2 Particles

> We have the tools, we have the talent!
>
> —Winston Zeddemore, *Ghostbusters* (1984)

Before we can build something, we need to be conversant with our tools. This chapter introduces the fundamentals of programming agent-based models with NetLogo, after which we will explore a simple toy model called "Particle World."

In this chapter and the next, we'll stick to agent-based models and leave aside purely mathematical formulations. For many people, simulating explicit agents provides greater intuition than mathematical models based on seemingly sterile equations. There's something visceral about seeing embodied agents moving around on your screen, which may explain some of the popularity of agent-based modeling. Many people also find the task of writing down models as equations intimidating. This is a hurdle we will have to vault, but not just yet. Hopefully, the intuitions gained from simulating social systems will provide a scaffold for understanding the equations. If you're raring to start writing down equations, sit tight and enjoy the ride for now. We'll get there soon enough.

This book is predicated on the idea of learning by doing. Therefore, I will not spend too much time on general principles of coding or modeling. Rather, I will try to give you just enough information to get going right away with coding and analyzing some simple models, and then, later, some more complicated models. There will be lots to learn from the models we study, and I want to get us started as soon as possible. In the next two sections, I will introduce some of the basics of NetLogo and some general principles of computer programming. If you are already broadly familiar with both NetLogo and programming, feel free to skip ahead to section 2.3.

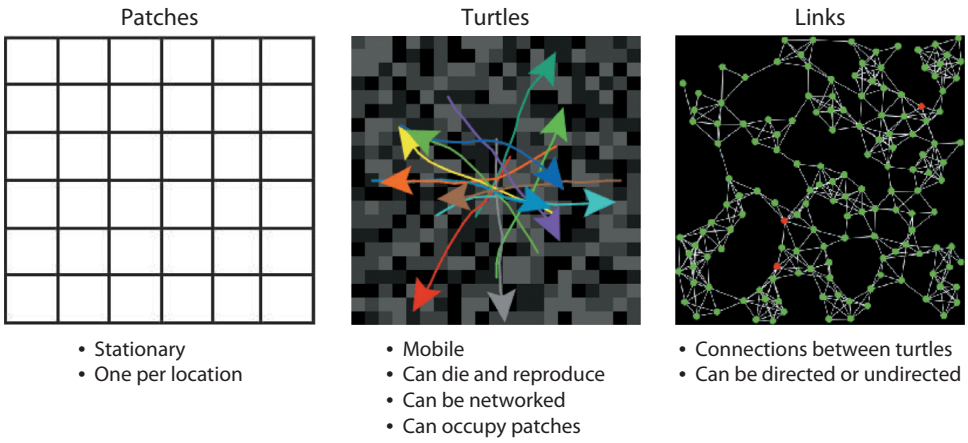

Figure 2.1 The building blocks of a NetLogo model: patches, turtles, and links.

2.1 NetLogo Basics

If you have not done so already, you should download NetLogo onto your machine, which you can do for free here:

https://ccl.northwestern.edu/netlogo/

As noted, this is not a book on how to use NetLogo. You should make sure to spend sufficient time on your own getting familiar with its workings.[1] First and foremost, you should work through the tutorials included in the NetLogo User's Manual. Doing this will give you a sense of how NetLogo works. In addition, the NetLogo Models Library comes loaded with lots of well-documented sample models that will give you a good sense of the program's possibilities. Finally, the NetLogo Dictionary at the end of the User's Manual contains information about all the NetLogo **primitives**—the built-in procedures that make coding models in NetLogo so efficient. This dictionary is a valuable resource. When I work in NetLogo, I refer to it often.

2.1.1 Patches, Turtles, and Links

NetLogo has three important building blocks for agent-based models that have many useful built-in properties: patches, turtles, and links (Figure 2.1). **Patches** are the default spatial organization in NetLogo: a set of discrete square cells laid out in a two-dimensional rectangular grid. Each patch has a fixed, unique location, but otherwise can be assigned arbitrary properties. Indeed, patches can be used as agents themselves in a model, for example when there are a fixed number of agents in stable locations.[2] More typically, patches are used to represent properties of the agents' environment.

When there is need for agents with a more versatile set of properties than those afforded by patches—such as the ability to move, die, reproduce, or form nodes in a network—NetLogo has a useful object class called **turtles**. Turtles will be used to represent agents

[1] If you are using a language other than NetLogo, with or without an explicit modeling library, make sure you are comfortable with it and able to create a simulated world with simple agents in it.

[2] Patches can easily be used to simulate one- and two-dimensional cellular automata.

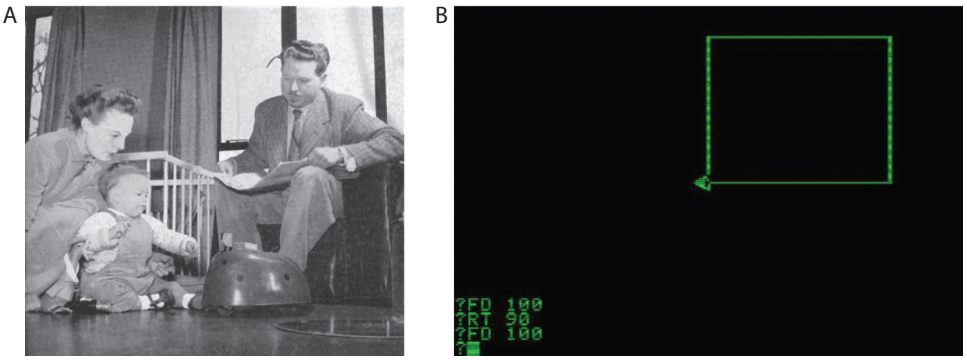

Figure 2.2 (A) W. Grey Walter introduces his cybernetic tortoise to a mother and her child. (B) Logo commands to draw a rectangle with the turtle.

in most of the simulations explored in this book. Turtles are spatially embodied and have a position in continuous space (which overlaps with the discrete space on which the patches exist) as well as a directional heading, though one may also ignore these pieces of locational information if they are not important to a model's dynamics.

Why are these objects called *turtles*? The term is an homage to the neuroscientist and roboticist W. Grey Walter,[3] who created a simple, programmable, mobile robot in the 1950s called a "tortoise." This idea was inspirational to the creators of the Logo programming language—developed in the 1960s—which allowed a user to issue commands to move and draw with a small triangular figure on a computer monitor.[4] The commands were similar to those given to Walter's tortoise, and the green triangle may have also given the impression of a turtle (many screens in those days were monochromatic and produced only green on black; see Figure 2.2). NetLogo uses many of the same commands to move its turtles as did the original Logo language; its agents are named in Logo's honor.

Finally, NetLogo conveniently allows turtles to be connected with the class of objects called **links**. More will be said about the use of links when we discuss network models in chapter 9.

2.1.2 NetLogo Tabs

The NetLogo window has three tabs: Interface, Info, and Code (see Figure 2.5). A NetLogo file contains information about all three tabs, and their union is properly considered the model code. The **Interface** tab displays the model visualizations and graphs, buttons that execute commands for initializing and running the model, and controllers (sliders and switches) for any variables that will be varied in the analysis of the model. You should make sure you are familiar with how to create, delete, and manipulate these aspects. The Interface tab allows for real-time inspection of the model output as well as rapid parameter

[3] Walter worked within the then-trendy discipline of cybernetics, which combined insights from psychology, neuroscience, and the nascent field of computer science. Cybernetics can be viewed as a precursor to the modern fields of artificial intelligence and cognitive science.

[4] Biographical note: Logo was the first language I ever programmed in, as a grade-schooler back in the optimistic heyday of the 1980s. Perhaps this helps to explain my affinity for NetLogo. For a history of Logo, see Solomon et al. (2020).

manipulation. The tab is also where the **Settings** window can be accessed, which allows the user to update the size and shape of the patch grid.

The **Info** tab is essentially a template for extensive documentation of the model and associated code. Nothing in this tab directly affects the run of the model itself. Rather, it is a place to describe the model for the benefit of others who may be using the model (including your future self, who will likely benefit from a refresher). In general, I will ignore this tab throughout the book. However, the library of example models included in the NetLogo download is very helpfully documented using the Info tab.

Finally, the **Code** tab is where the instructions for the model's operation are coded. All the code that will be presented throughout this book is entered in this tab. One thing worth noting is that calling this the "code" tab is somewhat misleading, because the interface is really also part of the code. For example, we will often instantiate global variables in the Interface tab using sliders and switches. How this works will become clear with practice.

2.2 Programming Basics

Once upon a time, a person could obtain an advanced degree in the social or biological sciences without ever typing a line of computer code. Those days are over, and rightly so. There are many skills a scientist can hone, and no one can do everything (at least not well). We create stronger sciences if specialization is allowed to flourish. Nevertheless, programming is simply too valuable a skill to leave entirely to specialists—gaining at least some programming experience is essential for the modern scientist. There are currently many, many resources available to help you learn to code, and it is likely you have already been exposed to some of these. As such, I will generally assume some familiarity with the conceptual language of computer programming. However, for novice programmers, it's also important to make sure that some of the foundational concepts are understood. For this reason, I present below a very short primer on some key programming concepts with examples of how they can be instantiated in NetLogo. Readers comfortable with programming can probably skip ahead to the next section.

Variables

Variables are the secret weapons that give mathematics its power—the ability to perform computations on an object without knowing the value of that object. In computing, variables often represent numbers, but they can also represent character strings, Boolean (true/false) values, or agents, as well as lists or sets of these things. In some languages, like Java or C++, when declaring a new variable you are required to specify what type of object will be represented, be it an integer, a floating point number, or a Boolean array. In other languages, including interpreted languages like Python or NetLogo, you are not required to do this, because the interpreter will automatically figure out what sort of variable it is dealing with.

When we write mathematical equations, we usually represent variables with Latin or Greek letters, like this:

$$y = 0.1x - 1$$

In this case, the variable y is a linear function of the variable x. Perhaps x represents the time you invest in the Skee-Ball game at your local penny arcade, and y represents the value of the knickknacks you can purchase with the tickets you win (the negative intercept is due to the entry cost for the arcade). We represent each value with a single character, because

only philistines use multiple letters to represent a variable (excluding subscripts) in purely mathematical formulations. However, in programming we can use almost any set of characters we want, excluding spaces. And we should. We want our variable names to be both succinct and easily identifiable. Code can be confusing, so let's lessen the confusion by using names that help us figure out what the variable refers to when we show our code to others or to our future selves.

NetLogo uses the equals sign (=) exclusively as a logical test (to check if quantities are equal), and instead uses the function `set` to change the value of a variable. So, in NetLogo, the code to assign the current value of y would look like this:

```
set value-win (0.1 * money-invested) - 1
```

NetLogo code 2.1

Variables are the heart and soul of any model. They represent the quantities we need to describe our study systems, and they represent the features of those systems whose dynamics we are interested in understanding. In an agent-based model, we often distinguish between several categories of variable: global variables, agent variables, and local variables. **Global variables** have a single value regardless of how they are accessed. These can be fixed for the duration of a simulation (such as the initial number of agents) or they can change over time (such as the current number of agents in a model with birth and death events), but their values apply globally in the simulation. **Agent variables** are values held by each member of a class of agents, with values that can vary between agents. For example, each agent might keep track of its group identity and current resource level. Global and agent variables are sometimes collectively described as a model's **parameters**. Within a block of code, it is also sometimes useful to define a **local variable**. This is a temporary marker that is used only within a particular block of code and then automatically deleted.

Functions

A function, also called a **procedure**, is a label for a set of commands that are run together whenever the function is called. A function may also take **arguments**, which are values that are assigned to variables within the function. For example, an addition function called `add(x, y)` might take the two arguments `x` and `y` and add them together, so that the output of `add(2, 3)` would be 5. Functions need not return any values, but they can, and when they do the output of a function can in turn be assigned to some other variable. In NetLogo, functions that return a value are also called **reporters**.

NetLogo has a number of built-in values and functions, called *primitives*. These are part of what makes NetLogo so powerful, because they make it easy to program the computations that make the dynamics of a model possible. For example, the primitive reporter `color` returns a turtle's color, while the primitive function `create-turtles` takes as arguments a number and a code block. The function then creates the given number of turtles, each of which executes the commands in the code block upon being created. There are many useful NetLogo primitives, all of which are described in the dictionary at the end of the NetLogo User's Manual.[5] We will be using many functions, both custom and primitive, throughout this book.

[5] For a useful introduction to NetLogo primitives, see here: https://ccl.northwestern.edu/netlogo/bind/article/what-is-a-primitive.html.

Loops

Looping is one of the major capabilities that make computers so amazing: the ability to perform the same computation over and over and over at great speed. Loops can take several forms, but are often of one of two varieties: **for-loops** perform a computation over a fixed set of values, and **while-loops** perform a computation for as long as some condition is met. For example, a while-loop in NetLogo can be instantiated like this:

NetLogo code 2.2

```
while [any? other turtles-here]
   [forward 1]
]
```

The code block above contains several NetLogo primitives—in fact, every word is a primitive except for the number 1. When a turtle executes this code block, it calls the function `any?`, which takes as an argument an agentset (a set of agents) and returns false if the set is empty and true if the set contains at least one agent.[6] The agentset is defined by the primitive function `turtles-here`, which returns the set of all agents on the same patch as the **caller**—the code object that initiates the function call (generally either an agent or the global observer). The function `other` takes an agentset and returns a copy of the same agentset excluding the caller. Thus, the set of brackets on the top line returns true if there are any other turtles on the same patch as the turtle executing the code, and false otherwise. If true, the turtle will move forward one unit (the width of a patch) in the direction it is currently heading. It will then check the conditional again to see if there are still any other turtles on its current patch, and it will keep moving forward until there are not. In this way, the turtle moves away from others, behaving in a somewhat antisocial manner.

NetLogo additionally has a very special sort of looping command that allows the user to loop over a set of agents in a random order. This is done with the primitive procedure `ask`. The randomization is key, as we usually don't want all of our agents to be scheduled in the same order every time step, because this could create undesirable artifacts in at least some cases. Most agent-based modeling libraries allow for a similar random scheduling of agents' behaviors. NetLogo is a very polite language, so we don't tell agents what to do, we ask them. The `ask` procedure allows us to create a list of instructions that each agent will then execute one by one. For example, suppose we had a population of agents located at various positions on a two-dimensional space and facing a particular direction, and we wanted each agent to turn 45 degrees to the right and then move forward one unit. We could do this with the following code:

NetLogo code 2.3

```
ask agents [
   right 45
   forward 1
]
```

[6] There is a convention among NetLogo coders that Boolean variables and functions that return Booleans have names that end with a question mark. This is not required, as NetLogo treats the question mark as a regular character, but it is often useful.

If agents all start at the center of the space and this command is their only behavior, looping over this ask command will yield our population of agents spiraling outward.

Conditional statements

Conditional statements are another major capability that makes computers awesome. Contingent rules allow for code to be executed only when specific conditions are met. NetLogo provides the function `if`, which takes a Boolean argument and executes a series of commands if and only if the argument is true. If the argument is false, nothing happens. The command `ifelse` is similar, but it involves two sets of commands, executing the first if the argument is true and the second if it is false. For example, consider a scenario in which agents have an energy reserve, stored as a variable called `energy`, and we want the agents to turn slightly to the right and move forward only if they have more than one unit of energy, otherwise they run some custom function called `forage` that we can imagine involves some other meaningful behavior. We can achieve this with the following code:

NetLogo code 2.4

```
ask agents [
   ifelse energy > 1
   [
      right 45
      forward 1
   ]
   [
      forage
   ]
]
```

More recent versions of NetLogo also allow `ifelse` to operate as a "switch" function, which allows the user to delineate an arbitrary number of commands with each to be run under some condition. Building models of behavior very often involves constructing sets of decision rules for how agents will behave under different circumstances. Conditional statements are a key part of making this happen.

Object-oriented programming

Early programs were simple lists of commands, performing sequential operations on stored variables one line at a time. And indeed, many programs today are still of this type. In these programs, variables typically refer to values or sets of values (such as a matrix of integers). **Object-oriented programming** allows for something more nuanced. The user can define an arbitrarily large number of object classes, each of which characterizes new *types* of variables for the program to work with. Each class can store its own variables and procedures, and all instantiations of that class (that is, the objects) will possess these variables and procedures.

This type of programming lends itself very well to agent-based modeling, because it is easy to define a class of agents with the desired properties. NetLogo uses a special type of object-oriented programming called **agent-oriented programming**, which differs primarily in that commands can be written as instructions directly to the agent. This allows for what many consider a more natural style of programming for agent-based models. We saw this above in our use of the `ask` command. A common problem faced by novice NetLogo

programmers is a "confusion of levels," in which a set of code is written at one level—for example, at the level of the observer (the perspective of you, the coder)—but is placed in the code in such a manner that it will be interpreted at another level—e.g., by a turtle. This usually generates an error, but if you're on the lookout for it, it is easily handled.

Through practice, the elements of programming—both those common to most languages and those specific to NetLogo—will gradually become more and more like second nature.

2.3 Particle World

Now that we've introduced our tools, let's get on with seeing how we can put them to work. We are going to build a very simple agent-based model. The goal is to get you comfortable with the tools of model building, so for now we won't worry about how well the model represents reality. Instead, we are going to start with a toy model of a make-believe world: Particle World.

This ignoring of reality may appear to run counter to what I said in the previous chapter about how models, by analogy, help us to understand the real world. However, we must learn to stand before we can run. To do this, we need to engage in a bit of play. Throughout the animal kingdom, young animals (humans certainly included) play with simplified versions of the objects or scenarios they will eventually need to take seriously as they grow into adults. Juvenile meerkats are given injured scorpions with their stingers removed by their mothers in order to become familiarized with the tasty but deadly prey. Young children in the West might play "family" or "cops and robbers" to become familiar with social roles in a low-risk context. Similarly, our first "model" is a sort of playpen, in which agent dynamics can be explored entirely within the world of the model without us worrying about how the model maps onto anything in the real world. These agents will be simple dots moving around on your computer screen. Of course, humans are natural pattern finders and storytellers, so you may not be able to stop yourself from imposing a narrative onto our model. I certainly couldn't.

2.3.1 The Story of Particle World

Particle World is a magical land inhabited by strange but simple creatures called particles, who require no sleep or food, only space to move around. And move they do, *constantly*. The space they live on is a **torus**, a donut-like surface, so a lone particle could happily move around forever. The torus (Figure 2.3) is a commonly used surface in agent-based modeling. We will represent space as a toroidal grid, sometimes referred to as a grid with **periodic boundaries**. This means that if you move past the rightmost edge, you end up on the left side, and if you move past the topmost edge, you end up at the bottom. If you've ever played a game of Pac-Man you will be familiar with this concept—when Pac-Man exits the maze through an opening on one side of the maze, he emerges on the opposite side. The grid is *toroidal* rather than a true torus because each cell or patch is presumed to be a perfect square with an area equal to that of all the other cells.

Particles vary in their propensity to wander around in new directions. Particle World psychologists have determined that the various cultural groups of Particle World exhibit varying levels of the personality trait they call "whimsy," which is the extent to which individuals fail to stick to their current directional heading (Figure 2.4). The opposite of whimsy is stubbornness. A stubborn particle heading due north at one moment in time will still be heading north in the following moment, barring a collision. A more whimsical particle, on

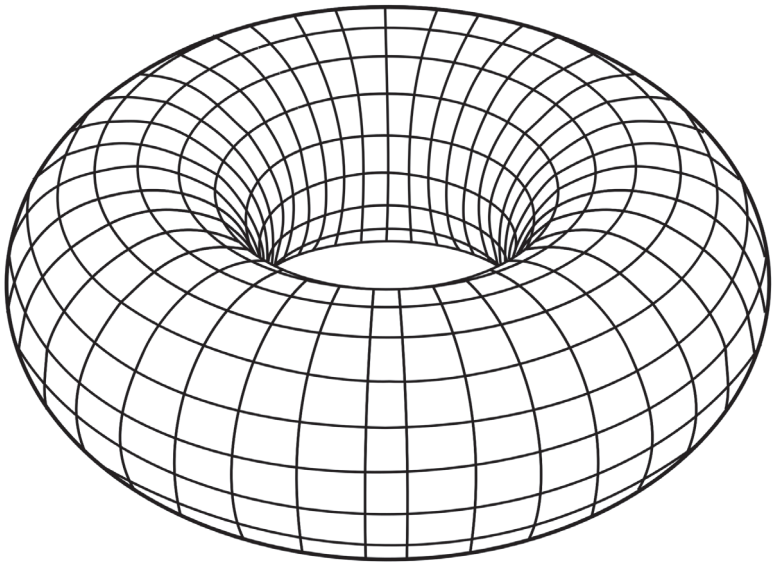

Figure 2.3 A torus.

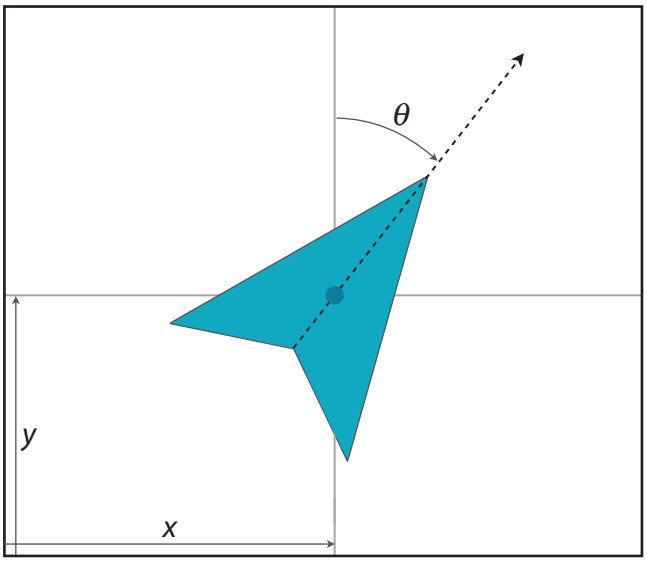

Figure 2.4 A depiction of a particle. The particle has position, (x, y), and a directional heading, θ. Its heading and position are updated on every time step.

the other hand, may have drifted some during the same time interval, so that in the moment following its northerly heading, it might be heading in a more easterly or westerly direction.

Things get hairy if a particle bumps into another particle. Keeping track of where they are heading takes a lot of concentration, which is broken by the collision. After they collide, both particles get confused and forget in which direction they were heading, and so each sets off in a new, randomly chosen direction.

Particle World physicians have become concerned about the long-term health consequences of these collisions. As such, they are particularly interested in how often the collisions occur. Some regions of Particle World are more densely populated than others, and they suspect that higher densities lead to more collisions. But how *many* more? Particle World scientists also know that individuals in some areas tend to be very whimsical, while individuals in other areas are more stubborn. They wonder: How does the whimsy of a region affect the number of collisions observed in that region?

We can make Particle World a reality and get to work answering these questions. Let's get coding!

2.4 Coding the Model

Now that we've got a good idea of how the model should work, let's get to coding it in NetLogo. For many of us, this is the fun part. I've provided a working version of the code in the repository,[7] titled **particles.nlogo**, and you may wish to simply examine that. However, there is also value in coding something up from scratch. For most of this book, I will not work through the NetLogo code in detail, but will instead focus on the algorithms and organizational aspects of the model code (this is also in service of readers who may wish to use other programming languages). This is our first time working through an agent-based model, however, and I want to make sure you understand what goes into coding up a new agent-based model.

Open up a new NetLogo model. You will be faced with a blank Interface window (Figure 2.5). We'll start by adding a few buttons and sliders for some of the commands and variables we know we'll need (you should be familiar with how to do this after going through the NetLogo tutorials). We'll need `setup` and `go` buttons, which will each correspond to procedures in our code. We'll also add sliders for the following global parameters: `num-particles`, `whimsy`, and `speed`. Before we head over to the Code window, we'll increase the size of our grid by clicking on the *Settings* button. The default grid in NetLogo is a 33×33 square. We'll increase the size to 101×101 to give our agents lots of space in which to move around (Figure 2.6).

With that accomplished, we can head over to the Code tab to start coding the model. If you look through my code, you'll notice throughout that I have added comments on many lines using semicolons. NetLogo ignores anything on a line that comes after a semicolon (two semicolons are often used as a convention, but one will work just as well). In general, documenting your code is a good habit to get into. It helps other people read and make sense of your code. Don't care about other people being able to understand your code? Well, consider one specific other person: your future self. You may know exactly how your code works now, but many a programmer has returned to code they haven't seen in months or years and found themselves baffled. Comment!

Moving on, NetLogo has the interesting feature that variables introduced in the Interface tab are treated as declared global variables, and so they do not need to be reintroduced in the Code window. In fact, there's only one other global variable we need to declare, and that is our outcome variable, `collisions`. Global variables are traditionally declared at the very top of the code, and so we can write the following:

[7] See the Preface for the repository URL.

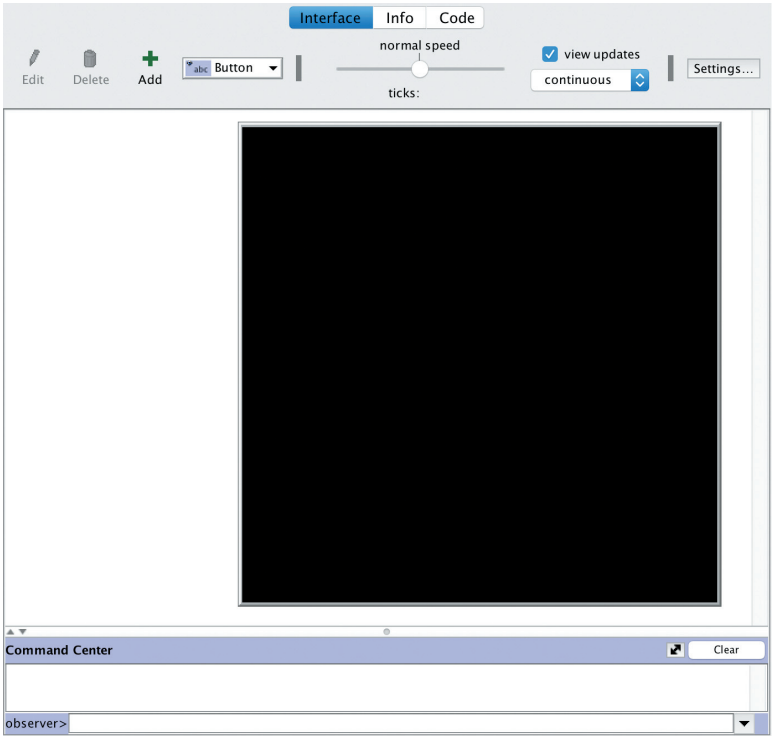

Figure 2.5 Blank Interface window in NetLogo.

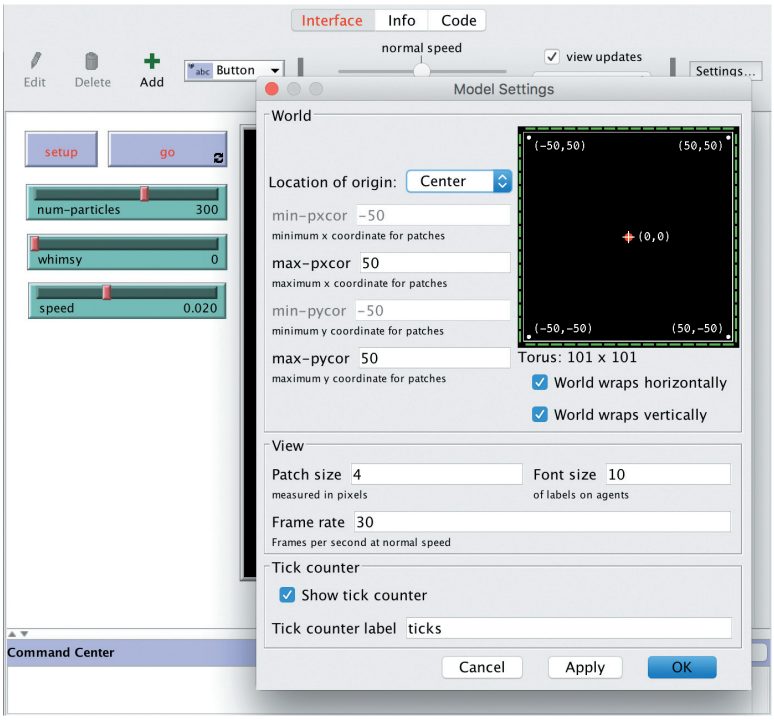

Figure 2.6 Interface window setting up for the Particle World model.

NetLogo code 2.5

```
globals [
  collisions
]
```

Next, we'll write the code for the function to initialize the model, the `setup` procedure, using the primitive function `to` to introduce the definition of a new procedure. The `setup` procedure must accomplish a few things. The first and last commands are standard and will be part of most NetLogo programs. `clear-all` resets all variables to their default initial values and removes all turtles and links from the simulation. `reset-ticks` resets the tick counter to zero and initializes all plots. These two commands are NetLogo primitives and are described in the NetLogo User's Manual. The first command after `clear-all` is aesthetic: we'll ensure that our agents take on the default shape of an arrow, which will allow us to easily see their directional headings.[8] We can then create the agents (as turtles). We make each agent green (or whatever color you like), make it larger to help us see the agents more easily, set it in a random location, and give it a random heading. The primitive reporters `random-xcor` and `random-ycor` are extremely useful, because each returns a random floating point number from the allowable range of spatial coordinates along the *x* or *y* axis, respectively. This means that you don't need to update the code if the size of the world is changed. After creating the agents, we will set our collisions counter to zero. After this procedure is run, all of our agents will exist in our simulated world, but they won't have done anything yet.

NetLogo code 2.6

```
to setup
  clear-all ;;clear/reset all variables
  ;;make the turtles look like arrows
  set-default-shape turtles "default"
  ;;make a bunch of green turtles
  create-turtles num-particles [
    set color green
    set size 2 ;;make them easier to see
    setxy random-xcor random-ycor ;;give them a random location
    set heading random 360 ;;give them a random heading
  ]
  set collisions 0 ;;initialize collisions at zero
  reset-ticks ;;set the clock to zero
end
```

Next, we'll code the model dynamics. These are laid out in the `go` procedure, which you should have set up to be called by a "forever" button in the Interface tab. This means that the procedure is run over and over until the button is pressed again or until some specified stopping condition is met. When this procedure is called, it will cause all of the agents to turn, move, and possibly collide (thereby updating our collision counter). We do this using

[8] The arrow is actually NetLogo's default shape for turtles, so declaring this isn't strictly necessary. However, doing this helps us to be conscious of the fact that other shapes are possible.

the command `ask turtles`, which takes as its argument a block of code that each turtle, in random order, will execute. First, each agent will turn to the right and then to the left, with the angle of each turn consisting of an integer randomly drawn from a uniform distribution between zero and (`whimsy`−1).[9] The agent will then move forward a distance specified by `speed`. I personally like the speed to be quite slow (around 0.02 or less) so that collisions are easy to visually observe and agents' behavior approximates smooth movement through space.

After an agent moves, we need to determine if it has collided with another agent. NetLogo has some neat tools for making this happen. The conditional statement here that determines a collision is `if count turtles in-radius 1 > 1`. The agent checks whether there are any other agents whose distance to the **focal agent**—the agent that we're focusing on at the moment—is less than or equal to one unit, indicating that they are touching. (For more information about any of the primitive commands used here and elsewhere in this book, see the NetLogo Dictionary.) If this condition is met, all of those agents with which the focal agent has now collided, as well as the focal agent itself, are asked to set their headings to a new random direction. Each of these agents then moves forward a small amount. Why do we do this? It's worth experimenting with removing this command (you can "comment it out" by adding semicolons at the front of the line) to see what happens. Agents will continuously jostle around, because they will remain near one another and are just as likely to move closer together as to move farther apart. Having each move a bit forward substantially reduces this possibility, making for a much more "natural" collision. It is debatable whether this sort of hack is reasonable in a model of behavior, and as a modeler that is the sort of decision you may often be faced with. The procedure ends with the command `tick`, which advances the simulation clock and updates the plots.

NetLogo code 2.7

```
to go
  ask turtles [
    ;;turn a random amount
    right random whimsy
    left random whimsy
    forward speed ;;move forward

    ;; if at least 1 other turtle near, set new headings
    if count turtles in-radius 1 > 1 [
      ask turtles in-radius 1 [
        set heading random 360
        fd 0.1
      ]
      set collisions (collisions + 1)
    ]
  ]
  tick
end
```

[9] Note that this will emphatically *not* produce a uniform distribution between negative and positive `whimsy`, but rather a binomial distribution bounded in ±`whimsy`. See Box 2.2: Correlated Random Walks and the Central Limit Theorem.

That is all the code you need to make the model work. Play around with it and see what kind of behavior you observe, fiddling with the various sliders in the Interface tab. We'll talk more about how to analyze the model below. Before we move on, however, it's worth saying something about code **modularity**. We have put all the model's code in just two procedures: `setup` and `go`. It may seem economical to reduce the number of procedures in a program, but in fact the opposite is often preferable. By having many small procedures, code becomes more modular. Modular code is usually easier to read and debug, and code snippets are more likely to be directly transferable to another piece of software. Below, the `go` procedure is rewritten to be more modular, supported by newly created supporting procedures: `move` and `bounce-turtle`.

NetLogo code 2.8

```
to go
  ask turtles [
    move
    bounce-turtle
  ]
  tick
end

to move
  right random whimsy
  left random whimsy
  forward speed
end

to bounce-turtle
  if count turtles in-radius 1 > 1 [
    ask turtles in-radius 1 [
      set heading random 360
      fd 0.1
    ]
    set collisions (collisions + 1)
  ]
end
```

Now that the model dynamics are coded, you can observe the simulation unfold in the Interface tab, watching the agents move around and collide. Wheeee! Visualization is useful when studying dynamic models, as watching those dynamics unfold under different parameter assumptions can give you intuition that is otherwise difficult to come by merely from reading the algorithmic model description or inspecting summary statistics. But summarize we must if we are to produce more than merely qualitative reports of the model behavior. We'll start very simply, by plotting the number of collisions as a function of time. This sort of plot is very easy to set up in NetLogo, which is yet another reason it's a nice platform for studying simulation models.

Back in the Interface tab, use the plus button or right-click to create a new plot. This is, by default, a plot of how one or more values change over time. Since we have already defined a variable to record the number of collisions, you can simply tell NetLogo to plot this variable. Filling in the window as shown in Figure 2.7 should do the trick. In addition to creating a plot, it may be useful to add a Monitor that reports the exact number of collisions at each time step. You can do this by means very similar to those you used to add the plot. You

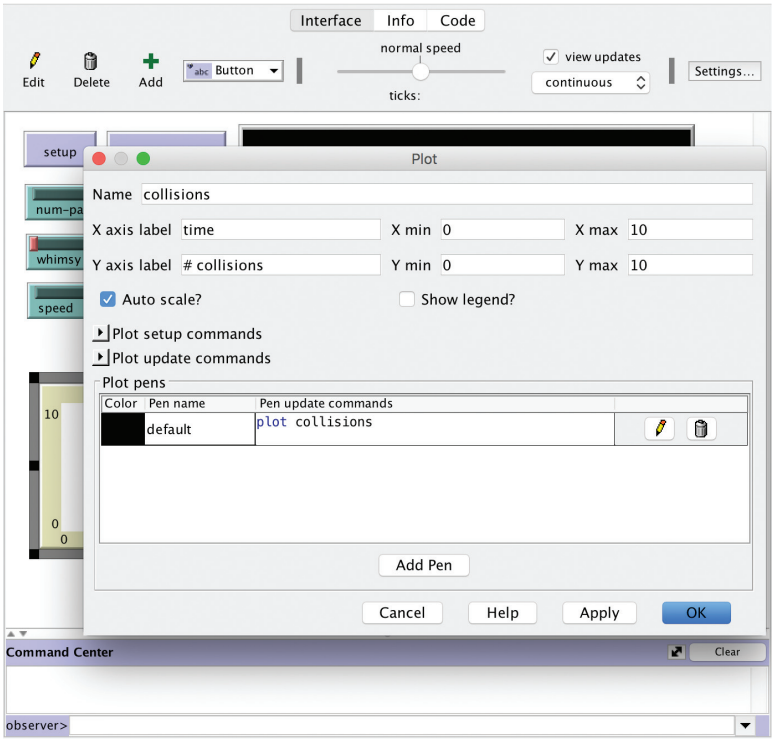

Figure 2.7 Adding a plot of the collisions as a function of time.

should end up with something that looks similar to what is depicted in Figure 2.8. You will observe that the number of collisions tends to increase linearly with time. Thus, you can use the *rate* of collisions per unit time to compare different scenarios of whimsy and population density—doing this is left as an exercise.

You now have a simulation model to take to the scientists of Particle World to help them study how the different population densities and cultural amounts of whimsy influence the rate of collisions each group experiences. You might even be able to use your model to suggest some interventions! For example, you might observe that while both population density and whimsy influence the number of collisions, the effect of density tends to dominate (Figure 2.9). This suggests that restricting density might be more practical than trying to change the (presumably strongly culturally and/or genetically imbued) degree of whimsy in each group.

The Particle World model is very simple and perhaps a little silly. However, the basic skills involved in putting it together constitute much of what will be needed to build models we can use to ask deeper questions about social behavior. I recommend you spend some time playing with the model parameters and seeing what kinds of outcomes you can get. Think about some other assumptions you could make about your agents, and try to code them. Get creative. Playing around to see what you can make your model do is a valuable habit to cultivate when designing and building models. I encourage you to play as much as you like, and I have provided some suggestions in the Exploration section of this chapter. In the next chapter, we'll study a slightly more serious model as well as a more serious approach to analyzing it.

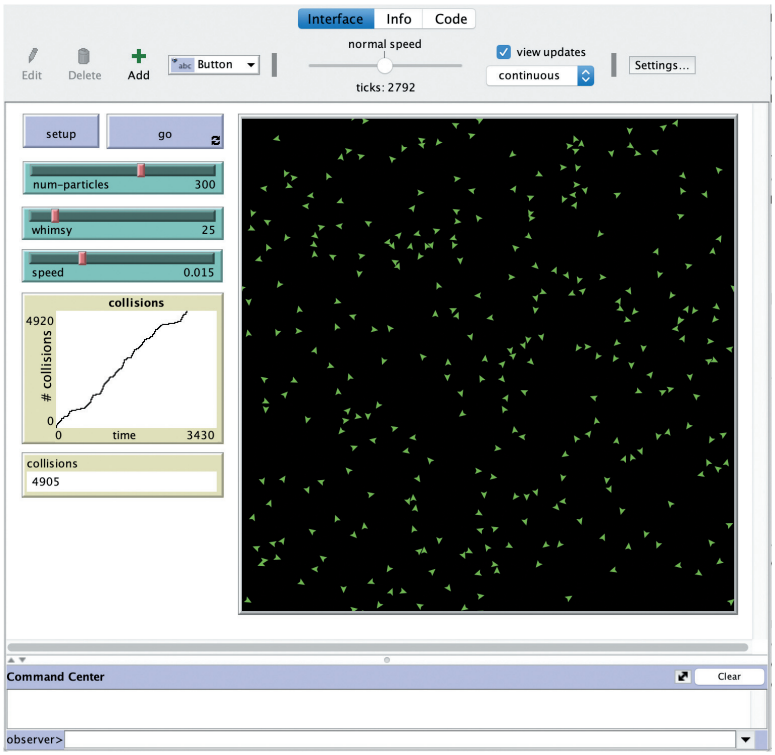

Figure 2.8 The Interface of the Particle World model in NetLogo.

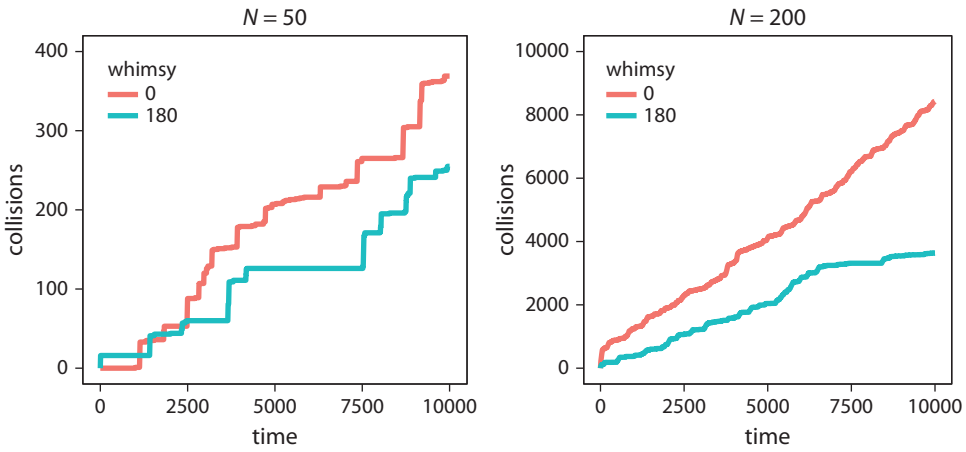

Figure 2.9 The cumulative number of collisions for several runs of the Particle World model over 10,000 time steps, for different values of the population size (N) and `whimsy`. In all cases, `speed` = 0.02.

You may notice that I have been a little vague about exactly how Particle World works. I could give the written description above to a few modelers to code up, and, without access to the NetLogo code, we might see slight or even prominent differences between their interpretations. This is a problem. If we are going to do serious science with models, we need to be able to talk with precision about how models work and to communicate that precision to others so that they can understand and reproduce our work. We therefore need to know how to talk about the components of a model—any model—and how to effectively write up a model description. In the next section, I'll discuss how to describe and communicate a model design, using Particle World as a case study.

2.5 The Components of a Model

Models are representations of a slice of reality, of some system of interest. A model decomposes the system into a simplified set of parts, properties, and relationships. As discussed in the previous chapter, there is no one right way to decompose a system. Rather, the value of a decomposition depends on how useful it is in explaining or illuminating some phenomenon or set of phenomena. Throughout this book, we will consider many systems involving social behavior and explore some representative decompositions. In doing so, we should always keep in mind this question: What are we assuming, and what are we excluding from those assumptions?

When it comes to presenting a model to others, our task is to help them to answer this question as easily and precisely as possible. Model descriptions should therefore be clear and transparent. In particular, they should include clear statements about (1) what the parts and properties of the model are, (2) how the model is instantiated and initialized, (3) how the dynamics of the model are scheduled, and (4) how the outcomes of the model simulations are measured or computed.

2.5.1 Parts and Properties

We need to lay out all the components of a model when describing it. What is being represented, and what is the nature of that representation? If we are talking about an agent-based model, what are the agents like? What properties do they have? What behaviors do they exhibit? If the agents interact, how are those interactions structured? What is the nature of their environment, and what are its properties?

The Particle World model consists of agents and their rather sparse environment. Let's start with the environment. The "real" Particle World might contain rocks and trees, lakes and rivers. However, we have made the simplification (which for now I will assert is reasonable) that particles move on large stretches of fairly flat land. So we'll ignore those geographic features and assume a fairly featureless landscape, and we'll model the topology of Particle World as a continuous square space with periodic boundaries. Let's unpack what that means. The world is square. For a spatially explicit simulation, a square is one of the simplest landscapes to program, analyze, and visualize.[10] As such, a square is one of the most commonly used spatial arrangements, particularly when a modeler doesn't have specific domain knowledge that would make the square a poor choice. The space is continuous, which means that

[10] A line is even simpler. However, when considering agents moving around and colliding, a line makes collisions unavoidable, while a 3-D space is unnecessarily complicated (not to mention more difficult to visualize). A 2-D space is best for our purposes.

agents can travel arbitrary distances in arbitrary directions. A commonly used alternative is a discrete grid, in which cells or patches are arranged in a regular pattern in the space and an agent's location is defined by its cell. Finally, and perhaps most notably for the novice modeler, the boundaries of Particle World are periodic, or toroidal (Figure 2.3). An agent that moves down past the lower bound of the space will emerge at the top, and an agent that moves right past the rightmost bound of the space will emerge at the left edge. Periodic or toroidal boundaries are commonly used in spatially explicit agent-based models. This may seem strange, given that almost no earthly environments are toruses. However, toroidal surfaces help us to avoid tricky artifacts like agents getting stuck in the corners of the grid, or asymmetries where agents near the edges have fewer spatial neighbors. Finally, we'll have to specify the size of our square, denoting the length of one side as L in arbitrary units. The spatial length unit is arbitrary, but it serves to determine the meaning of things like speed and distance in the model, and in some models this unit may correspond to real lengths or distances. In NetLogo, this is the width of a patch.

Onto this space we'll place our agents, whose number we will specify with one of our global parameters. The population size will also determine the population *density* if we hold the size of the space constant. Each agent has only two intrinsic properties: its location and its directional heading. That is, each agent will keep track of where it is and where it is heading. These need not be the only agent properties. For example, we discussed the importance of the speed at which the particles move. However, for our analysis, we are interested in speed as a property of the cultural group, so we will code speed as a property of the environment that governs the movements of all the agents in the population. We similarly discussed whimsy as a property that varied primarily between cultural groups, so we will likewise code whimsy as a global variable (that is, a property of the entire simulation) rather than as a property of the individual agents.

Finally, let's briefly talk about visualization. For reasons that will hopefully become clear, it is often useful to visualize the model dynamics in detail. Dynamic visualization isn't always practical for some model designs or some programming language choices—the ease with which dynamic visualizations are possible is a strength of NetLogo. When it is possible, it can help the modeler gain intuition about the system they are studying, and even observe outcomes they otherwise wouldn't think to test for. When making a visualization, we'll have to make some arbitrary choices concerning the aesthetic appearance of our agents and their world, including agents' color and shape. These aspects are often purely cosmetic, because they do not affect the model's behavior or its outcomes, and such cosmetic aspects are typically not reported in formal descriptions for simulation models. Indeed, these aspects of the code are often bypassed when running batches of simulations to reduce computing time. For the Particle World model, I have chosen to represent agents as green circles on a black background, but you can choose any appearance you like. Now that we know the components of the model, we need to specify how they are arranged.

2.5.2 Initialization

You cannot tell someone how to get somewhere if you don't know where they are starting from. Similarly, you cannot fully understand the dynamics or end states of a model if you don't know what things were like at the start. What's going on when a model simulation begins? How many agents are there, and what are their properties? Where are they in their environment and in relation to each other? What does the environment look like? Research on nonlinear dynamics has shown that in complex, interconnected systems, long-term dynamics can be highly sensitive to initial conditions. Although many social systems

are also quite resilient, the social systems we are interested in are often complex and subject to feedback processes that can amplify small variation. It is therefore vital to answer these questions about initialization thoroughly.

At the beginning of a Particle World simulation, the spatial environment is established and the agents are placed upon it. Recall that agents have only two uniquely held properties—location and heading—and so each agent will keep track of its own values for these variables (NetLogo does this automatically for turtles). How should they be initially assigned? There are lots of possibilities. For example, all the agents might start in the same location, or be evenly spaced in a grid pattern, or be arranged in a circle, or in the shape of a T-rex, or be placed on the grid in locations representing actual places in the real world. Similar concerns apply to their directional headings. If we don't have good reasons to choose one of these, or if we have a good reason to think it doesn't matter, we might as well draw both locations and headings at random from a uniform distribution of choices, and this is what we have done. That is, `num-particles` agents are created and placed at random x and y locations on the square grid. Our agents are ready to go!

2.5.3 Dynamics

Once we know what the parts of the model are and how they are initialized, we need to know how they change. In other words, how does the state of the model system update from one moment to the next? We usually think of time as progressing continuously, and this sort of continuous change *can* be modeled to some extent by mathematical formulations using infinitesimal time increments, as with coupled differential equations. In this book, we will generally stick to modeling time as advancing in discrete increments, though these increments can be arbitrarily small. This assumption also fits the nature of computational representation, which is naturally discrete. In NetLogo, discrete temporal increments are usually called "ticks," and I will use this term interchangeably with the more widely used phrase "time steps." Time steps can represent short units of time like a second or a day, or longer units like a year or a reproductive generation. The description of a model's dynamics typically involves what happens in one time step of a model simulation.

You should carefully consider the specifics of what happens during a time step and in what order those things occur. This ordering of the computations performed during each time step is called **scheduling**. Choices about scheduling can be consequential. For example, all agents might calculate their next move before any actions are taken, so that each agent is responding to the exact same environment. In this case, the ordering in which agents make their decisions probably doesn't matter. Alternatively, each agent could calculate and execute their move in one fell swoop, so that each agent potentially responds to the actions of the agents who are scheduled before it. In this case, the order in which agents are scheduled can matter, and it is usually wise to randomize the order in which agents are "stepped" at each tick[11] (NetLogo's `ask` procedure does this automatically). Whether agents respond **synchronously** (all responding to the same environment) or **asynchronously** (one at a time) can affect how the dynamics of the model unfold, though the qualitative results of most models are usually robust to both styles of scheduling.[12] A related issue concerns

[11] For example, Turner and Smaldino (2018) studied how stochastic decisions in initialization and scheduling could dramatically affect the dynamics of a single simulation run in a model of opinion dynamics.

[12] An interesting debate on this issue can be observed by reading Nowak and May (1992), Huberman and Glance (1993), and Nowak et al. (1994) in sequence, which deal with a spatial model for

the scheduling of multiple actions within a single time step. Should all the agents complete one action before any agent can move to the next action? Or should each agent complete all actions before the next agent is scheduled? There is no right answer to this question without knowing the details of the system being modeled, but the modeler should be mindful that different answers can lead to different outcomes. Finally, specification of a model's dynamics includes any stopping conditions for the model—that is, when a simulation is considered finished. Sometimes this will be after a fixed number of time steps, while in other cases it may be whenever a particular system state is reached (such as an equilibrium). In the former case, the number of time steps should be justified. For example, it may reflect sufficient time such that all observed simulations have reached a roughly stable state.

For each time step of the Particle World model, each agent, in a random order, turns, moves, and potentially collides. If the agent is stubborn, it will not deviate much from its previous heading when it moves. If it is more whimsical, it will turn quite a bit more. More precisely, the variable `whimsy` denotes an agent's maximum turning angle. Each agent will adjust its heading at each time step by first turning to the right an amount randomly chosen from a uniform distribution between zero and `whimsy` degrees. It then draws another number at random from the same distribution and turns that many degrees to the left. Thus, more whimsy translates into more frequent wide turns and less correlated random walks, while less whimsy translates into smaller turning angles and more highly correlated random walks, such that at the limit of zero whimsy, each agent simply travels in a straight line. After turning, the agent will move forward `speed` units in the direction of its current heading (ending up on the other side of the space if it moves across the edge of the grid). If no other agents are located where the agent has moved to, the agent is finished for the current time step. However, if another agent is sufficiently close to the agent in question (within one spatial unit), then a collision occurs. Recall that when an agent collides with another, it gets confused and heads off in a new direction. So, when two agents collide, they both receive new directional headings chosen at random from a uniform distribution between 0 and 359 degrees. A time step is completed when all agents have performed these actions.

2.5.4 Outcomes

The design of a model is driven by the questions we are asking about our system, and the process of modeling must therefore include deciding how the model's outcomes will be quantified and how the model's behavior under different conditions will be characterized for comparison. In other words, we need to know about our outcome measures. Sometimes this is as easy as counting the proportion of agents in the population exhibiting some trait. Other times more computation will be necessary, as when we calculate the structural properties of an evolving network.

In the Particle World model, we are concerned with the number of collisions that occur over the course of the simulation, as a function of both `whimsy` and `num-particles`. That is, our outcome is the number of collisions that occur over some standardized length of time. We measured this by creating a variable that is initialized to zero at the beginning of a simulation and is then incremented by one every time a collision occurs.

---

the evolution of cooperation that initially used synchronous updating. Some results were robust to a change to asynchronous updating, others were not.

2.6 Describing a Model

If you are sharing your model with others, as in a scientific paper or even in a blog post, you should describe it well. Make sure you've included details about the parts and properties, how the model is initialized, how the dynamics work, and any outcome measures you are collecting. A careful reader who is a competent coder and familiar with models—but not necessarily familiar with *your* model—should be able to replicate your model in a programming language of their choice, based solely on your written description. That is, the model description should be clear and complete, and should minimize ambiguity.[13] This is really important. The lessons one can draw from a model come directly from understanding how the model's assumptions lead to the consequences highlighted by the modeler. If a model is described poorly, the reader won't be able to discern exactly how it works, and any results of the model's analysis border on useless.

Writing up a clear description of a complicated model is a skill that requires practice to hone. There are many good suggestions in the **ODD protocol**, widely used in ecology, for describing agent-based models (Grimm et al., 2010, 2020). The protocol suggests a three-stage strategy of model description: the Overview (the "story" of the model), the Design (the computations involved in the model's dynamics), and the Details (all of the model's algorithmic details, sometimes relegated to an appendix). I am hesitant to recommend elaborate all-purpose protocols for describing research conducted across disciplines, but I quite like the general approach of an iterated description with increasing detail given at each stage. Nascent modelers often forgo the Overview stage, preferring to plunge ahead with the formal mathematical or computational details of the model. This is usually a mistake for all but the simplest models. A model is a representation of something else. When we make assumptions about a model system, we are mapping them onto our representation of the corresponding real-world system. However, as noted, a model simplifies—it omits details, and it even introduces falsehoods in the service of simplicity (for example, the falsehood that there are only two types of people). So, in order to make sense of how the model details map onto the real-world system, it helps to explain the model system verbally. The subsequently presented formal details can then be used to clarify how the model works without requiring the reader to simultaneously figure out the underlying analogy to the real world.

Because my intent in this book is pedagogical, I will often introduce models in a more narrative style than I would use were I to describe them in a scholarly article written for experts. In order to demonstrate more clearly what I am talking about, however, I have presented a complete formal description of the Particle World model in Box 2.1. This information is redundant with what is described above, but it illustrates how a model might be described in a publication.[14] The box also uses the convention of labeling parameters using single letters rather than the names actually used in the computer code. I find this approach easier to read, and it offers continuity between mathematical and agent-based modeling.

---

[13] For very complicated models, such as those used in systems science or artificial life, it may not be practical for every presentation of the model to include a full description. Even in these cases, however, that description should be accessible *somewhere*, and that somewhere should be directly referenced in all write-ups.

[14] For recent examples of how I recommend describing agent-based models of much greater complexity, see Smaldino and Turner (2022), Smaldino et al. (2019a), and Smaldino et al. (2019c).

BOX 2.1: Particle World Description

This model features a population of spatially embodied agents moving through continuous space, each using a correlated walk. When agents collide, they become confused, and each sets off in a new direction. We run each model simulation for 10,000 time steps and compare the number of collisions during that time.

Initialization

A population of N agents is initialized with each agent placed at a random real-valued locations on an $L \times L$ grid with periodic boundaries. Each agent i has a direction heading θ_i, which is initially chosen at random from a uniform distribution of integers [0, 359]. Each agent is fully defined by its location and directional heading. Other model parameters are the whimsy, w, which determines the turning angle agents use on each time step, and the speed, s, which determines the size of the step they take when moving. Finally, we keep track of the cumulative number of collisions over time, $C(t)$.

Dynamics

At each time step, each agent, in a random order, turns, moves, and collides. An agent first *turns* by adding to its direction heading an integer value that is randomly drawn from a uniform distribution in [0, w]. The agent then subtracts a newly drawn value from the same distribution from its directional heading. In other words, its new directional heading is $\theta_i + \epsilon$, where ϵ is randomly drawn from a binomial distribution bounded in $[-w, w]$. The agent then moves s units forward. If there are any other agents with a position within one unit of the focal agent's new location (defined by the Euclidean distance between the centers of each agent), a *collision* occurs between the focal agent and all of these other nearby agents. In this case, all of the agents involved in the collision update their heading to a new value randomly selected from the uniform distribution [0, 359]. Each of the involved agents then moves forward 0.1 spatial units in order to move away from the site of the collision and avoid cycles of perpetual collision. If a collision occurs, the cumulative collision counter $C(t)$ is incremented by one.

BOX 2.2: Correlated Random Walks and the Central Limit Theorem

In our Particle World model, the agents use a kind of **correlated random walk**, which just means that an agent's directional heading at time $t + 1$ is correlated with its heading at time t. We implemented this by having agents turn a random amount to the right and then a random amount to the left. Both of these turning angles are drawn from a uniform distribution bounded between 0 and `whimsy`. So, the resulting distribution of agent turning angles (their right turn minus their left turn) should be uniform as well, right?

Actually, it's not. To see that it's not, we can plot the distribution of the resulting turning angles. I wrote a simple script that repeatedly generates two random numbers between 0 and 90, and then subtracts the second number from the first, which is

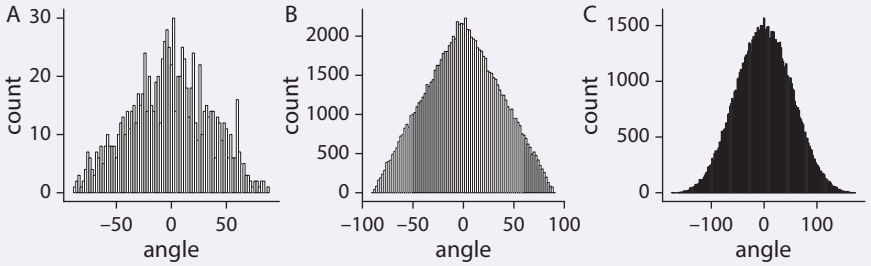

Figure 2.10 (A) Distribution of 1,000 turning angles generated by subtracting one random value from the other, where both random values are drawn from $U(0, 90)$. (B) The same, but with 10^5 turning angles generated. (C) Summing more numbers better approximates a normal distribution. Here 10^5 turning angles were generated by adding two positive random values together and then subtracting two positive random values from that sum, with each value drawn from $U(0, 90)$.

exactly what our agents do if we assume `whimsy` $= 90°$. I repeated this random process a thousand times and then plotted the distribution of the sums generated, shown in Figure 2.10A. There seem to be a lot more values close to zero than at the extremes. If we repeat the process a few more times and get more data points, we can see that the distribution of turning angles is essentially triangular (Figure 2.10B). This definitely isn't a uniform distribution. Why not?

It turns out that the distribution of the sum of draws from a uniform distribution is not itself a uniform distribution. This is because there are simply more ways to get values near the center of the range than values at the extremes. Consider a simpler case: rolling two standard six-sided dice. Even if you are not a professional gambler, some familiarity with dice or board games has likely given you the impression that rolling a seven is more common than rolling a two or a twelve (Figure 2.11). This is more than an intuition, of course, it's a mathematically necessary fact of probability. Consider a case where you want to roll a two: snake eyes. Your first die must come up one, and your second die must also come up one. This is the *only* way to roll a two. Next, consider a case where you want to roll a seven. If the first roll is a one, the second must be a six. But the first roll could also be a two, in which case the second must be a five. For *any* roll of the first die, there is a one in six chance that the second roll will yield a total of seven. There are thirty-six possible outcomes when you roll a pair of dice ($6 \times 6 = 36$), so one in every six rolls will come up seven, while only one in every thirty-six rolls will produce snake eyes.

When we repeatedly sum integers drawn at random from uniform distributions, the distribution of those sums will be given by the **binomial distribution**. For a large enough number of integers being summed, the binomial distribution is well-approximated by the normal distribution. And indeed, the continuous version of this scenario is the **central limit theorem**: the sum of a large number of uniformly distributed quantities will yield a normal distribution. Technically, the variables being summed need not be uniformly distributed, but merely *identically distributed* and independent. That is, each number must be drawn from the same probability distribution and be independent of all other draws.

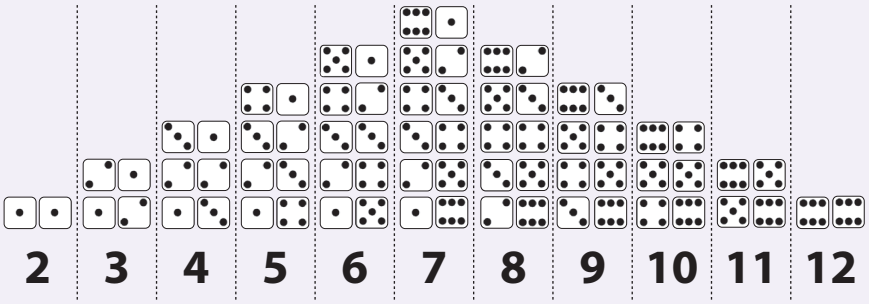

Figure 2.11 The many ways to roll two dice. Outcomes near the middle of the range of possibilities are more probable.

The purpose of this divergence is to illustrate the importance of examining the consequences of seemingly trivial modeling decisions. The distribution of turning angles produced by summing two uniformly distributed turns is not itself uniformly distributed, but highly concentrated so that small turns are always more likely than large turns even for highly "whimsical" agents. If turning angles were uniformly distributed, large turns would be much more common for those agents, which might dramatically change the relationship between whimsy and the resulting number of collisions. In general, it is wise to invest time considering the consequences of even the most minor or seemingly trivial assumptions when modeling.

2.7 Flocking

In our basic Particle World model, the agents just move on their own and crash into one another willy-nilly. Each agent is just moving along without any heed for what others are doing. As a result, there are tons of collisions. Indeed, the only social behavior we have modeled is collisions, and this behavior is confrontational to say the least. But now let's make things more interesting and imagine that our agents are a bit more civic-minded. They go with the flow. In practice, that means that they go the same way that others go.

One of the big ideas behind agent-based modeling is that coherent population-level phenomena can emerge from the aggregation of local, decentralized behaviors. In the last chapter, we briefly discussed the boids model, which illustrated how realistic flocking behavior could emerge from three simple rules for mobile agents: separation, alignment, and cohesion. Here we will explore the idea of emergence by adding just one of these rules to our Particle World model: alignment. The particles will observe their nearby neighbors and adjust their headings to match.

We'll implement flocking by adding two new global variables to the Particle World model: a Boolean variable called `flock?`, which will allow us to toggle between the original Particle World model and the version with flocking, and a parameter to control the size of the local neighborhood that agents can observe, `vision-radius`. In general, when extending a model it is often desirable to set up the code so that you can recover previous or alternative incarnations of the model. The code for this model in the repository is titled **particlesflock.nlogo**.

In our `go` procedure, we'll have the agents call a new procedure called `flock`, which, if flocking is turned on, will be executed before the agents whimsically turn and move forward. The `go` procedure now looks like this:

NetLogo code 2.9

```
to go
  ask turtles [
    if flock? [flock]
    move
    bounce-turtle
  ]
  tick
end
```

Note that `flock` is only called when `flock?` is true, so that our code allows us to directly compare conditions with and without flocking. When the `flock` procedure is called, the agent scans an area described by a circle with radius `vision-radius`, centered on itself. The agent first checks whether there are any other agents in that circle. If there aren't, the agent does nothing. If there are, it records the directional headings for each of the other turtles and averages them. It then adjusts its own heading to match that average. The NetLogo code for this procedure is below. Note that we define a local variable called `mean-heading`, which is the average of the headings of the other agents in the focal agent's vision radius.

NetLogo code 2.10

```
to flock
  if any? other turtles in-radius vision-radius [
    let mean-heading (mean [heading] of other turtles
     in-radius vision-radius)
    set heading mean-heading
  ]
end
```

What happens when flocking is turned on? Even when agents respond only to very close spatial neighbors, cohesive flocks encompassing most or even all of the agents in the simulation emerge, with large groups of agents all traveling in the same direction (Figure 2.12, top). If we compare the simulation before and after flocking is enabled, we also see that flocking allows the agents to avoid all but a few rare collisions (Figure 2.12, bottom).

The flocking algorithm we have implemented is a highly simplified version of the boids algorithm introduced by Craig Reynolds in 1987. This algorithm is also based on simple particles moving at a constant speed, but the full version requires not one but three observational radii. Within the smallest circle, an agent turns to avoid collisions. If no collisions are imminent, the agent looks within the second radius, and turns to align its heading with its neighbors. It is this aspect—alignment—that we have added to the Particle World simulation. Finally, if there are no nearby agents with which to align, the agent looks in a wider radius and heads toward the center of mass of any agents observed, thereby maintaining social cohesion. Having all three rules generates collective behaviors that are a bit more realistic than what we observe in our version. This simple flocking model forms the basis of more detailed explanations of collective behavior in a wide variety of species, including schooling fish, flocking birds, and swarming humans (Sumpter, 2010). It is also the basis

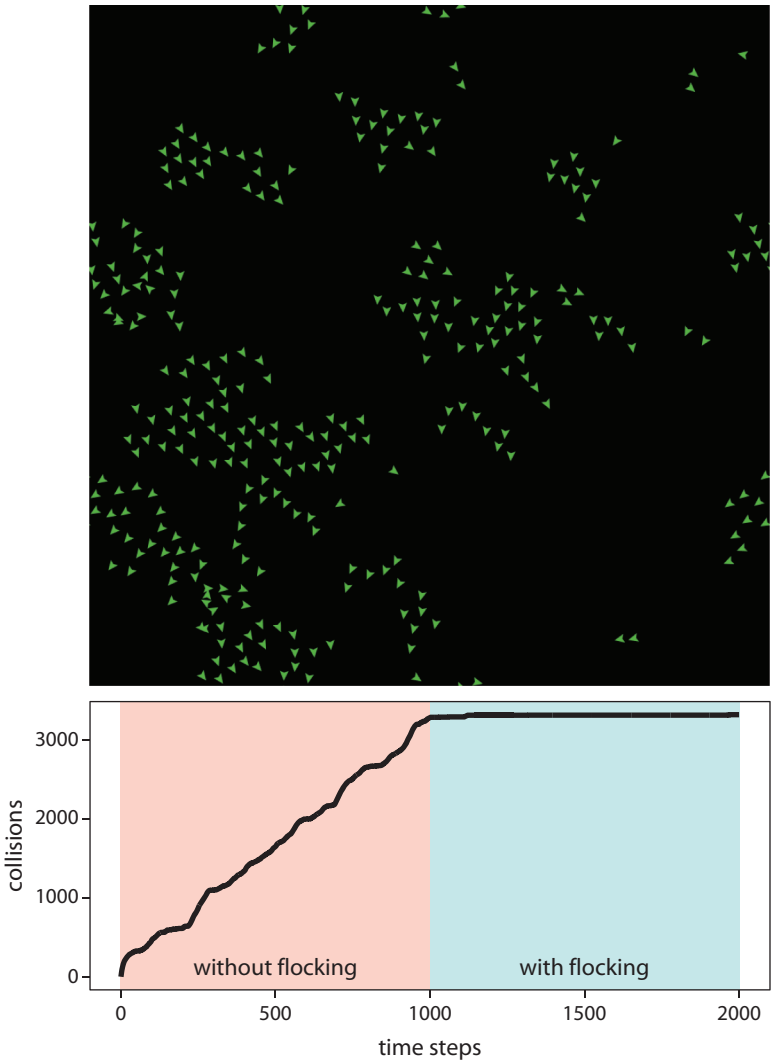

Figure 2.12 Top: The emergence of coherent flocking when agents align with those nearby. Bottom: Number of collisions over time. Simulation starts out without flocking, but flocking is turned on (that is, `flock?` is set to true) at $t = 1000$. Flocking effectively eliminates collisions. For this simulation, $N = 300$, `vision-radius` $= 3$, `speed` $= 0.03$, and `whimsy` $= 10$.

of almost all CGI (computer-generated) schools, flocks, and swarms in movies and video games (Gerdelan, 2010). Play around with the simple model we have constructed and see what behaviors you can produce by altering parameters and modifying rules.

2.8 Reflections

Throughout this chapter I have emphasized the importance of play. It is hard to overstate its value for becoming a competent modeler. Play allows you to test the failure modes of your code and provides opportunities for solving novel problems you might not otherwise

encounter until the consequences are more serious. Spending time exploring in a domain without severe consequences provides opportunities to build up competence and confidence in navigating within that domain.[15] We can learn so much when we give ourselves license to simply try things for the sake of trying them. Play is important even when coding artificial worlds, partly because play helps you to develop stronger competence in turning your ideas into reality. This facility can, in turn, help you to be less constrained by the details of any particular model or software package, and instead allows you to be constrained only by what you can imagine. Play also helps to cultivate your imagination. For any system of interest, there are many ways to model it. Even once you have specified your parts, properties, and relationships, there are still details to be decided upon, and these details sometimes matter. Gaining some familiarity with how seemingly small decisions affect the behavior of a model is a valuable muscle to train.

2.9 Going Deeper

In this book we will use NetLogo to code and analyze agent-based models. However, there are also other languages and libraries you might take advantage of. Of these, the most complete and best supported is probably the MASON library (Luke et al., 2005), which is a Java library for agent-based modeling that has a rich collection of demo code and a broad community of users. Several newer libraries are available in Python, notably Mesa (Kazil et al., 2020) and the more recent AgentPy (Foramitti, 2021). The Agents.jl library, written for Julia, is also promising (Datseris et al., 2022). At the time of this writing, these Python- and Julia-based packages are still not as well developed or as widely used as MASON, but due to the popularity of these languages among social scientists and data scientists, they are growing their user bases along with all the trappings that come with that growth. NetLogo, MASON, and Mesa can all be integrated with geographic information systems (GIS), allowing agent-based models to be mapped onto real-world landscapes, including cities, roads, and ecosystems. Of course, you don't actually need a specific software library to do agent-based modeling (though it often helps, particularly with visualization and scheduling). A competent programmer should be able to write a working model in any language they choose. For example, Acerbi et al. (2022) have recently provided an open access textbook on coding simple agent-based models of cultural evolution using R. For a deep conceptual discussion on modeling complex social systems, see Miller and Page (2007).

There is a very rich literature on modeling the behaviors and patterns that emerge from the movement behavior of embodied agents. To the extent that we will model mobile agents in this book, their movements will generally be quite abstract and represent movement through social space rather than physical space. But there are large and important literatures on using models to understand the movement of collectives, including the behaviors and patterns that emerge from various movement strategies. These include flocking and schooling, the dynamics of crowds, and the decisions of social foragers. For a deeper look into these topics, see Sumpter (2010) and Ball (2004).

[15] This is supported by a wide range of work in child development (Gopnik et al., 2015), animal behavior (Smaldino et al., 2019b), and even robotics (Cully et al., 2015).

2.10 Exploration

1. **I'm walkin' here.** Create a new NetLogo model in which a user-defined number of agents are created in initially random locations and then walk around randomly.

   (a) Create a new NetLogo model with a `setup` procedure that creates turtles.
   (b) Create a slider for a parameter called `num-turtles` that controls the number of turtles created.
   (c) Write a `go` procedure that makes the turtles wander around the screen randomly. To do this, have each turtle turn a random angle and then walk forward one spatial unit.

2. **Pen down.** Observing the pathways of collisions in the Particle World model is not so easy when all the agents look the same and you can't see their trajectories over time. This is easily fixed, however. NetLogo has primitives you can use called `pen-down` and `pen-up`, which trace the movement trajectory of an agent using the same color as the agent itself. To implement this tracking, change the `setup` procedure so that all the agents are assigned a random color. Then add a Boolean switch to the Interface and corresponding code in the Code tab that allows you to toggle whether or not the agents leave colored trails as they move. Report your code and some screen captures of your output. You're a regular Jackson Pollack.

3. **Collision analysis.** Characterize how density (`num-particles`) and `whimsy` influence the number of collisions in 1000 ticks. Collect the data for at least three values of each of the two variables, and report them in a plot and/or a table. Characterize the results verbally. What did you learn? What are some of the limitations of your ability to draw conclusions from this sort of analysis, and how might they be improved?

4. **Exploding collisions.** Create a new version of your Particle World model called Particle Smash. In this version, you will let the space be bounded rather toroidal, so that agents will bounce off the walls. Moreover, crashing into other agents will now have more dire consequences. To keep things relatively simple, restrict agent movement to straight lines in the absence of collisions (`whimsy` $= 0$).

   (a) Set the boundaries to fixed instead of toroidal. Change the dynamics so that the turtles "bounce" off the walls if they contact the edge of the world. Recall from physics that the angle of incidence equals the angle of reflection. For example, when a moving turtle hits the wall on a shallow angle, it should bounce off at a similarly shallow angle.
   (b) Alter the code so that every time an agent collides with either a wall or another agent, it changes color. Report your code.
   (c) Update the code so that the first time an agent collides with another agent, each agent splits into two smaller agents heading in random directions. It may be useful to know that the NetLogo primitives `hatch` and `die` cause an agent to spawn new agents and to be removed from the simulation, respectively, and that the primitive `size` controls an agent's size. When smaller agents collide, they should remain the same size.

5. **Bust a move.** Study the properties of random walks. How far will one agent tend to move from its initial location using a correlated random walk as a function of the maximum turning angle, θ (this is whatever parameter controls the turning angle)? Create

a model in which a single turtle is initialized in the center of the grid. Each time step, the agent should turn first to the left and then the right, where each turning angle is a random draw from a uniform distribution between zero and θ. The agent will then move forward some fixed distance. Ensure the grid is sufficiently large, relative to the step length, that the agent cannot reach an edge in 1000 time steps.

(a) Code this model.
(b) Create trajectory plots for several values of $\theta = \{0, 15, 30, 60, 180\}$.
(c) Create a plot or monitor that displays the agent's Euclidean distance from the origin. You may wish to use the NetLogo primitive `distancexy`.
(d) What do you conclude about the relationship between turning angle and distance traveled for correlated random walks?

6. **Color spread.** This is a more challenging task to test your ability to translate a verbal description into a working simulation. Make sure to document your code and describe how each aspect works. You will create a new NetLogo model called Color Spread. When the model is initialized, all patches will start as white or black (your choice) except a single patch of some other color either at the center of the space or in one of the corners. Your first task will be to allow the color to spread to adjacent patches.

   (a) Build this model. At each time step, the color should spread from colored patches to any adjacent noncolored patches (there are many possible ways to do this). Create a button to launch the procedure. Describe verbally how the color spreads, and include screenshots of the process.
   (b) Add a plot that graphs the number of colored patches as a function of time.
   (c) Create a chooser to allow the user to select which color will spread. NetLogo has primitives for several colors, but it can also represent colors both as single numbers from 0 to 139 and as RGB triplets.

7. **Rainbow spiral.** Here's the tricky one. Create a NetLogo model that produces a rainbow spiral. When the model is initialized, all patches will start as white or black (your choice) except a single patch of some other color either at the center of the space or in one of the corners. When the model runs, it should produce a spiral (either inward, starting in a corner, or outward, starting in the center) of spreading colors in which patches change color one by one. Each colored patch should have its color chosen at random. In this version, only one new patch should become colored at each time step, and the model should stop running when the spiral is complete. For the ambitious: you can choose to leave "layers" of black patches so that the spiral pattern is easier to discern, as in Figure 2.13.

8. **Flock of seagulls.** Let's do some experiments with our flocking model.

   (a) Consider the parameter `vision-radius`, which controls how close agents need to be to each other in order to influence each other's heading. Fix the population at $N = 300$, `speed` = 0.03, and `whimsy` = 10. Turn on flocking and initialize the model with different values of `vision-radius`: {0, 1, 2, 3, 4, 5}. Plot the number of collisions over 5000 time steps for each of these values, and then plot the rate of average collisions per time step against the value of `vision-radius`. Is the effect that flocking has on collisions linear with the radius of vision? If not, why not?
   (b) Now consider how a propensity for random turning interacts with flocking. Consider the following values for `whimsy`: {0, 15, 30, 45, 60, 75, 90}, and how each of

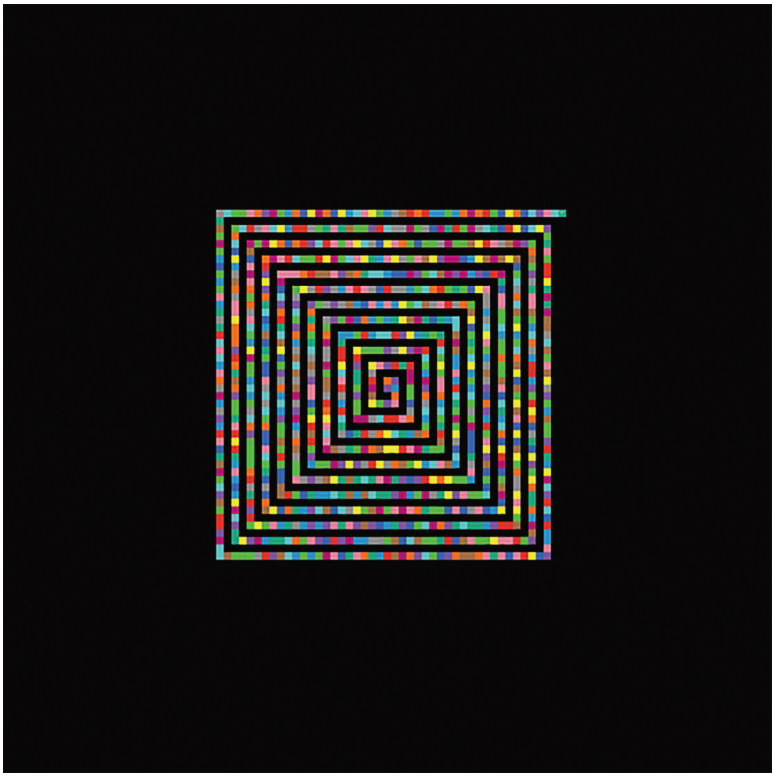

Figure 2.13 A run of the rainbow spiral model in a 101×101 grid at $t = 46$.

these values interacts with the following values of `vision-radius`: {2, 4}. From visual inspection, what sort of relation do you see between the parameters and the emergent behavior of the agents? Next, run simulations out to 5000 time steps, and plot rate of average collisions per time step against `whimsy` for each value of `vision-radius`.

(c) Relate the quantitative patterns you saw in your plots to the dynamic visual patterns you observed from simply watching the model run. How well do the graphs accurately capture the relationship between your observed patterns and the parameters used? What sort of important information about agent behavior or emergent patterns is not captured by these graphs? Can you think of other metrics that might capture this information?