1 Introduction

Social interactions are rich, complex, dynamic – and endlessly fascinating. One way to understand them is to design and build theoretical models of interactions that fascinate us. Some of the more realistic and powerful models are computer simulations. The good news is that simple, elegant, and powerful tools are available right now to help you design and build your own simulations of the social world. Even better news is that these tools are easily available in user-friendly free software. This software allows you design, build and run your own theoretical model of any social interaction that intrigues you. And you can do all this on the most basic laptop computer. The ability to build powerful computer models of social interaction, once the preserve of a privileged few, has moved down to street level. This Element is about how to unleash that power to develop and explore your own theoretical ideas about how the world works.

Computer models of social interaction typically analyze the behavior of autonomous artificial agents. Each agent is a simulated human decision maker with actions determined by computer code written by the model builder – by you. Each agent has its own brain (referring to agents using the gender-blind “it”) and each agent’s brain is designed by you! In simple models, you might program all your agents’ brains to have exactly the same response in the same situation, as if the real humans you model were all clones of each other. Since real humans are not all clones of each other, however, you can easily program different types of agent to respond to the same situation in different ways. Such diversity is at the heart of human interaction. You will already be well aware that real-world social interactions typically evolve into settings where very different types of people coexist side by side.

After you have programmed your agents’ brains, specifying their responses to changes in their environment and encounters with other agents, you hit the “go” button and set your model running. You can then sit back and observe, record, and analyze the resulting social dynamics. Computers never get tired, so you can go to bed and run the model overnight if you want to see how things evolve over the very long run. You can run your model for weeks until your laptop smokes and glue runs out of its seams.

The beauty of all of this is that you, the model’s author, have great freedom to develop and explore your own theories of social interaction, programming your agents to behave in ways you think are interesting and instructive. You can also take anyone else’s theoretical model and adapt it to take account of your own ideas. This is why what is now known as agent-based modeling is so downright addictive – pure fun that at the same time allows you to investigate theoretical
2 Quantitative and Computational Methods for the Social Sciences

ideas about social interaction in a way that goes far beyond the capabilities of most alternative approaches.

My aim in what follows is to seduce you into designing and programming your own agent-based models (ABMs) of social interactions that fascinate you. We’ll get your models up and running, then see how to exercise them in a careful way that lets you draw confident conclusions about social interaction. Everyone is a theorist in their own head, but the gold standard is to develop theoretical models that are widely accessible to others yet rigorous in the sense that different people, using the same model in the same way, see the same results.

My approach in this Element is hands-on. You’ll be encouraged from the very start to design, program, and run theoretical models that address social interactions you care about. The modeling platform we use to do this is NetLogo, a stable and intuitive programming environment, available as a free download at http://ccl.northwestern.edu/netlogo. It’s also important that NetLogo is public domain, open source software (https://github.com/NetLogo/NetLogo). Code files for all named NetLogo models (set in bold text) referred to in this Element are available at www.cambridge.org/laver1. NetLogo’s greatest virtue is that it is both simple and powerful. Some who work with ABMs are coding ninjas who like work as close as possible to the bare metal of the machine, valuing raw speed above all else. But a core value of this Element and of NetLogo is that ABMs are for everyone and should be as transparent and accessible as possible. The computer code powering them must be legible to anyone with just a modicum of effort. Having built a shiny new model of some important social interaction, what you want above all else is for as many people as possible not only to admire it, but to use and understand it. You want others to modify, improve, extend, and, yes, criticize it. Your ideal is that your theoretical model provokes vigorous and wide-ranging discussion. Agent-based modeling is transformative precisely because it puts a powerful modeling technology into the hands of so many people — democratizing social theory in the process. It should be inclusive, not exclusive, and the way to achieve this is to make tools that are as accessible as possible. NetLogo does this.

Depending on your interests and expertise, you may eventually run up against the limits of NetLogo and feel the need for fancier software. If you like to code in Python, for example, you’ll find a Python ABM framework at https://pypi.org/project/Mesa/. I’m a simple person, holding the view that if it’s too complicated for NetLogo, it’s too complicated. For many reasons we’ll discover later, fancier models are often worse models. I wrote a book-length treatment of party competition as an evolving, complex system using nothing but NetLogo. NetLogo will get you a very long way, and in my view its very simplicity encourages you to specify simpler and better models. Furthermore, if you ever
feel you’ve reached the end of the road with NetLogo, you’ll be well able to make your own software choices, and I will wish you all the best.

Plan of Campaign

This Element is the first of a two-part introduction to agent-based modeling. This part will hopefully take you from zero to ninety in agent-based modeling in as fast and friendly a way as possible. It assumes nothing about your prior experience with either modeling social interactions or writing computer code. Old hands at doing these things might be tempted to skim early sections. My humble suggestion is to read them anyway because agent-based modeling is different from other approaches to modeling social interaction, and writing code for agent-based models is different from other types of computer coding.

My aim is to have everyone writing NetLogo code for agent-based models as fluently as if they were writing in their own native tongue. I want you to be thinking about the theoretical model and the problems it addresses, not the code. This involves lots of hands-on work. We’ll get our hands dirty right away by analyzing, torturing, and extending an early and influential model of social segregation designed by Thomas Schelling. The core intuition from this model is that, if we have only a very mild preference for having people “like us” as neighbors but move house when we’re unhappy with our neighborhood, then the resulting social interactions lead to the evolution of startlingly high levels of social segregation. We find ourselves living in much more segregated neighborhoods than we want. The Schelling model is simple and powerful – the best type of model – and you’ll find a faithful rendition of it in the NetLogo models library.

Given its simplicity, accessibility, and power, I use the Schelling segregation model throughout the next five sections as a vehicle for learning about many aspects of designing and programming ABMs. In the next section we look at how the model was coded in NetLogo and play with it a little to get some basic intuitions. This is as far as many people get with agent-based models, treating them essentially as discovery toys. That’s not a bad thing, but we’ll be going a great deal further. In Section 3 we move beyond casual play to deploy the segregation model as a serious research tool. We do this by specifying a suite of computer simulations designed to exercise and investigate the model, analyzing and presenting the results of these simulations, and drawing well-founded conclusions you would be happy to publish to the whole world.

4 Quantitative and Computational Methods for the Social Sciences

After we have gotten serious with the baseline segregation model in Section 3, Section 4 moves on to consider how we might improve and extend it. The baseline model everyone has been talking about for years has only two social groups, treats all agents as clones of each other, and considers only hyperlocal social neighborhoods. What if there are more groups? Or if different agents have different preferences? Or if social neighborhoods are more extensive? Would these improvements change longstanding core intuitions from the Schelling model?

The great beauty of ABM is that it’s a simple matter to modify and reinterrogate this famous model in search of systematic answers. When you’ve read previous theoretical accounts of social interaction you may well have thought, “What about this, or that?” Now, using ABM, you can derive your own rigorous answers to such questions.

Indeed, this is why agent-based modeling is such a potent tool for anyone interested in theorizing about social interactions. It’s one thing to develop a nice, simple, and plausible theory about some social interaction and then speculate about ways this might be extended. But the great advantage of computationally powered theory is that you must be very precise about theoretical extensions before you can turn these into computer code. And you will often find that the theoretical extension you were thinking about is rather woolly and vague, looking quite different when you make it precise enough to code. The need to program your agents’ brains in a very precise way is a great source of discipline when you theorize about social interactions.

We will quickly find, as we extend it, that the baseline Schelling model is very path-dependent. In particular, as soon as all agents are happy with their locations, no agent ever moves again and model output reaches a stationary state. Each time agents stop moving because they’re all happy, however, the configuration of their locations, the stationary state, is somewhat different. The real social world is a much more uncertain place, in which no stationary state persists forever. We take account of this in Section 5 by adding a random, or stochastic, component to the segregation model. This models the plain fact that at least some people move house for random reasons that have nothing at all to do with being unhappy with their local neighborhood. For example, they may get a new job in another city, cross the country to care for a sick relative, or move for many other reasons. We model this randomness by giving agents some small probability of looking for a new location, even when they’re happy with their local neighborhood. Adding this stochastic component changes the model in fundamental and desirable ways. Not only does it make the model more realistic but, since some agents are always moving, the model never stops in an arbitrary location. Since each agent now has some small probability of moving to any location in the NetLogo world, the model becomes much less path-dependent. Left running for a very long time, all
run repetitions tend to converge on the same general outcome, no matter what the starting configuration of agent locations. It may seem counterintuitive when you first think about this, but adding some randomness to your model in this way is a good thing, because it can increase the tendency of outputs from different model run-repetitions to converge on each other. And this allows you to be more confident about the insights you derive from the model.

Up to this point, we extended the Schelling segregation model by adding new features in a well-disciplined way, one at a time. Each time we specified a new feature, we added this to the baseline model on its own so that we could carefully assess the effects of doing this. If what we want is the most realistic model we can achieve, however, we might well want to add many new features at the same time. We conclude this introduction to agent-based modeling, in Section 6, by combining several innovations we discussed into a single, more realistic model of social segregation. We quickly see that, while first instincts might favor deploying the most realistic model we can specify, more realistic models typically have more moving parts. Models with more moving parts, especially when these interact with each other, are typically much harder to understand. They may therefore be less fertile sources of intuition. If, however, we want our model to help us estimate something important about a particular real-world setting, such as where a particular hurricane is most likely to make landfall, we above all want our model to be as realistic as possible. We may then well be prepared to pay a price for this in lost intuitions about the interactions we are modeling. We conclude this Element, therefore, by thinking about a trade-off between realism and intuition, which lies at the very heart of the modeling enterprise.

I finish this section with an observation that may seem geeky, but which informs these Elements and may interest those concerned with the distinction between computational theories of social life, like the ones we explore later, and the formal or logical theories more often found in economics or political science. There is a longstanding argument linking computer science and logic, known as the Curry-Howard isomorphism. Simply put, this says that a computer program is essentially a logical proof, and a logical proof is essentially a program for deriving a result. Both are sets of logical statements that imply a particular outcome or outcomes. Google this topic and read up on it if this argument interests you, but the overall point is not to obsess about this distinction. Contrary to what some who don’t know much about ABMs may think, an ABM is an analytical model of the world, logically resolved by the computer code and research design that express it. If you take ABM seriously, you will generate results every bit as true as propositions derived mathematically from formal models. These Elements are about helping you do that.
2 Social Segregation: Basics

The Core Intuition

When people belong to social groups that are important to them for some reason or another, we often see the evolution of local neighborhoods where most people live surrounded by others of the same group. The tendency to associate with similar others, called homophily, can generate a pervasive pattern of geographic social segregation. The surprising and disturbing thing about how this segregation evolves is that, even when people have only a mild preference for living close to others like themselves but move to another location if this mild preference is not satisfied, they often find themselves living in much more segregated neighborhoods than they really want.

Nobel Prize winner Thomas Schelling developed one of the first agent-based models of this important social phenomenon. He did this before the era of personal computers, “running” his model by moving pieces by hand around a large checkerboard that functioned, just like an abacus, as a physical computer. His model adapts very well to modern electronic computers, and there is a faithful implementation (called Segregation) in the NetLogo models library. Look for the models library in the NetLogo “file” menu and find the Segregation model, along with many other nice models, in the library’s “Social Science” folder. This model is beautiful, being both very simple and very powerful, and will therefore be a real pleasure for us to use as a vehicle for learning about ABM.

We’re going to learn by doing, so let’s find out about the Schelling segregation model by jumping right in and running it. Download the model from the NetLogo models library. Open it. Hit the “setup” button. The top panel of Figure 2.1 shows what you see. The big multicolored square to the right of the screen is the NetLogo world. In this model, the world represents a geographic area, say all or part of a city. The little squares inside it represent places, called patches in NetLogo, where people might live. Black patches are unoccupied. Agents belonging to one of two social groups, red and green, live on colored patches. There are approximately equal numbers of red and green agents, and when you hit the “setup” button you scatter these randomly around the patches in the NetLogo world. Keep hitting “setup” and you see a different random scattering of red and green agents, but always about the same number of each.

Now hit the “go” button and set the model in motion. Watch agents move as they sort themselves into more or less segregated neighborhoods – much more

---

Figure 2.1 Screen shots from the default NetLogo implementation of the Schelling segregation model, before (top panel) and after (bottom panel) a model run segregated than their random start. The bottom panel of Figure 2.1 shows an example of what might happen. (It may happen quite fast on your computer. To see it in slow motion, go to the “normal speed” slider above the NetLogo world.
The greenish-blue slider called \%-similar\text{-wanted} shows that agents in this model want 30 percent of their neighbors to belong to the same social group as themselves. (I distinguish NetLogo code and variables from other text by printing them in a font like this.) The beige \% similar monitor and Percent Similar plot show what happens when agents move house whenever fewer than 30 percent of their neighbors belong to the same social group and keep moving until this preference is satisfied. This is the simple but powerful logical engine of the Schelling segregation model. For the example in the bottom panel of Figure 2.1, people who want just 30 percent of their neighbors to come from the same group as themselves, and move whenever they don’t find this, end up living in much more segregated neighborhoods where about 76 percent of their neighbors belong to the same group. This is Schelling’s core theoretical intuition, one that has been extremely influential in many different settings since it was first published.

Hit the “setup” and “go” buttons until you’re exhausted and you always see something a bit different but nonetheless quite similar, always involving the evolution of segregated neighborhoods. The more times you do this, the more confident you are about the core intuition: people who behave as Schelling assumes sort themselves into much more segregated neighborhoods than they want. This, albeit in a very simple form, is exactly how you derive reliable theoretical conclusions from an ABM. Random things happen. So, outcomes of different model run repetitions, like outcomes in the real world, are never precisely the same. Here, this happens because there are random (stochastic) elements in the model, which we’ll shortly consider. Nonetheless, randomness in both the model and the real world does not prevent us from drawing reliable conclusions about important patterns in social behavior.

We just saw that the NetLogo implementation delivers the core intuition of the Schelling ABM of social segregation. What you really want to do is modify this model to test your own ideas about the evolution of social segregation. First, however, we must find our way around NetLogo. Go back to the NetLogo screens in Figure 2.1. Near the top of each you’ll see three tabs: “Interface,” “Info,” and “Code.” The “Interface” tab is selected, and you’ve already been working with the interface when you hit “setup” and “go.” We’ll be returning in more detail to this. The “Info” tab shows lots of information about the computer model, supplied by the model’s author. Read this to find out what is going on and see suggestions of things you might do with the model. The “Code” tab contains the NetLogo code that runs the model. You can see at once that the language of this code is close to English. Once you get used to it, you’ll read NetLogo code almost as easily as you’re
reading this sentence. Think of NetLogo as a power tool. The interface is what you see and use on a daily basis, the code is the internal (logical) engine, and the “getting started” manual is in the info tab.

We’re now going to break all the rules and tinker with the engine before we even read the manual, just to see how easy it is to code in NetLogo. Look at the NetLogo world on the interface and you will see, immediately after you hit setup and before you hit go, that some little squares have an “x” in them and some do not. The squares with an “x” show unhappy agents who plan to move. The squares with no “x” show happy agents who plan to stay right where they are. This is rather boring, so let’s give happy agents a smiley face and unhappy agents a sad face. This is easy, if rather silly. Click the “Code” tab and peer at the logical engine. Find the procedure called to update-turtles. It’s in some lines of code near the bottom, of which the first is to update-turtles and the last is end. (We’ll soon discuss procedures, which always begin with to and end with end.) Shortly before end you’ll see the following line of code:

```plaintext
ifelse happy? [set shape "square"] [set shape "square-x"]
```

The ifelse instruction is a logical fork in the road and is very useful in both NetLogo and other coding environments. It tells the machine, “If the following condition is satisfied, carry out the first instruction that follows; else, carry out the second instruction.” The condition here is happy? and the code tells NetLogo, “If the agent is happy, show it as a square; else show it as a square with an ‘x’ in it.” Replace square with face happy and replace square-x with face sad. That gives you:

```plaintext
ifelse happy? [set shape "face happy"] [set shape "face sad"]
```

Click the green “Check” button at the top of the code to make sure you haven’t made a mistake. This catches most typos. Go back to the interface tab and hit “setup.” Instead of squares you see happy and sad faces. Hit “go” and see the agents move around until they all have happy faces. You’re now successfully coding in NetLogo! Fresh from this minor triumph, let’s get more familiar with the NetLogo modeling environment. Before doing anything else, however, save your modified model under a new name, such as Segregation 1.01 faces.

**VERSION CONTROL ALERT!**

If you want to save your smiley face model before moving on, use “save as” in the file menu and give it a new name, such as “Segregation model with faces.”
Never, ever, ever, overwrite your original model with code changes! You always want to be able to go back to the original when something goes wrong, and, sooner or later, something will go wrong.

Now, go back to the baseline model you downloaded from the NetLogo models library and make a copy called “Segregation 1.0.” Every time you change this model, save it as “Segregation 1.1,” “Segregation 1.2,” and so on. When you make a really big change that generates a different version of the model, you can proudly celebrate this by saving it as “Segregation 2.0,” “Segregation 2.1,” and so on.

This practice may seem overcautious but, trust me, a time will come when you will be really glad you did this. It will save you untold hours of exasperation and misery.

The NetLogo World

We have already seen the NetLogo world in the Segregation model, the big square to the right of the interface tab. This is a rectangular grid of patches, small squares that combine to make up the world inhabited by NetLogo’s artificial agents. You can give these patches all sorts of properties that define the environment in which your agents live. Different patches may supply different amounts of energy, for example, or different levels of danger or difficulty. We’ll come back to all that later. Depending on what you want to model, this world may be a physical space, as in the segregation model. ABMs often do have a physical geography, distinguishing them from other models of social interaction, which often have no geography. In the Schelling segregation model, agents have physical neighbors – people who live close by – and they are more likely to interact with neighbors than with those who live farther away.

The NetLogo model world can also be a conceptual space. For example, in some of the sections in the second of the two Elements, we’ll think of horizontal and vertical dimensions not as east-west and north-south, but as “left- versus right-wing” preferences on economic issues, or “liberal versus conservative” preferences on social issues. And you don’t need to give your model any geography at all, whether physical or conceptual. Because geography does structure many social interactions in the real world, however, it’s nice that you can easily take this into account in NetLogo. Click the “settings” button over the NetLogo world. Figure 2.2 shows what you see.

We’re mostly interested in the top half of this panel. First, you see that the world is 51 patches wide and 51 patches high, 2601 patches in all. This value is an odd number because positions in the NetLego world in this particular example are