, there are difficult choices involved with deciding what to include in an article and how best to structure that paper to communicate results effectively. Further, a successful article will likely have two types of audiences: context specialists and other modelers (Romanowska 2015b). Thus, the paper is a synthe-

sis, and needs to be accompanied by a technical description, the finished model code, a folder full of output files, and the analysis code.

To begin with, the article should aim to increase the understanding of the research questions to modelers and nonmodelers alike. Lake (2010) expressed frustration with the low citation rate of ABM papers in archaeology beyond the domain of other ABM papers. This seems less true today, but highly technical papers full of specifics of coding and data tables are unlikely to be understandable to anyone who is not already well versed in ABM methodology. In fact, some of the most successful model-based papers spend very little space on the model details, instead focusing on the research question, the context, and how the results help to increase the understanding of the research topic (e.g., Brughmans 2015; Premo 2010; Dean et al. 2000). This is not to say that model details do not need to be made available to the reader, but it is common practice for ABM papers to include code and a comprehensive description of all technical aspects of that code in an appendix, a separate code repository, or a preprint server. This helps to limit the model description in the main text to just the components necessary for the reader to understand the model dynamics while still fulfilling the open-science goals of transparency of methods and replicability of results (Marwick et al. 2017).

Repositories most commonly used by modelers include github.com, comses.net, figshare.org, zenodo.org, and socarxiv.org.

In general, a publication of an ABM should include two major sets of information: what you have done and why you have done it.

First: What have you done? Think of the model's code description as the methods section of the paper. The method is not just "we used an agent-based model" but "we simulated a fixed population of 1,000 agents with unbiased cultural transmission of farming-related cultural traits." A good description of the model is crucial for enabling readers to assess the scope and robustness of the results. While the paper still needs to include an adequate description of the model and its dynamics, it is sometimes impossible to predict all doubts that may emerge in the readers' heads.

Making code (model and analysis scripts) available with a paper already at the peer-review stage may also go a long way toward satisfying reviewers' concerns.

Second: Why have you done this? In the methods section, you will need to justify the algorithms used and coding decisions you made. That one place where you just put in a value of 10 for the length of the cultural trait list, rather than any other number? You will have to justify that choice. If you used a correlated random walk instead of a fully ran-

dom walk or targeted movement, justify that too with references to previously published models, ethnographic research, evolutionary theory, or the published archaeological hypotheses that you are testing. It is paramount to include and justify all aspects of the model that a reader or reviewer needs to know in order to assess your results and conclusions.

A common and standardized format for describing both the model and the intentions behind it is the ODD (Overview, Design concepts, and Details) Protocol (Grimm et al. 2010; Grimm et al. 2020). It has been widely adopted in archaeology and other disciplines. Müller et al. (2013) expanded on the original ODD Protocol to include extra sections for human decision-making that may be useful for specific applications. It is generally a good practice to have an ODD attached to the code but not necessarily included as the main text of an article. ODD Procotol descriptions are commonly included in supplementary information and/or as a file with the code in a separate repository.

PUBLISHING CODE

In our experience, openness and transparency involved in making code available has been unanimously commended by academic reviewers and, later, by readers of the published work. Including model code alongside a paper means that others can build on your work, not just indirectly by being inspired by your ideas and citing your work but literally by modifying and adding additional code to your model.

Building your model from an existing simulation is efficient, reduces error incidence, and makes the results easier to communicate.

For example, Wren and Burke (2019) built on a previously published model by Barton et al. (2011) , converting it from a model of Neanderthal and *Homo sapiens* dynamics to a model of interregional interaction among multiple populations of humans approximately 20,000 years later during the Last Glacial Maximum. They also used the GIS extension (ch. 7) to add a heterogeneous habitat landscape that affected agent mobility and survival. In earlier chapters, we mentioned other projects that were inspired by, or directly based on, earlier publications of code, such as the Piaroa Swidden model by Riris (2018), or even the foundational *Artificial Anasazi* (Dean et al. 2000) built on the *SugarScape* model (Epstein and Axtell 1996).

To make this all possible, it's important to publish the code in a format that guarantees its long-term sustainability. The code should be well

documented with inline comments and clarifications and use meaningful variable names. In addition, every published model should:

- Have a digital object identifier (DOI) and a citation file, so other researchers can easily refer to your work and give you credit when they use it;
- Come with a license to make it clear to everyone how can they use the code (whether they need to cite the original, can use it for commercial purposes, etc.[5] It's a good idea to also have a `Read-me` file that spells it out in simple language); and
- Be updated occasionally to make sure it continues to work as major NetLogo updates are released.

See this guide on how to mint a new DOI for your model: guides.github.com/activities/citable-code.

TIP
Adding a license to a GitHub repository is as simple as ticking a box.

For further information on best practices in computational science and sharing code, see the reviews by Eglen et al. (2017) and Stodden and Miguez (2014).

Effective communication of results also means having appropriate and clear figures in your paper. It's unfortunate that so few journals have the capacity for video clips of example model runs, interactive R plots,[6] or even full ABMs embedded as web applets. Mostly, we are still limited to including NetLogo's exported map views and static plots exported from our data analysis packages. That said, a well-designed plot can sometimes communicate the complex dynamics and output of a model far more effectively to a nonmodeler than a textual description or even a video of a particular run. In addition, Crabtree et al. (2019) gives a detailed description of how to communicate to a broad range of audiences outside of the traditional academic journal articles using the visual interest of ABMs, their natural dynamic storytelling nature, and the ability of audiences to interact directly with them.

Internet Archaeology (intarch.ac.uk) is a notable exception in its inclusion of 3D models, interactive maps and figures, video, and other nontraditional media.

9.7 Summary

This chapter has covered a lot of the complexity of turning a functioning model into something that rigorously produces output data, is analyzable,

[5] Creative Commons has an easy-to-follow tool to determine which license you need: creativecommons.org/share-your-work.

[6] See, for instance, https://shiny.rstudio.com/ and https://bokeh.org/.

and can be published. However, we have only touched on the vast field of data science that is dedicated to the effective analysis of complex datasets, such as ABM output tables, and the creation of useful data visualizations. But it's not enough to learn to code an ABM; now you must become a data scientist as well. This chapter provided you with the keys to begin this journey and reminded you that at this point your skills in programming in NetLogo are transferable to many other computer languages. Begin by using the code provided here, but start increasing your literacy in R, Python, or another language suitable for data analysis. Eventually, you may decide to write your ABMs in them as well. It is not uncommon for the number of runs necessary to fully analyze the results to explode, necessitating that you deploy supercomputing resources (known as HPC, or high-performance computing). These are actually not as difficult to access as you may think, since most large universities belong to networks granting HPC access to their researchers. The native BehaviorSpace application can be cumbersome in HPC environments but the OpenMOLE framework will allow you to package your NetLogo code to be ready for a supercomputer.

Find out more about OpenMOLE at openmole.org.

Above all else, remember that while writing and running a model can be fun, the reality is we are using these tools to examine scientific theories. To get the most out of your model, you'll need to strictly adhere to the principles of scientific practice such as well-defined research questions, proper experiment design, parsimony, and transparency.

End-of-Chapter Exercises

1. Go all the way back to chapter 1 and run a full parameter sweep of the Y&B model, varying growth-rate and distance moved per generation. As your output variable, use the time needed to arrive to Southeast Asia. Upload the results to your software of choice and analyze which factors speed up and which slow down the dispersal. Now change the output to a point in northern France and repeat the analysis. Are the results similar?

2. Design a new BehaviorSpace experiment to replicate the results included in Crabtree (2016). From the text of her article, figure out what the input parameters were and what output data she needed, then plot out some of the results to compare to her published figures.

3. Find a couple of the models online that we mentioned in the book. Look at the paper, model documentation, code, and additional documents associated with it. You can even attempt to replicate one of them from their ODD and/or code if not originally a NetLogo model.

Further Reading

- D. L. Carlson. 2017. *Quantitative Methods in Archaeology Using R.* Cambridge, UK: Cambridge University Press. doi:10.1017/9781139628730

- S. A. Crabtree. 2016. "Simulating Littoral Trade: Modeling the Trade of Wine in the Bronze to Iron Age Transition in Southern France." *Land* 5, no. 1 (February): 5. doi:10.3390/land5010005

- A. Downey. 2018. *Think Complexity: Complexity Science and Computational Modeling.* Second edition. Beijing, China: O'Reilly.

- A. Downey. 2016. *Think Python.* 2nd edition, updated for Python 3. Sebastopol, CA: O'Reilly Media.

- V. Grimm et al. 2020. "The ODD Protocol for Describing Agent-Based and Other Simulation Models: A Second Update to Improve Clarity, Replication, and Structural Realism." *Journal of Artificial Societies and Social Simulation* 23 (2): 7. doi:10.18564/jasss.4259

- F. Karsdorp, M. Kestemont, and A. Riddell. 2021. *Humanities Data Analysis: Case Studies with Python.* Princeton, NJ: Princeton University Press.

- I. Lorscheid, B.-O. Heine, and M. Meyer. 2012. "Opening the 'Black Box' of Simulations: Increased Transparency and Effective Communication through the Systematic Design of Experiments." *Computational and Mathematical Organization Theory* 18, no. 1 (March): 22–62. doi:10.1007/s10588-011-9097-3

- B. Marwick et al. 2017. "Open Science in Archaeology." *SAA Archaeological Record* 17 (4): 10–14. https://osf.io/3d6xx/

▷ J. Salecker et al. 2019. "The NLRX R Package: A Next-Generation Framework for Reproducible NetLogo Model Analyses." Edited by L. Graham. *Methods in Ecology and Evolution* 10, no. 11 (November): 1854–1863. doi:10.1111/2041-210X.13286

NOTES