

Agent-Based Modelling for the Self Learner

Tutorials Edition

Jennifer Badham

These tutorials are extracted from a book I am writing (but probably will not finish). This document is v1.0 (June 2019) and downloadable from:
www.criticalconnections.com.au/ABMbook.

Please contact me at research@criticalconnections.com.au with any comments or suggestions, including errors you find.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/> or send a letter to:
Creative Commons
PO Box 1866
Mountain View, CA 94042
USA

Preface

This book arose because I was fortunate to be a researcher at the Centre for Research in Social Simulation (University of Surrey) when Professor Nigel Gilbert and Dr Corinna Elsenbroich decided that it was time to offer a new short course in agent-based modelling for social scientists, and that such a course should have a significant component teaching *NetLogo*. I volunteered to develop such a component in the form of a tutorial that has participants gradually write a (pre-written) *NetLogo* model.

The idea was to support a 'learn by doing' approach, to move participants through the two-fold change of perspective required to build their own models: agent-centric thinking, and computer programming. Programming itself has many strands, including explicitly and specifically designing what the program is supposed to do (which overlaps with agent-centric thinking), good programming practices such as continuous verification, and the keywords and syntax of the programming language used. Those three programming strands were woven into the *NetLogo* tutorial. We used *NetLogo* because it is a standalone language (so no need to also learn python or java) that is designed for agent-based modelling, relatively easy to learn, and many resources are available for the participants once the course is over.

The initial tutorial was a housing model, which required far too much code to deal with technical aspects of the UK housing market. I then took up a position with the Centre for Public Health, Queen's University Belfast and was given the opportunity to update the course. As the revised course was to be targeted to public health researchers, the housing tutorial had to be replaced with something more relevant. With the experience of the previous courses, I was able to develop a shorter and simpler tutorial model that (I believe) still covered the most important agent-based modelling concepts and *NetLogo* language.

That public health tutorial is the main tutorial of this book. The interwoven structure of the tutorial is also maintained for the book. The material is presented in the same way as for one of the courses. Topics are introduced in the order in which they are needed for the model. New concepts are described immediately before being included in the model, then coded into the model to provide a concrete example. The reader is expected to run the model at multiple points as the code is developed. Such a structure leads to a certain amount of repetition. It also means that topics that may be introduced together in other books due to their similarity are instead introduced far apart. The payoff of this structure is that the novice reader is able to experience the process of building a *NetLogo* model in the same way as they would build their own model: iteratively.

This tutorial would not be possible without the commitment of both the University of Surrey and Queen's University Belfast to provide a course (and the time to prepare materials for that course). In addition, I would like to acknowledge the many students who asked interesting questions, colleagues who taught the course with me, and several alpha-testers who worked through earlier versions of the main tutorial. They all improved the tutorial in uncountable ways, as well as supported me in making the tutorial available.

Contents

1	Orientation	1
1.1	Complex Systems	2
1.2	What is an Agent-Based Model?	3
1.3	Agent-based modelling for complex systems	5
1.4	Introducing Netlogo	5
1.5	Rabbits Grass Weeds	7
1.6	Virus on a Network	11
1.7	More Library Models	13
1.8	Model Design	14
1.9	Doing the Tutorial	17
1.10	Tutorial Progress Check	17
2	Model 1: Model Entities	19
2.1	The <i>NetLogo</i> World	19
2.2	Fundamental Coding Concepts	21
2.3	Start and Continue	22
2.4	Create Model Control Buttons	23
2.5	Variables	24
2.6	Describing the Physical Environment	26
2.7	Doing Mathematics	30
2.8	Turtle Agents	31
2.9	Testing with Inspect Windows	34
2.10	Colours	35
2.11	Communication through Interface Design	36
2.12	Tutorial Progress Check	40
3	Model 2: Introducing Time and Space	43
3.1	Model Design: The Epidemic Process	43
3.2	Methods of Time Keeping	44
3.3	Working with Agent Variables	45
3.4	Conditional coding: Updating the colour scheme	49
3.5	User Input Widgets	50
3.6	The ask Command	51
3.7	Making the Model a Simulation	51
3.8	Commands, Reporters and Procedures	55
3.9	Spatial Awareness	57
3.10	Output Widgets for Model Results	61
3.11	Tutorial Progress Check	63
4	Model 3: Agents Making Decisions	65
4.1	Model Design: Protective Behaviour Decisions	65

4.2	Context and Perspective	66
4.3	Procedures with Arguments (Inputs)	69
4.4	Agentsets	71
4.5	Finishing off the Core ABM	74
4.6	Tutorial Progress Check	75
5	Model 4: Representing Relationships	81
5.1	Model Design: Influential Friends	81
5.2	Creating a Social Network	82
5.3	Using Links	84
5.4	Tutorial Progress Check	87
6	Models 5-8: Enhancements	89
6.1	Model 5: Efficiency	89
6.2	Model 6: Visualising Decisions	90
6.3	Model 7: User-Controlled Perception	90
6.4	Model 8: Infecting individuals	91
6.5	Tutorial Progress Check	94
7	Bringing it all Together	95
7.1	Model Design	95
7.2	Good practice in programming	97
7.3	<i>NetLogo</i> language	98
7.4	Where to from here?	100
8	Turtles on the Move	103
8.1	Mobility Tutorial Setup	103
8.2	Co-ordinates	105
8.3	Direction	107
8.4	Movement	109
8.5	Adding Obstacles	111
8.6	Comments	114

List of Figures

1.1	Empty NetLogo interface	6
1.2	Accessing the Models Library	7
1.3	Rabbits Grass Weeds: Interface	8
1.4	Rabbits Grass Weeds: go	10
1.5	Rabbits Grass Weeds: move	11
1.6	Rabbits Grass Weeds: eat-grass	11
1.7	Virus on a Network: spread-virus	12
1.8	Virus on a Network: become-infected	12
1.9	Conceptual design: protective behaviour in an epidemic	16
2.1	Settings Dialogue Box	20
2.2	Passage of time	22
2.3	Adding buttons	23
2.4	Buttons dialogue	23
2.5	Error message example	25
2.6	Model 1a	25
2.7	Monitor dialogue	28
2.8	Model 1b	30
2.9	Model 1c	34
2.10	Inspect turtle	35
2.11	Colour values	36
2.12	Model 1d	39
3.1	SIR epidemic transitions	43
3.2	NetLogo passage of time	45
3.3	Command Center	48
3.4	ifelse-value compared to ifelse	49
3.5	Model 2c	50
3.6	Slider dialogue	52
3.7	Model 2d	54
3.8	Step through of epidemic completed test	56
3.9	Patch neighbourhoods	59
3.10	Model 2f	61
3.11	Plot dialogue	62
3.12	Model 2g	63
4.1	Contextual procedures	67
4.2	Attitude histogram	71
4.3	Using truth results of conditions directly	74
4.4	Model 3c	75


4.5	Model 3d	76
4.6	Model 3e	77
5.1	Inspect turtle	83
5.2	Printing to screen	86
5.3	Model 4c	87
6.1	Model 8	94
7.1	Using with and if	99
7.2	<i>NetLogo</i> entity types	99
8.1	Co-ordinate system	106
8.2	Model M4	110
8.3	Model M7	114

*

Orientation

You are presumably reading this book because you have seen an agent-based model (or read about one) and decided that agent-based modelling is the most appropriate simulation methodology to help you investigate some research question. Or maybe one of your colleagues has suggested it? Unfortunately, there are few courses available, and many people building their first model are doing so with limited programming experience and without the assistance of experienced colleagues. If you are in this situation, this book is intended for you.

The book is structured like a course rather than a textbook or reference book. The core of the book is a tutorial that will gradually take you from a ‘blank page’ (what you see when you open *NetLogo*) to a fully developed model. Like an in-person course, there is a certain amount of foreshadowing of future material and reviewing material already presented to help you connect what you are doing in the tutorial to the broader theory of agent-based modelling.

This tutorial is constructed so as to teach three aspects of agent-based modelling simultaneously. **(1) Agent-centric thinking** is interpreting the world to be simulated as individual agents taking actions that implement some process. The particular computer programming language we use to build the model is **(2) NetLogo**. However, programming is not simply a matter of learning what each defined keyword in a language does, but **(3) good programming practices** are required to use that language effectively. Throughout the tutorial, highlight boxes (marked with the  icon) extract key ideas for each of these aspects.



Key points

Boxes like this contain definitions, concept summaries, programming tips, and other important points. They do not, however, describe *NetLogo* language keywords, which are displayed in a different type of box.

The tutorial model concerns protective behaviour (such as good hand hygiene) when faced with an epidemic. It is unlikely that the model that you want to build is about the same topic. However, regardless of your topic, the core process being simulated is common to many agent-based models: people making decisions based on their own motivations and circumstances and those decisions in turn affecting the world around them.

It is essential that you actually do the tutorial. This book will not help you learn agent-based modelling or *NetLogo* if you simply read it. Furthermore, ‘doing the tutorial’ does not mean just following the instructions, typing in the programming code and running the model. It is also important to think about how the code works. The particular commands are explained in the text and I describe what each section of code does, but you need to be sure that you understand how that code achieves its objective.

While the tutorial model was developed to use many of the most important parts of the *NetLogo*

programming language, it does not cover all of them. In particular, the agents in the tutorial model do not move, so mobile agents are dealt with separately (see chapter 8). The full book will also cover important topics such as lists and extensions, but they are not included in the *Tutorials Edition*.

Before starting the tutorial, it is worth considering in what situations agent-based modelling is useful. This requires an understanding of what is meant by complex systems, and what agent-based modelling can offer as a lens to examine complex systems.

1.1 Complex Systems

There are many descriptions and definitions of complex systems. While these descriptions emphasise different aspects, the key to complexity is how the parts of the system interact. If the behaviour of the system cannot be understood by examining the parts separately, and arises instead from the interaction between parts, the system is exhibiting complex behaviour. That is, a complex system is more than just a complicated system with many parts that are related in some way, the interaction between the parts must be critical in generating the behaviour of interest (Miller and Page, 2007).

Consider a traffic jam. This is a behaviour generated not just because there are many moving cars, but because of the way in which interactions between the drivers of those cars introduce cascades of braking and accelerating. A single blockage, such as a set of traffic lights, braking to avoid an animal on the road, or correcting a drift too close to the car in front, can trigger a traffic jam. Differences in drivers' preferred speeds, separation and reaction speeds all amplify any slight variations. Analysing individual cars would not provide any understanding of the phenomenon of a traffic jam. In different circumstances, such as moving smoothly on a highway, the same cars can be considered somewhat independently as their behaviour does not significantly affect the behaviour of others.

The modern term of complexity was introduced to describe a set of *problems which involve dealing simultaneously with a sizable number of factors which are interrelated into an organic whole* (Weaver, 1948, pg 539). However, the essence of the idea was encapsulated much earlier by Aristotle in the phrase *the whole is one, not in the manner of a heap but of a syllable* (Aristotle, 2011, 1041b, more familiar as 'the whole is other than the sum of its parts').

Weaver (1948) recognised that complex problems represented a relatively unexplored area in science and were particularly prevalent in biological, medical, psychological, economic and political sciences. A modern commentator would add social and environmental sciences, at least, to this list. Complex systems questions include some of the most pressing policy issues: how to limit the extremes of economic cycles, what species are critical to maintaining the food web, where is the point at which we cannot recover from climate change, how to avoid an influenza pandemic. All of these questions involve behaviour arising from interactions. For example, influenza is transmitted between people, which can only occur through contact between infected and susceptible people. Economic cycles are the consequence of millions of transactions where institutions, firms and individuals purchase goods and services from other institutions, firms and entities.

While economies, food webs and epidemics may be referred to as complex systems, this language is somewhat misleading. What is of interest is a particular behaviour of the system, not the obvious entities within the system. To bridge this gap, the term 'system' must be understood in its widest sense, so that the parts of the system are not just the individual components, but also the rules of interaction. To return to the traffic jam example, the system does not consist only of cars, drivers and roads, but must also include the characteristics and behaviour of the drivers that contribute to the traffic jam. To make this comprehensive scope of 'system' clearer, this book uses a definition of complex systems that explicitly focuses on behaviour.



Complex System

A system is complex if the system's behaviour of interest is driven by the interactions between the individual parts, not simply the behaviour of the parts independently.

As well as identifying the class of complex problems, Weaver (1948) also suggested that progress could only be made by multidisciplinary teams developing new computer assisted methods. Agent-based modelling is one such method.

1.2 What is an Agent-Based Model?

Before defining agent-based modelling, we must be clear about what is meant by the term 'model'. Models come in many forms, including: a small collectible replica of a car, a set of equations to describe climate change, and a person who wears the clothes being displayed at a fashion show. While these examples differ greatly, they share some features.

The first is that the model is standing in for something else. Even the fashion model is a substitute for the people who will eventually purchase and wear the clothing line being presented.

The second is that the model is an imperfect stand-in. The imperfections take two forms: some parts of the system are not included in the model at all and some are included but in a simplified way. The model car may look like a car, but it can't be used to travel. The equations only describe some of the processes involved in climate change. And most people who will eventually wear the clothes being modelled have different body shapes than most fashion models. Ultimately, the choice of what to include in the model, and at what level of detail, is guided by what the model is for. For example, the lack of an engine is immaterial in a collectible model car, the goal is a faithful representation of the body shape and styling.

These commonalities suggest a definition for model that focuses on representing something. In addition, we wish to exclude mental models because a model is only of use in understanding complex systems if it can be shared.



Model

A model is a formal representation of relevant features and relationships of some target system.

This definition has three elements:

1. **Formal representation:** the model is constructed using a language, diagram or some other communication medium that is independent of the modeller;
2. **Relevant features and relationships:** the model is simpler than the full target system and there is some purpose or question that guides the decisions about what to include and exclude in the representation;
3. **Target system:** the model is 'of' something.

How does this definition of model relate to the specific type of model that is dealt with in agent-based modelling? An agent-based model is constructed in a computer programming language (such as *NetLogo*). But the specifics of the target system and its relevant features are less immediately clear.

Agent-based models are a type of simulation, and a simulation is simply a model of a process. By this, we mean that the target system for a simulation is a series of actions, decisions, consequences and other changes that occur through time.



Simulation

A simulation is a model of a process.

There are at least two ways to think about a process, and therefore to represent it. One way is to describe the process from the perspective of an external observer. Alternatively, the process can be described from the perspective of the participants what actions they take and why. Returning to the traffic jam example, an observer may describe it as a line of vehicles that are moving slowly (or stalled) over a distance of several hundred metres. The drivers, however, are within the traffic jam and do not know its length or other characteristics. Instead, they perceive only the local traffic, and respond to the vehicle in front of them by starting and stopping as appropriate. Some drivers may look further ahead, but any information gained has little influence on their progress.

Agent-based modelling adopts the perspective of the agents within the system. The model is comprised of many agents, each representing an individual person, car and driver, household, business, or some other entity. The model moves through time, with each agent taking a series of actions that are chosen by that agent, without central control. The computer program is written to model the world from this agent-centric perspective.

The implementation of this perspective in code is best understood with specific code examples, so will be revisited throughout the tutorial (for example, see section 1.5.2). At this stage, it is sufficient to recognise that agent-based models represent the target system from the agent-centric perspective.



Agent-centric perspective

I, the agent, have certain characteristics and beliefs of my own as well as information about the world around me, and therefore will decide on some action.

Putting these elements together constructs a brief definition of an agent-based model. It is a computational model of a process from the perspective of the participants. The relevant features and relationships that are to be included in the model are those that contribute to the actions that the agent must take in order to represent their equivalent real-world entities. Furthermore, characteristics that have no effect on actions do not need to be included in the model.



Agent-based model

An agent-based model is a computer program that represents individual entities taking actions in accordance with their own characteristics, resources, beliefs, and perception of their social and physical environment.

While agent-based models are typically run on a computer, that part of the definition is descriptive rather than prescriptive. For example, the segregation model (described in section 1.7) is an agent-based model that was originally implemented physically with a checkerboard and counters (Hegselmann, 2017). Descriptions of agent-based modelling may also include non-definitional aspects of the method, such as experimentation with different scenarios (Abdou et al., 2012), or advanced agent features such as the capacity to adapt (Macal and North, 2010).

To confuse the definition further, there are other simulation methods that overlap with agent-based modelling. Microsimulation (or Markov chain) models have simulated individuals independently change states over time, typically based entirely on probability. These simulations therefore have heterogeneity but no interaction. In cellular automata simulations, cells in a grid take on different states (such as alive or dead) based on the states of the cells around them.

Thus, interaction is limited to a fixed set of other cells that are defined spatially. These are (generally) considered to be distinct model types, but agent-based modelling tools can be used to implement these simulations.

1.3 Agent-based modelling for complex systems

Agent-based modelling is a particularly suitable methodology for simulating complex system behaviour (Gilbert, 2008; Railsback and Grimm, 2011; Luke and Stamatakis, 2012). The agent-centric perspective allows the interactions that drive the behaviour of a complex systems to be directly represented in a natural way. Returning again to the traffic jam example, the simulated car/driver is able to interpret its environment (distance to the car in front, speed of the car in front), and combine that with its own characteristics (preferred speed, current speed) to determine an action (accelerate, brake). Agents can also be programmed to interact directly with other agents, such as transmitting an infection, buying goods, or influencing behaviour. By directly representing the drivers or mechanisms of behaviour at the individual level, the complex behaviour of the overall system that emerges from these interactions can be observed.

Understanding the connection between individual (micro-level) behaviour mechanisms and overall (macro-level) system behaviour is an important use of agent-based models. It is particularly valuable where interactions are combined with heterogeneity because the methodology enables different agents in the same situations to take different actions, and the same agent in different situations to take different actions. Real-world richness can therefore be represented in the model.

For the purposes of this book, we will assume that both interaction and heterogeneity are important in the process that you wish to simulate. The processes modelled in the tutorial includes both aspects to help reinforce the relationship between agent-based modelling and complex system behaviour.

So let's jump in . . .

1.4 Introducing Netlogo

Before starting to code the tutorial model, you must first install and become familiar with the (free) modelling software *NetLogo*. The *NetLogo* software includes a library of example models and we will use these to demonstrate the concepts already introduced, including agent-centric thinking and complex system behaviour.

NetLogo (Wilensky, 1999) is specialist agent-based modelling environment that includes its own programming language. It can be downloaded from <https://ccl.northwestern.edu/netlogo> and is available (for free) for Windows, Mac OS X and Linux. This book assumes you are using version 6 (the latest available at the time of writing). Download and install *NetLogo*.

There is also an online version of *NetLogo* (called *NetLogo Web*) that can be run over the internet, without installing any software. However, only some *NetLogo* keywords are included, and the tutorial cannot be completed over *NetLogo Web*.



Keyword

A keyword in a computer programming language is a word that is part of the language and has a special meaning. Keywords allow the programmer to tell the computer what to do. The programmer cannot use those words for other purposes (such as variable names), they are reserved for their specified meaning. In *NetLogo*, the keywords that operate on agents (rather than the model as a whole) are also referred to as primitives.

1.4.1 Navigating a *NetLogo* model

Open *NetLogo*. As no model is loaded, an empty model template is displayed (see Figure 1.1). The main menu at the top (labelled 1 in the figure) is used to open an existing model, save the model you are working on (both in the File menu) and run experiments (with the batch simulation tool BehaviorSpace). It also provides access to resources such as the *NetLogo* Dictionary, that describes all the keywords (in the Help menu).

Immediately under the menu, there are three tabs to switch between the ‘Interface’, ‘Info’ and ‘Code’ screens (labelled 2 in the figure). The interface is the visible screen when NetLogo starts. When a model is open, the interface contains the buttons, sliders, plots and other model controls for the user to operate the model and interpret its results. The info page contains descriptive information about the open model. As this information is written by the model developer, detail may be limited. Models in the sample library included with the *NetLogo* software have useful information similar to a basic users guide for the model. The Code tab accesses the page with the program code that implements the model.

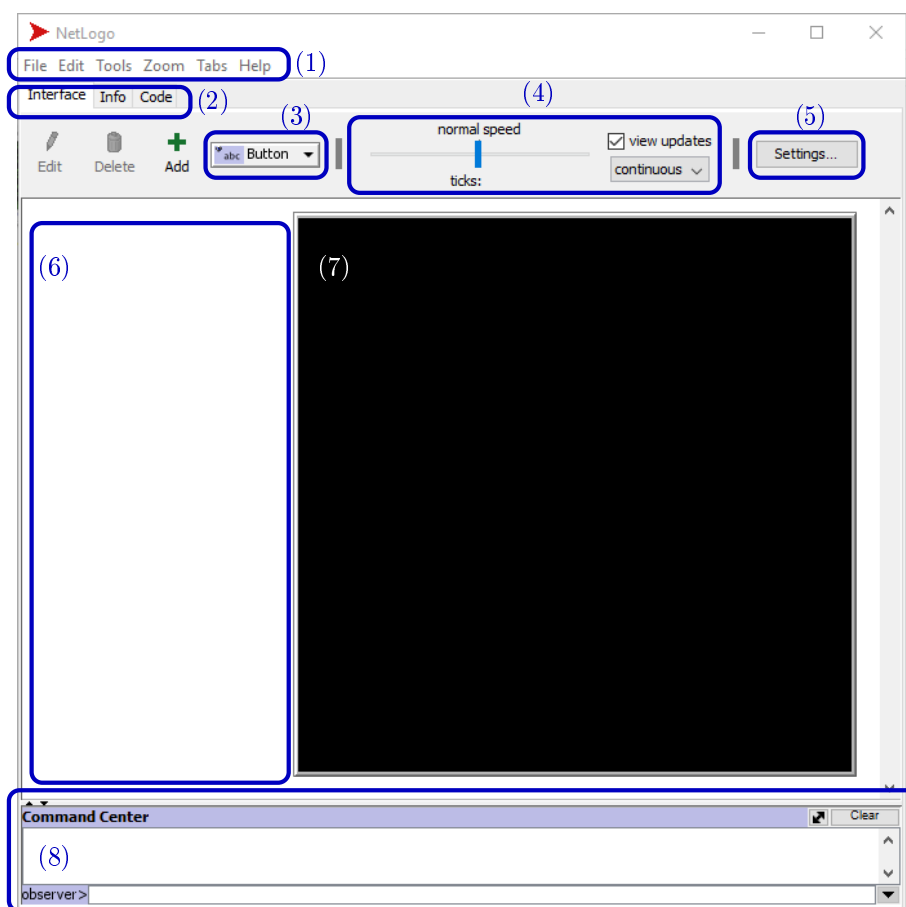


Figure 1.1: Screenshot of an empty *NetLogo* model with parts of the interface labelled. (1) is the menu. (2) is the tabs to switch between the Interface, Info and Code views. (3) adds widgets (input controls and output reporters) to the interface. (4) contains the speed slider and update controls. (5) is the settings controls. (6) is a blank area of the interface where input and output widgets are added. (7) is the world, or main view of the model. (8) is the command center, with input (bottom) and output (upper) areas.

The functions of the other areas will be described as they are needed. Briefly, the model developer uses the widget (3) and settings (5) items to design the interface for the user to interact with the model. The main interface contains a view of the world (7), various input and output widgets to control and interpret the model (in the currently blank area 6) and controls for the simulation speed (4). Finally, the Command Center (8) allows the developer or the user to directly enter NetLogo code for immediate running.



Widget

A widget is an interface component such as a button, plot or slider.

1.4.2 NetLogo model entities

For historical reasons, the main agents in a *NetLogo* model are referred to as *turtles*.¹ These are the agents that perceive their spatial and social environment and change their behaviour accordingly. In this tutorial, the term ‘agent’ and ‘turtle’ will be used interchangeably except where otherwise stated.

There are three other types of entities in a *NetLogo* model. The spatial environment is comprised of *patches* and the social environment is comprised of *links* that represent a relationship between two turtles. More abstractly, *NetLogo* is aware of the *observer*, who is able to issue instructions to the other entities.

Technically, all of these entities are different types of agents within the architecture of the *NetLogo* software. However, referring to *patches* and *links* as agents is unnecessarily confusing for novices because they are conceptually different from the agents whose behaviour is actually being simulated. Nevertheless, it is important to be aware of this usage because the official documentation refers to all of the entity types as agents.

1.5 Rabbits Grass Weeds

The next step in *NetLogo* familiarisation is to open (load into *NetLogo*) an existing model and play with it. There is a library of sample models installed with the software, which contains full models and short examples of code to achieve specific results. Open the *Models Library*. This can be found in as an item in the *File* menu (see Figure 1.2). From the library, load the model named *Rabbits Grass Weeds* (located in the Biology section of the Sample Models section, or search for ‘rabbits’ using the search box at the bottom of the models list).

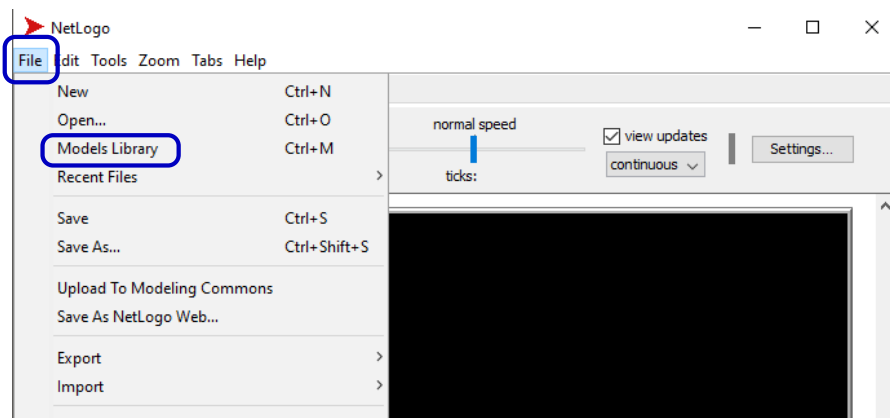


Figure 1.2: Partial screenshot of an empty *NetLogo* model. The menu option for the models library is highlighted.

When initially opened, the interface has widgets on the left (in the area marked (6) in Figure 1.1) but the world (area (7)) remains empty. Once the model is running, it will look like Figure 1.3.

The user controls consist of two buttons (labelled ‘setup’ and ‘go’) and six sliders with various labels (section (1) of Figure 1.3). In addition, there are two output widgets, a plot labelled

¹The Logo programming language was used in the 1980’s in education settings to program robot turtles to draw by crawling on a piece of paper with a pen. Creating a pattern requires taking the turtle’s perspective instead of an observer’s.

‘Populations’ and a small box labelled ‘count rabbits’ (section (2) of Figure 1.3).

Drag the ‘number’ slider to 180. Do this by placing the mouse pointer on the red marker and hold down the mouse button to move the marker. You can also move the marker a small amount by clicking the mouse cursor on the right or left of the marker. Press (with a mouse click) the button labelled ‘setup’. The ‘setup’ button is used to initialise the simulation, creating some patches with grass and a starting population of rabbits. The world should now have some small green squares and 180 white rabbit icons scattered (numbered (4) in Figure 1.3).

Start the simulation running by pressing the ‘go’ button. To pause (or stop) the simulation, press the ‘go’ button again and then continue the run with the same button.

Once the simulation is running, the plot will display the number of patches with grass (green) and rabbits through time. The current number of rabbits is in the monitor (small box below the plot). Use the speed slider (numbered (4) in Figure 1.3) to speed up or slow down the simulation.

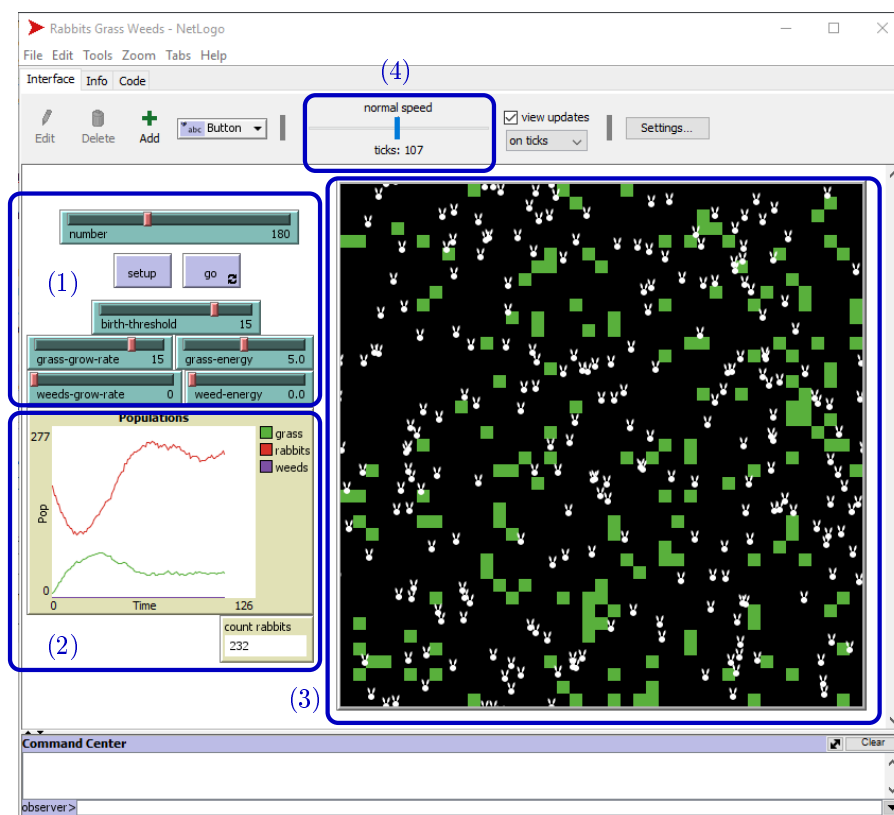


Figure 1.3: Interface for the *Rabbits Grass Weeds* model part way through a simulation run. In this model, the input controls are in the top left section of the interface (1), key information about the simulation is reported in the bottom left section (2) and the world (3) displays the location of rabbits and patches of grass.

Play with the model for at least several minutes. Try out different values for the sliders. Remember to press the ‘setup’ button to initialise a new simulation and then the ‘go’ button to start it running. Here’s some things you should try, and some questions to consider while playing:

- How would you describe the system level behaviour to someone who can’t see the model running? For example, is there a pattern over time for the rabbit population? Does there appear to be a relationship between the different populations (curves in the plot)?
- What happens if you move the slider labelled ‘grass-growth-rate’?
- What about the other sliders?
- Try moving several sliders at once and then just one slider at a time.

- While the simulation has been running for a while, move the ‘birth-threshold’ slider while the model is running.
- Can you work out what is happening within the model just by playing with it?

Do not read past this sentence until you have completed playing with the model. The next paragraphs will discuss some of these experiences and you will not learn as much if you simply read the discussion.

The next step is to look at the Info tab. This contains text written by the modeller. As *Rabbits Grass Weeds* is a library model included with the *NetLogo* software, the text follows a specific format that is geared toward education. It includes a brief description of the behaviour mechanisms within the model, what to adjust and what to look for. There is also some information about aspects of the *NetLogo* language that are particularly interesting. The language discussion can be ignored at this stage and will be revisited at the appropriate point in the tutorial.

1.5.1 Complex Behaviour

As well as introducing you to the basic operation of a NetLogo model, this example is intended to clarify what is meant by complex behaviour. *Rabbits Grass Weeds* is an example of a classic type of model referred to as a predator-prey model, often modelled mathematically with the Lotka-Volterra differential equations (Lotka, 1925; Volterra, 1926). These models have linked cycles of population counts. In this case, rabbits move around the world and eat any grass or weeds they happen to land on; rabbits are the predators and grass (and weeds) patches are the prey. When there are few rabbits, the grass is able to grow. But as there are more grass patches available, the rabbits are able to acquire energy and reproduce. With more rabbits, there is over-exploitation of the grass and a collapse in the rabbit population, starting the population cycle again.

The sliders have meaningful names, so it is likely that you understood at least some of what was happening within the model just from playing with it and thinking about what happens under different conditions. The mechanics of grass growth and rabbit reproduction are relatively straightforward, as the rabbit finds food, it gains energy from that food and then reproduces if its energy is sufficiently high (birth-threshold). What is less clear is that simply being alive costs a small amount of energy and that rabbits die when they run out.

The rabbit population pattern can only be understood through its relationship with the quantity of grass (and conversely). It is possible to observe that the curves move in the opposite direction of each other, and that the grass curve is slightly later than the rabbits curve. Furthermore, the oscillations stabilise over time.

There are several other observable relationships. These are easier to extract by systematically moving one slider and holding the others constant. Return the weed-growth-rate slider to 0 (if you have moved it off 0) and set grass-growth-rate, grass-energy and birth-threshold to moderate values. Then start with different numbers of rabbits. You should notice that the simulation eventually stabilises with about the same number of rabbits, but the shapes of the initial curves are different. This sort of exploration is easy to accomplish with a model, but not feasible in the real-world system that the model is intended to represent.

One aspect of complex systems that you may have experienced is the difficulty in theorising about the micro-level behaviour mechanisms when only the macro-level system behaviour can be observed. Without a clear understanding of causes, it is also difficult to understand the consequences of intervening in the system.

Imagine that you had observed a short period of the cyclic rabbit population in the real world. Many potential interpretations are possible including some unknown predator, a disease that affects fertility, and changes in food quality. Reductionist research techniques such as examining individual rabbits will not help to understand what is happening because those techniques do not examine context and interactions. The understanding difficulty is exacerbated where there are multiple simultaneous changes. Experimenting with the model allowed you to isolate the effect of individual changes and simplify the behaviour (for example, by excluding weeds with

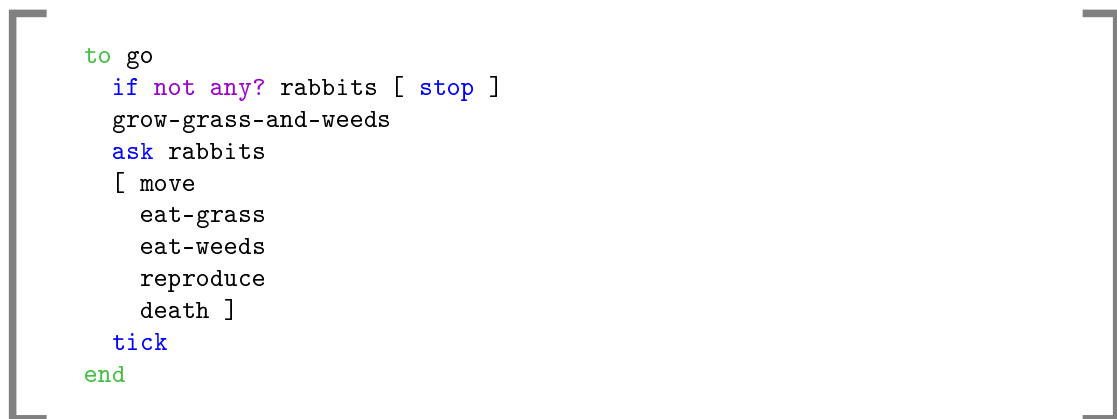
weed-growth-rate set to 0). You also had the benefit of informative variable names. None of this is true when trying to understand real world complex behaviour.

1.5.2 Agent-centric Code

To clarify the concept of agent-centric thinking, we need to look at the programming code. Press the Code tab (in area (2) of Figure 1.1). What you see is the *NetLogo* program that implements the *Rabbits Grass Weeds* model. You are not expected to understand any of the code at this stage. However, you will see that there are several words that are coloured, and that many of these words are standard English words such as ‘ask’. These coloured words are the keywords in the *NetLogo* programming language.

The scroll bar at the right allows you to move through the entire program. It may be surprising to you that the program is very short to achieve so many things, including: grow grass, have rabbits be born and die, respond to the user’s inputs, and visualise the results. The drop down box labelled ‘Procedures’ allows you to move to specific sections of the code.

The ‘go’ procedure is displayed at Figure 1.4. This is the procedure that is repeated while the model is running. In the middle of this procedure is the command to ‘ask rabbits’ to do a list of five things delimited by square brackets. Each of these five tasks is actually the name of a separate procedure (which you can see because all the names are in the ‘Procedures’ drop down box). The first of these is named ‘move’ and is displayed in Figure 1.5 and the second is ‘eat-grass’ (Figure 1.6).



```

to go
  if not any? rabbits [ stop ]
  grow-grass-and-weeds
  ask rabbits
  [ move
    eat-grass
    eat-weeds
    reproduce
    death ]
  tick
end

```

Figure 1.4: The ‘go’ procedure of the *Rabbits Grass Weeds* model contains the main structure of the program. Part of this code instructs each rabbit to do the five actions described in the procedures named in the square brackets.

Looking first at the ‘move’ procedure, this code has the rabbit turn a small random amount to the right and left, thereby ending up facing in a similar but no identical direction as it started, and then move forward. The rabbit’s energy is also reduced slightly.

So the ‘go’ procedure instructs the computer to focus on one rabbit and have it do five tasks. First, the rabbit follows the instructions in the ‘move’ procedure, to slightly change direction, go forward a little, and consume some of its energy store. The next two tasks instruct the rabbit to look at where it is standing and, if there is grass or weeds available, eat the grass or weeds and obtain the energy that it holds (see Figure 1.6). If the rabbit has acquired enough energy, it births another rabbit (not shown). But if there was no food and its energy runs out, the rabbit dies (not shown). After running through all those actions for one rabbit, the program changes focus to the next rabbit until all rabbits have completed the actions.

This group of procedures demonstrates several aspects of agent-centric thinking, and how to implement such a perspective in a programming language. The ‘go’ procedure explicitly changes the focus to each rabbit in turn and the model is implemented through the actions of the rabbits. The rabbit is able to perceive its own environment, in particular whether there is grass or weeds to eat. If so, the rabbit eats that grass or weeds and increases its own energy. The rabbit’s

```

to move ;; rabbit procedure
  rt random 50
  lt random 50
  fd 1
  ;; moving takes some energy
  set energy energy - 0.5
end

```

Figure 1.5: The ‘move’ procedure of the *Rabbits Grass Weeds* model contains the first action that rabbits are instructed to follow in the ‘go’ procedure. First the rabbit performs a right turn (‘rt’) of a randomly chosen amount up to 50 degrees, then a similar left turn (‘lt’). The rabbit then moves forward (‘fd’) a distance of 1 unit. Its energy is reduced by the amount 0.5.

```

to eat-grass ;; rabbit procedure
  ;; gain "grass-energy" by eating grass
  if pcolor = green
    [ set pcolor black
      set energy energy + grass-energy ]
end

```

Figure 1.6: The ‘eat-grass’ procedure of the *Rabbits Grass Weeds* model contains the second action that rabbits are instructed to follow in the ‘go’ procedure. The rabbit looks at the colour of the patch where it is standing (‘pcolor’) and, if it is green (indicating grass), then the patch colour is changed to black and the rabbit’s energy is increased by the amount of energy contained in grass.

action changes its environment, eating the weed or grass means that any other rabbits on the same patch are not able to obtain food. Therefore, although each rabbit operates independently, the behaviour of each rabbit affects the others and there is interaction between rabbits as well as between rabbits and their environment.

As an aside, the code extracts also demonstrate the use of comments. The semicolon indicates to *NetLogo* to ignore the rest of the line (such as the ‘;’ at the beginning of the second line in Figure 1.6). The programmer includes comments to assist human readers to understand what the code is intended to achieve. Comments are discussed in more detail within the tutorial (see section 2.2.2).

1.6 Virus on a Network

The next model to explore is *Virus on a Network*. The *Rabbits Grass Weeds* used a spatial environment, as the rabbits moved around a world of patches with grass. In contrast, this model demonstrates a social environment, represented by links between agents.

As before, open the *Models Library* from the *File* menu (see Figure 1.2). From the library, load the model named *Virus on a Network* (located in the Networks section of the Sample Models section, or search for ‘virus’ using the search box at the bottom of the models list).

After pressing the ‘setup’ button, a network is created. Each node (small filled circle) represents an individual and edges (lines) represent a relationship between pairs of individuals. At the start, some of the nodes are infected and all others are susceptible to infection. Infected nodes have a probability of infecting any susceptible nodes to which they are connected. Once a node is infected, there is a probability of recovering each time step, either becoming susceptible again or immune. In epidemiology, these models are referred to as SIS or SIR models, but are both examples of the broader class of diffusion models. The names denote the available states; SIS for susceptible, infected, susceptible, and SIR for susceptible, infected, removed (indicating death or

immunity).

There are two important sliders to explore the system behaviour. The slider named ‘average-node-degree’ varies the number of edges for each node (good values are about 10 to 30). The slider named ‘gain-resistance-chance’ sets the probability that a node will become immune to further infections when it recovers from the infection.

Play with the model. Remember to press the ‘setup’ button to initialise a new simulation and then the ‘go’ button to start it running. Focus particularly on the how you would describe the system behaviour, and how it varies in response to changes in the slider values. Once you have experimented a little, read the Info tab to get some additional understanding of the model and further ideas about what to explore.

1.6.1 Agent-centric code

The ‘spread-virus’ procedure (Figure 1.7) implements transmission of the infection along network edges. The program selects one of the nodes that is currently infected. It then identifies all nodes that have an edge with that node. If one of these nodes is susceptible, it becomes infected with some probability controlled by a slider on the interface (named ‘virus-spread-chance’).

```

to spread-virus
  ask turtles with [infected?]
    [ ask link-neighbors with [not resistant?]
      [ if random-float 100 < virus-spread-chance
        [ become-infected ] ] ]
end

```

Figure 1.7: The ‘spread-virus’ procedure of the *Virus on a Network* model is the main procedure called by the ‘go’ procedure. Infected network nodes (turtles) check the nodes at the other end of each of the links to which they are attached and, if the node is not resistant, there is a probability of infecting that node.

The actions entailed in becoming infected are implemented in the ‘become-infected’ procedure (Figure 1.8). The node’s infected status is set to ‘true’ so that it will be included in the set of infected nodes in the next iteration of the ‘spread-virus’ procedure. The node is also coloured red for the convenience of the model user, providing a visual representation of the spread of the virus.

```

to become-infected ;; turtle procedure
  set infected? true
  set resistant? false
  set color red
end

```

Figure 1.8: The ‘become-infected’ procedure of the *Virus on a Network* model is called by the ‘spread-virus’ for those nodes that are to be infected. It changes the colour and some status indicators of the node.

1.6.2 Complex Behaviour

In the *Rabbits Grass Weeds* model, the rabbits interacted only indirectly; a rabbit’s action in eating grass made that grass unavailable for other rabbits. In the *Virus on a Network* model, however, interaction is direct. A node transmits the infection to another node with which it has a relationship. This interaction is the mechanism that generates a system level behaviour, that of an epidemic. An epidemic is characterised as widespread infection within a population, and

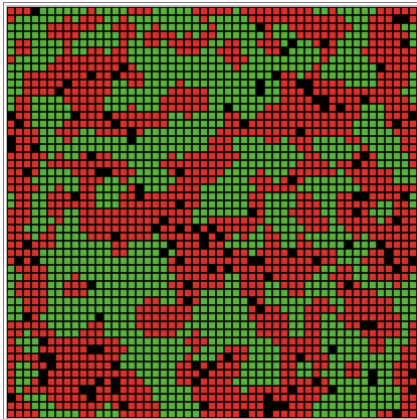
can be seen in the interface plots by an increase then decrease in the infected (red) proportion of nodes. That is, an epidemic is an example of complex system behaviour.

In the classic SIR model of differential equations (Kermack and McKendrick, 1927), the infection is spread by contact but a key assumption is that everyone has an equal chance of coming into contact with anyone else. That is, social networks are ignored. The initially infected people are embedded in a population of susceptible people and are able to transmit with every interaction. Consider the case where, on average, a person is infected long enough to transmit to 2 other people. This means that initially the infection spreads fast, doubling the new infections each generation. Eventually, however, infected people are coming into contact with people who are already infected (or recovered and immune), so the spread slows down. This sequence generates the classic diffusion curve that is observed in the *Virus on a Network* plots, where the number of nodes currently infected initially increases and then decreases. If the number of nodes ever infected is plotted, it would display an S-shaped curve. This is characteristic of diffusion models generally, such as adoption of a new product or other innovation (Bass, 1969; Rogers, 2003).

An SIR epidemic on a network shows similar behaviour, but the transmission is attenuated. As the infection can only be transmitted along network edges, the area of the network where the epidemic is active has a relatively high proportion of nodes that are already infected or immune. Slow the *NetLogo* model (using the speed slider, numbered (4) in Figure 1.1) and try to identify areas where the epidemic is being blocked by an immune node. For an SIR model, the ‘gain-resistance-chance’ slider should be set at 100%. Blocking of the epidemic is more visible (and more extreme) with a lower number of edges per node because there are fewer options for the infection to access other areas of the network.

1.7 More Library Models

Playing with these two models was intended to familiarise you with the NetLogo interface, particularly buttons, sliders, plots and the world display. In addition, you should understand how to access the Code tab and return to the Interface.



Those two particular models were selected because they demonstrate different features of NetLogo: physical and social environments. In both models, the agents (rabbits and turtle nodes) interact with the environment and take some action accordingly. That action can be a decision, such as eating grass, or simply a reactive status change, such as becoming infected. The sequence of actions is the simulated process that generates the complex system behaviour.

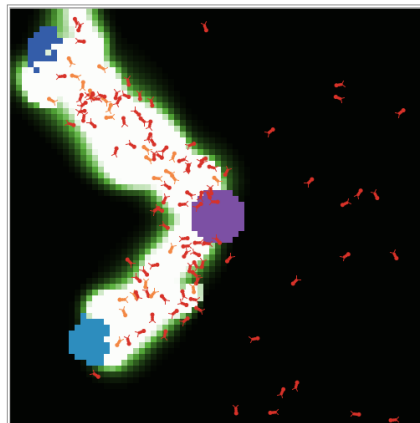
The Models Library contains many other models, demonstrating a wide variety of complex system behaviours. Some are described briefly below, concentrating on the agent-centric perspective and how that generates system behaviour. For these, it is sufficient to read the Info tab and run the model with a few different settings. There is no need to examine the code.

The *Segregation* model (in the Social Science section) implements a classic model (Sakoda, 1971; Schelling, 1971; Hegselmann, 2017) that generates spatial segregation where two types of people end up in mostly separated areas of the world despite being randomly located at the start. The underlying mechanism is that a person moves if the proportion of their neighbours who are of their own type falls below a user controlled threshold. When the simulation stabilises, the average proportion of neighbours who are of the same type is much higher than the given threshold.

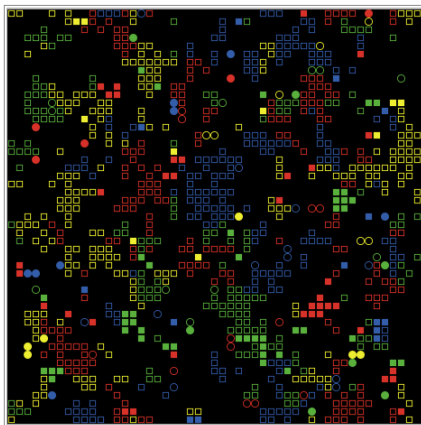
In this model, all agents are identical. The only characteristic of an agent is their desired proportion of neighbours of the same type, and that proportion is the same for all agents.

However, there is heterogeneity in agent locations and therefore in their environments. The agents interact through their contribution to each other's neighbourhoods. When an agent moves, it can trigger a cascade because it occupies a previously empty space and therefore changes the proportion of neighbours in each of the two groups for each of the agents surrounding that previously empty space.

The *Ants* model (in the Biology section) implements a model where each ant interacts only with its own local environment but the entire colony efficiently finds food and returns it to the nest. The ants communicate locally with pheromones to inform each other where the food is located. The ants explore randomly until one (or more) finds food. When an ant finds food, it brings the food back to the nest. While carrying food, the ant leaves a pheromone trail, which gradually spreads out and evaporates. Ants that cross a pheromone trail move in the direction of the strongest pheromone and then reinforce that trail when they find the food. However, ants only respond to trails with a positive but not large amount of pheromone; this ensures they react to the edges of the trail rather than simply following a recently passed ant back to the nest.



The *Ethnocentrism* model (in the Social Science section) is a particular example of evolutionary game theory. This class of models explores the effect on the evolution of the population of repeated interactions between individuals who are able to assist each other at some personal cost. They were initially developed to explain how altruism could arise despite its evolutionary disadvantage (Smith, 1964).



In the *Ethnocentrism* model (Hammond and Axelrod, 2006), each agent has one of four strategies: whether or not it assists (cooperates) with agents of the same colour as itself, and whether or not it assists with agents of a different colour. Each agent also has one of four colours (unrelated to the strategy). Assistance takes the form of giving up some of its own chance to reproduce to increase its neighbour's chance to reproduce. When an agent reproduces, the new agent has the same colour and strategy as its parent, so strategies that confer an evolutionary advantage come to dominate the population.

The final library model to explore is the *Traffic 2 Lanes* model (in the Social Science section). This model returns to our initial presentation of complexity, demonstrating how a traffic jam can emerge from drivers with

different desired speeds reacting to the variations of speed of the driver in front of them.

1.8 Model Design

Now that you are more comfortable with agent-centric thinking and the way in which behaviour mechanisms at the individual level have consequences for complex system behaviour, we are ready to think about the design of the tutorial model, concerning protective behaviour in response to an epidemic. The tutorial model is obviously predefined, but I will go through the same design steps and questions that you would need to follow for your own model. This includes identifying the key entities, interactions and behaviours that are to be included in the code. Such design is an important step in any modelling process and the design is typically refined as the model and the modeller's thinking evolve. The design will be revisited throughout the tutorial to orient the

coding of each element.



Thinking about your model: conceptual design

1. What process (or processes) is to be simulated?
2. How to identify the relevant features and relationships for each process? Possibilities include: existing subject matter knowledge, advice, experiment, observational data, literature.
3. What agents are required to implement the process?
4. What characteristics of the agents are relevant to the process? This includes both characteristics that affect actions, and status indicators.
5. What form does the environment take: abstract spatial, realistic spatial, social network...?
6. What environmental features influence the process or are affected by the process?
7. How do the agents and environment interact? This includes indirect interaction such as perception as well as input to actions and consequences of actions.
8. How to operationalise the process(es)?

The first consideration is what to model. In most situations, the research question is decided first and then agent-based modelling would be chosen as the methodology because it is the most appropriate to respond to that particular question. Many questions concern whether an individual level mechanism is a plausible explanation for an observed system behaviour (such as the *Segregation* and *Ethnocentrism* described in section 1.7). Agent-based models are also useful for ‘what if’ explorations, for example to gain insight into the potential consequences of policy options.

The question for this tutorial is of the ‘what if’ type: How is epidemic spread affected by protective behaviour? Potential scenarios could examine the effect of increasing the attitude of all individuals by some amount to inform a communications campaign, or identify whether there is a minimum efficacy for a protective measure to have any significant reduction in epidemic impact.

Having determined the question, it must be refined into a conceptual design for the model. There are three key elements: process, entities, and interactions. The conceptual design would typically include operationalisation of these elements, but these aspects will be partly deferred until required during model construction.



Operationalisation

Operationalisation is the process of specifying the detailed model rules that are to be translated into code. This includes how each relevant feature is measured within the model, how the relevant features that influence any action are combined, what is known to each agent, and each possible situation and its consequences.

As an agent-based model is a simulation, the focus of the design is the process or processes to be modelled. There are two processes in the tutorial model, adoption (and abandonment) of protective behaviour, and the epidemic (which includes both infecting new people and disease state changes for infected people). In a real model, the design for each process would be informed by some combination of existing subject matter knowledge, advice from subject matter experts (including disciplinary and local stakeholders), experiment, empirical data, and published literature.

Consider first the protective behaviour process. This process is a series of decisions by individuals or people, the main agent in the model. For the purposes of this tutorial, we assume that there are only three influences on a person’s behaviour: their attitude or belief that the protective

behaviour is useful, the behaviour of people around them, and the extent to which they feel threatened by the epidemic. This brings in one personal characteristic, one for social norms, and one that responds to the other process being modelled - the epidemic.

Attitude is to be operationalised as a value from 0 to 1, norms as the proportion of nearby people who have adopted protective behaviour, and threat as a function of recent nearby new infections. In addition, people communicate with their friends about threat in their location. If the weighted sum of attitude, norms and threat is above a threshold value, then the person will adopt (or maintain) the protective behaviour, and will abandon it once that weighted sum falls below the threshold. Further details of the protective behaviour operationalisation are provided in Chapter 4, where this process is coded.

The influenza epidemic process is well studied and the standard mathematical operationalisation is the SIR model introduced for the Virus on a Network model (see section 1.6). Although influenza is spread from person to person, it is not necessary to individually model infectious people coming into contact with susceptible people. The virus survives on surfaces and in the air for several hours so it is reasonable to instead treat the epidemic at the population level, and infectivity as a characteristic of the spatial environment.

Epidemic spread is to be operationalised with differential equations that describe the change in the proportion of the population in each of three states: susceptible (never infected), infected, and recovered. The equations will be spatially explicit, so that the infected population can only infect others who are located nearby. Further details of the epidemic operationalisation are provided in Chapter 3), where this process is coded.

The two processes interact with each other. If the epidemic is active nearby, with a large number of new infections, that increases a person's likelihood to adopt protective behaviour via the threat component in the behaviour operationalisation. As the proportion of people who have adopted protective behaviour increases, epidemic transmission reduces, with the reduction also depending on the efficacy of the protective behaviour. The influence flows of the conceptual design are summarised in Figure 1.9.

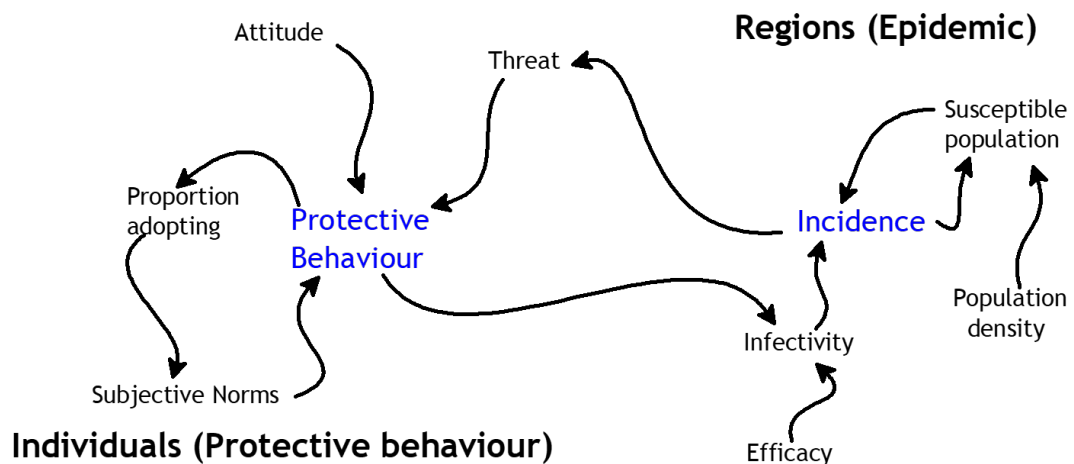


Figure 1.9: The broad design of the tutorial model. Individuals adopt protective behaviour based on some combination of their own attitude, the extent to which they observe other individuals adopting protective behaviour, and their perceived threat. Protective behaviour affects the transmission potential of the epidemic, which in turn affects the actual spread and hence the perceived threat.

The key elements of the design are:

- Agents: people
- Environment: spatial for epidemic, social for information about threat
- Resources: none

- Interactions: agent/agent - norms, environment/agent - threat, agent/environment - reduction in infectivity
- Attributes: agent - attitude, environment - population in epidemic states

The conceptual design is only one of the steps in designing a model (Badham et al., 2019). It is also important to design the interface, identifying the parameters that the user may wish to adjust and creating easily interpreted results and visualisations. The design may also consider the data that will be available for calibration to ensure that there is sufficient detail to set appropriate values for the operationalised processes. Only a basic interface will be developed for the tutorial.

1.9 Doing the Tutorial

The remainder of the tutorial starts from an empty model. You will enter sections of code into the Code tab and create controls on the Interface tab (like the buttons, sliders and plots in the library models).

The complete instructions for building the model are presented in snippets, each of which contains both instructions (upper section) and text to be typed into the Code tab (lower section). Simply typing in the text will not create a working model. The controls on the Interface tab are automatically linked to the code and instructions for creating those controls must also be completed.

The snippets are organised so that the model is constructed iteratively. At the end of each group of snippets, there will be an instruction to run the model. Trying to run the model at other points in the tutorial will not break anything, but may generate errors.

Key concepts are introduced immediately before you are expected to enter code to implement that concept. Don't worry if the concept does not immediately make sense, there will be repetition. However, you should spend some time thinking about how the code relates to the concept that has just been introduced.

In addition, any keywords that you are expected to learn are highlighted in separate box. Early in the tutorial, only some of the keywords will be highlighted and the others will be deferred to later. By the end of the tutorial, all new keywords will be highlighted. Whenever you see a highlighted keyword, you should try to understand it, but also look it up in the *NetLogo* Dictionary (see the Help menu, or the *NetLogo* website).

1.10 Tutorial Progress Check

That completes the preliminaries, you are now ready to start constructing the model. You should be able to navigate an existing *NetLogo* model, switching between the interface where you use the model and the Code tab that contains the program. You should also be able to use a *NetLogo* model to explore the behaviour of a complex system by running the model with different inputs, and understand why an agent-centric perspective of a process provides insight into that behaviour.

You should also have a general sense of the structure of the tutorial. There will be a short discussion of some agent-modelling or programming concept, then a set of instructions (including code to enter) that provide a demonstration of that concept. At various points in the tutorial, you will be instructed to run the model and also to reflect on the most important concepts.

Finally, you should have a broad picture of the model that is to be built in the tutorial. In particular, the model will contain individuals making decisions about protective behaviour based on their own attitude, the behaviour of other individuals around them, and their perception of the epidemic threat. Simultaneously, an epidemic will be simulated in the environment. These two processes influence each other.

Model 1: Model Entities

The first version of the model establishes the world and the agents that populate the world. However, there is no behaviour or simulation as there is no progression of time, so no processes are modelled. This version will also be used to introduce some basic *NetLogo* concepts, and take the first steps in good programming practices.

2.1 The *NetLogo* World

We will be starting with a completely empty model. There are two steps: creating a file, and defining the world. These are described in detail below, and are summarised at Snippet 0 (page 21) for completeness.

2.1.1 Create a new model

Open *NetLogo*. If you already have it open from the previous activity of running library models, create a new model ('New' in the 'File' menu). You should have an empty model as in Figure 1.1.

Save the model ('New' in the 'File' menu) in whatever file location you prefer. The name is arbitrary, but an appropriate name is *epidemic*, or *epidemic v1* if you want to save multiple versions at different points in the tutorial. If you do choose to save multiple versions, simply use the 'Save As ...' option in the 'File' menu when you want to start a new filename.

The file will be created with the extension *nlogo* automatically. That file extension lets the computer know that it is a *NetLogo* file. It also means that you can in future open *NetLogo* with the model already loaded by double-clicking on the model in the computer's file manager.

2.1.2 A world of patches

The *NetLogo* world (the area numbered (7) in Figure 1.1) is comprised of a grid of patches. Patches are one of the fundamental objects in *NetLogo*. They store information about the spatial environment and are also used for visualisation.

Each patch has a pair of co-ordinates, *pxcor* and *pycor*. The default geometry has 1089 patches in the 33x33 layout, with patch 0 0 at the centre of the world and both *pxcor* and *pycor* ranging from -16 to +16. Furthermore, the world is wrapped, for example an agent that moves off the right of the world reappears at the left.

pxcor, pycor

pxcor and *pycor* are the horizontal and vertical co-ordinates respectively of a patch.

For the tutorial model, we want to increase the number of patches to 51x51 and turn off the wrapping. We also want to reduce the size (amount of screen) for each patch so that the additional patches don't make the world too large for the screen. These changes are all made from the settings dialogue box.

Press the 'Settings...' button (top right of screen) to open the dialogue box that controls the World. The dialogue box is displayed at Figure 2.1, with the settings to be changed highlighted. Change the values of 'max-pxcor' and 'max-pycor' to 25, and the patch size to 8. Uncheck both the horizontal and vertical wrap checkboxes. Once the settings are correct, press the 'OK' button to apply the settings and close the dialogue box.

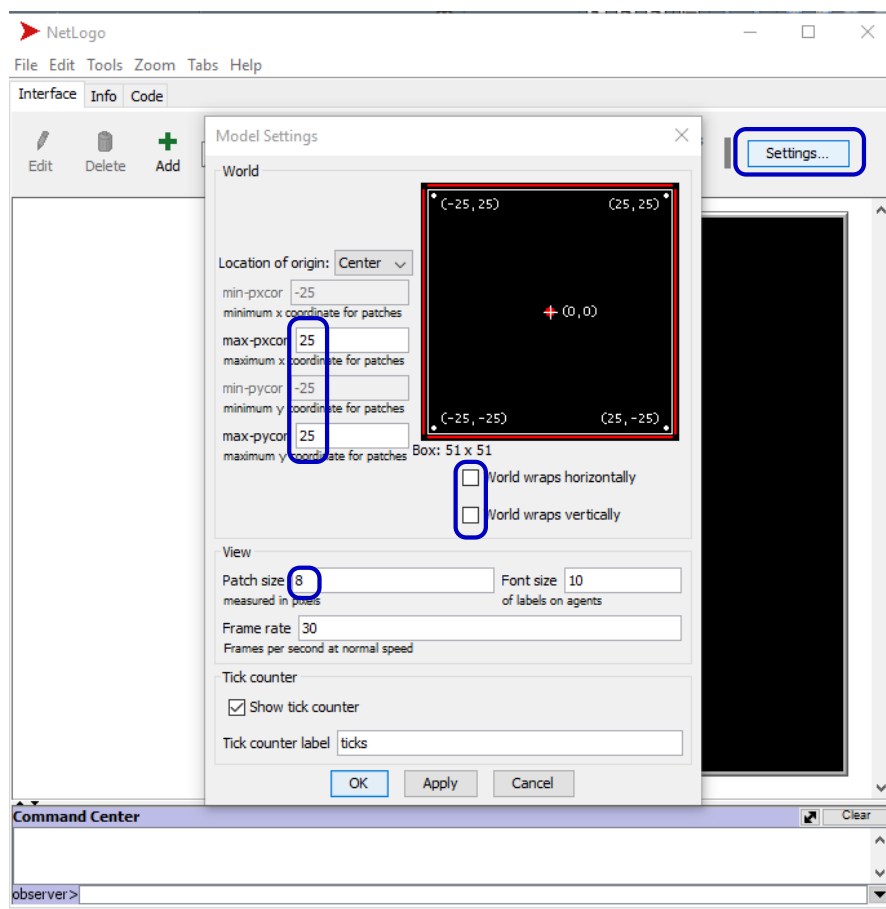


Figure 2.1: Screenshot of an empty *NetLogo* model with the Settings dialogue box open. Click on the 'Settings...' button at the top right of the screen to open the dialogue box. The highlighted settings should be changed to match the settings displayed.

The World and other interface components can be positioned anywhere within the *NetLogo* interface. A pop-up context menu is available (typically accessed by right-click on Windows systems, Ctrl-click on Mac OS) that has choices for Edit, Select and other options. Note that the Edit option is an alternative way of accessing the Settings... dialogue box because these are the settings to edit for the World.

First, make the Interface larger by dragging its edges or corners (with the mouse). Second, choose the Select option from the context menu. A grey box will surround the World with eight small black squares. That box indicates that the World is currently selected. Dragging the grey box will reposition the World and dragging the small black squares will resize it. Drag the World to the right side of the now larger Interface.

Snippet 0: Create new model

1. Open *NetLogo* (if closed) or create a new model (if open).
2. Amend world settings as shown in Figure 2.1.
3. Adjust the window size and World location.
4. Save the (empty) model.

2.2 Fundamental Coding Concepts

At this point, we are ready to start entering text into the Code tab. The name *NetLogo* is used for both the software that runs models like those already opened, but also to the programming language in which the model is written. This means that *NetLogo* is more than the set of keywords that provide instructions about how the model works, but also the architecture that allows moving a slider on the Interface tab to change the way in which the model interprets those instructions. Both the code and the interface controls will be introduced iteratively.

2.2.1 Procedures

The main *NetLogo* code is comprised of procedures: blocks of code that perform a task. Each procedure is delimited by the keywords `to` and `end`, and the name of the procedure immediately follows the keyword `to`. You, as the model builder, can name the procedure whatever you like, but good practice is to use a name that is reasonably short but also descriptive. For example, the names of procedures already seen include ‘eat-grass’ (Figure 1.6) and ‘spread-virus’ (Figure 1.7).

to <procedure-name> <instructions> end

A *NetLogo* procedure is a block of instructions to the computer. The keywords `to` and `end` mark the beginning and end of the block respectively. The first word following the keyword `to` is the name of the procedure.

A second type of procedure is delimited by the keywords `to-report` and `end`. These are discussed later in the tutorial (see section 4.3).

Many procedures call other procedures. That is, the block of code for one procedure includes the name of another procedure. *NetLogo* implements the instructions line by line starting at the top of the procedure. When it gets to a call to a different procedure, implementation moves to the start of that other procedure, running from the top to the bottom. Control is returned to the original procedure when the `end` keyword is reached, and any remaining instructions are completed. *NetLogo* programs can have several levels of procedure, with one procedure calling several others, which each call further procedures and so on. For example, in the *Virus on a Network* model, the ‘go’ procedure called the ‘spread-virus’ procedure, which (with some probability) called the ‘become-infected’ procedure. Eventually, the procedure that is deepest in the calls gets to its `end` statement and the implementation steps back up a level and continues until either a call to another procedure or the `end` of that higher procedure.

2.2.2 Comments

All programming languages have a way to leave comments in the code. These are for a human reader to help them interpret what the code is intended to do (Knuth, 1984). The instruction to the computer is to simply ignore them.

In *NetLogo*, a comment is indicated by a semicolon (;). The semicolon and anything after it on the same line is skipped when the computer interprets the program.

Comments are useful to explain what a section of code does but, also, why that section of code is needed. For example, the comment in the *Rabbits Grass Weeds* move procedure (Figure

comment (;)

The computer ignores anything appearing between a semicolon (;) and the end of that line. It is used to indicate a comment from the programmer to any humans reading the code.

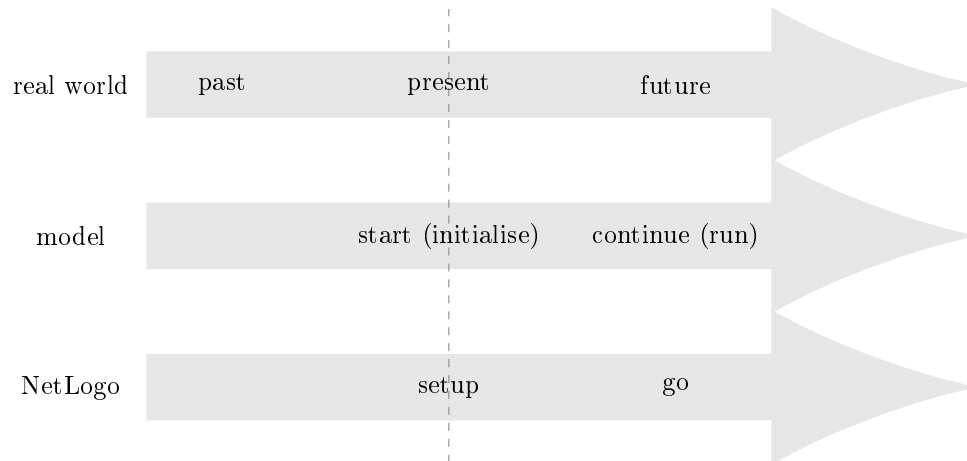


Figure 2.2: A model must be initialised to describe the state of the world at some arbitrary start point, which also captures the outcomes of actions taken in the past. It is the run to estimate the future. In *NetLogo*, this separation is typically constructed in a *setup* procedure for the initialisation, and a *go* procedure that represents the actions occurring during the model run.

1.5) states that ‘moving takes some energy’, which explains the purpose of the following line of code where energy is reduced. The line of code is straightforward to interpret, so it would be uninformative to have a comment that simply says ‘reduce energy’.

In this model, we also use comments as headings for groups of procedures. The dropdown list of procedures is ordered alphabetically. This is sufficient for small models or if you are able to remember the name of the procedure you want to find, but grouping procedures provides an alternative way to find the one you need. Procedures can be organised by however is convenient for the modeller. The group heading is written over multiple lines to make it easily visible when scrolling through the model.

2.3 Start and Continue

The current state of the real world is a snapshot in time. Focusing on the process of interest, the current situation incorporates what has already occurred in the past. In some sense, time can be broadly broken into past, current and future.

A simulation, however, must start at some specific point in time so there must be an initial state. The past is represented in the simulation only by its effect on the initialisation. Of course, the past from our current perspective can be simulated by setting the initial point of time even further in the past, but that simply moves the problem.

All simulations differentiate between initialisation (the starting state) and the rules that step through the process being simulated (see Figure 2.2). In *NetLogo*, it is conventional to construct this differentiation as two main procedures.

The *setup* procedure contains all the code necessary to initialise the model. This would typically include creating all the agents and environment, and setting values for attributes of the agents and features of the environment. For example, in the *Rabbits Grass Weeds* model, the *setup* procedure created rabbits and grass, and assigned starting energy values for each rabbit.

The *go* procedure contains all the code necessary to implement the process. This is the code that

contains the interactions, decisions and actions that describe the model behaviour and change the agent attributes or environmental features. For example, in the *Rabbits Grass Weeds* model, the *go* procedure had the rabbits move, eat, reproduce and die.

2.4 Create Model Control Buttons

Almost all *NetLogo* models have two main control buttons; by convention these are named ‘setup’ and ‘go’. The setup button is used to generate the starting point of the simulation, and the go button is used to run the simulation by stepping through time.

Buttons are just one of the many widgets available. A widget is any component of the interface, typically used to control the model or report information from the model. In the library models, you have used buttons to start the model, sliders to enter values for model settings, and seen plots of values generated by the model. Buttons, sliders and plots are all types of widgets. Technically, the World display is also a widget.

To add a button, make sure the dropdown box (numbered (1) in Figure 2.3) is set to Button. Click the Add icon (numbered (2) in Figure 2.3) and then click on the interface where the button should be positioned. Alternatively, you can add a widget by opening the pop-up context menu in the interface (right-click for Windows, Ctrl-click for Mac OS) and selecting ‘Button’. This will open a dialogue box (Figure 2.4) to request the settings for the button you wish to add.

First create the setup button. As in Figure 2.4, enter the word ‘setup’ in the *Commands* box, and also in the *Display name* box. Press *OK* and a button will be created labelled ‘setup’. The display name is the text to place on the button. The command box can contain any valid *NetLogo* commands. In this case, the button will call a procedure named ‘setup’. However, that procedure does not exist so the label will be in red text to indicate an interface error.

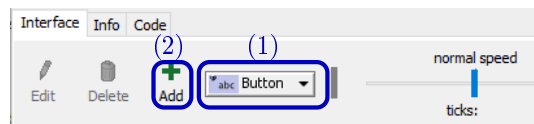


Figure 2.3: The menu icons to open the dialogue box to create a button on the Interface. The dropdown box (1) is used to select the type of control widget to add. Then click on the ‘Add’ icon (2) to change the mouse cursor into a + and click on the Interface to position the button.

Create a second button. For this one, the command is ‘go’. In a typical *NetLogo* button, the ‘go’ button would call a ‘go’ procedure and be labelled with the text ‘go’. For this tutorial, however, to demonstrate that this consistency is not a requirement, set up the button to call a procedure named ‘go’ (command box) but label it with ‘do it!’ (display name box). For this button, check the ‘Forever’ checkbox. Checking ‘Forever’ adds a circular pair of arrows icon to the button but will not actually do anything until time is introduced to the model. Press *OK* to complete the second button.

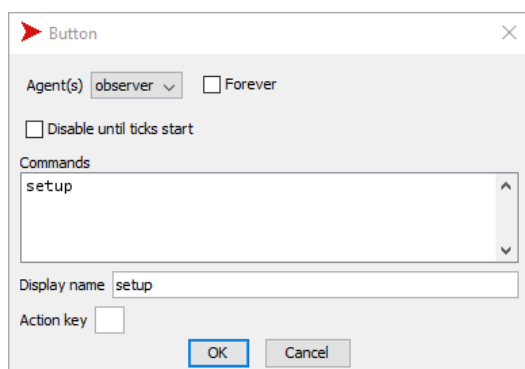


Figure 2.4: Dialogue box for setup button.

The next step is to enter the code in the bottom part of Snippet 1a (page 24). Switch to the *Code* tab and simply type it in. The code contains the two procedures called by the buttons, but neither procedure has any content. Instead, each procedure simply has a comment. In addition, there is a heading comment stating that these are the main control procedures (make sure you have a semicolon at the beginning of each line). The box style frame and upper case is intended to make the heading more visible as you scroll through the code.

As soon as you start typing in the *Code* tab, the *Check* icon (a tick mark) turns green. Once you have completed typing in the code, click on that *Check* icon. This will check the

code for errors such as mismatched brackets and words that *NetLogo* doesn't know what to do with. These are typically typographical errors such as misspelling a keyword or procedure name. If there is such an error, a yellow bar will be placed at the top of the screen together with an error message (see Figure 2.5). In addition the *Code* tab label will be red to indicate there are unresolved errors. Unfortunately, there are many errors of other types (such as logic errors) that *NetLogo* does not identify automatically.

If you do have errors, make the appropriate changes to the code and *Check* the code again. Once there are no errors (and the *Check* icon returns to grey), save the model. Look at the interface, the button labels should be black text. Now that the procedures exist, *NetLogo* is able to interpret the button command to call the procedure.

Snippet 1a: Buttons

1. Create a button labelled 'setup' to call the procedure named 'setup'.
2. Create a button labelled 'do it!' to call the procedure named 'go'.
3. Enter the following code to create the 'setup' and 'go' procedures.
4. Check the code (green tick) and Save the model.

```

;-----
; MAIN CONTROL PROCEDURES
;-----

to setup
  ; initialisation procedures go here
end

to go
  ; model step procedures go here
end

```

You can now run the model. Go to the interface and press the *setup* and *go* buttons. Nothing will happen, but *NetLogo* should not give an error either.

At this point, the model looks like the screenshot at Figure 2.6. Buttons can be moved in the same way that the World can be moved. Access the context menu, press 'Select' and the grey box can be used to drag the button as required. Once the widgets are in their appropriate places, model version 1a is complete. Save the model (Save item in the File menu).

2.5 Variables

A variable is simply a way of storing information. You can think of it as a box with a labelled lid. The actual contents of the box can change, and the box may even be empty, but the label stays the same. Variables provide a convenient way of keeping track of information that the computer needs to know, such as the energy of a rabbit in the *Rabbits Grass Weeds* library model, and referring to that information as required in the code.

The value of the variable (or contents of the box) can be a number (eg 10), string (eg "high"), boolean value (**true** or **false**), agentset (see section 4.4) or list. There is no need to specify the type of contents for the variable.

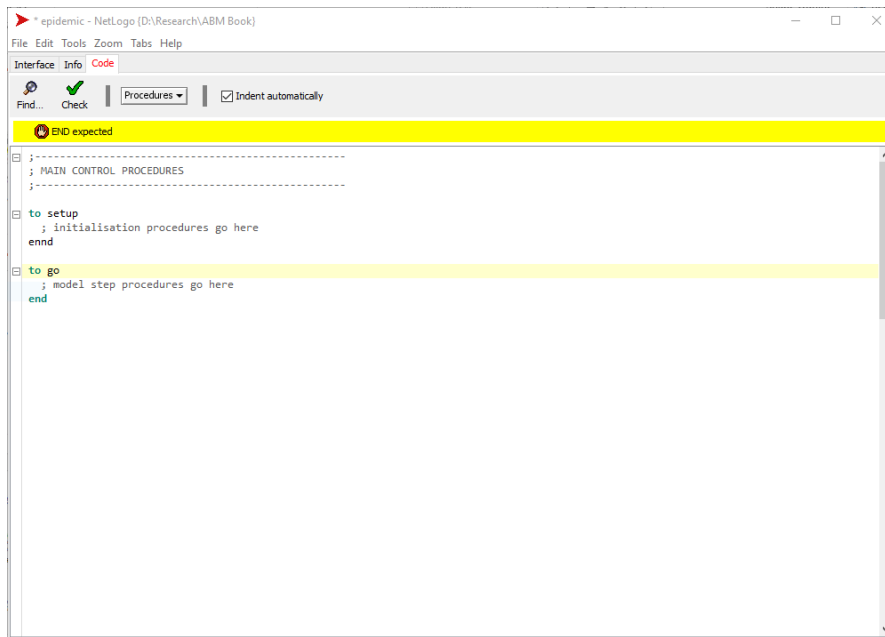


Figure 2.5: Screenshot of a typical error message presented after clicking on the green tick *Check* icon. In this case, the word 'End' is mistyped to end the 'setup' procedure, so the error message is reporting that a new procedure is being started (with the **to** keyword) before the previous procedure has been terminated (with the **end** keyword).

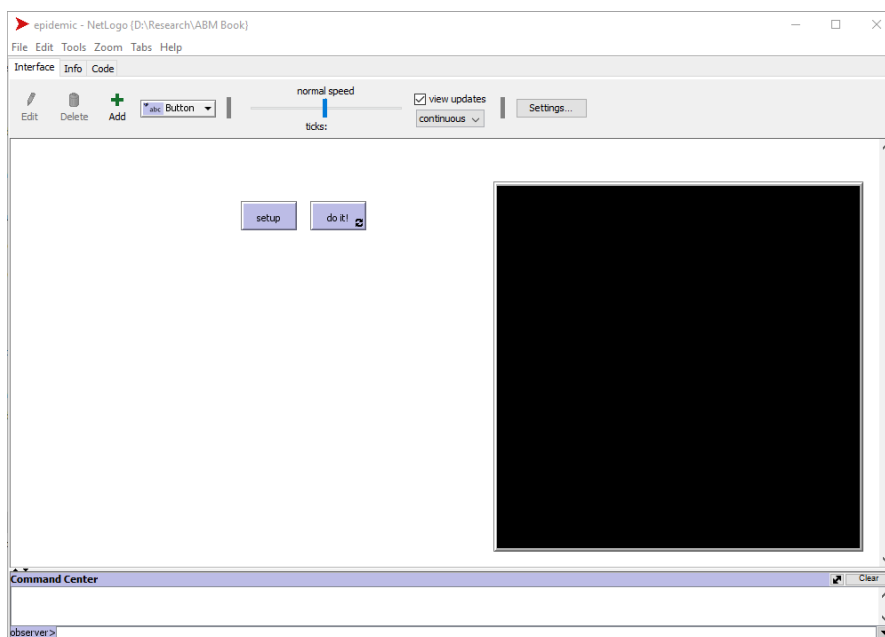


Figure 2.6: Screenshot of the interface after completing version 1a of the model.



Variables

A variable is a way to link a meaningful name to a piece of information that can be stored, changed and retrieved. There are three types of variables in *NetLogo*: global variables contain information available to all model entities, agent variables store information about individual agents (including patches or links), and local variables are temporary storage.

2.5.1 Variable names

Variable names are for the convenience of the programmer (and model user), and should therefore describe the information being stored. On the other hand, variable names should be fairly short so that they can be accurately typed when writing the code, and also not interfere with reading the code. Only certain characters are permitted in names: letters (a-z, A-Z), numbers (0-9) and some special characters (including `?` `#` `-` `_` `%`).¹ Other special characters (including `@`) will create an error if included in a variable name. The name can start with any of the permitted characters except for a number.

The most common convention in *NetLogo* variable (and procedure) names is to use all lower case and to join words with a hyphen (-) for more descriptive names. This leads to some potential confusion between variable names and procedure names but, in practice, variable names tend to be single words and procedure names tend to be several words so the confusion does not arise. This is the convention used in the *NetLogo* Models Library and in this tutorial.

Some other naming conventions for joining words are to use an underscore (`_`) or camel case (upper case at the start of each word, for example `ExampleVariableName`). The convention you adopt is not important as long as you adopt a convention; consistency reduces opportunities for error.

Apart from connecting words, symbols are not generally used in *NetLogo* names, with three important exceptions. The first is that a question mark (`?`) is always used as the last character for a variable that is only true or false, and never used in names otherwise. The second exception is that some models use a special character to start variable names within procedures that take arguments. In this tutorial, the hash character (`#`) is used for that purpose (see section 4.3), but the underscore (`_`) is also common. Finally, variables that contain percentage amounts often have the percent character (`%`) at the end.

2.5.2 Accessing variables

NetLogo must be informed about a variable's existence before it is actually used. This allows *NetLogo* to allocate some storage and link the name to that storage. Declaring a variable's existence is done differently for each of the variable types.

Global variables are declared at the top of the code with the `globals` primitive followed by a list of variable names. In addition, widgets such as sliders that take values from the model user automatically create a global variable of the name given to the widget. Variables that store information about an agent or link are created when the agent or link is created. However, *NetLogo* needs to know what variables to create and they are also declared at the top of the code, but with `turtles-own` and similar statements. Similarly, a `patches-own` statement is used to name all the variables associated with each patch.

Local variables are temporary storage that is forgotten once the relevant piece of code has finished running. These are created with a `let` statement, which both declares that the variable exists with the specified name, and stores the results from some piece of code into that variable.

The contents of the variable that is known to a model entity are accessed simply by using the name of the variable. This includes global variables and its own variables. In addition, turtles are aware of the variables owned by the patch where they are located. However, if an entity wants to use information that it does not automatically know, then the code must also specify the entity that owns the relevant variable (using the `of` primitive).

2.6 Describing the Physical Environment

A key factor in the spread of an influenza-like epidemic is population density. As the epidemic is to be simulated by the patches, representing the physical environment, the patches will need a

¹Technically, mathematical operators are also available, but I recommend avoiding them entirely as it is too easy to confuse manipulating multiple variables with referring a single variable (for example, `income + capital` adds two variable values together, and `income+capital` is a variable name).

variable for their population density. However, all patches are of the same size, so we will instead use population because that makes the transmission equations easier conceptually.

We will also introduce a global variable for the maximum population of any patch and show its contents on the interface. This allows a sanity check, which is a quick assessment of whether results are reasonable. For example, if we write code that is intended to assign population values in the thousands, but the maximum population shows millions, then there is an error in the assignment method. Similarly, if the maximum population shows 0, then population values are not being assigned. Such sanity checks are essential while building the model to make sure that the model being built is actually consistent with its design.

Create a variable to store the maximum population value (Snippet 1b-1 (page 27)). This snippet introduces the `globals` keyword, which is used to list the global variables. Note the short descriptive variable name, and that a comment has been used to provide a more detailed description.

Snippet 1b-1: Define global variables

1. Enter the following code, starting at the first line at the top of the *Code* tab.
2. Check the code (green tick) and Save the model.

```
globals
[ max-popn                ; maximum population of a patch (for
  colour scale)
]
```

globals

The `globals` keyword is used to identify all the global variables except those that are controlled by the widgets that provide information to the model (sliders, switches, choosers, and inputs). The variable names are listed between square brackets.

This code only instructs *NetLogo* to create the variable. We need to separately create a monitor widget to display its value, and set a value to display. A *monitor* is an interface widget to show the contents of a single variable. It is created in a very similar way as a button. Go to the *Interface* tab and add a monitor (change the widget dropdown to ‘Monitor’ and click on ‘Add’, or access the context menu).

The dialogue box (see Figure 2.7) has a box for *NetLogo* code and the Display name, which is the same as the button widget. It also has settings for how the results are displayed, such as decimal places. As the contents of a variable are accessed simply from the name, the variable name is the only code required in the Reporter box.

At this point, the monitor should display 0. A *NetLogo* variable, once created, has a value of 0 until it is `set` to some other value. The global variable named `max-popn` is created by the `globals` statement and assigned the value of 0 by default. But patches do not yet have a variable for population, so no maximum can be calculated and assigned to `max-popn`.

For *patches* (and *turtles* and *links*), the variables available are declared with the `patches-own` keyword (and similarly for the other entities). Each patch has its own copy of the variable with potentially different values. First, we must create a variable to store the population value for each patch, to be named ‘popn’ (see Snippet 1b-2 (page 28)).

Having made the variable available, the next task is to assign a suitable value of population to each patch. In the real world, population density varies greatly. Typically, most people will live in one of a small number of cities. For the model, three patches will be assigned large population values, with small values for all other patches.

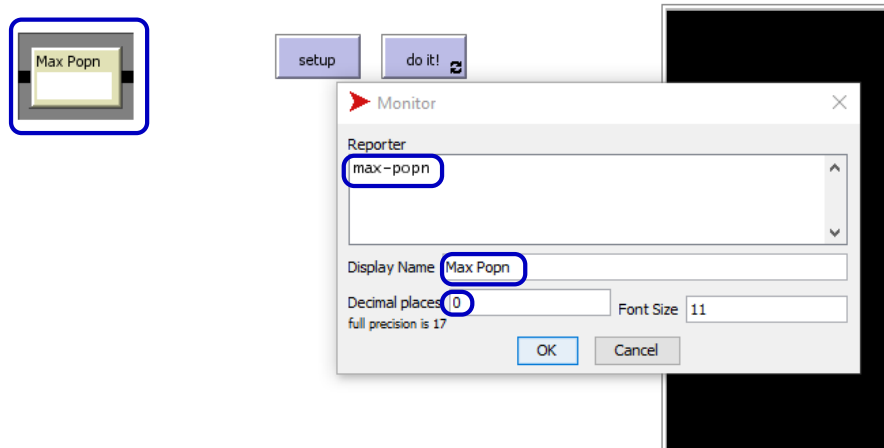


Figure 2.7: Partial screenshot with dialogue box for maximum population monitor. Note that the title of the monitor matches the text in the ‘Display Name’ box.

Snippet 1b-2: Declare patch attributes

1. Enter the following code below the globals declaration but above the heading for the main procedures.
2. Check the code (green tick) and Save the model.

```
; patches have epidemic information
patches-own
[ popn                ; population of patch
]
```

Snippet 1b-3 (page 29) creates a new procedure (called `setup-patches`) that uses the `random` primitive to generate random numbers and assign them as population values. It also calculates the maximum of these and changes the value of the `max-popn` variable to that maximum. The new procedure is called by the `setup` procedure simply by including the procedure name in the code for the `setup` procedure. This means that clicking on the `setup` button starts the `setup` procedure, and the `setup` procedure in turn diverts to the `setup-patches` procedure.

There are several new primitives introduced in this snippet. In `setup-patches`, `random 10000` returns a randomly generated integer from 0 to 9999 and that number is added to 2000 to assign a population count (with the `set` primitive) to each patch. Three of the patches are randomly chosen for a higher population count, representing cities. For these, `random 3` is used to generate a random number from the set {0, 1, 2}, which is then added to 3 and multiplied by 1 million. So the city patches each have a population value of 3 million, 4 million or 5 million. Other new primitives are explained later in the tutorial.

Having added some code, the next step is a sanity check, does it work as expected? The green tick only checks syntax. Logic errors in agent-based models can be difficult to diagnose, so it is important to add code gradually and verify it before adding any further code. You should have

patches-own

The `patches-own` keyword is used to identify all the variables (listed between the square brackets) that are available to patches to describe attributes of the spatial environment. Each patch has its own value for each of the variables named.

Snippet 1b-3: Assign patch populations

1. Edit the 'setup' procedure to delete the comment and replace with the code below, which also calls a new procedure named 'setup-patches'.
2. Create the 'setup-patches' procedure (in a new section at the bottom of the code with the heading comment "Implementation Procedures").
3. Check the code (green tick) and Save the model.

```

to setup
  clear-all
  setup-patches
end

to go
  ; model step procedures go here
end

;-----
; IMPLEMENTATION PROCEDURES
;-----

to setup-patches
  ask patches [ set popn 2000 + random 10000 ]
  ask n-of 3 patches [ set popn 1000000 * (3 + random 3) ]
  set max-popn max [popn] of patches
end

```

random

Generates an integer between 0 and n-1 where n is the number immediately following the word `random`. Each integer in the range is equally likely to be generated.

a clear idea of what to expect; after all, the code is intended to implement the model design.

The most obvious check is to press the *setup* button and look at the number in the Max Popn monitor. From the discussion of how the setup-patches procedure is intended to work, the value should be 3, 4 or 5 million. Because the maximum is randomly generated, repeated button pressing should also lead to some changes in the number returned. Note that a common error is to create a new procedure, but forget to include the call to that procedure in one of the main procedures. In this case, the value of max-popn would never change from its default of 0. You can see this by commenting out the call to the setup-patches procedure in the setup procedure.

**Sanity check**

Whenever you add code to your model, do a sanity check. Test the code to see if it does what you expect it to do. The green tick only checks syntax, not logic.

This check only confirms that appropriate values are assigned for the patches representing cities. What about the others? Checking can be made more rigorous by commenting out the line that generates the city populations (by placing a semicolon at the beginning of the line), the maximum population value generated for other patches will be displayed in the monitor. You should expect to see a value slightly below 12,000. Remember to delete the semicolon after checking.

At this point, the model looks like the screenshot at Figure 2.8. Once the maximum population is correctly calculated after clicking the *setup* button, model version 1b is complete. Save the model (Save item in the File menu).

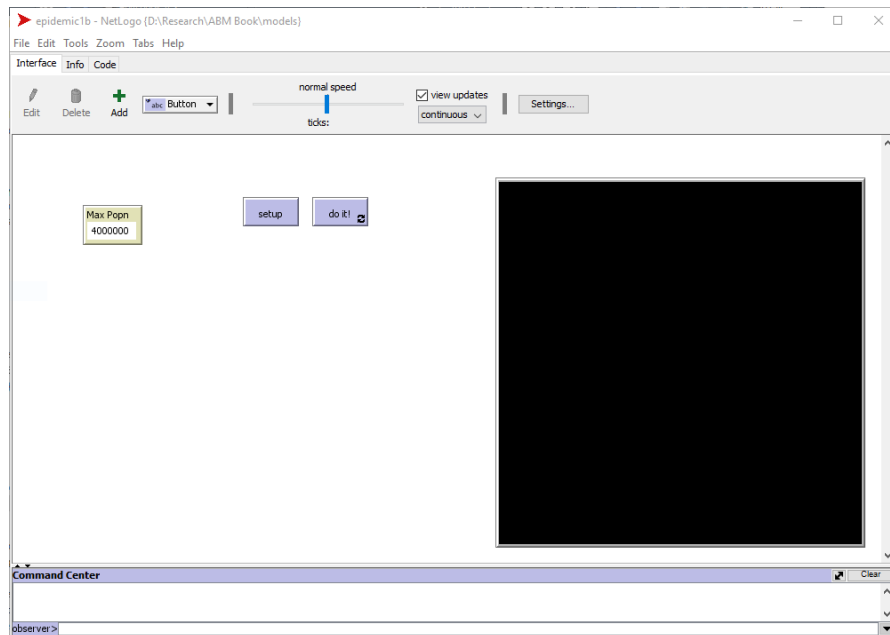


Figure 2.8: Screenshot of the interface after completing version 1b of the model.

2.7 Doing Mathematics

Consider equation 2.1:

$$x = 1 + 2 \times 3 - (4 + 5) \div 6^2 \quad (2.1)$$

How would you calculate the value of x ? I'm hoping you don't start at the left and work your way along (so $1 + 2 = 3$ then $3 \times 3 = 9$ then ...). You should start with the brackets, then do the 'power of' (called exponentiation), multiplication and division, then finish with the addition and subtraction. Breaking the calculation into small steps:

$x = 1 + 2 \times 3 - (4 + 5) \div 6^2$	step done:
$= 1 + 2 \times 3 - 9 \div 6^2$	$4 + 5 = 9$
$= 1 + 2 \times 3 - 9 \div 36$	$6^2 = 36$
$= 1 + 6 - 0.25$	$2 \times 3 = 6$ and $9 \div 36 = 0.25$
$= 6.75$	$1 + 6 - 0.25 = 6.75$

In mathematics, the convention for ordering of the calculations is referred to as precedence. Programming uses the same term, but the operators that must be considered are more than simply $+$, \times and so on. Precedence also applies to logical operators (such as **and** or \geq) and primitives that work on agentsets (see section 4.4).

NetLogo, along with other programming languages, also uses specific symbols for these mathematical operators. Your computer keyboard has plus ($+$) and minus ($-$) symbols, but there isn't a key for the 'divided by' sign (\div), it would be confusing to use the letter ' x ' for multiplication, and how would you indicate the superscript that is used for exponentiation when you have a pen and paper? The symbols used in *NetLogo* suggest the pen and paper symbols, but are not the

same. Exponentiation uses the ‘caret’ or ‘hat’ symbol (^), to point up. The asterisk (*) is used for multiplication, a sort of \times with extra arms; and division is indicated with the forward slash (/), a standard alternative for \div in equations.

Because *NetLogo* is very flexible with variable names, all mathematical operators must have a space immediately before and after. For example, 6 to the power of 2 would be written as **6 ^ 2**, not as **6^2**. The only exception to spacing is brackets, no space is required before or after either a normal bracket or a square bracket.



Precedence

Precedence is the order in which pieces of an equation are calculated. Brackets are used to change precedence, anything in brackets is done first. The mathematical operators follow the standard mathematical precedence order of exponentiation (^), then multiplication (*) and division (/), then addition (+) and subtraction (-).

Putting the symbols and spacing rules together, and remembering that **set** is used to assign a value to a variable, equation 2.1 would be written as follows in *NetLogo*:

```
set x 1 + 2 * 3 - (4 + 5) / 6 ^ 2
```

NetLogo also offers a range of other mathematical functions. These are all implemented with primitives rather than symbols. For example, the square root of 4 would be written as **sqrt 4** and the logarithm can be found with **log** (base 10) or **ln** (base *e*). Also available are statistical functions for manipulating multiple values (eg **sum**, **mean**), trigonometric functions for geometry (eg **sin**) and logical operators (eg **and**) and comparisons (eg **>=**). The full list is available in the *NetLogo* Dictionary (Help menu).



Spaces in code

NetLogo is very sensitive to the presence of space. Every keyword, operator, procedure name, variable name, mathematical operator or other code element must be separated with at least one space. The only exception is brackets; so any of (or) or [or] can be used without leaving a space. The number of spaces does not matter, any number of spaces is treated as a separator.

2.8 Turtle Agents

Having established a world for our agents to occupy and perceive, we need to create agents to make decisions. In *NetLogo*, these decision making agents are referred to as turtles.

The model developer is able to specify multiple types of turtles, referred to as breeds. This contrasts with patches, where there is only one type. Each breed of turtles has its own list of agent variables.

For the tutorial model, our agents are referred to as ‘people’ because it is people who are deciding whether to adopt or abandon protective behaviour. Those people need a variable to store their protective behaviour status. Snippet 1c-1 (page 32) defines a **breed** of turtles called people (or person when referring to a specific turtle), then declares the variables available to people with **people-own**. Note that the variable is to store whether the person has adopted protective behaviour, so its value will be **true** or **false**, and therefore has a question mark at the end of its name.

Patches are automatically created simply by starting *NetLogo*, they are part of the provided environment. However, turtles must be explicitly created. That is, the code must inform *NetLogo* of the number and type of turtles to create.

breed

Turtle agents can be of different types, referred to as a breed. To define a breed, the **breed** keyword is followed by the plural form of the breed name, then the singular form.

turtles-own

The **turtles-own** keyword is used to identify all the variables (listed between the square brackets) that are available to turtles to describe their attributes. Different sets of attributes can be defined for different breeds of turtles, with the name of the breed used in the keyword instead.

Multiple people are needed so that there is the opportunity for people on a patch to make different decisions and therefore contribute to different levels of protection. A larger number of people allows finer gradations in the proportion protected.

There are three different primitives for creating turtles, depending on the type of agent issuing the instruction (or the agent whose perspective is being taken). The observer can **create-turtles**, patches can **sprout** and turtles can **hatch**. To create/sprout/hatch turtles of a particular breed, then the (plural form) name of the breed is used instead of turtles. For example, the tutorial model will create turtles using **sprout-people** to instruct the patches to create turtle agents of the people type (in Snippet 1c-2 (page 33)) shortly).

Why does *NetLogo* need three different ways of doing the same thing, creating turtles? Because they represent different process mechanisms, allowing better matching between the real world and simulated behaviour. In particular, **hatch** is reproduction, which occurs while the simulation is running, it is part of the behaviour. In contrast, **create-turtles** and **sprout** are typically used to create turtles during initialisation, to make them available for the simulation. The key difference between these two is that **sprout** is convenient for spatially distributed turtles.

Regardless of the primitive used to create the turtles, it is followed with a number, indicating the number of turtles to be created, then an optional block of code in square brackets (like []) that each turtle is to run on creation. This block of code is used to set values for the attributes. In addition, turtles created with **sprout** inherit their location from the patch doing the sprouting, and turtles created by **hatch** inherit variable values from its parent. In both cases, those values can be overwritten using a **set** statement in the block of code.

Follow the instructions for Snippet 1c-2 (page 33). Think about what the code is doing before continuing. As already noted, this snippet uses the **sprout** primitive to create agents. It demonstrates several other things about *NetLogo* as well.

Snippet 1c-1: Define agent attributes

1. Add the following code below the **globals** definition and above the **patches-own** statement.
2. Check the code (green tick) and Save the model.

```
; people deciding about protective behaviour
breed [people person]
people-own
[ protect?                ; true / false for protected
]
```


create-turtles, sprout, hatch

The primitives `create-turtles`, `sprout` and `hatch` are all used to create turtle agents. They differ by the type of entity responsible for creation: observer, patch and turtle agent respectively. Variable values such as location are inherited by the new turtle from its parent patch or turtle when created with `sprout` or `hatch`. For all three primitives, a `breed` can be specified for the turtle(s) to be created. A code block following the creation contains any code that is to be run immediately on the turtle's creation, typically containing initial values for variables.

Snippet 1c-2: Create agents

1. Create the 'make-people' procedure by adding the following in the implementation procedures section of the code.
2. Add call to 'make-people' procedure in the 'setup' procedure. That is, add a line 'make-people' immediately following the line 'setup-patches' in the 'setup' procedure.
3. Check the code (green tick) and Save the model.

```
to make-people
  ask patches
  [ sprout-people (2 + floor ( popn / 10000 ))
    [ set protect? FALSE
    ]
  ]
end
```

The turtles being created are of the `breed` people. Comment out² the code created in Snippet 1c-1 (page 32), then check syntax (green tick) and *NetLogo* will highlight the `sprout` statement with an error. So the `breed` statement defines a type of turtle that is then available to be used. Remember to uncomment when finished.

The number of people to create is included in the code as a calculation rather than a specific number. Each patch has a value for the `popn` variable and that value is used so that patches with a larger population also have more people. The `floor` primitive³ is a mathematical way of saying 'take the integer part'. For example $19/10$ is 1.9 and rounding to the closest integer would result in 2, whereas `floor 19/10` is 1. In this case, the value of the `popn` variable will be a few million in the city patches, which will consequently have a few hundred turtle agents, and a few thousand in the other patches, which will have either 2 or 3 agents. *NetLogo* will report an error if the number of turtles to be created is not an integer. If you accidentally have the call to the `make-people` procedure before the call to the `setup-patches` procedure (in the `setup` procedure), then the `popn` variable values will all be 0 when the number of turtles is calculated and two turtles will be created at each patch.

As each person is created, the new person runs the code in the square brackets to `set` the `protect?` variable to `FALSE`. This variable is available because of the `people-own` statement in Snippet 1c-1 (page 32). It needs to be `set` to `FALSE` to indicate that the person has not adopted protective behaviour.

²It is easy to comment or uncomment a block of code by highlighting the code (click and drag the mouse over the block) and then use the menu item Edit > Comment/Uncomment.

³See also the related functions: ceiling, round, precision



Use a breed for turtles

Even if you only intend to have one breed of turtle agents, it is worth creating it with a **breed** statement rather than using the default. Firstly, this allows you to use a meaningful name for your agents, which makes the code more readable. More importantly, no recoding would be required if you decide to add an extra breed of turtles later.

After pressing the setup button, the model looks like the screenshot at Figure 2.9. Each coloured icon in the World is one person. Once turtle agents are being created successfully, model version 1c is complete. Save the model (Save item in the File menu).

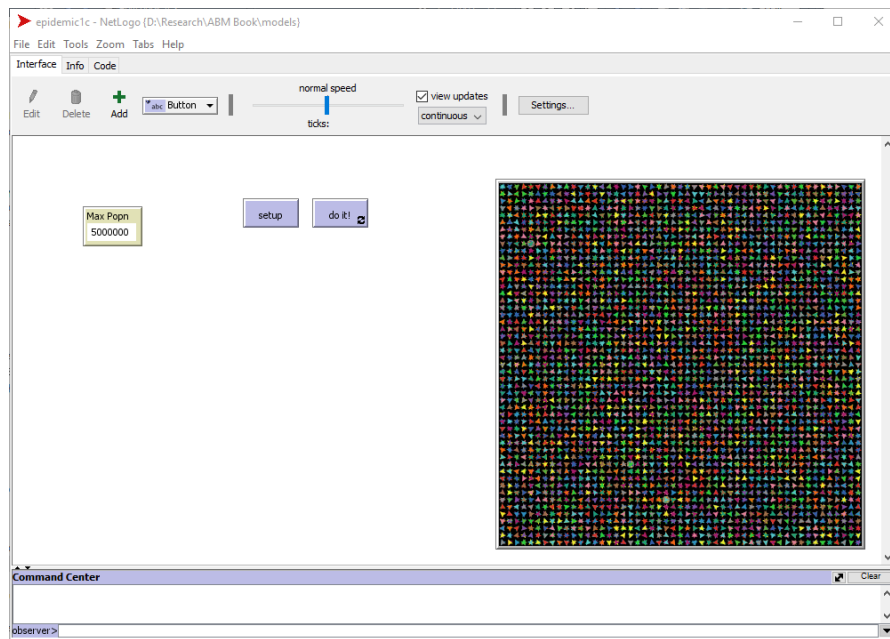


Figure 2.9: Screenshot of the interface after completing version 1c of the model.

2.9 Testing with Inspect Windows

What sort of sanity check could be used here? When you have learned more code, it would be easy to check the minimum and maximum number of people on any single patch. However, that option is not available yet. What about getting a closer look at the World instead?

Press the setup button so that the World is populated. Place the mouse pointer on one of the people icons and open the pop-up menu (with right-click in Windows and Ctrl-click in Mac OS). If you have the pointer on a turtle agent, the bottom of the menu will have some submenus, one for each person at the pointer. If the bottom line is 'inspect patch...', then move the pointer and try again. Open the submenu for one of the people and select the *inspect* option. This opens the inspect window (see Figure 2.10), which lets you look at individual agents in the model.

The person you chose to inspect is identified at the top of the window - person 4445 in my example. The top section has a small area of the World, 7x7 patches at the default zoom level. Immediately below this display is a slider to control the zoom. This World section is 'live', it is not just an image of the World. Any change in the World is also seen in the display and, for example, you can use the display to open further inspect windows.

The next section is a list of all the variables owned by the turtle (or person in this case). The first 13 variables are built-in variables that are automatically created for all turtles. They include icon controls (color, shape, size), the breed, and position (xcor, ycor). The variables created by the **turtles-own** keyword are then listed. The only variable so far created is named 'protect?'

and can be seen as set to `false`. At the bottom of the window is a box where *NetLogo* commands can be entered.

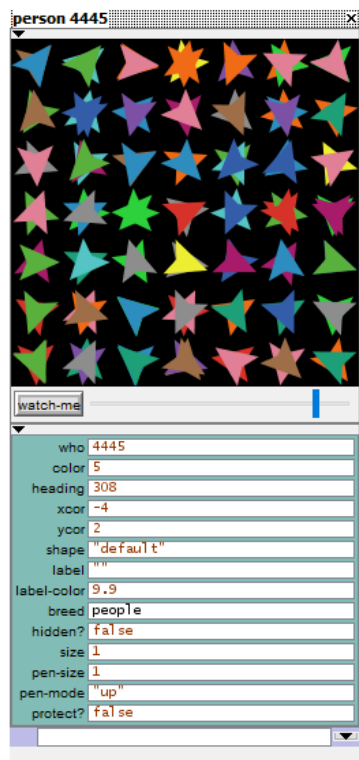


Figure 2.10: Inspect window for person.

Amongst other things, it deletes any agents that have been created and sets all global and patch variables to 0 (their default value).

Open an inspect window for an agent. In the code box at the bottom, type `set size 10` and then press Enter. The icon for the person you are inspecting will increase to size 10 in both the inspect window and the main World widget. This is an example of controlling *NetLogo* interactively, entering code for immediate implementation.

clear-all

The primitive `clear-all` is used to reset *NetLogo* to its default state, as if no code has been run. It is sometimes seen in code in its abbreviated form, `ca`.

2.10 Colours

Unless otherwise specified, patches are coloured black and turtle agents are randomly coloured. The next model iteration makes the interface informative by using colours effectively. First, however, we need to understand how *NetLogo* defines colours.

Open the tutorial model and press setup. Open an inspect window for a person and one for a patch. In the person window's code box, enter `set color red` (note the US spelling). In the patch's code box, enter `set pcolor 15`. What happened? What should happen is that both the person icon and the patch background change to red.

Colours in *NetLogo* have equivalent numerical values, and the name and number can be used

You can also open inspect windows for patches (and links, once there are some in the model). Open the context menu again, and this time select the 'inspect patch...' option. The inspect window for a patch has the same structure as for a turtle: the title bar identifies the patch, with a section of the World immediately below, variables and then a code box. For patches, there are only five built-in variables instead of the 13 for turtles.

The only `patches-own` variable defined so far is named `popn`. Look at the `popn` value for the patch you are inspecting. From previous discussion, you should expect the value to be near 10000 or a few million. For cities, there will be several hundred people on the patch and therefore in the centre of the World display. But for most patches, there will be two or three people displayed in the centre of the World. For the sanity check for turtle creation, open inspect windows for several patches and check that the number of turtles is consistent with the value of the `popn` variables.

We can also use the inspect window to understand the effect of the `clear-all` primitive that you included as the first line in the setup procedure. Comment out that line, then press the setup button a few times with an inspect patch window open. You should see additional people being created. That is, the setup procedure is not resetting the model back to an appropriate initial condition. Uncomment the `clear-all` line and try again. The `clear-all` resets *NetLogo* to its default or empty state, as if the model had just been loaded.

interchangeably. Valid colour numbers are from 0 to 140⁴. Each interval of length 10 refers to a broad colour such as red, which runs [10,20) or magenta, which runs [120,130). The lowest numbers in an interval are dark, and the higher numbers are pale. The mid number (that is, integer ending in 5) are the named colours.

The colour map is available from the *NetLogo* menu, at Tools > Color Swatches. Selecting that item will open the map displayed at Figure 2.11. Move the mouse pointer to the number 15 and click. The colour map will update, with the word ‘red’ in the bottom left text box, and a red turtle icon against various coloured backgrounds down the right column. Now try the number 123. In the text box, this colour is described as ‘magenta - 2’, confirming the equivalence of a colour and number. Note that the colour number does not have to be an integer. In the bottom right, change the radio button to 0.1 increment to see finer gradations.

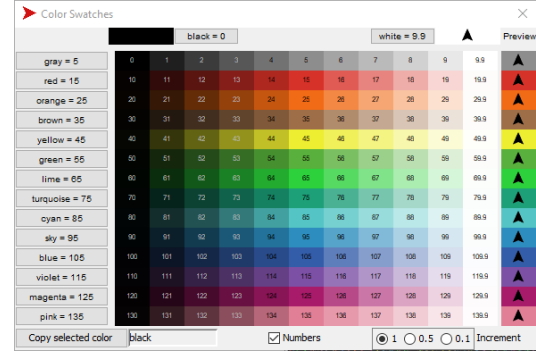


Figure 2.11: Colours available in *NetLogo* with their numerical equivalents.

The use of numbers for colours supports great flexibility in using colour to provide information about the model. We have already seen patches coloured to display whether food is available in the *Rabbits Grass Weeds* library model. The next step of the tutorial model is to use colour and other visual elements to communicate with the model user, highlighting important information.

2.11 Communication through Interface Design

The tutorial model will use colour to indicate population density. The display takes advantage of the numerical equivalence of colours with the `scale-color` primitive, which rescales a numerical variable and assigns a colour based on that variable value.

scale-color

The `scale-color` primitive is used to map a value to a colour range. The full syntax is `pcolor <colour-group> <number-to-map> <minimum-value> <maximum-value>`. The minimum and maximum values can be swapped, and the colours will invert accordingly.

Follow the instructions for Snippet 1d-1 (page 37). We first create a local (temporary) variable named `max-scale` and assign it a value higher than the maximum that will be required for scaling the colour. Remember that the `let` primitive is used to simultaneously create a variable and store a value in that variable (with `set` used to change it later if required). A logarithmic transformation is applied to the population values to reduce the extreme difference between the city patch populations and the other patch populations. The primitive `ln` is the mathematical function of natural logarithm. The multiplication by 1.3 ensures that even the highest values of population density do not generate colours that are too pale to be visible.

let

The primitive `let` is used to create a local variable and assign a value. The full command is `let <variable name> <value>`. The value can be specified explicitly (eg `10` or `"hello"`) or as the output of some piece of code. The variable created by `let` is temporary and will be deleted at the end of the block of code (delimited by `[]` square brackets) or procedure in which it is created.

⁴Other numbers are not treated as errors, they are instead mapped back to the relevant range using modulo arithmetic.

Snippet 1d-1: Colour patches

1. In setup-patches: add call to colour-patches procedure after the call to setup-patches in the setup procedure.
2. In the (new) utility procedures section at the end of the code, create the following procedure to colour patches.
3. Check the code (green tick) and Save the model.

```

;-----
; UTILITY PROCEDURES
;-----

to colour-patches
  let max-scale 1.3 * ln max-popn
  ask patches
  [ set pcolor scale-color blue ln popn max-scale 3.2
    ]
end

```

The variable `pcolor` is the built-in colour variable for patches. So `set pcolor ...` instructs the patch to change its colour to the stated value. In this case the stated value is derived from the `scale-color` primitive, with the logarithm of the value of the patch variable `popn` mapped to the `blue` colour range. In this example, the maximum possible `popn` value is 5 million, which would result in a value of approximately 20 for `max-scale`. So the population values are transformed logarithmically and then mapped to a scale between 3.2 and 20.05 and applied to the blue range of the *NetLogo* colour system (numbered 100 to 110).

pcolor

The built-in variable for patches to store their colour is named `pcolor`. It can be changed with `set pcolor <value>`. The colour value can be specified with either a name (eg `red + 1.3`) or the numerical equivalent (eg 16.3).

After completing this snippet, press the setup button and see what happens. The black background in the World is now blue, but the turtle icons are obscuring any variation in the shades, so the next step is to consider how to colour the turtle agents or people. The most relevant information about people is whether they have adopted protective behaviour. However, the people on a patch are all in the same place, only the last created person is visible as it obscures all the others. It is therefore more sensible to hide the turtle agents entirely.

There are two ways to hide turtle agents from the World. The first way is using the `hide-turtle` primitive. Follow the instructions for Snippet 1d-2v1 (page 38).

After completing the snippet, press the setup button. The colouring to indicate population is now visible because the people agents are not visible. They are still created though. You can see this by opening an inspect window for a person. However, the context menu is not available. Try it, and you will find that the menu does not include items for turtle agents. This is because `hide-turtle` hides the turtle from the World, not simply turning off the visible display. Instead, go to the command center (bottom of the interface, see section labelled (8) in Figure 1.1) and enter the code `inspect person 100` at the bottom space.

A more useful approach (for our purposes) is to make the turtle icons so small they are invisible but not actually 'hide' the turtles. Complete Snippet 1d-2v2 (page 38) to set the size of the turtles to 0. Make sure you delete the line you added in Snippet 1d-2v1 (page 38). Now press the setup button. The World should look the same but you should also have access to the context

Snippet 1d-2v1: Initial approach: Hide agents

1. Change the make-people procedure: add the line marked with **««** (within the code block run with sprout-people).
2. Check the code (green tick) and Save the model.

```

to make-people
  ...
  [ sprout-people ...
    [ ...
      hide-turtle ««
    ]
  ]
end

```

menu to inspect people.

Snippet 1d-2v2: Revised approach: Minimise agents

1. Change the make-people procedure: replace **hide-turtle** with the line marked with **««** (within the code block run with sprout-people).
2. Check the code (green tick) and Save the model.

```

to make-people
  ...
  [ sprout-people ...
    [ ...
      set size 0 ««
    ]
  ]
end

```

Now that you can see the variation in population (blue shading for patches), does anything strike you as odd? Remember that the population was assigned randomly for each patch. This means that cities with high population can be adjacent to other patches with very low population. But this is not realistic. In the real world, big cities are surrounded by relatively high population areas and, similarly, low population areas tend to congregate. The next step is to create a smoother transition in population values instead of sharp differences at patch boundaries.

The **diffuse** primitive moves part of a variable's value to the surrounding patches. Complete Snippet 1d-3 (page 39), which uses **diffuse** to smooth population. Each patch takes 40% of its own value of popn and shares it between the eight neighbouring patches. This means that the city centres create an initial ring of relatively high population values. This spreading operation occurs three time (due to the **repeat** primitive), so that there are three rings with gradually decreasing population values around each city. The city loses 40% of its population each time (leaving it with 0.6^3 or 21.6% of its initial value) but also gains some population from the sharing of its neighbours.

Informative interface design is not only useful for communicating model results to the user, but it also reduces the need for explicit sanity checking as the code is constructed. The World displays the population values with the blue shading, making it clear whether the objective of smoother population has been achieved. But some checking is still appropriate. Inspect a dark blue patch and also patches in one direction outward, is the population value decrease reasonable? Also, is

Snippet 1d-3: Diffuse population

1. Change the setup-patches procedure: add the line marked with **««** after setting the population values.
2. Check the code (green tick) and Save the model.

```
to setup-patches
  ask patches ...
  ask n-of ...
  repeat 3 [ diffuse popn 0.4 ] ««
  set max-popn ...
end
```

diffuse

The **diffuse** primitive removes the specified proportion of a patch's variable and shares that portion equally between each of the eight neighbouring patches, adding the share to whatever value the receiving patch already has for that variable. The full syntax is **diffuse** <variable-name> <proportion-to-share>.

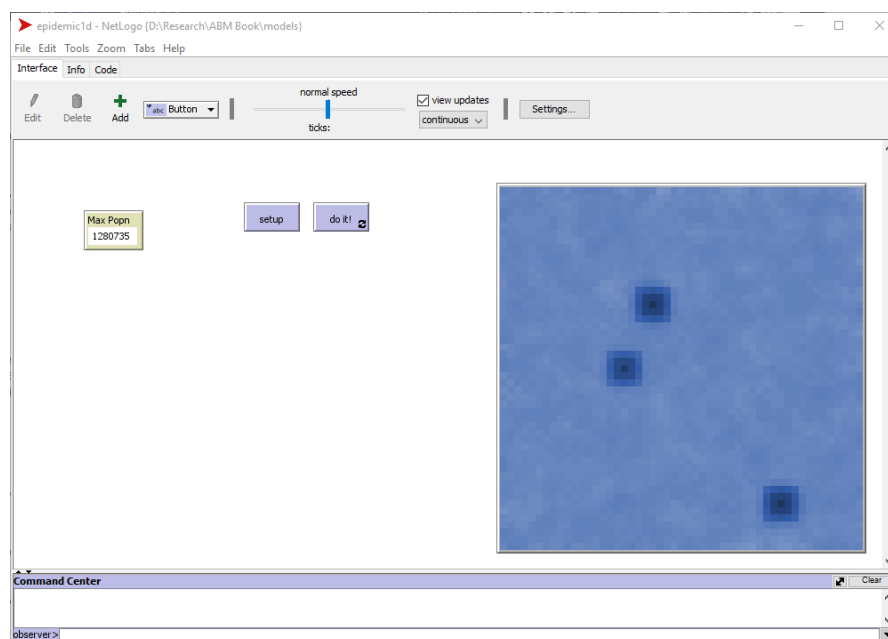


Figure 2.12: Screenshot of the interface after completing version 1d of the model.

the new maximum population in the monitor at the expected level?

repeat

The **repeat** primitive instructs *NetLogo* to perform the following code block multiple times. The full syntax is **repeat** <number-of-occurrences> [code-to-repeat].

2.12 Tutorial Progress Check

Version 1 of the model is now complete. The model creates turtle agents to represent people making decisions about their protective behaviour, and people are able to store their protective behaviour status. There is also a background population available to transmit the epidemic. However, no decisions are actually being made. In fact, this model is not yet a simulation as there is no passage of time and no processes are being modelled. Nevertheless, many important concepts have been presented.

2.12.1 Programming practices

At this stage, you should be much clearer about the iterative nature of developing code. You first think about what you want to achieve, then write code to achieve that objective and then test the code to make sure it works. Each addition should be as small as possible while still being complete, in the sense that the model must run with the code added.

NetLogo provides a syntax checker (green tick). This will find problems of construction where the entered code does not follow the syntax described in the *NetLogo* dictionary. For example, the `sprout` primitive must be followed by a number for how many turtle agents to create. If that number is missing from the code, the syntax checker will report an error. It will also identify errors such as missing `end` for a procedure, unpaired brackets, and mistyped variable or procedure names.

Each code snippet has included a reminder to apply the syntax checker and then save the model. Those reminders will no longer be included in snippet instructions, you should be checking and saving regularly as part of your programming habits.

This checker does not help with logic errors. Common errors include creating a procedure but forgetting to include the call to run the procedure, and forgetting to set sensible initial values for variables. However, most logic errors are more subtle, and can only be detected by explicit testing. So far, tests have involved checking that generated values are consistent with expected values, either displaying a variable value or by opening an inspect window.

Code should be modular, each procedure (bounded by `to` and `end`) is independent. Each procedure is comprised of primitives and calls to other procedures. A primitive is a defined word of the *NetLogo* language that instructs patches, turtles or links to do something or reports some information about these entities. Primitives are written by the *NetLogo* developers. In contrast, procedures are written by you, the model developer. Procedures that are invoked by higher level procedures may in turn call other procedures even further down the levels.

In *NetLogo*, procedures can be included in the *Code* tab in any order. By convention, *NetLogo* has two main procedures: the setup procedure contains the code to initialise the model, and the go procedure (which has not yet been used) contains the code to step through time and implement the processes. In this tutorial, the procedures are separated into groups in the following order: main control procedures (setup and go), implementation procedures and utility procedures. Implementation procedures are called by setup or go to perform some task that is specific to the model. Utility procedures are calculations and other general purpose functions that could conceivably be used in many different models.

Before the procedures, there are specific code sections that define the operating environment of the model by describing the types of agents and the variables that are to be available. The recommended order for these sections is:

- `globals` statement listing the global variables.
- `patches-own` statement listing the variables available to each patch.
- `breed` statement identifying the types of turtle agents available to the model, paired with the appropriate ...
- `turtles-own` statement listing the variables available to the specified type of turtle. Note that these use the breed name rather than the word 'turtles' in the primitive.

A key part of the developing the model is designing the interface. In *NetLogo*, the interface is automatically connected to the code. Interface widgets that are used to input variables (such as sliders) automatically define a global variable simply by their presence on the interface. Buttons are able to call procedures to run. And output widgets have a code box that contains the *NetLogo* code to identify the variable value or other information to report.

The World widget displays the model's world. It is comprised of a grid of patches, each of which represents a physical location and contains information about the environment. This world is populated by turtles, which represent decision makers or actors in the processes to be simulated. Turtles come in different breeds.

Information in the model is stored in variables. Global variables contain information that is accessible to all model entities. There is only a single copy of each global variable, with each variable defined by either an interface widget or by being named in the `globals` list. Agent variables contain information about a specific patch or turtle agent. There is one copy for each patch or turtle, and are defined by `patches-own` or `turtles-own` lists. Finally, local variables are temporary storage created as required for convenience and then destroyed at the end of the code block or procedure where they are created.

2.12.2 *NetLogo* language

Comments are used to describe the purpose of a procedure or a section of code to the human reader. They are created with the comma (;), with *NetLogo* ignoring the comma and the following text to the end of the line. Comments are also common to provide additional information about a variable when it is created, and multiple line comments are used as headings in the code to make navigation easier.

White space is important in *NetLogo* for both the computer and the human. For the code to be interpreted correctly, mathematical operators must have space on either side. This is additional to the usual requirement that variable names, primitives and other code elements are separated with a space. Brackets also separate code elements, and space is not required around brackets. *NetLogo* does not care about the amount of space, and multiple spaces are treated the same as a single space.

Space is also used to indent code, to make it readable for humans. *NetLogo* would interpret the code correctly if all the code was on a single line with a space between each element. However, it would be extremely difficult for a human to understand the code. Refer to Snippet 1c-2 (page 33) for a more readable presentation of the code. This demonstrates several good practices:

- Each instruction is on a new line.
- Each level of nesting (either the procedure delimiters `to` and `end`, or a block of code surrounded by square brackets) is indented, with further indenting at the next nesting level.
- The starting bracket is on a new line and lined up with the command that implements the block, and the closing bracket is on its own line and also lined up. This makes it easy to see that the brackets are paired correctly, and the limits of the code block.

Good indenting makes it easier to identify errors, update the code for new features, and adapt the code to different projects. As you have entered the code so far, you may have noticed that *NetLogo* does some indenting automatically. For example, if you end a line, the next line will have the same indent. Further, if you type a closing square bracket on a new line and then press 'Enter', the bracket will outdent, reversing the expected indent of the code block.



Make your code neat

Readable code is easier to understand and maintain.

New *NetLogo* keywords explained in this model version:

- code structures: `to`, `end`, `clear-all`
- defining and creating turtle agents: `breed`, `sprout`
- creating and manipulating variables: `globals`, `patches-own`, `turtles-own` `let`, `set`,
- built-in patch variables: `pxcor`, `pycor`, `pcolor`
- mathematics: operators (+ * - / ^), `random`, `ln`, `scale-color`
- miscellaneous: `diffuse`, `repeat`

Model 2: Introducing Time and Space

Model version 2 implements the epidemic process, at which point the model is a simulation. Key to implementing a simulation is modelling the passage of time and changes in model states through time. This version also introduces some relatively simple interaction, the actions occurring in one patch affect other patches. Such influence arises because of the spatial relationships between patches, and this version will also introduce the key spatial primitive of `neighbors`.

3.1 Model Design: The Epidemic Process

Before trying to model the epidemic process, the process must be specified in more detail. The detailed design is then implemented in *NetLogo* code.

The tutorial model epidemic process is the simplest version of the SIR model with no births or deaths (Kermack and McKendrick, 1927; Diekmann and Heesterbeek, 2000). Ignoring births and deaths essentially assumes that the epidemic process occurs in a much shorter time-frame than the time-frame for population change.

How does a person become infected? Consider a typical person's activities during a day (or some other arbitrary but fixed unit of time). They interact with some number of people sufficiently closely to permit transmission if one person is infected and the other susceptible. Of those contacts, only a proportion will lead to transmission because of other factors such as the viral load of the infected person and healthiness of the susceptible person. Let β denote the combination of the number of such contacts and the chance of infection given such a contact.

Once a person becomes infected, they have a fixed probability per unit time of recovering, with the probability denoted by γ . Once recovered, they are immune from further infection. Note that the full SIR model allows a proportion of infected people to die rather than recover, so the state is referred to as removed from the population. These state changes and parameters are summarised at Figure 3.1.

With these definitions, the change in the population in each epidemic state can be described by a set of equations. The change in the susceptible population (ΔS) is the number of people

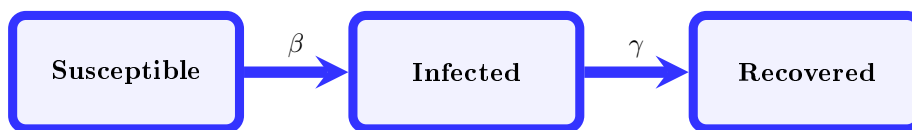


Figure 3.1: Epidemic state transitions for a population in the SIR model. There are no births or deaths, so the population is constant. For the given unit of time, the transition parameter β denotes the probability that a susceptible person becomes infected, and γ denotes the probability that an infected person recovers (and becomes immune).

changing states from I to S. From above, β is equivalent to the number of people able to be infected by a single infectious person (in the given time period) if everybody else in the population is susceptible. But only a proportion of their contacts are actually susceptible, that proportion is S/N . Further, the number of people instigating such infections is given by I. This leads to $\Delta S = -\beta I S/N$. Similarly, if γ is the probability of an infected individual recovering (in the given time period), then the infected population recovering is given by $\Delta R = \gamma I$. Finally, the change in the infected population is the gains from those becoming infected and the losses from those recovering. The relevant equations are therefore:

$$\begin{aligned}\Delta S &= -\beta I \frac{S}{N} \\ \Delta I &= \beta I \frac{S}{N} - \gamma I \\ \Delta R &= \gamma I\end{aligned}$$

Some mathematical manipulation (see (Diekmann and Heesterbeek, 2000) or another epidemiology text) provides a more intuitive pair of parameters that can be used to derive β and γ . The duration of infection is $1/\gamma$ and the basic reproduction ratio is β/γ . The basic reproduction ratio is the number of new infections that would be generated by an average infected person if every contact they had was with a susceptible person.

Note that time is built in to these equations, change (Δ) in the population counts requires a time period over which the change occurs. Further, this process is complex, new infections arise from the interaction between the susceptible and infected populations. However, the representation is not agent-centric. The population is simply counted by epidemic state, and all people are treated as identical apart from their state. There is no concept of local interaction as occurred with the *Virus on a Network* library model, and individuals are not represented in the equations.

Refer to Figure 1.9, both protective behaviour and the efficacy of that behaviour also influence the epidemic process. However, those elements cannot be introduced until the model includes protective behaviour. This is an example of starting with the simplest model possible and then refining it gradually.

As part of the detailed design, the modeller should identify the requirements to implement the process to be represented and the expected behaviour of the model. This planning helps to organise the order in which to introduce new code, and the tests to conduct to assess whether the code is correct.

Looking at the equations, each patch needs a variable for the populations in each state and global variables are required for the state transition parameters β and γ . From general epidemic theory, an SIR epidemic has a characteristic pattern of infections, with incidence (new infections) and prevalence (current infections) increasing and then decreasing in the classic diffusion curve. Eventually, the epidemic dies out because of the lack of accessible susceptibles. In addition, a basic check is that the total population in each patch does not change, just the allocation to different states.

3.2 Methods of Time Keeping

Just like any other aspect of the world that is to be included in the model, time must be represented in some way. Mathematically, time is continuous and many processes (including epidemic spread) occur in continuous time. This means that activity could occur at any arbitrary time.

One way to model continuous time is to track discrete events in a schedule. These events can be generated by some function or triggered by other events. For example, in an epidemic, a person becoming infected is a discrete event, which also triggers a later discrete event of that person recovering. In this example, when the person becomes infected, the duration of the infection is calculated and the recovery event is added to the schedule to occur at the appropriate time.

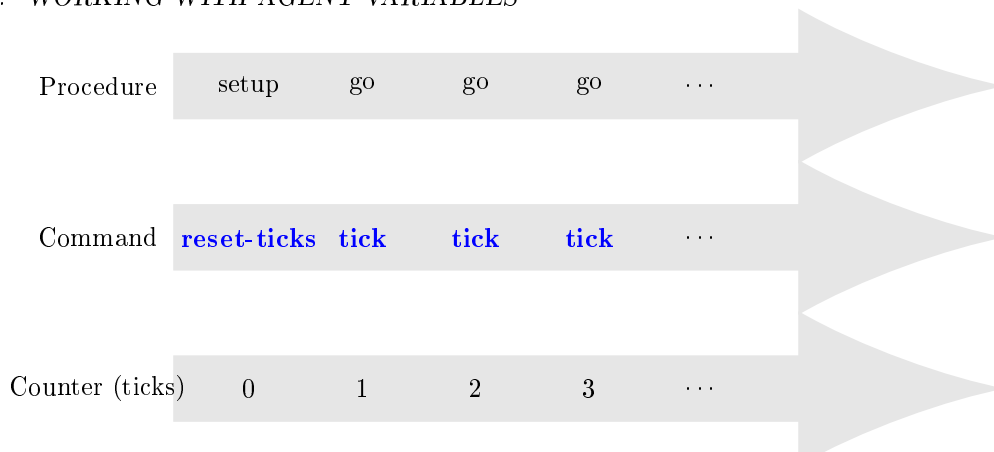


Figure 3.2: *NetLogo* represents time with a clock counter, accessed with the **ticks** reporter. The clock is initialised with the **reset-ticks** command, usually the last command in the *setup* procedure. It is incremented with the **tick** command, usually the last command in the *go* procedure.

Processes that occur in continuous time can also be modelled mathematically in continuous time by measuring activity or changes over a given interval of time.

In practice, however, simulations generally model time as a discrete variable. That is, the model includes an internal clock and each tick of the clock represents the same period of time such as an hour or a week. The advantage of this method is that empirical observations about processes are typically recorded for some regular period of time, such as quarterly economic activity.

3.2.1 Watching the Clock - Ticks

In *NetLogo*, the **tick** command advances the internal clock, and the **ticks** primitive (note the 's' at the end) reports the number of ticks that have occurred since the simulation (and hence the clock) started. In addition, the **reset-ticks** command is used to prepare the clock, setting the tick counter to 0.

This clock analogy relates to the discussion at section 2.3 about initialisation (or start) in the *setup* procedure, and simulation (or continue) in the *go* procedure. The *setup* procedure contains all the instructions that must occur at time 0. Conventionally, the **reset-ticks** command is the final instruction. The *go* procedure contains all the instructions that must occur in a time interval of whatever duration is represented by a tick. Conventionally, the **tick** command is the final instruction.

The **ticks** primitive is used when the time something occurs is important. For example, a seller might offer a discount to recent buyers and this rule would require a comparison between the **ticks** value of the most recent transaction with that buyer against the current **ticks** value. An example is included later in the model tutorial.

3.3 Working with Agent Variables

The first step in spreading an epidemic is establishing some infected people to initialise the epidemic. Remember that the epidemic in this simulation is carried in the environment. This means that the patches must include variables for the number of people (actually population density) in each epidemic state. Follow the instructions for Snippet 2a-1 (page 46) to create these variables and for Snippet 2a-2 (page 46) to set appropriate default values (that is, before the epidemic is seeded).

Remember that the snippet instructions will no longer include a reminder to check syntax (with the green tick) and save the model. However, you should check syntax and save after every snippet, and the model should pass the syntax check after each group of snippets with the same prefix (eg 2a-1 and 2a-2). The model may also pass the syntax check after adding one one snippet from a group, but you should at least try to understand any error messages.

Snippet 2a-1: Define patch attributes

1. Modify patches-own list to add new variables for epidemic state counts. New lines are marked with ««.

```

patches-own
[ popn                ; population of patch
  incidence            ; new infections this tick ««
  popn-S              ; susceptible population ««
  popn-I              ; infectious population ««
  popn-R              ; recovered population ««
]
```

Snippet 2a-2: Initialise patch attributes

1. Modify the setup-patches procedure to assign initial values for the patch variables. New lines are marked with ««.

```

to setup-patches
...
repeat ...
ask patches ««
[ set popn round popn ««
  set incidence 0 ««
  set popn-S popn ««
  set popn-I 0 ««
  set popn-R 0 ««
] ««
set max-popn ...
end
```

In these snippets, the letters ‘S’, ‘I’ and ‘R’ refer to the three possible epidemic states: susceptible, infected and removed (or recovered) respectively. The variable ‘incidence’ is the number of new infections. The population is initialised as entirely susceptible, so the values for I and R are set to 0 and the value for S is set to the patch population.

The tutorial has already used the `set` primitive to assign values to variables (such as the `popn` variables in Snippet 1b-3 (page 29)) with limited discussion. This primitive is used to assign a value to an existing variable, regardless of the variable type (global, agent or local) or the type of value (string, number, agentset, list, ...).

Here, `set` is being used to initialise the global variables. Note that some of these values are being set to 0 even though a variable is automatically assigned the value 0 when it is created. Systematically assigning values to all variables reduces the likelihood of missing a variable accidentally. It is also used to round¹ the population values to an integer after the `diffuse` command introduces non-integers.

¹See also the related functions: `ceiling`, `floor`, `precision`



Initialise variables explicitly

It is good practice to explicitly set variable values even if they are to be set to 0 (the default value) so that the value is chosen consciously and to reduce the chance of leaving some variables out of the assignment process accidentally.

set

The primitive `set` is used to assign a value to a named variable. The full command is `set <variable name> <new value>`. The new value can be specified explicitly (eg `10` or `"hello"`) or as the output of some piece of code such as a calculation.

The changes introduced in these snippets are not visible in the model. After pressing the setup button, the model interface will be identical to that for model 1d (Figure 2.12).

Now that there is a way to store information about the epidemic, the epidemic can be seeded. In the real world, epidemics require a critical mass of infected people to act as a source. To represent this in the model, a small number of high population patches must be selected as seeds, with a proportion of their population set to infected status. This is implemented in Snippet 2b (page 47).

Snippet 2b: Trigger epidemic

1. Create the start-epidemic procedure (in implementation procedures section of the code).
2. Add the call to the start-epidemic procedure into the setup procedure (before colour-patches).

```
to start-epidemic
  ask n-of 3 max-n-of 20 patches [ popn ]
  [ set popn-I round (0.005 * popn)
    set popn-S popn-S - popn-I
  ]
end
```

This snippet introduces two important primitives used for selecting a subset of model entities. `max-n-of` is used to identify the 20 patches with the highest values of the `popn` variable. From these 20 patches, 3 are randomly selected with `n-of`. The population state counts are adjusted in these three patches: the number of infected (`popn-I`) is set at 0.5% of the population (rounded to the nearest integer with `round`) and the susceptible population (`popn-S`) is reduced accordingly.

n-of

The full syntax is `n-of <number> <agentset>`. Selects a subset of specified size (number of entities) from the specified agentset, with each entity having an equal chance of selection. See also `one-of`.

3.3.1 Testing with the Command Center

What sort of sanity checks can be done to test this code? The code is intended to change the epidemic status counts in 3 of the 20 highest population patches, so one option is to open inspect windows at some of the dark blue patches until one has a non-zero value for `popn-I`. However, this is inefficient. Instead, we want to open a useful inspect window on the first attempt.

max-n-of

The full syntax is `n-of <number> <agentset> [<variable-name>]`. Selects a subset of specified size (number of entities) from the specified agentset, selecting those with the largest values of the specified attribute variable. Ordering is random for those entities with the same value. This primitive has been presented as using a variable to order the entities, which is a common use. However, the primitive is more general, and any calculation that can be made for each entity can be used for ordering. The equivalent for selecting the entities with the smallest values is `min-n-of`. See also `max-one-of` and `min-one-of`.

The Command Center (labelled (8) in Figure 1.1) has an area at the bottom to enter *NetLogo* code to be run immediately. You can use this to report the number of patches with a non-zero value for `popn-I` and open an inspect window for one of these patches.

Press the setup button to initialise the epidemic. Enter the code `count patches with [popn-I > 0]` into the input box. The output area repeats back the entered code (with `show` at the front) and reports the results of the `count` (see figure 3.3; as expected, there are three patches with `popn-I` values greater than 0).

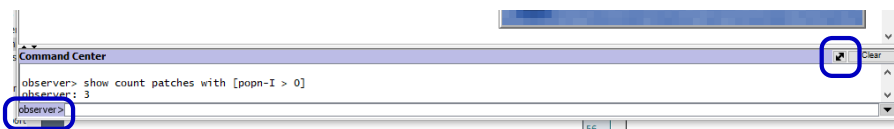


Figure 3.3: Command Center with results from code. The blue lines mark the perspective dropdown box and the Command Center expansion toggle.

count

The `count` primitive reports the size (number of entities) in the specified agentset. The agentset can be specified explicitly by name (such as `patches`) or as the result from a piece of code.

There are other aspects of the Command Center to notice. Both the input area and the output have the word ‘observer’ at the start. This is to inform the user which perspective is being taken. Change the perspective to ‘patches’ using the dropdown box at the left of input area. Enter the code `set pcolor red` and you should see that all patches change their colour to red. Change back to the ‘observer’ perspective and enter `colour-patches`. This will run the procedure named `colour-patches` in exactly the same way as it is called from the ‘setup’ procedure when the setup button is pressed. As only the entered procedure is run, the maximum population and the location of the cities is unchanged.

At the top right of the Command Center there is a double ended arrow. Click on that and the text area will expand to take up the right half of the model interface. Click again and it will return to the small area at the bottom. The bar between the main interface and the Command Center can be dragged to change the size.

With the perspective set to ‘observer’, enter the code `inspect max-one-of patches [popn-I]`. What does this code do? First, it searches through the patches and finds the one with the largest value of `popn-I` (infected population) and then opens an inspect window for that patch. You should check that the values of the population in the different epidemic states look reasonable. In particular, check that `popn-I` approximately 0.5% of `popn`, and that `popn` is the sum of `popn-I` and `popn-S`.

Figure 3.4: `ifelse-value` can be used instead of an `ifelse` construction where the two paths are setting the same variable to different values. The left code box is the `ifelse-value` as included in Snippet 2c (page 49), and the right code box shows the `ifelse` with separate `set` commands. Note also the use of numbers for colours: 16 is a darker shade of red than 18.

3.4 Conditional coding: Updating the colour scheme

So far the colour scheme displays population, but this can be extended to display the epidemic. Blue will continue to be used if the patch is mostly susceptible, with green if mostly removed (so the epidemic has passed). In addition, red will be used to indicate where the epidemic is active. This is implemented in Snippet 2c (page 49).

Snippet 2c: Visualise epidemic

1. Amend colour-patches procedure to colour by epidemic state. New lines are marked with `««`.

```
to colour-patches
  let max-scale 1.3 * ln max-popn
  ask patches
  [ ifelse popn-S > popn-R ««
    [ set pcolor scale-color blue ln popn max-scale 3.2 ]
    [ set pcolor scale-color green ln popn max-scale 1.5 ] ««
    if popn-I >= 1 ««
      [ set pcolor ifelse-value (popn-I > 0.03 * popn) [16] [18] ]
    ««
  ]
end
```

This snippet introduces all three versions of the conditional branching IF / THEN / ELSE. This is a fundamental construct of many programming languages and consists of a test, with one set of commands run if the test returns true and a different set run if the test returns false. It is essential for agent-centric behaviour because it translates the different characteristics and circumstances of each agent into potentially different actions.

The first use is the `ifelse` format. The test compares the susceptible and removed population counts. If the test is true then the first code block is run (using a blue colour scale when susceptible is larger) and if it is not true then the second code block is run (green colour scale). The second use is the `if` format to test whether there is any infected population. The difference between `if` and `ifelse` is that nothing happens if the test is false with `if`, the code block is simply skipped. Finally, `ifelse-value` is used if the two actions to be taken are simply to set the same variable to two different values. Figure 3.4 displays the full `ifelse` alternative for `ifelse-value`.

After completing this snippet, press the setup button. Three patches should be coloured light red (see Figure 3.5). Further, those three patches should all be in the high population areas of the model. It is now straightforward to open an inspect window for one of these patches to confirm that the population values in each state are sensible. This example also demonstrates that sanity checking is much easier with an informative interface design.

ifelse, if, ifelse-value

The most general of these primitives is `ifelse`, for which the full syntax is `ifelse <condition> [<instructions-if-true>] [<instructions-if-false>]`. This allows the program to branch, running one set of instructions if the condition is `true` and a different set if the condition is `false`. If no code is to run when the condition is `false`, the primitive `if` can be used. If the two branches set the same variable to different values, then `ifelse-value` is the appropriate primitive.

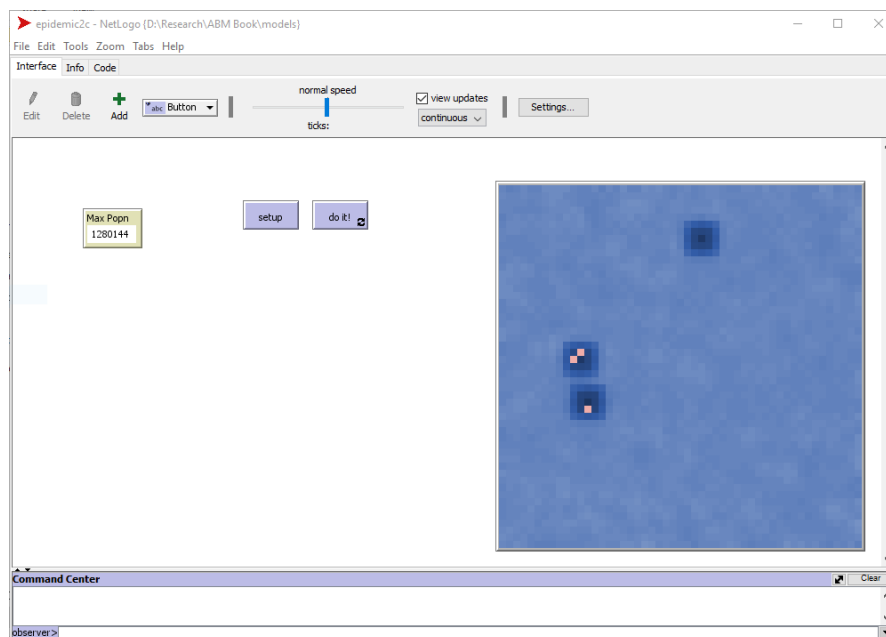


Figure 3.5: Screenshot of the interface after completing version 2c of the model.

3.5 User Input Widgets

Examining the epidemic process equations in section 3.1, there are two parameters that control the transitions between epidemics states: β and γ . Clearly, these could be added to the list of global parameters and assigned values with the `set` command. However, this approach would require the model user to edit the code to obtain different values. A better alternative is to assign them with interface controls.

There are four types of widgets that allow values to be entered by the user into the model. A *slider* allows only numbers to be entered, and the specific number is selected by moving a slider. The developer sets the range of permitted values (maximum and minimum), which can be used to restrict entry to sensible values. A *switch* is used to enter a value that is either true (switch on) or false (switch off). A *chooser* (or dropdown box) is used to select between multiple pre-defined choices, which could be numbers, text or some combination. Finally, an input box allows any value to be directly typed by the user, and the only restriction is the form of the value (such as number or string).

Regardless of the type of input widget used, the top of dialogue box has a space titled ‘Global variable’. The text entered in that box must satisfy the variable naming conventions (see section 2.5). This is because the widget creates a global variable with the given name. The variable is available to the code in exactly the same way as a variable listed in a `globals` statement. The value is assigned by the input rather than a `set` statement. Note that a `set` statement in the code will change the variable’s value, both internally and on the interface. That is, the interface widget is ‘live’ throughout the simulation and can even be changed during a simulation run.



User Inputs

An important part of the interface design is identifying the key parameters that users will want to change. Those parameters should be included in the interface with sliders or other input widgets, and less important parameters included within the model code as global variables with sensible default values applied with a `set` command.

3.6 The ask Command

The `ask` command is the essential primitive in making *NetLogo* models. It is the primitive that implements the agent-centric approach by connecting the instructions to the model entities (turtles, patches or links) that are to perform the instructions.

The model entities being asked are referred to as an agentset (discussed in more detail at section 4.4). The agentset can be all of a particular type of entity, such as `ask patches [...]` in the `setup-patches` procedure of Snippet 1b-3 (page 29) or some identified subgroup, such as `ask n-of 3 patches [...]` in the same procedure. Agentsets can also consist of a single entity.

ask

The full syntax is `ask <agentset> [<instructions>]`. The `ask` command instructs the specified agentset to implement the code in the square brackets.

The `ask` command steps through each entity in the agentset and runs all the code for that entity before moving on to the next entity. Each `ask` moves through the entities in a random order. In a simulation, much of the code is run each time step (or tick), so any `ask` is also run multiple times. The random ordering avoids artefacts arising with same agent always accrues any benefits of going first.

3.7 Making the Model a Simulation

The next few snippets concern the process of spreading the epidemic. This is the point at which the passage of time is introduced into the model. There is a conceptual leap involved, so it is important to think about how time is integrated with the code.

The first decision is the amount of time that a tick is to represent. For abstract models, the time step is arbitrary. However, if the simulation is representing a real world system, then it will be calibrated to some sort of data that incorporates time.

The time step should be short enough that it is quicker than the fastest process being simulated. That is, agents should not be taking multiple actions within a single time step unless all the actions are part of one larger action (such as moving and eating in the *Rabbits Grass Weeds* model). But making the time step too quick is inefficient. A good rule of thumb is the time step should be set so that a small but non-trivial proportion of the agents should change state (that is, take some action) each time step.

For the tutorial model, a convenient time step is one day. This is because the time step should be shorter than the typical infectious, which is about five days for an influenza-like epidemic is about five days.



How long is a tick?

A `tick` should represent a time step during which a meaningful proportion of the agents change their behaviour or state.

3.7.1 Input epidemic parameters

For the tutorial model, both input parameters are numbers, so sliders are the most appropriate widget. As β and γ can be derived from the duration of infection and basic reproduction ratio, sliders will be constructed for these more intuitive measures and the epidemic parameters calculated within the model.

A slider is added to the interface in the same way as other widgets, like the monitor added at 27. Select slider in the widget dropdown box and click on the ‘Add’ icon then the interface where the slider should be located. Alternatively, open the context menu on the interface and select slider. The slider dialogue box will open.

Snippet 2d-1 (page 52) describes the two sliders to be added to the interface. The first of these is for basic reproduction ratio, which is denoted by R_0 in epidemiology. Figure 3.6 displays the relevant slider dialogue box filled in.

Snippet 2d-1: Epidemic parameter sliders

1. Create a slider with variable name ‘R0’, minimum 0.25, increment 0.25, maximum 5, and value 2.
2. Create a slider with variable name ‘recovery-period’, minimum 1, increment 1, maximum 10, value 5, and units ‘days’.

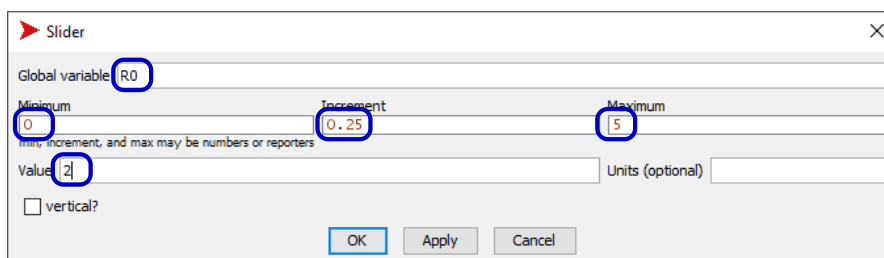


Figure 3.6: Screenshot of the slider dialogue box for the variable R_0 , which is the basic reproduction ratio. The top text box is used to enter the name of the global variable that is created by the slider.

3.7.2 Stepping through time

The next step is to implement the design equations at section 3.1, the key procedure for the epidemic process. With D as the variable recovery-period, then $\gamma = 1/D$ and $\beta = R_0/D$. In addition, new infections (incidence) is given by: $\beta IS/N$. For convenience, the equations are:

$$\begin{aligned}\Delta S &= -\beta I \frac{S}{N} \\ \Delta I &= \beta I \frac{S}{N} - \gamma I \\ \Delta R &= \gamma I\end{aligned}$$

Before implementing Snippet 2d-2 (page 53), it is worth thinking about how you would translate these equations into *NetLogo* code. What order would you update these equations and why is that order important? Are there any checks you need to make during the calculations?

The individual primitives in this procedure are relatively simple and have all been introduced already, comprising only **ask**, **if**, **let** and **set**. Operating with information stored in variables, careful ordering of these simple commands is able to create complex behaviour.

The logic in this procedure is:

Snippet 2d-2: Mechanism for epidemic spread

1. Create the spread-epidemic procedure (a suitable location is as the first procedure in the implementation section).

```

to spread-epidemic
  ask patches
  [ ; calculate the number of new infections generated by the patch
    let beta (R0 / recovery-period)
    set incidence popn-I * beta * popn-S / popn
    ; update the population counts in each epidemic state
    let recovered popn-I / recovery-period
    if incidence > popn-S [ set incidence popn-S ]
    set popn-S popn-S - incidence
    set popn-I popn-I + recovered + incidence
    if popn-I < 1 [ set popn-I 0 ]
    set popn-R round (popn-R + recovered)
  ]
end

```

1. Calculate expected incidence first, before the S and I population counts are adjusted.
2. Make sure that incidence does not exceed the number of susceptible available. This can occur because, conceptually, each infected person finds their own susceptible targets simultaneously so susceptibles can be infected multiple times. Reduce incidence if required.
3. Calculate the population to recover before the newly infected are added to the infected population.
4. Mathematically, the infected population would never reach 0 after it is non-zero because all the relevant equations simply multiply numbers together. Instead, both incidence and popn-I would approach 0, though rounding to nearest integer may resolve this. To guarantee resolution, the infected population is set to 0 when it falls below 1.

Note that rounding (to the nearest integer) is included in this code as it is not possible to have fractional people infected in the real world. However, this can introduce subtle errors when dealing with small numbers. This is the reason that popn-R is rounded rather than the local variable 'recovered'; the number of people recovering as the epidemic dies out will be less than 1, and rounding to 0 would mean that the last infected person never recovers.

Snippet 2d-2 (page 53) implements the spread of the epidemic in a single time unit (day). The next step is to set up the passage of time, instructing *NetLogo* to repeatedly run the procedure to spread the epidemic spread. This is implemented in Snippet 2d-3 (page 54). Once coded (and syntax checked), press the *setup* button and then the *do it!* button. After the simulation has run for awhile, press the *do it!* button again to stop it. The interfact should look similar to the screenshot at Figure 3.7.

NetLogo links together the *do it!* button and the *go* procedure. The button calls the procedure when pressed. Checking the 'Forever' box means that the *go* procedure continues to get called until the button is pressed again. That is, a single press of the button requests multiple iterations through the *go* procedure (as depicted in Figure 3.2).

Each iteration through the *go* procedure first runs the spread-epidemic procedure, which updates the epidemic status counts for each patch. It then calls the colour-patches procedure, which we previously created so that the patches were appropriately coloured during initialisation (that is, the *setup* procedure). Finally, it instructs *NetLogo* to increment the tick counter with the *tick* command.

Snippet 2d-3: Step through time

1. Add `reset-ticks` at the end of the setup procedure.
2. Confirm that the 'Forever' checkbox is set for the *do it!* button.
3. Amend the go procedure.

```

to go
  spread-epidemic
  colour-patches
  tick
end

```

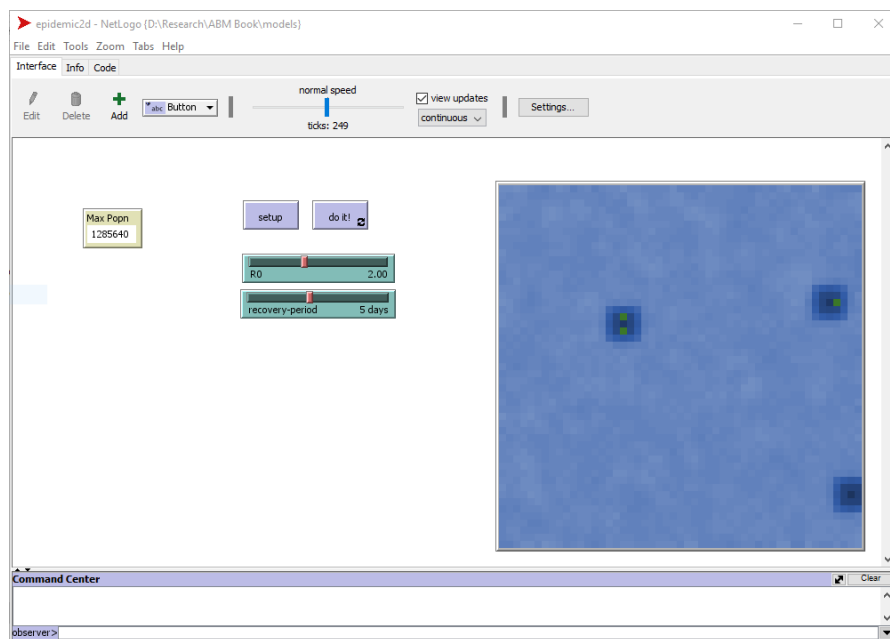


Figure 3.7: Screenshot of the interface after completing version 2d of the model. Notice the tick counter (below the speed slider) reading 249 in this example and the green colour of the patches where the epidemic occurred.

If `reset-ticks` command has not already occurred, then `tick` will generate an error. So the setup procedure is amended to include `reset-ticks`.

tick

The `tick` command increments the tick counter. The number of ticks is used to represent the passage of time and can be accessed with the `ticks` reporter. See also `reset-ticks`, which is used to initialise the tick counter.

**Code order**

It is standard practice to locate `clear-all` as the first command in the setup procedure, `reset-ticks` as the last command in the setup procedure, and `tick` as the last command in the go procedure. As `reset-ticks` also initiates any plots, this order ensures that all the model entities are initialised before their values are plotted.

To understand what is happening and as a sanity check, you can edit the *do it!* button to uncheck 'Forever', then press the setup button once and then the go button. Open an inspect window for one of the red patches. Each press of the *do it!* button will run through the go procedure once, updating the epidemic state population counts and incrementing the tick counter. Watch the incidence and population values in the inspect window as you press the button. See if you can use the equations to calculate the expected values for the epidemic state populations for the next go step then press the go button and see if you are correct. Try it again with different parameter settings; for example, you would expect a more intensive epidemic with a higher value of R_0 and a shorter recovery-period. When you have finished, check 'Forever' again for the *do it!* button.

What did you notice while running the simulation? First, you had to stop the simulation by pressing the go button, even though the epidemic had run its course. Second, the epidemic didn't spread out, it was trapped in the patches where it started. Fixing these two issues, and plotting some simulation results, are the remainder of the tutorial model version 2. However, some additional theory is required first.

3.8 Commands, Reporters and Procedures

There are several types of keywords in the *NetLogo* language. At the first level, *NetLogo* distinguishes between keywords and primitives. Keywords establish the *NetLogo* program rather than run it, and include **globals**, **patches-own**, **to** and **end**. In contrast, procedures are constructed with primitives.

Those primitives that are used to obtain information are referred to as reporters. Important reporters include **n-of** and **patches**, as well as several that will be introduced in the next version of the model. Other primitives that can be used within procedures are referred to as commands. Commands are primarily used to instruct one or more turtles or patches (or links) to do something. Important commands include **ask**, **set** and **ifelse**. Looking at this paragraph, you will see that all the reporters are purple and commands are blue, this is part of the automatic formatting in the *NetLogo* code area. And keywords are formatted in green.

In the same way that the primitives are split between reporters and commands, two types of procedures are available. A reporter procedure runs the contents and then provides some piece of information back to the procedure that called it. Command procedures implement actions. All the procedures included in the tutorial model so far are command procedures, and have been delimited with **to** and **end**. Reporter procedures are delimited with **to-report** and **end** and must include a **report** command identifying the value to be returned to the calling procedure.

to-report <procedure-name> <instructions> **report end**

A reporter procedure is a block of instructions that concludes with calculating some piece of information and returning that value to the procedure or model entity that called the reporter procedure. The keywords **to-report** and **end** mark the beginning and end of the block. In addition, a **report** command must be included within the block to identify the information to provide to the caller.



Command or reporter?

It is not always obvious when to construct a procedure as a command procedure (using **to**) or as a reporter procedure (using **to-report**). A good rule of thumb is that, if the procedure changes anything in the model, it should be a command procedure. Reporter procedures should be completely passive, accessing information but not affecting the model in any way.

```

to-report all-done?
  let test-result 0
  let total-infected sum [popn-I] of patches
  ifelse total-infected > 0
    [ set test-result FALSE ]
    [ set test-result TRUE ]
  report test-result
end

```

Figure 3.8: An expanded equivalent to the code in Snippet 2e-1 (page 56) to assist with understanding.

3.8.1 Ending the epidemic

There is no point in continuing the simulation after all the infected people have recovered. Snippet 2e-1 (page 56) and Snippet 2e-2 (page 58) combine to test whether the epidemic has ended and, if so, to override the ‘Forever’ checkbox to end the iterations through the go procedure.

The first step is to check whether the epidemic is over. Look at the code at Figure 3.8 and step through the logic:

1. Create a reporter procedure named ‘all-done?’ with a **to-report** declaration. Note that, like the people attribute ‘protected?’, the procedure name ends with a question mark. This is because the information that will be reported is the value **TRUE** or **FALSE**.
2. A local variable called ‘test-result’ is created to store the value that is to be reported at the end of the procedure.
3. A local variable called ‘total-infected’ is created. The total of all the patches’ infected population values (popn-I) is calculated and assigned to this variable.
4. If the total is more than 0 (tested with **ifelse**, the the epidemic is still active and the value **FALSE** is assigned to the variable ‘test-result’. Otherwise, the value **TRUE** is assigned.
5. The value of test-result (**TRUE** if the epidemic is over, and **FALSE** if it is ongoing) is provided to the procedure or model entity that called the ‘all-done?’ procedure.

Add Snippet 2e-1 (page 56) to the model. This implements the same logic as the code in Figure 3.8, but in a single line. The primitive **ifelse-value** can be used instead of **ifelse** because the action for either result is to **set** the same variable with different values. More subtly, the two values calculated in this procedure are not explicitly stored in named variables, but are simply used as soon as they are calculated.

Snippet 2e-1: Test if epidemic completed

1. Create the all-done? procedure in the utility section.

```

to-report all-done?
  report ifelse-value (sum [popn-I] of patches > 0) [FALSE] [TRUE]
end

```

In Snippet 1b-3 (page 29), the maximum of the population values over patches was calculated and assigned to the global variable max-popn. This procedure takes a similar approach, calculating the **sum** (total) instead of the **max** (maximum). Both **sum** and **max** are mathematical primitives that operate over a list. A list is a particular type of data structure, where multiple data items (numeric values, text strings, turtles, other sub-lists) are stored in a single variable in a fixed order. While lists are useful, they are outside the scope of this tutorial. What is important here is that the primitive **of** returns a list. In this snippet, the list contains the values of the popn-I

variable for each patch (in whatever random order the `ask` runs through the patches). That list is used as soon as it is created by the `sum` operator, without being explicitly stored.

of

Model entities automatically know the values of their own variables, including global variables and turtles have access to the variables of the patch where they are located. The reporter `of` is used to access variables owned by other entities; it identifies the owner of the variable to be accessed. The full syntax is [`<reporter>`] `of` `<agentset>`.

3.8.2 Testing truth values

Boolean variables are those that only take the values `TRUE` or `FALSE`. Unlike some other languages, *NetLogo* does not actually know which variables are boolean. The convention of using a question mark at the end of the variable name is for the humans reading the code. The values of `TRUE` and `FALSE`, however, are special and are recognised by *NetLogo*. Note that `TRUE` is not the string `"TRUE"`, it is a specific truth value. Furthermore, capitalisation does not matter, any mix of upper and lower case that spells the word ‘true’ or ‘false’ is interpreted as `TRUE` or `FALSE` respectively.



Truth values

`TRUE` and `FALSE` are special values recognised by *NetLogo*. They are created by logical tests or conditions and can be manipulated with logical operators.

If you want to test whether a variable has a specific value, then the test would have similar syntax as `if varname = "xyz" []`. Further, to test whether it any value other than the specified one, the code would be `if varname != "xyz" []` (the combination ‘!=’ is the operator for ‘not equal to’). Equivalently, for a variable being used to store boolean values, the syntax would be `if varname? = TRUE []`.

However, if the test is of a truth value (that is, either `TRUE` or `FALSE`), it is sufficient to drop the ‘=’ part of the test. The code `if varname?` will resolve to `TRUE` or `FALSE` as appropriate, and return an error if the variable ‘varname?’ has any other value (such as 0 or the string `"TRUE"`). But what if you want to test for `FALSE`? Just as there are mathematical operators to manipulate numbers (such as +), there are logical operators to manipulate truth values. In particular, `not` converts `TRUE` to `FALSE` and `FALSE` to `TRUE`. So, `if not varname?` would resolve to `TRUE` only when the value of ‘varname?’ is `FALSE`. Another common logical operator is `and`.

Instead of explicit variables, Snippet 2e-2 (page 58) tests the truth value of the procedure created in Snippet 2e-1 (page 56) to break out of the ‘go’ loop once the epidemic has run its course. Each loop through the go procedure, the all-done? procedure is called and, if that procedure returns `TRUE`, then the condition is met and the `stop` command breaks *NetLogo* out of the forever instruction issued by the button for the go procedure.

This snippet completes model 2e. No screenshot is presented as it is identical to Figure 3.7. However, the simulation should stop running automatically. Press the *setup* button and then the *do it!* button to check the termination.

3.9 Spatial Awareness

The spatial infrastructure of a model is created automatically by the *NetLogo* system. This comprises the grid of patches that make up the World, and a co-ordinate system that allows each turtle to recognise its location. The *NetLogo* language has many primitives that take advantage of this spatial awareness to interpret concepts such as distances, directions and neighbours.

Snippet 2e-2: End simulation

1. Amend the go procedure to call the all- procedure as its first line. The new line is marked with `««`.

```

to go
  if all-done? [ stop ] ««
  spread-epidemic
  colour-patches
  tick
end

```

stop

`stop` forces the model entity (including the observer) to terminate the code block that it is running. If implemented within a code block delimited with square brackets, the effect is the same as jumping to the closing bracket (and then continuing). It can also be used to terminate a procedure called by a button with forever checked.

3.9.1 Co-ordinate system

Patches are identified by a pair of numbers such as `patch -5 3`. The first number is the `pxcor` (or patch x co-ordinate where x refers to horizontal or left / right) and the second number is the `pycor` (up / down). The numbers refer to the number of patches away from `patch 0 0`, which is in the centre of the world by default.² So this example would be the patch 5 patches to the left and 3 above the centre patch. The variables `pxcor` and `pycor` are automatically created with the World.

Similarly, turtle agents locations are specified with a pair of variables named `xcor` and `ycor`. Turtle positions are continuous, for example `xcor` of -5.2345 and `ycor` of 2.6789. The co-ordinate system for turtles is the same as the co-ordinate system for patches (despite the different names).

The integration of the co-ordinate systems means that a patch is exactly 1×1 spatial units in size. The patch that a turtle occupies is the one identified by rounding the co-ordinate values for the turtle, with 0.5 always rounded up. For example, the turtle with `xcor` of -5.2345 and `ycor` of 2.6789 would be on patch -5 3. Similarly, patch 0 0 covers $-0.5 \leq \text{xcor} < 0.5$ and $-0.5 \leq \text{ycor} < 0.5$.

This built in co-ordinate system allows spatial concepts such as distance and directions to be meaningful. Turtle agents also have a heading automatically, even if they are not moving; it is a number from 0 to 360 (degrees) with 0 as 'up'. The spatial structure also supports concepts such as 'here' (referring to the same patch) and, for turtles, 'forward' and 'the patch in front of me'. For the tutorial model, the turtle agents will be stationary. Movement is discussed in Chapter 8.

3.9.2 Neighbourhoods in NetLogo

NetLogo has two primitives to identify neighbouring patches, depending on whether you want all eight with a common corner (`neighbors`) or only the four with a common edge (`neighbors4`). Turtles are also able to use these primitives to refer to the patches bordering the patch where the turtle is located. These are demonstrated at Figure 3.9, with the code on the left generating the model world on the right. You can test this yourself by starting a new model, entering the procedure into the code tab, and then running the procedure by typing the procedure name into the Command Center.

²The position of 0,0 can be moved with the Settings dialogue box

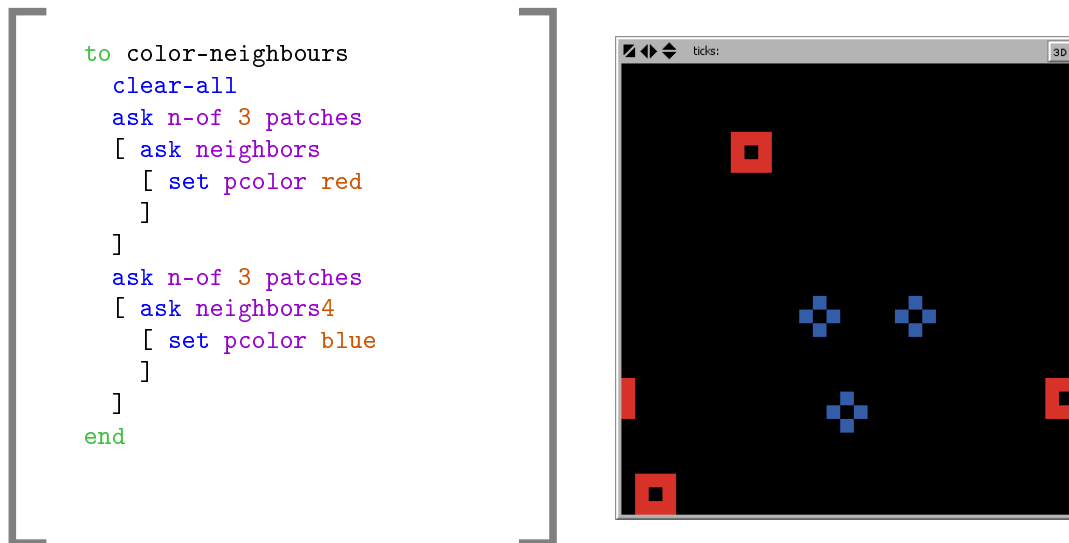


Figure 3.9: Demonstration of the **neighbors** (8 patches) and **neighbors4** (4 patches) reporters. Each of three randomly selected patches instruct their **neighbors** to turn red, and another three instruct their **neighbors4** to turn blue.

neighbors

The eight patches adjacent to a patch are referred to as the patch's **neighbors**. A turtle can also access the **neighbors** of the patch where it is located. See also **neighbors4**.

3.9.3 Spatial epidemic

In order for the epidemic to spread spatially, some proportion of new infections must be created on neighbouring patches. The first step is to create a variable to store that proportion. That could be created on the interface with a slider or other input widget. However, to reduce clutter, the interface should be reserved for important settings that a user may wish to adjust to test out different scenarios.

Instead, the variable will be created as a global variable within the code. Creating it this way means that it will have a value of 0, so the value must also be explicitly set (or initialised) to something more appropriate. Both these tasks are implemented in Snippet 2f-1 (page 59).

Snippet 2f-1: Establish travel

1. Create new global variable 'travel-rate' (by adding it to the **globals** list at the top of the code).
2. Create the initialise-globals procedure. While specific location is not important, it must be after the breed statements and other declarations (a suitable location is immediately above the setup procedure).
3. Amend the setup procedure to call the initialise-globals procedure as the first line after **clear-all**.

```

to initialise-globals
  set travel-rate 0.1
end

```

The **globals** statement declares the existence of this new variable. The initialise-globals procedure uses **set** to assign a value. Setting all the global variable initial values in a single procedure

allows them to be easily found and changed if necessary, even by a model user. The call to that procedure must occur before any of the variables are used by other procedures, otherwise the value 0 will be used instead. Incorrect global values can introduce subtle logic errors that are difficult to detect and *NetLogo* will not report an error (unless, for example, the variable has a value of 0 and is the denominator in a division).



Setting global variable values

Setting the values for all the global variables in a single procedure reduces the chance of accidentally neglecting to set the value for one of them. It is good practice to include the variables that are to have their values set to 0, even though *NetLogo* would set the value to 0 automatically. The procedure to set the values should be called as the first procedure to run after the **clear-all** command.

Snippet 2f-2 (page 60) uses the new global variable to allocate some of the change in total infected population to neighbouring patches rather than just the patches where the infections are generated. The patch does this by randomly selecting one of the eight patches surrounding it and changing the incidence value in that patch. However, a patch could receive new infections from more than one patch, so care must be taken with the sequencing of actions.

The various sources of new infections must be summed within each patch to calculate the total incidence for that patch. This means that all patches must have the opportunity to generate new infections before the value of new infections is used to update patch epidemic status counts. In addition, since incidence is being cumulated rather than calculated, it must be explicitly set to 0 at the beginning of each time step so that only the new infections in the new time step are added together. Therefore, the spread-epidemic procedure must have multiple **ask patches** [] code blocks.

As well as the already discussed **neighbors** primitive, this snippet introduces **one-of**. This is similar to **n-of**, discussed at 47, but randomly selects only one member of the specified set of agents (patches, in this case).

This snippet also introduces **myself** but this is a difficult primitive to understand, and some further work is required before it can be fully explained. At this point, it is sufficient to recognise that it is referring to the patch that did the asking.

Snippet 2f-2: Effect of travel

1. Amend epidemic-spread procedure. New or amended lines are marked with <<<

```
to spread-epidemic
  ask patches [ set incidence 0 ] <<<
  ask patches
  [ ; calculate the number of new infections generated by the patch
    let beta (R0 / recovery-period)
    let new-cases popn-I * beta * popn-S / popn <<<
    ask one-of neighbors [ set incidence incidence + travel-rate *
      [new-cases] of myself ] <<<
    set incidence incidence + (1 - travel-rate) * new-cases <<<
  ] <<<
  ask patches <<<
  [ ; update the population counts in each epidemic state <<<
    ...
    ...
  end
```

After implementing Snippet 2f-2 (page 60), press the *setup* and then *do it!* buttons to see the effect of the changes. You should have the epidemic spread throughout the world, as shown in Figure 3.10.

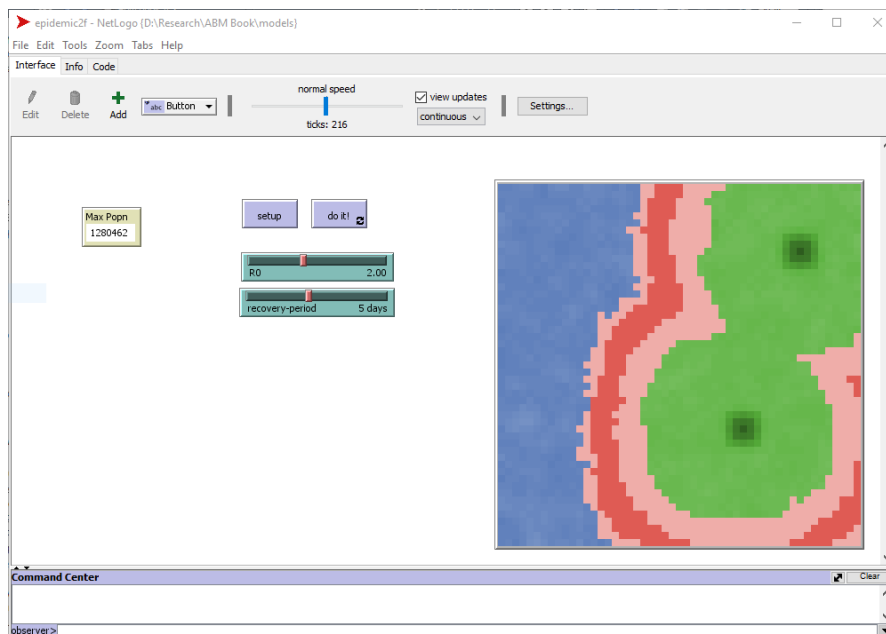


Figure 3.10: Screenshot of the interface after completing version 2f of the model. The epidemic should spread throughout the world, with a darker red epidemic frontier and patches turning green as the epidemic has passed.

3.10 Output Widgets for Model Results

So far, reporting of model results has focused on diagnostic uses of widgets (such as the maximum population monitor) and ensuring that the world displays the most useful information in informative ways (colours). However, there are several widgets available to report results to help users to understand what is happening in the simulation.

The world view presents the state of the simulation at a specific tick (describing what is happening ‘now’). Simulations represent a process, and it is often important to summarise how some aspect of that process changes over time. Other summary information can also help the user interpret the model. Output widgets are constructed in the same way as the input widgets (see section 3.5), by selecting a type from the widget dropdown box and placing it on the interface.

The most common use of a *plot* in *NetLogo* is as a line chart that displays some model information on the y-axis against time (ticks) on the x-axis. Other plots are available, however, if displaying over time is less important. Histograms are reasonably common (to be demonstrated at Snippet 3b-3 (page 71)) and it is also possible to plot two values against each other.

A *monitor* is used to display a single value, such as the contents of some variable or the results of a calculation. A *note* places fixed text on the interface, such as a heading. Finally, text can be generated by the model and written to an *output* area on the interface.

3.10.1 Reporting the epidemic

Key information about an epidemic is the incidence (new infections) and prevalence (current infections). A simple epidemic follows the classic diffusion curve, with an initial increase, slowing and then decrease. Plotting these curves will provide further verification that the epidemic is behaving as expected. It will also show the way in which changes in epidemic parameters affects the spread of the epidemic more easily than can be seen in the world view. However, it is easier

to compare numbers than curves across multiple simulations and a monitor will be added to the interface to report the impact of the epidemic, the total population ever infected. You should think about how to calculate these values before moving on.

Note that you may need to move the existing widgets to make room for the plot. This can be achieved by selecting the widget (popup menu) to highlight it and then dragging it with the mouse. Selecting a widget also allows it to be resized by dragging an edge or corner.

Snippet 2g (page 62) describes the plot and monitor to be added to the interface. The plot has two pens (different coloured lines), one for incidence and the other for prevalence. Figure 3.11 displays the relevant plot dialogue box with the code and other details filled in. The monitor will display the total of the popn-I and popn-R values.

Snippet 2g: Summarise epidemic progress

1. Delete the maximum population monitor (popup menu on the widget).
2. Create a plot titled 'Epidemic' with pens for 'Incidence' and 'Prevalence'.
3. Create a monitor titled 'Impact'.

```
plot sum [incidence] of patches / sum [popn] of patches ; code for
incidence pen
plot sum [popn-I] of patches / sum [popn] of patches ; code for
prevalence pen
(sum [popn-I] of patches + sum [popn-R] of patches) / sum [popn] of
patches ; code for impact monitor
```

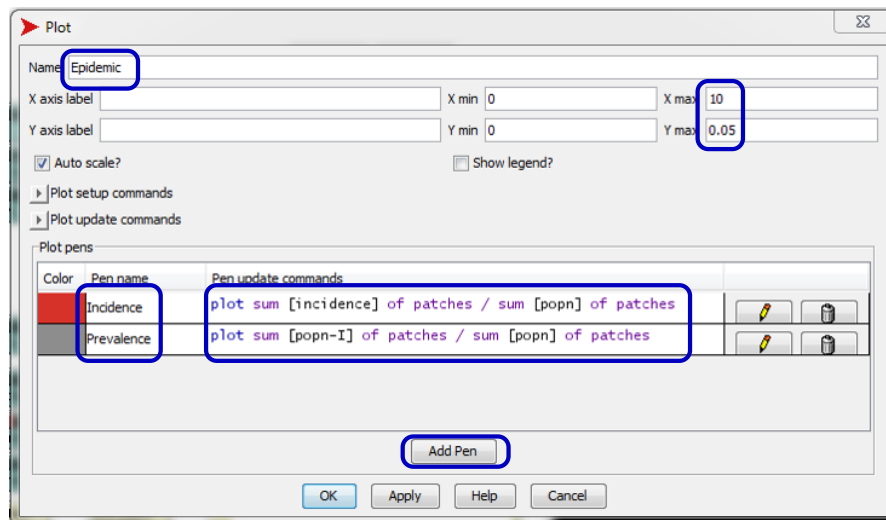


Figure 3.11: Screenshot of the plot dialogue box for the epidemic progress. The top text box is used to enter the title of the plot. To the right are the (starting) limits of the axes. The x-axis is for ticks, so the default value of 10 is reasonable. However, the y-axis is proportion of the total population, and a small number is required to make sure the plot curves are visible. The Auto scale? checkbox allows these axis limits to increase as required, and is essential when plotting against ticks. The button at the bottom of the dialogue box creates extra pens. Each pen is a line with a colour (changed by clicking on the coloured area), name, and code (commands). If there is insufficient space in the code box, press the pencil icon button at the right to access a further dialogue box.

After implementing Snippet 2g (page 62), press the *setup* and then *do it!* buttons to see the effect of the changes. As the epidemic spreads, you should see the plot curves increase and then decrease (see Figure 3.12), potentially with additional peaks as the epidemic reaches areas with high population density.

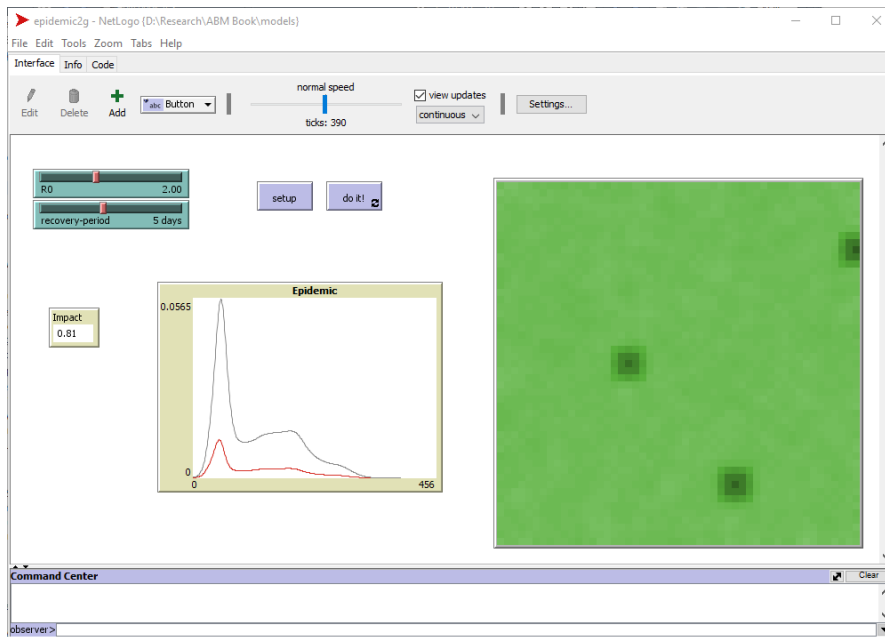


Figure 3.12: Screenshot of the interface after completing version 2g of the model, with the plots and impact monitor. This screenshot was captured at the end of the simulation, when the epidemic is over, to show the classic shape of the epidemic curves.

Try changing the epidemic parameters and seeing the effect on the epidemic. You should find that a larger value of R_0 and/or shorter recovery period increases the intensity of the epidemic (steeper curves) and also the impact (total population affected). However, regardless of the parameter values, the impact will not reach 1, some population will remain susceptible. What happens if you reduce R_0 to below 1? This means that each infected person infects less than one other person on average, so the epidemic dies out very quickly. Because of the spatial effects (where infected people are concentrated), this dying out will actually occur at some value greater than 1.

3.11 Tutorial Progress Check

Version 2 of the model is now complete. The model represents the process of an epidemic spread and is therefore a simulation. There are also aspects of an agent-based model; changes on each patch are influenced by the surrounding patches and patches have different characteristics. However, there is no sense of agency, taking actions, or decision making, the patches are simply responding to their environment. It would satisfy only a broad definition of agent-based modelling.

Space and time are critical elements in agent-based modelling introduced in this model version. *NetLogo* is equipped with a spatial structure (and co-ordinate system) and a counter for the passage of time, as well as a range of keywords to access and control time and space.

Space is necessary to support a representation of the environment, an important source of influence in many agent-based models. The important spatial primitive `neighbors` uses the spatial structure inherent in the *NetLogo* platform to implement local actions.

Time is necessary for any representation of a process. In *NetLogo*, time is measured in arbitrary units referred to as ticks. In the tutorial model, a tick represents a day, but some other interval could have been used. Such connection to a real world time unit is necessary where empirical data must be translated for the model, so that appropriate parameter values can be assigned.

This version also introduced conditional testing and boolean variables. These both require that *NetLogo* recognise `true` and `false`, not simply as words but as truth states. Conditions are

used to control the flow of the code, so that different commands can be implemented in different situations. So far, this has only been used for visualisation, to colour the world. In the next version of the model, conditional code will allow agents to make decisions that reflect their own characteristics and circumstances, the essence of agent-based modelling.

Also formally presented was the fundamental command `ask` to pass instructions to specific agentsets, which had been used in version 1 code snippets without comment. Detailed discussion is deferred until agentsets have also been presented.

3.11.1 Programming practices

In this version of the model, the programming practices introduced in the previous version were reinforced. Each new snippet added as small a change as possible and both the syntax and logic were tested before making further changes.

The code is modular, with each separate procedure implementing a specific action and potentially calling other procedures for subsidiary tasks. The *go* procedure comprises a series of calls to the procedures that implement the epidemic process. This is equivalent to the *setup* procedure controlling the model initialisation. The main control procedures are placed near the top of the code tab to make it easy to see this highest level ordering of tasks.

Control flow with `if` conditions is a standard programming approach. Other control flow structures included in *NetLogo* are `while` and `loop`, neither of which is used in the tutorial. They all follow the same structure, the code block is run if some condition is satisfied. Unlike `if`, `while` and `loop` allow multiple iterations through the code block. The previously presented primitive `repeat` controls multiple iterations through a code block without conditions.

The final aspect of programming practice reinforced in this model version concerns good interface design. This includes use of colours (to highlight the epidemic front) and explicit consideration of the information most useful to the model user for presentation in plots and summary values.

3.11.2 *NetLogo* language

There are two types of primitives in the *NetLogo* language, commands and reporters. Commands change some aspect of the model, such as storing a new value in a variable. In contrast, reporters simply extract information, such as recovering the stored variable value.

It is not a coincidence that this distinction exists for both primitives and procedures. Fundamentally, procedures and primitives are handled the same way by *NetLogo*; when *NetLogo* reaches one, it jumps to the piece of code with that name and receives instructions on what to do. The difference is that the code to run with a named primitive happens to be written by the *NetLogo* developers, while procedures are written by model developers (like you).

New *NetLogo* keywords explained in this model version:

- time: `reset-ticks`, `tick`
- space: `neighbors`
- code structures: `to-report`, `if`, `ifelse`, `ifelse-value`
- working with variable values: `set`, `of`
- working with agents: `ask`, `n-of`, `max-n-of`
- mathematics: `count`, `sum`, `!=`, `>`
- miscellaneous: `stop`

In addition, the primitive `myself` was used but not explained. A full understanding requires the concepts of agentsets and context, both presented in the next chapter.

Model 3: Agents Making Decisions

Model 3 implements the behaviour decision process: people adopting (and dropping) protective behaviour. Once implemented, this version is a full agent-based model because it represents individuals enacting a process based on their situation and characteristics. That situation involves interaction with other individuals, so all the essential elements of the definition of an agent-based model are in place.

From this point in the tutorial, there are fewer additional keywords introduced in each model version. While *NetLogo* is a rich language, only a small subset is necessary to implement many agent-based models. However, only a small part of writing code is knowing the relevant keywords, and there are many important concepts remaining.

4.1 Model Design: Protective Behaviour Decisions

As for the epidemic process, a detailed specification of the behaviour decision process is required before considering how to implement that design in code. The broad design (see section 1.8) states that people will behave protectively while the average of their attitude, behaviour of nearby people, and threat perception is above some threshold. Each of these behaviour inputs must be operationalised, as well as the calculation of the average.

The first input is attitude, if a person is more positive about protective behaviour, then they would adopt such behaviour more easily. This is a personal characteristic that varies across individuals but is constant for each over time. While any scale can be used, it is convenient to operationalise attitude as a number in the interval $[0,1]$.

The second input is the behaviour of nearby people. The most direct way to include this is to simply measure the proportion of people on nearby people that are currently enacting protecting behaviour. Initially at least, nearby will be operationalised as the same patch and neighbouring patches. As a proportion, the behaviour of nearby people will also be a number in the interval $[0,1]$, but it will change during the simulation as protective behaviour is adopted or dropped.

The final input is threat perception. The perception of threat should increase as the local infections increase and reduce as the epidemic passes. One way to implement this pattern is a discounted cumulative sum of local incidence, so that the influence of previous incidence levels decreases over time. Incidence is the new infections this time step as proportion of population. If δ is the discount, then $(1 - \delta)$ is the incidence to be retained for threat perception in the next time step, and such discounting continues indefinitely. For example, at time $t = 0$, threat would be initial incidence I_0 , then $I_1 + (1 - \delta)I_0$ at time $t = 1$, where I_j is the incidence at time j . At time $t = 2$, threat would be $I_2 + (1 - \delta)I_1 + (1 - \delta)^2I_0$, which can also be written as $\sum_{j=0}^2 (1 - \delta)^{t-j} I_j$.

The relative importance of each of these factors in the decision can be controlled with weights.

An individual's behaviour score is therefore given by:

$$B_i = \omega_A A_i + \omega_N N_{r1} + (1 - \omega_A - \omega_N) \sum_{j=0}^t (1 - \delta)^{t-j} I_{t,r2}$$

where i specifies the individual, $r1$ and $r2$ denote the relevant regions for the perception of norms and threat respectively, t is the time step, A , N and I represent attitude, norms and incidence, and ω_A and ω_N are the weights for attitude and norms. The behaviour score is compared to a threshold and behaviour is adopted or dropped accordingly.

In addition, people's protective behaviour must influence the epidemic process. The capacity of the epidemic to spread in a region (r) should reduce with increases in the proportion of people adopting protective behaviour (P_r) and the efficacy of that behaviour (E). Efficacy is the reduction in probability that a protected person is able to become infected or, if already infected, transmit the infection. For example, if 100% of the population have adopted protective behaviour and that behaviour is 40% efficacious, then infections should be 40% lower than they would be without such behaviour and, if 40% of the population have adopted protective behaviour and that behaviour is 100% efficacious, then infections should also be 40% lower than they would be without such behaviour. What happens if 60% of the population have adopted protective behaviour and that behaviour is 50% efficacious? This would reduce infections by 30% (calculated as 50% of 60%). Since β in the epidemic transmission equations represents that capacity, this parameter must be adjusted for local behaviour, with the adjustment given by:

$$\beta_r = \beta (1 - P_r E)$$

What is needed to implement this design? Try to identify the information (variables) and actions (procedures) required before moving on. Are the variables global or do they potentially have different values for some patches or turtle agents? Which procedures required for initialisation (called by setup) or during the simulation (called by go)? All of this detail is part of the conversion from a design to a fully operationalised specification suitable for translating into code.

4.2 Context and Perspective

The basic representation in agent-based modelling takes the agent's perspective of its situation and acts accordingly. The need to take the perspective of different entities is incorporated in the *NetLogo* language, and is formally referred to as context. As previously described (see section 1.4.2), there are four types of model entities: observer, turtles, patches and links. Each of those entities also supports a context.

Consider the code at Figure 4.1. Each version changes the colours of the patches along the diagonal to either red (left) or blue (right). You can test this by adding the procedure(s) to the code tab, then typing the name of the procedure in the Command Center.

The diag-observer procedure is written from the perspective of the observer, an external entity that can pass instructions to the model and, in some sense, stands in for the model developer or user. The first step is to **ask patches** to follow the instructions in the code block (delimited by []). The **ask** instructs the simulation to randomly choose a patch, apply the code in the code block to that patch, then move to another randomly chosen patch until all patches have run the code. When interpreting (or writing) the code, you might find yourself imagining yourself as a patch and thinking about what happens to you. That is, all the code in the code block is from the perspective of the patch, you don't have to specify the patch to check its **pxcor** and **pycor** and changes colour if they are equal, it is automatically whichever patch is the current patch in the **ask** order. In *NetLogo* terms, the opening bracket changes the context from the observer to the patch, and the closing bracket changes it back.

```

to diag-observer
  ask patches
  [ if pxcor = pycor
    [ set pcolor red
    ]
  ]
end

to diag-bypatch
  ask patches [ do-patch ]
end

to do-patch
  if pxcor = pycor
  [ set pcolor blue
  ]
end

```

Figure 4.1: Running procedure `diag-observer` (left) and `diag-bypatch` (right) achieves the same outcome. The code on the left takes the perspective of the observer, instructing some patches to change colour to red. In the code on the right, the procedure `diag-bypatch` instructs each patch to run the `do-patch` procedure. The `diag-bypatch` procedure is observer context, and the `do-patch` procedure is patch context.



Square brackets - context change

The square brackets `[]` are used to change contexts in *NetLogo* (and for other purposes). For example, following a command to `create-turtles` by the observer, the context changes to turtles and the code delimited by the square brackets is run for each turtle as it is created.

The `diag-bypatch` procedure is also written from the perspective of the observer. It starts the same way, but then transfers control to a separate procedure. As before, the context changes with the square brackets. In this version, however, the different context occurs in a separate procedure. The `do-patch` procedure is written from the perspective of an individual patch and is run once for each patch (controlled by the `ask patches` in `diag-bypatch`).

The primitive `myself` can only be interpreted from an understanding of context. When one model entity instructs a second model entity to do something, that second entity may need information that is known to the original entity. But the switch of context with the `ask` statement means that *NetLogo* is running code from the perspective of the second entity. So the `myself` primitive provides a link back to the first entity, the one issuing the `ask`.

myself

`myself` is used inside an `ask` code block. It refers to the agent (or model entity) that issued the `ask`. See also `self`.

NetLogo keeps track of the context automatically. While most keywords can be used in any context (such as `ask` or `if`), others can only be used within certain contexts. These are marked by an icon in the *NetLogo* dictionary: observer has an eye (👁️), patch has a checkerboard (♠️), turtle agent has a turtle (🐢) and link has a chain link (🔗). For example, `tick` can only be used by the observer, and `xcor` is only accessible from the turtle context.

In the `diag-observer` procedure, the first line of `ask patches` signals to *NetLogo* that the procedure is intended to run in the observer context. *NetLogo* does not allow a patch (or turtle or link) to `ask` all the patches (or turtles or links) to do something. In the `do-patch` procedure, the first line refers to the patch only primitive `pxcor`, signalling to *NetLogo* that the procedure is to be run by patches.

4.2.1 Updating threat, a patch procedure

Threat is calculated as a discounted sum of local incidence. While it is used by turtle agents in their decision making, it is the same value for all turtles on a patch. It is therefore more efficient (fewer calculations) to calculate at the patch level and simply have the turtles access the relevant threat value.

Remember from section 4.1 that the threat (at patch r) at time t is given by a discounted cumulative sum of incidence (as a proportion of population) over time, with δ representing the discount. But calculating the sum would require keeping track of all the values of incidence since the start of the simulation. A different way of thinking about this is that the new value of threat is given by the sum of the new incidence and the discounted value of the threat at the previous time step.

$$\begin{aligned} T_{t,r} &= \sum_{j=0}^t (1 - \delta)^{t-j} I_{j,r} \\ &= (1 - \delta)^0 I_{t,r} + (1 - \delta) \sum_{j=0}^{t-1} (1 - \delta)^{(t-1)-j} I_{j,r} \\ &= I_{t,r} + (1 - \delta) T_{(t-1),r} \end{aligned}$$

The update-threat procedure at Snippet 3a (page 68) uses this relationship to calculate the new value of threat each tick. The procedure is a patch context procedure because the call to the procedure will be inserted within an `ask patches []` block in the epidemic-spread procedure. Threat must be updated after the epidemic has been updated because it requires the value of incidence for the patch.

Snippet 3a: Calculating threat

1. Create new global variable ‘threat-discount’.
2. Add threat-discount to the initialise-globals procedure (with value 0.1).
3. Add ‘threat’ to the patches-own attributes.
4. Create update-threat procedure in the implementation procedures section.
5. Add the call to update-threat at the end of the last ‘ask patches’ in the ‘spread-epidemic’ procedure.

```
to update-threat          ; patch procedure
  set threat incidence / popn + (1 - threat-discount) * threat
end
```

Note that the same variable is being used for both the assignment and within the calculation. Mathematically, this would introduce an error for circular reasoning. However, this construction in *NetLogo* and many other programming languages is interpreted as: calculate the expression using the current value of the variable named ‘threat’ and then assign the results of that calculation to the variable named ‘threat’.

This snippet makes no visible change so the model remains as in Figure 3.12. However, a sanity check is necessary to ensure the code works as intended. Press *setup* and then open an inspect window for one of the patches where infections are present. Type ‘go’ into the Command Center to run the go procedure once. Look at the values of incidence and threat, then type ‘go’ again and look at the updated values. At tick 1, the value of threat should be incidence / popn. At tick 2, the value of threat should be $0.9 \times \text{tick 1 threat} + \text{incidence} / \text{popn}$. More generally, inspect a patch outside of the epidemic area and note that the threat is zero until the epidemic is close then increases as the epidemic reaches the patch and decreases once the epidemic has passed.

4.3 Procedures with Arguments (Inputs)

So far, procedures have simply run the same set of commands each time, with the only variation in effect arising from the differences in the values of patch variables or environment. However, the full syntax of procedures allows inputs to be provided to the procedure.

This is similar to mathematical functions. For example, the function $f(x) = x^2 + 3$ returns different values for different values of x . For $x = 2$, $f(x) = 7$ and for $x = 5$, $f(x) = 28$ and so on. In the same way, procedures can be written that include an input (referred to as an argument) in the processing of the code. This is true for both command procedures (defined by `to`) and reporter procedures (defined by `to-report`).

The arguments are named immediately after the procedure name and included in square brackets. When the procedure is called, the values for those arguments follow the procedure name without any special syntax.



Procedure argument naming

The arguments should have meaningful names. However, variables being passed could have very similar names, leading to confusion. It is therefore sensible to have some sort of naming convention that distinguishes between the agent variables and the procedure arguments. One such convention is to start the argument name with a special character such as a hash mark (#).

4.3.1 Assigning attitude, a procedure with arguments

In attitude surveys, a common response is ‘agree’, which results in a skewed distribution. To replicate this pattern, the attitudes will be randomly assigned from a triangular distribution with a mode of 0.8. *NetLogo* does not have a random number generator for the triangular distribution, but it is easy to convert a uniform distribution. To convert a uniformly generated number U on $[0,1]$ to a triangular X on $[0,1]$ with mode m :

$$\begin{aligned} X &= \sqrt{Um} && \text{for } U < m \\ X &= 1 - \sqrt{(1-U)(1-m)} && \text{for } U \geq m \end{aligned}$$

If a procedure might be relevant to many different models, it is useful to have a general purpose procedure with arguments rather than fix values within the code. This makes it more flexible. Snippet 3b-1 (page 70) creates a general procedure for generating random numbers from a triangular distribution with three parameters for the limits and the mode. The `triangular0to1` procedure calls the general procedure, passing the mode and setting the limits as 0 and 1. This latter procedure is used to generate attitude values in Snippet 3b-2 (page 70).

How can this code be checked? Random number generators can only be tested by drawing many random numbers and examining the distribution of the values. However, there is no need for strict tests of randomness; that is the responsibility of the *NetLogo* developers in their construction of the `random-float` primitive. It is instead sufficient to examine whether the conversion to a triangular distribution is working as expected.

Snippet 3b-3 (page 71) constructs a histogram of the attitude values. It will populate after you press the *setup* button. What do you expect? Since you have assigned a mode of 0.8, you should see a triangular shape with the highest point at the value of 0.8 similar to Figure 4.2. Note that if you have only a single bar, you probably did not follow the instruction to set the interval (bar width) to 0.05.

Snippet 3b-1: Customised random numbers

1. Create random-triangular procedure in the utility procedures section.
2. Create triangular0to1 procedure in the utility procedures section.

```

to-report triangular0to1 [ #Mode ]
  report random-triangular #Mode 0 1
end

to-report random-triangular [ #Mode #Lower #Upper ]
  let uRand random-float 1
  let scale (#Mode - #Lower) / (#Upper - #Lower)
  report ifelse-value (uRand < scale)
  [ #Lower + sqrt(uRand * (#Upper - #Lower) * (#Mode - #Lower)) ]
  [ #Upper - sqrt((1 - uRand) * (#Upper - #Lower) * (#Upper -
    #Mode))]
end

```

random-float

The full syntax is `random-float n`. Generates a number anywhere in the continuum from 0 to n. The random number is uniformly generated along the continuum.

More thorough testing would vary the mode and see if the histogram adjusts accordingly. The most comprehensive testing would check the random-triangular procedure with different limits to make sure that the limits of 0 and 1 are not the only ones that work. Try code such as `set attitude random-triangular 3 2 5` for assigning attitude values. Remember to change the code back after testing.

**Operationalising probability**

The `random-float` reporter is particularly useful for implementing code that should be run with some probability, replicating the outcomes of rolling a fair die with an appropriate number of sides. For example `if random-float 1 < 0.2` will be `true` with a 20% probability.

Snippet 3b-2: Assign (random) attitudes

1. Create a variable named 'attitude' for people.
2. Add call to triangular0to1 to assign attitude in make-people procedure. New line is marked with `««`

```

to make-people
  ...
  set size 0
  set attitude triangular0to1 0.8 ««
]
...
end

```

Snippet 3b-3: Verify with histogram

1. Create a plot titled 'Attitude' with x-axis upper limit (X max) of 1 and the given code to create a histogram.
2. Use the pencil icon (at the end of the code entry line) to access the detailed settings and select bar mode with 0.05 as the interval.
3. Once the triangular distribution tests are complete, delete the plot (context menu on the plot).

```
histogram [attitude] of people
```

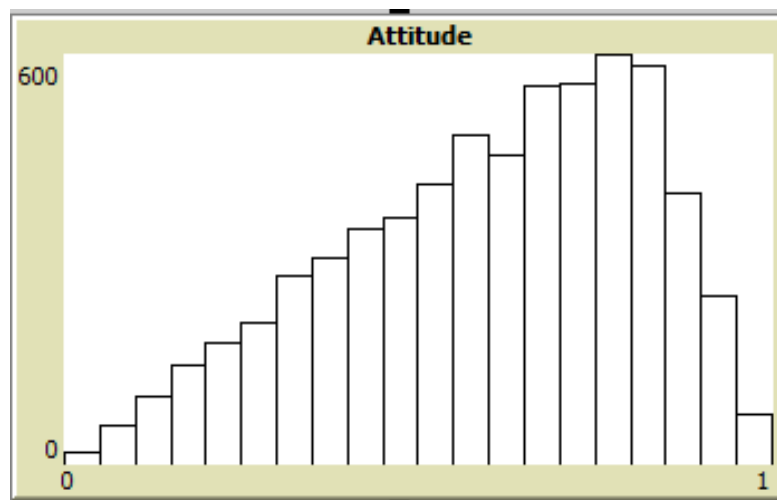


Figure 4.2: Screenshot of the histogram displaying the distribution of attitude values. Note the peak at 0.8 and the triangular shape.

4.4 Agentsets

An agentset is simply a group of patches or turtles or links but not a mix. An agent can only appear once in an agentset; if an attempt is made to add an agent to an agentset in which it is already a member, nothing happens.

Whenever an agentset is accessed, for example with an `ask` command, the access is in a random order. That order changes each time the agentset is accessed. This means that sequencing effects are automatically avoided; for example, no agent gets the advantage of always being the first to make a choice.

Agentsets provide much of the power of the *NetLogo* language. They support clear and concise short coding of the fundamental representation in agent-based modelling, taking the perspective of each agent individually so that it can take some action.

So far, the tutorial model has extensively used the built in `patches` dataset, which is the group of all the patches. There are also equivalent agentsets for all `turtles` or all `links`. In addition, if breeds have been defined, then there is also an 'all members' type agentset for each breed. These agentsets are not only automatically available for any model, they are updated as agents are added (for example, with a `hatch` command) or removed (with a `die` command). Another important agentset built into the *NetLogo* language is `neighbors`.

Agentsets can also be created explicitly with a `patch-set` or `turtle-set` or `link-set` command. This command instructs *NetLogo* to combine the agentsets that are specified into a single agentset. That is, these commands are the set operator 'union'.

patches

The `patches` agentset consists of all patches. It is automatically generated and updated (for example, if the size of the World is changed) and available to all model entities. See also `turtles` and `links`.

The final method of creating agentsets is by selecting agents from an existing agentset. This has also been used already in the tutorial model with `n-of` and `max-n-of`. Another common subsetting command is `with` to select those agents that meet some condition, which will be used in Snippet 3e (page 77).

Agentsets can be assigned to a variable in exactly the same way as any other variable value, using a `set` or `let` command. But they can also be used in the code without being assigned to a variable. For example, in Snippet 1b-3 (page 29), the instruction to assign a high population value to some patches was in the form: `ask n-of 3 patches [set ...]`. A set of 3 patches was created by the `n-of` statement and used by the `ask` command without ever being assigned to a variable.



Assigning agentsets to variables

If an agentset is to be used multiple times, it should be assigned to a variable. This is more efficient as the agentset only needs to be created once. While it is good practice to also store other information used multiple times, it is particularly important for agentsets as their creation can be computationally expensive if there are many agents in the model.

4.4.1 Agentset application

Snippet 3c-1 (page 73) adds the procedure for people to make decisions about adopting and dropping protective behaviour. It uses all three types of agentsets discussed: built in, explicitly constructed and subsets. All the new primitives included in the procedure concern agentsets.

The procedure will be setting the ‘protect?’ attribute owned by the turtle agent people. This variable has already been created in Snippet 1c-1 (page 32), when demonstrating the `turtles-own` primitive, but the other variables needed to support the procedure must be created. Once you have complete the snippet, try and identify every agentset in the procedure before moving on.

There are several built in agentsets used. Both `patches` and `people` are complete sets of patches and the turtle agents people respectively. The agentset `self` has only a single member, the agent running the code. The final built in agentset used is `neighbors`, introduced in section 3.9.2.

The `patch-set` primitive is used to construct a 3×3 group of patches as the union of the patch creating the patchset at the centre (with `self`) and its eight neighbours (with `neighbors`). `self` is an example of an agentset with only a single member. The union patchset is stored in the variable named `visible`, which is temporarily created for each patch in turn.

The other agentsets are more subtle. The new primitive `people-on` is a breed version of `turtles-on`, which constructs an agentset comprised of all the turtles that are located on the specified patchset, in this case the 3×3 group of patches named `visible`. So the new agentset is all the people that are nearby, either on the same patch or one of the neighbouring patches. A related agentset is constructed by selecting only those people with the variable named ‘protect?’ set to `true`. The number of people in each of these agentsets is calculated with the primitive `count`, and the quotient of these counts gives the proportion of nearby people who are currently adhering to protective behaviour.

Snippet 3c-1 (page 73) sets the value of ‘protect?’ to `true` or `false` depending on the results of a conditional test, which also returns `true` or `false`. The truth value has been explicitly assigned for clarity, but there is a shorter equivalent (see Figure 4.3) that takes advantage of the fact that

Snippet 3c-1: Decide behaviour

1. Add two variables to the **patches-own** list: prop-protect, max-threat
2. Create three sliders for weights and threshold with range 0 to 1: wt-attitude (value 0.4), wt-norms (value 0.2), threshold (value 0.35)
3. Create decide-behaviour procedure (in the implementation procedures section)
4. Add call to decide-behaviour as the last line in the go procedure before **tick**

```

to decide-behaviour
  ask patches
  [ let visible (patch-set self neighbors)
    set prop-protect (count (people-on visible) with [protect?]) /
    (count people-on visible)
    set max-threat max [threat] of visible
  ]
  ask people
  [ let decision-value wt-attitude * attitude + wt-norms *
    prop-protect + (1 - wt-attitude - wt-norms) * max-threat
    set protect? ifelse-value (decision-value >= threshold) [TRUE]
    [FALSE]
  ]
end

```

self

self is the agentset comprised of one agent, the one from whose perspective the code is being implemented. See also **myself**.

truth values are consistent; it doesn't matter whether it's a boolean variable or a conditional test, **true** is **true**.

By now, you should be considering what sanity check to use to check this code. One option is to run the model and stop it when the epidemic is very active. You can then simply count the people currently protected by typing code into the Command Center. Suitable code would be **count people with [protect?]**. You could also **inspect** some people and the patch where they are located to see the values of the various decision inputs and the protection state of the person. Or you could add a monitor to the interface to count the protected people and let the model run, deleting the monitor when you are confident the model is operating correctly.

**Interface design**

Organise the interface so that related widgets are close to each other. Another option is to have the basic controls in one area and more advanced settings in another area. The goal is to make the model easy to use and understand.

patch-set

The full syntax is **(patch-set input-patchset1 input-patchset2 input-patchset3 ...)**. It is a union operator to combine multiple specified patchsets into one patchset. Even if a patch appears in more than one of the input patchsets, it will only appear once in the combined patchset. See also **turtle-set** and **link-set**.

turtles-on

The full syntax is `turtles-on <some-patchset>`. Creates an agentset of all the turtles that are located on the patches in the specified patchset. The turtles can be all turtle agents, or just those of a specific breed.

with

The reporter `with` subsets an agentset, creating a new agentset that only contains the agents that meet the specified condition.

An easier way to check whether the protective behaviour process is working as expected is to construct a plot of the proportion of the population who are protected. This is also useful information for users, and the plot to add is described at Snippet 3c-2 (page 75). However, such a plot does not negate the need to inspect some people to ensure the detailed calculations are correct. With a plot, it is easy to test the effect of different threshold values; with a low threshold almost every person should be protected, and none would be protected if the threshold was set very high.

Once the plot is created, press the *setup* and *do it!* buttons. The model should be similar as shown in Figure 4.4. Notice that the interface widgets are grouped so that epidemic controls are together, as are protective behaviour settings.

4.5 Finishing off the Core ABM

The processes of the ABM are now in place. However, protective behaviour is not yet influencing the epidemic. From the model design at section 4.1, a patch's capacity to transmit the epidemic must be adjusted for the level of protective behaviour and its efficacy with: $\beta_r = \beta(1 - P_r E)$. This is implemented in Snippet 3d (page 76).

This adjustment introduces a new agentset primitive. `people-here` is a breed specific version of `turtles-here`, which creates an agentset of all the turtles at the patch issuing the instruction. This enforces the local effect of protective behaviour, as only people on the relevant patch are able to influence the epidemic process.

It is straightforward to test this snippet, by simply adjusting the efficacy slider and seeing the effect on epidemic impact. Experiment with different levels; an example where the behaviour stopped the epidemic from fully taken off is shown at Figure 4.5. Remember also that the effect of protective behaviour on the epidemic depends on both the efficacy and the population adopting protective behaviour (which can be controlled with the threshold).

Protective behaviour comes in many forms including vaccination, increased hand washing, or reducing social contact. The model should be able to represent any of these behaviours. They vary in efficacy, which can be adjusted with the slider that has just been added. However, they also vary in permanency. Once a person is vaccinated, they can't abandon the behaviour.

The final step of model version 3 is to add a user control to set whether the protective behaviour is reversible. This is to be implemented with a new type of input widget called a switch. A

```
set protect? ifelse-value (decision-value >= threshold) [TRUE] [FALSE]
set protect? decision-value >= threshold
```

Figure 4.3: These two lines of code are equivalent. The first explicitly sets the truth value of the variable based on whether the conditional is true. The second simply assigns the result of the conditional and is shorter, but more difficult for a human reader.

Snippet 3c-2: Summarise protected people

1. Create a plot titled ‘Proportion protected’

```
plot count people with [protect?] / count people ; code for pen
```

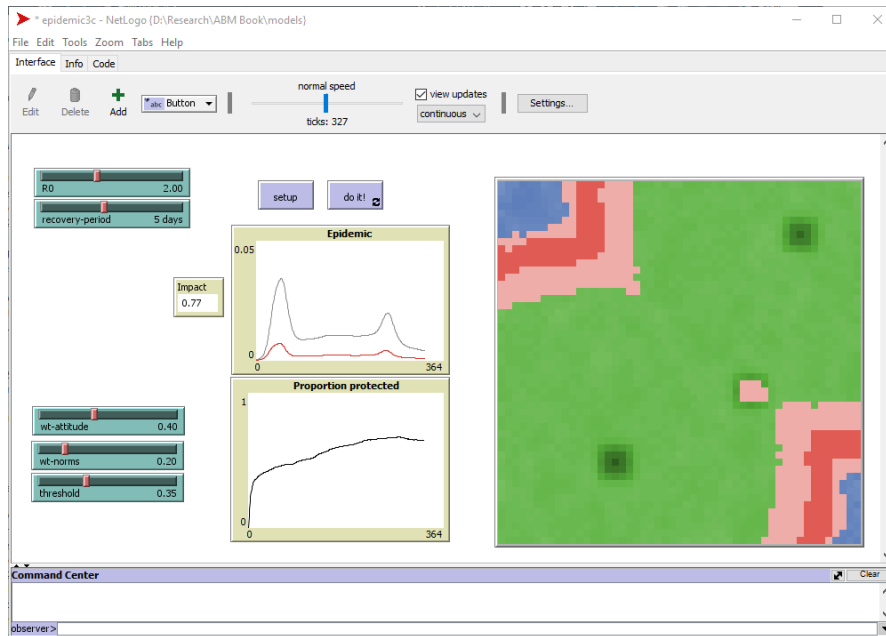


Figure 4.4: Screenshot of the interface after completing version 3c of the model. Approximately half the population quickly adopts protective behaviour, with protection levels starting to decline as the epidemic is dying out.

switch is used for boolean variables, it is either on (**true**) or off (**false**). As with other widgets, it is added directly to the interface and its value assigned to a global variable by entering an appropriate variable name.



Working with truth values

Use logical operators when working with boolean variables because *NetLogo* will expect a logical value (**true** or **false**) and report an error if it encounters some other value. For example, **if** [test?] will report an error if ‘test?’ has value 0, but **if** [test? = **true**] will appear to run correctly.

Change the efficacy slider to 0 and run the model twice, once with the switch on, and once with the switch off. Because efficacy is set to 0, the level of protective behaviour has no effect on the epidemic impact and there should be little difference in model results (averaged over many runs). However, in the runs with the switch set to not reversible, the proportion protected should increase (or remain stable), even when the epidemic has passed. This can be seen in Figure 4.6.

4.6 Tutorial Progress Check

Version 3 of the model is now complete. Referring back to the definition (see section 1.2), it is now an agent-based model, as it:

Snippet 3d: Behaviour influences epidemic

1. Create slider for ‘efficacy’ (0 to 1 by 0.01, value 0.8)
2. Amend spread-epidemic procedure. Amended line marked by <<<

```

to spread-epidemic
...
ask patches
[ ; calculate the number of new infections generated by the patch
  let beta (R0 / recovery-period) * (1 - efficacy * (count
people-here with [protect?] / count people-here)) <<<
  let new-cases popn-I * beta * popn-S / popn
...
end

```

turtles-here

`turtles-here` creates an agentset of all the turtles that located on the instructing patch, or the patch where the instructing turtle is located. The turtles can be all turtle agents, or just those of a specific breed.

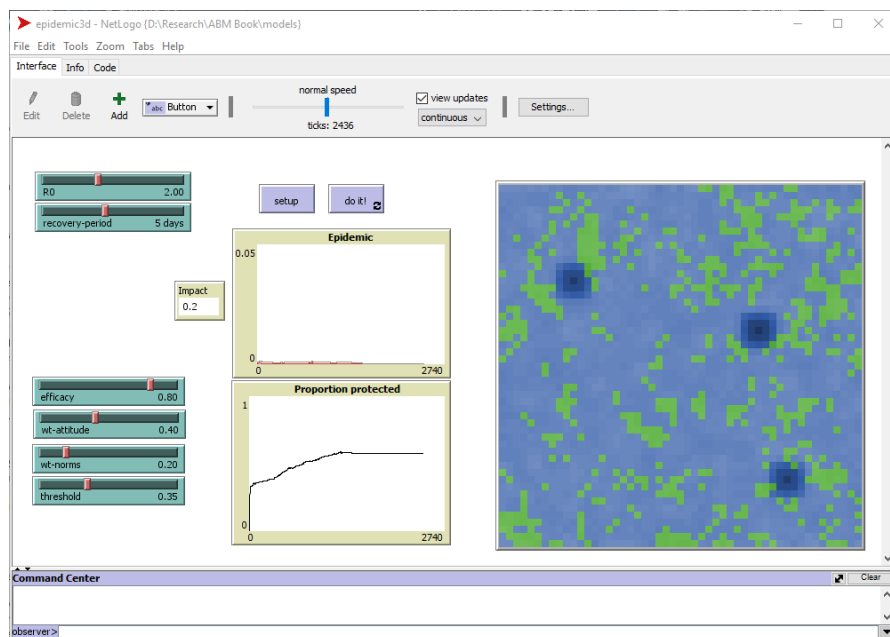


Figure 4.5: Screenshot of the interface after completing version 3d of the model. The effect of protected behaviour is visible, with much of the World coloured blue at the end of the epidemic, indicating that less than half the population were ever infected.

- is a computer program...: yes, it is written in the *NetLogo* language and runs on the computer;
- that represents individual entities...: yes, the turtle breed ‘people’ are individuals;
- taking actions...: yes, the simulated people adopt and abandon protective behaviour;
- in accordance with their own characteristics, resources, beliefs...: yes, the personal attribute of attitude is a component in their choice of action;
- and perception of their social and physical environment.: yes, the simulated people are able to perceive the proportion of people nearby that are behaving protectively and the

Snippet 3e: Some decisions are final

1. Create a switch for the variable ‘reversible?’
2. Amend decide-behaviour procedure. New and amended lines are marked by **««**

```

to decide-behaviour
...
ask people
[ let decision-value wt-attitude * attitude + wt-norms *
  prop-protect + (1 - wt-attitude - wt-norms) * max-threat
  ifelse reversible? ««
  [ set protect? ifelse-value (decision-value >= threshold)
    [TRUE] [FALSE] ] ««
  [ if not protect? and (decision-value >= threshold) [ set
    protect? TRUE ] ] ««
]
end

```

Logical operators

Logical operators manipulate truth values in the same way that mathematical operators (such as +) manipulate numbers. Common logical operators are: **not** which converts **true** to **false** and **false** to **true**; **and** which is **true** only if all input values are **true**; and **or** which is **true** if any input value is **true**.

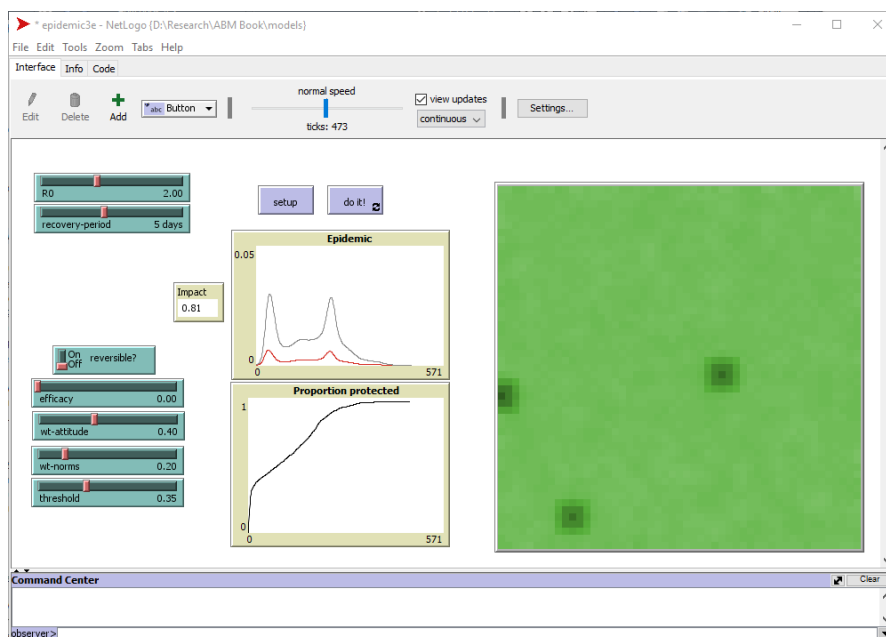


Figure 4.6: Screenshot of the interface after completing version 3e of the model. The effect of irreversible behaviour is displayed in the proportion protected plot, which increases throughout the epidemic and does not decline once the epidemic is over.

epidemic threat, both of which contribute to the choice of action.

Having now directly implemented these features of the definition in the tutorial model, the meaning of each aspect should be reasonably clear. Agent-based modelling is able to represent complex systems because of these features, as they allow interaction and its effects to be rep-

resented. The behaviour of the epidemic is driven by the interactions between the epidemic and protective behaviour processes. This was immediately apparent when the processes were connected in Snippet 3d (page 76) and the epidemic was throttled by effective protection.

Two important concepts were introduced in model 3: agentsets and context. Agentsets are fundamental to the *NetLogo* approach to coding an agent-based model. They allow concise identification and selection of relevant agents, both overtly or more subtly underpinning the flexibility of *NetLogo*. Agentsets can be combined or subsetted, stored in variables, and issued instructions.

Each time *NetLogo* accesses an agentset, the context changes and the perspective is taken of each individual agent in turn. The order in which *NetLogo* iterates through the agents is random, and is reordered each access.

When developing code, it is often easiest to imagine yourself as an agent (such as a turtle) and think about what information you would need and what you would do to perform the required task. It is also worthwhile to consider whether the same logic applies for all agents, particularly the ones taking the first turn, a middle turn, and the last turn. Some of the primitives only make sense for some agent types and are therefore only available in particular contexts.

In addition, two previously introduced concepts were extended: procedures and working with truth values. Procedures are used to modularise the code and either instruct agents to perform some task (command) or provide some information (reporter). The more advanced feature introduced is the use of arguments, inputs that the procedure can use for more flexibility. The tutorial model used this capacity to construct a procedure that provides random numbers from a triangular distribution with the limits and mode of that distribution passed as arguments. This potential for arguments emphasises the equivalence between user written procedures and the primitives built in to the *NetLogo* language. Primitives are simply (efficiently written) procedures with arguments.

Truth values (that is `true` and `false`) are included in models in different ways. They are used as status flags, such as the variable 'protect?'. They are also generated as the outcome of conditional tests. Such tests are overt in the construction of `ifelse` and similar constructions, but are also fundamental to agentsets constructed with the `with` filter. These truth values are their own type of value. That is `true` is not the same as the text string "true", nor is it stored as a number (such as 1) with a specific indicator on the variable type. Logical operators should be used to work with truth values, in the same way that mathematical operators are the most appropriate for numbers (or string operators for text).

4.6.1 Programming practices

Version 3 of the model is now at the point that complex behaviour is being simulated. Each step has been only a small addition that is tested before moving on. Such iterative construction and testing is important when programming in any language, but it is essential for *NetLogo*. This is because the normal methods available to debug programs in other languages, such as stepping through the code and examining variable values, are much harder to do in *NetLogo*. Furthermore, logical errors are very difficult to detect (in contrast to syntax errors, which immediately generate an error message).

Each version of the model is presented with less detailed instructions. At this point you should be comfortable with adding widgets to the interface, and automatically syntax checking (green tick) and saving your model. You should also be starting to recognise the syntax of common primitives such as `ask` (which must be followed by an agentset and a command block) and `set` (which must be followed by a variable name and the value to be assigned).

If you are not already doing so, you should try and develop the habit of thinking about algorithm design and testing for every additional section of code. Algorithm design is the name given to working out each step that must be programmed to instruct the computer how to implement the model. The tutorial presents the full design of the model and also some tests so you are sure the design is implemented as intended. But these instructions are not for you to blindly follow, you need to think about why you are doing what you are doing.

Another programming practice built into the tutorial is interface design. The instructions provide details of what widgets to include on the interface, and the screenshots display one option for placement. What to include and where to place it are part of the model design. The model must be accessible for users. Parameters that need to be adjusted should be available, but too many widgets adds to the clutter and makes it difficult to find the correct one. The world displays the current status of the model, but colours and other features can be used to emphasise or hide particular aspects of that status. Finally, plots and other output should be focused on the information needs of the user.

This does not mean, however, that the interface cannot include widgets for the convenience of the model developer, they should just be removed from the released version of the model. In fact, it can be very useful to have an area of the interface devoted to diagnostic information to make sure that changes do not introduce errors in already tested code. The attitude distribution plot and maximum population monitors are both examples of such diagnostic widgets.

4.6.2 *NetLogo* language

Most of the new primitives introduced in this chapter concern the construction of agentsets. New *NetLogo* keywords explained in this model version:

- agentsets: `patches`, `turtles`, `patch-set`, `turtles-on`, `turtles-here`, `with`
- context: `self`, `myself`
- mathematics: `random-float`
- logic: `not`, `and`

Model 4: Representing Relationships

While version 3 of the model is a complete agent-based model, it does not fully implement the original design. In particular, section 1.8 states that people communicate with their friends about threat. This element is to be added in version 4.

Friendships and other relationships between entities are represented mathematically with networks, comprised of nodes for entities and edges for relationships Newman (2010). Nodes can represent people or firms or countries or many other real world entities. Relationships are equally broad, such as friendship or shared board membership or international trade. In *NetLogo*, network nodes are turtle agents and links are used for edges.

Links are a new type of *NetLogo* model object (referred to as types of agents in the official documentation). The four objects are observer, turtle, patch and link. The observer is external to the model, patches are used to represent space, turtles to represent agents (as meant in this tutorial), and links to represent relationships between pairs of turtle agents.

Much of the apparatus that you have already been using for patches and turtles is easily adapted to links. Some primitives work in exactly the same way, including `ask` and `set`. Others have equivalents, for example `link-neighbors` creates the agentset of turtles at the other end of the links connected to the asking turtle(s), which is the link version of the spatial `neighbors`. Therefore, while there are many primitives introduced in this chapter, they are mostly related to previously seen primitives rather than new programming concepts.

Like turtles, different breeds can be defined for links. Each breed has variables, defined with a `links-own` statement (or its breed equivalent). These variables store attributes of the relationship, not attributes of the turtles that are related. For example, if the relationship represented by the link is friendship, then an appropriate variable would be the number of times per month that the two simulated people contact each other.

5.1 Model Design: Influential Friends

In order to communicate with friends, people must have friends. Friendship is a relationship between people, not an attribute of a particular person or a feature of the environment, and will be modelled with links. While not particularly realistic, the model will randomly allocate the same number of friends for each person. Furthermore, the friendships will be undirected, which means that if person A considers person B a friend, then person B also considers person A a friend.

The other design decision is how to interpret communication about threat. Currently, threat is one of the three inputs to the protective behaviour decision. That threat is based solely on the local epidemic incidence and is the same for all people on a patch (and consequently calculated by patches). Instead, threat is to be the maximum of the incidence based threat across all a person's friends. That is, if a person is in a location where the epidemic is active and therefore

perceives a high threat level, that heightened threat is communicated to all their friends and the friends adjust their perceived threat accordingly.

5.2 Creating a Social Network

Creating a network with a fixed number of edges per node is not straightforward. If you simply ask every node to create (say) two edges with randomly selected other nodes, then some nodes may be selected more often than others. Instead, each node must be aware of the target number of edges and only accept selection if they have not yet reached that target. This algorithm is implemented in Snippet 4a (page 82).

Snippet 4a: Create a social network

1. Add slider for ‘num-friends’ (0 to 5 by 1, value 2)
2. Create make-network procedure
3. Add call to make-network procedure in setup procedure (think about where)

```
to make-network
  ask people
  [ let needed num-friends - count my-links
    if needed > 0
    [ let candidates other people with [ count my-links <
      num-friends]
      create-links-with n-of min (list needed count candidates)
      candidates
      [ hide-link]
    ]
  ]
end
```

The primitive `my-links` is the agentset of links connected to the specific turtle. For the first person, the count of this agentset is 0, so the local variable ‘needed’ is set to the value of the num-friends slider. For later people, some links may already have been created and needed is accordingly lower. If any links need to be created for that person, the code block to create links is run.

my-links

The agentset of links connected to a specific turtle is generated with the `my-links` primitive. See also `link-neighbors`, which is the agentset of turtles at the other end of those links.

That code block selects all the people who still have capacity for further links, and excludes the person looking for friends (with the `other` primitive), assigning that agentset of people to the variable named candidates. But what if there’s not enough people available to connect to? *NetLogo* will report an error if it tries to select too many agents (for example, if you are asking for 5 random turtles from a set with only 3). So the value for the `n-of` is the minimum of the number to be selected and the number available.

other

The primitive `other` excludes the model entity (turtle, patch, link) from the agentset being selected, even though it satisfies the conditions for inclusion.

But the `min` primitive takes the minimum of a list. The tutorial has previously used `max` to find the maximum popn value. In that case, a list was automatically created by the `of` reporter requesting the values of a variable from all patches. Here, however, the list must be explicitly created with the values that are to be its items. For example, `(list 1 2 3)` would create a list of three items, the numbers 1, 2 and 3.

The smaller of the number of links needed and the number of available turtles to link to is passed to the `n-of`. That number of people are randomly selected from the agentset stored by the variable `candidates` to generate an unnamed agentset.

The primitive `create-links-with` instructs the person to create an undirected link with each of the people in the unnamed agentset. Like the turtle creation commands discussed in section 2.8, the following code block is run for each link as it is created. In this snippet, that code block hides the link so it is not shown in the world (or available in the popup menu).

create-links-with

The full syntax is `create-links-with <agentset> []`. Instructs a turtle to create an undirected link with every turtle in the specified agentset. Any code within the square brackets is run for each link as it is created. See also `create-links-from` and `create-links-to` for directed link versions. Similar primitives are also available that specify the breed of the links to be created.

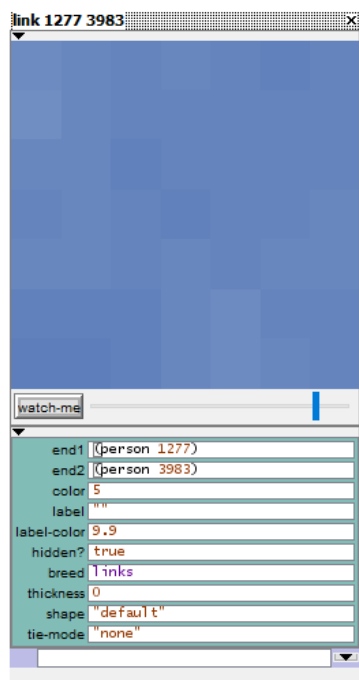


Figure 5.1: Inspect window for link.

Since the links are hidden, they are not available for the popup menu. Instead, type `inspect one-of links` into the Command Center to open the inspect window for a random link. You can see the built in variables for a link. A specific link is identified by the `who` numbers of the two turtles it joins (top left corner of inspect window).

No screenshot is presented as it is identical to Figure 4.6. The most obvious effect is that initialisation (the code run with the `setup` button) takes much longer to run. This is because each link being created requires *NetLogo* to construct an agentset by checking every person to check its current number of friends. You may also see the world apparently flickering as the new links briefly appear before being hidden.

Knowing that the code creates links is not, however, a check that it creates the correct links.. With a little thought, however, you may realise that setting the slider for `num-friends` to 2 gives you access to a simple test. If each link connects two people and each person has two friends, then the number of people and the number of links should be identical. You can test this by (after running `setup`), typing 'count people' into the Command Center and then 'count links'. It may be that the number of links is slightly lower; if the last person asked does not already have any friends, then the only candidate is itself and links cannot be made from a turtle to the same turtle.

Finding the person agentset and the linkset to be the same size is not a conclusive test because links created at random will still give rise to the correct number of links. What you want to check is that all people have exactly two friends (or equivalently, exactly two links), with the possible exception of the failed self link attempts. One possibility is a histogram of `count [my-links] of people`. But it is easier to simply find the maximum of the number of links. If the maximum is 2, then all people must have two friends. The appropriate code is `max [count my-links] of people`.

hide-link

The command `hide-link` makes a link invisible in the world view. It is reversed with `show-link`. See also `hide-turtle` and `show-turtle`.

5.3 Using Links

Using the links to obtain information from friends introduces a new *NetLogo* agentset, that of `link-neighbors`. This is the links equivalent of `neighbors`, generating an agentset of turtles that are socially connected to the instructing turtle, instead of patches that are spatially next to the instructing patch. The tutorial has previously used the reporter `of` to access information from agentsets (see Snippet 2e-1 (page 56)). Snippet 4b (page 84) uses the same primitive to access information about threat values where friends are located. No screenshot is displayed as the changes have no visible effect.

Snippet 4b: Communicating along links

1. Amend decide-behaviour procedure. New and amended lines are marked by `««`

```
to decide-behaviour
  ...
  ask people
  [ let friend-threat ifelse-value any? link-neighbors ««
    [ max [threat] of link-neighbors ] ««
    [ 0 ] ««
    let mythreat max (list max-threat friend-threat) ««
    let decision-value wt-attitude * attitude + wt-norms *
    prop-protect + (1 - wt-attitude - wt-norms) * mythreat ««
    ifelse reversible?
    ...
  ]
end
```

link-neighbors

`link-neighbors` is the agentset of turtles at the other end of the links connected to the specified turtle. See also `my-links`, which is the agentset of links representing those connections.

The user may set number of friends to 0. *NetLogo* will fail if it is instructed to run code on an empty agentset. Therefore the calculation of the maximum of the friends' threat values is preceded by a test as to whether there are friends to calculate over. If not, the friends' threat value is set to 0. This protection also deals with the case where a non-zero number of friends is selected but there are no potential friends available for the final person to have edges created.

**Error protection on empty agentsets**

If it is possible that an agentset is empty, it is good practice to precede any `ask` on that agentset with a test using `if any?`.

This snippet also uses the explicit `list` constructor. Applying `of` to an agentset creates a list of the variable values so that the maximum of the threat values of friends can be calculated.

any?

any? is a logical operator that reports **true** if the specified agentset has at least one member, and **false** if is empty.

The explicit **list** is used to create a list with two members, that maximum and the previously calculated maximum threat from the directly observed patches. Once these two values have been placed in a list, the **max** operator can choose the larger to be the relevant perceived threat. The snippet requires only minor amendments to the previous code because the revised maximum is inserted into the behaviour decision by simply replacing the maximum from only the observed patches.

list

The full syntax is (**list** <item0> <item1> <item2> ...) to create a list with the items specified as its contents, in the order specified. List member items can be numbers, text strings, other lists, agentsets, or some combination of these data types.

There is no obvious way to test this code as the threat combining the sources is stored only as a local variable and therefore cannot be queried while the model is stopped. However, there are primitives that print information to the Command Center and these can be used to report relevant values at specific points in the code. Such primitives are commonly used to construct a basic test.

Save the model so that you can revert to the saved version after testing. Make the changes at Figure 5.2 but do not save the model. This code modifies the decide-behaviour procedure to provide information about variable contents. The primitives **type** and **print** are used to report the (labelled) threat inputs and then the threat value used. There are also equivalent commands to send information to an output widget on the interface or to a file. Note that the **ask people** has been modified to **ask n-of 10 people** to limit the amount of information sent each tick.

Press the *setup* button and type *go* into the Command Center (to run the go procedure once only). Remember (from section 3.3.1) that you can expand and collapse the reporting area with the double ended arrow in the top right corner of the Command Center. As only 10 people are being queried, you may need to enter *go* more than once to find a person whose friends have higher threat values and therefore see a change in the person's threat value. Once you have checked the code, close the model without saving (assuming you haven't saved these changes) or reverse the edits.

print, type

print and **type** both instruct *NetLogo* to provide information to the Command Centre. The difference is that **type** sends only the information asked, while **print** sends the information and ends the line. See also **show** and **write**, which also send information to the Command Centre but additionally includes the identifier for the model entity.

5.3.1 Measure the effect of extended threat perception

The final step in this version is to report the effect of friends on behaviour. The effect of each friend is to potentially increase perceived threat and therefore trigger adoption of protective behaviour earlier or delay cancellation. Therefore, a higher number of friends could be expected to increase protective behaviour overall.

Potential ways to measure this include the average proportion protected over the duration of the run or the maximum adoption level achieved. Both of these summarise the whole of the simulation, not simply measure the current status. Snippet 4c (page 86) implements tracking

```

to decide-behaviour
  ask patches
  [ ...
  ]
  ask n-of 10 people
  [ ...
    [ 0 ]
    type "patch threat is: " print max-threat
    type "friend threat is: " print friend-threat
    let mythreat max (list max-threat friend-threat)
    type "threat is: " print mythreat
    let decision-value...
  ]
end

```

Figure 5.2: Modifications to use output printed to the Command Center to confirm code is implemented correctly.

the maximum proportion protected achieved during the run and when that maximum is (first achieved). Each time step it calculates the current proportion of people with protective behaviour. If that value is higher than the stored maximum, that maximum is replaced and the current time is also stored.

Note that there is a complication arising from the effect of that protective behaviour; with the epidemic restricted, threat is lower, confounding the effect of the friends. To avoid this, any simulations to test the hypothesis should be run with efficacy set to zero.

Snippet 4c: Measure protection

1. Create two global variables: max-protect and when-max-protect.
2. Initialise both new variables to 0.
3. Create the update-globals procedure.
4. Add call to update-globals procedure at end of go procedure.
5. Add monitors to the interface for each new variable.

```

to update-globals
  let protected count people with [protect?] / count people
  if protected > max-protect
    [ set max-protect protected
      set when-max-protect ticks
    ]
  ]
end

```

This snippet demonstrates the use of `ticks` to obtain the current time during the simulation (as compared to `tick` to advance the tick counter). Each tick, the current value of protected proportion is compared to the stored maximum already achieved. If the new value is higher, both the maximum and the time at which that maximum is achieved are updated.

ticks

`ticks` queries the tick counter and reports the current time.

Once implemented, this snippet completes version 4 of the model, displayed at Figure 5.3. No

explicit test is required for this code, because it is easy to check whether the new monitors are correct by comparing their reported values with the plot of proportion protected. Note that hovering over the plot (placing the mouse pointer near the line) will reveal the plotted values.

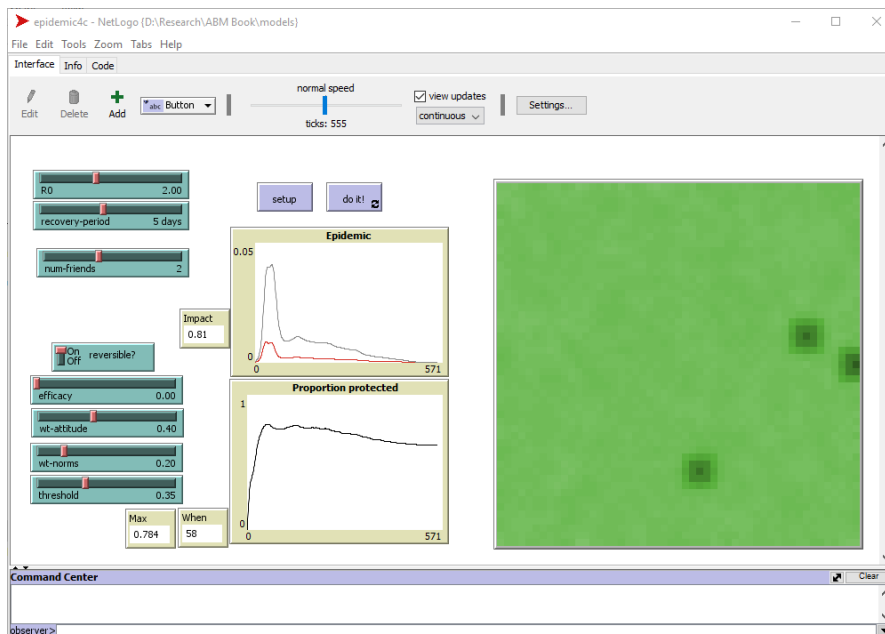


Figure 5.3: Screenshot of the interface after completing version 4c of the model. Note that the values in the two monitors concerning protective behaviour coincide with the peak in the protective behaviour plot.

5.4 Tutorial Progress Check

Version 4 of the model is now complete. The purpose of this version was to introduce links, the final *NetLogo* model entity type. Links can have breeds and attributes, though neither are included in the tutorial. All of the relevant primitives you have already seen in previous versions for patches and turtles work in exactly the same way for links, such as `ask` and `set`. Other turtle or patch primitives have equivalents for links with appropriately modified names, such as `link-neighbors` and `create-links-with`. Finally, there are primitives such as `my-links` that have no clear equivalent for other entities.

Programming practices continue to be reinforced. These include detailed design before coding, modularisation of code, minimum feasible change, syntax and logic testing for each change. A new way to test was introduced, printing variable values to describe changes that occur within a time step.

5.4.1 *NetLogo* language

Most of the new primitives introduced in this chapter concern the construction and application of links. New *NetLogo* keywords explained in this model version:

- links: `create-links-with`, `my-links`, `link-neighbors`, `hide-link`
- agentsets: `other`, `any?`
- other: `list`, `print`, `type`, `ticks`

Models 5-8: Enhancements

The core *NetLogo* concepts and language have been presented in model versions 1 to 4. Version 1 established the model entities to represent agents and space: turtles and patches. Version 2 introduced time and space, and implemented the process of epidemic spread in the patches. Version 3 implemented the process of adopting and dropping protective behaviour, relying extensively on the key concept of agentsets. Finally, version 4 introduced links to represent relationships.

There are many other *NetLogo* primitives that will not be covered. The majority of these are specialised and used very rarely. However, there are two topics that are essential for many models. Any model that includes mobile agents will require primitives that turn, move, look ahead and similar actions (see Chapter 8). Lists have been introduced, but their true power is unexplored. They are required where sequence is important, such as remembering only the most recent interactions. There are also specialised sets of primitives that are not part of the core language but are instead added with extensions, including primitives to deal with geographic information or networks. Lists and extensions are not covered in this tutorial.

The rest of the main tutorial enhances the existing model. Each enhancement is independent. They are intended to stimulate ideas for what can be done with models rather than demonstrate features of the language. While new primitives are used, they are not formally discussed. Instead, you should look them up in the *NetLogo* dictionary.

6.1 Model 5: Efficiency

The first enhancement highlights the importance of good design. Creating the links takes considerable time. Snippet 5 (page 90) establishes edges with a different algorithm. In Snippet 4a (page 82), each person separately creates the agentset of potential friends as the code iterates through `ask people []`. The new algorithm constructs the agentset of potential friends only once, and removes people as their friend spots fill up.

The agentset of potential friends is called ‘lonely’. It starts with all people as members. Each person first removes itself from the agentset of potential friends. It then selects the required number of friends from that agentset and creates links with them. If any of the linked people have now met the quota of friends, that person is also removed from ‘lonely’.

Model version 5 should initialise noticeably quicker than version 4. Efficiency is particularly important for large models with many agents, where even a small difference in time taken for one agent scales up to considerable time for many agents and time steps. There are tools to identify which procedures are consuming the most computer time (see the profiler extension).

As well as the benefits of good algorithm design, this snippet demonstrates the scope of local variables. The agentset named lonely is assigned at the start of the procedure and is then available to all the code at deeper levels in the `ask people` code block within the procedure.

Snippet 5: Faster network creation

1. Rename make-network procedure to make-network-original.
2. Create the new make-network procedure.

```

to make-network
  let lonely people
  ask people
  [ set lonely other lonely
    let needed round num-friends - count my-links
    if needed > 0
    [ let targets n-of min (list needed count lonely) lonely
      create-links-with targets [ hide-link]
      ask targets [ if count my-links = num-friends [ set lonely
        other lonely ] ]
    ]
  ]
end

```

6.2 Model 6: Visualising Decisions

Model version 6 enhances the visualisation to display those patches where some specified proportion of people have adopted protective behaviour, providing a barrier to the epidemic spread. This is implemented with an additional breed of turtle agents that provide patch markers. The marker breed is defined in Snippet 6a (page 90), and two procedures are added to create the markers (Snippet 6b (page 91)) and control their display (Snippet 6c (page 91)). New *NetLogo* keywords are the primitive `show-turtle` and the built in turtle variables `shape` and `color`.

Snippet 6a: Define a breed for markers

1. Define a new breed for the marker called barrier.

```

; markers for protected patch
breed [barriers barrier]

```

With these snippets implemented, a small black X will be displayed in those patches where the protected proportion of the people is at least the level set by the slider. As many patches have only 2 or 3 people, a useful level is 0.75 so that a single high attitude person does not trigger the visualisation. You should expect to see a higher density of marked patches near the epidemic frontier, where threat is highest, particularly if the number of friends is set to 0 so that only local threat is perceived.

6.3 Model 7: User-Controlled Perception

Model version 7 allows the user to control the distance over which the spatial environment influences protective behaviour decisions. Currently, the model constructs an agentset named `visible` comprising the patch issuing the instruction and its eight neighbouring patches. That patchset is used for the calculations of the social norm (proportion of people protected) and threat. In version 7, the agentset `visible` is a variable owned by patches, created during setup to comprise the patches within a specified distance.

The only new primitive is `in-radius`. The purpose of storing the visible patchset for each patch

Snippet 6b: Create the markers

1. Create the make-barrier procedure.
2. Amend the setup procedure to call the make-barrier procedure.

```

to make-barriers
  ask patches
  [ sprout-barriers 1
    [ set shape "x"
      set size 0.7
      set color black
      hide-turtle
    ]
  ]
end

```

Snippet 6c: Visualise the markers

1. Create a slider called 'show-protect' (0 to 1 by 0.01, value 0.75) for the proportion of people protected at which the marker will be visible.
2. Create the colour-barriers procedure.
3. Amend the go procedure to call the colour-barriers procedure.

```

to colour-barriers
  ask barriers
  [ if-else count people-here with [protect?] / count people-here
    >= show-protect
    [show-turtle]
    [hide-turtle]
  ]
end

```

is for efficiency. The patchsets never change so can be created once during setup and stored, which is quicker than the previous approach of creating it each time step. This improvement is particularly important when calculating the patches in range with `in-radius`, a relatively slow operation.

6.4 Model 8: Infecting individuals

In this model, the people exist in a world where an epidemic is occurring, but they do not become infected themselves. The epidemic is transmitted through the spatial environment rather than from person to person. There are pedagogical benefits of this representation; for example, it facilitates the separate consideration of time, space and agent characteristics.

This representation was not chosen because of those benefits, however. Idealising transmission as a local environmental process is reasonable due to the low level of interaction required for transmission, even though influenza and similar airborne diseases are spread from person to person. This contrasts with blood born diseases such as hepatitis, where the network of contacts is critical to transmission.

Nevertheless, it may be appropriate to also model the epidemic status of the people agents. For example, you may wish to report the proportions protected by epidemic status, or increase

Snippet 7: User controlled perception

1. Add a slider called see-distance (0 to 5 by 1, value 2).
2. Add a patch-own variable called visible.
3. Amend the setup-patches procedure to initialise visible (shown).
4. Remove the line that generates the visible patchset from the decide-behaviour procedure.

```

to setup-patches
  ...
  ask patches
  [ ...
    set visible patches in-radius see-distance
  ]
  ...
end

```

perceived threat if a friend becomes infected. This does not require changing the transmission representation. Instead, the local incidence can be used to trigger simulated people becoming infected.

The basic approach is to allow susceptible people on a patch to become infected with probability given by the incidence on that patch (as a proportion of the susceptible population). However, people who have adopted protective behaviour should derive benefit from that behaviour, so the probabilities must be adjusted to average at the appropriate proportion but also recognise the efficacy of protective behaviour. Once a person has become infected, the probability to recover is the same as used for the patch based epidemic.

If E is the efficacy of the protective behaviour, P_r the protected proportion of people, and q the susceptible fraction becoming infected, the probability of a susceptible person becoming infected is given by:

$$P = \begin{cases} q & P_r = 0, P_r = 1 \\ q \frac{1}{1 - P_r E} & \text{unprotected} \\ q \frac{1 - E}{1 - P_r E} & \text{protected} \end{cases}$$

As the infection probability depends on the proportion of susceptible becoming infected in that time step, the new procedures to infect people must be called after the spread-epidemic procedure. Snippet 8 (page 93) implements infection and recovery, as well as monitors to report the proportion infected by protective behaviour status.

The only new primitive introduced is `all?`. The equation to be implemented is expressed with one of the cases arising for proportion protected of 1. However, it is bad practice to test whether two calculated numbers are equal as they may differ slightly at the limits of precision. The snippet instead checks whether all people on the patch are protected directly instead of checking whether the proportion protected equals 1.

This snippet also demonstrates turtles accessing patch variables. The susceptibles agentset has turtle members, so the `ask` shifts *NetLogo* into the turtle context. Nevertheless, patch variables such as ‘incidence’ are used directly, there is no need for a construction such as `[incidence] of patch-here`. This capacity for turtles to access the variables owned by the patch where they are located is built into the architecture of *NetLogo*.

Snippet 8: Infecting individuals

1. Add a variable for people called state.
2. Initialise the state variable to be "S" when people are created.
3. Create the epidemic-people procedure.
4. Amend the go procedure to call the epidemic-people procedure.
5. Create monitor for the proportion of protected people who are infected.
6. Create monitor for the proportion of unprotected people who are infected.

```

to epidemic-people
  ; infected to recovered
  let infecteds people with [state = "I"]
  if any? infecteds
  [ ask infecteds
    [ if random-float 1 < (1 / recovery-period) [ set state "R" ]
    ]
  ]
  ; susceptible to infected
  let susceptibles people with [state = "S"]
  if any? susceptibles
  [ ask susceptibles
    [ let protect-here count people-here with [protect?] / count
      people-here
      let infect-here incidence / (incidence + popn-S)
      ifelse all? people-here [protect?]
      [ if random-float 1 < infect-here [ set state "I" ] ]
      [ let prob ifelse-value protect?
        [ infect-here * (1 - efficacy) / (1 - protect-here *
        efficacy) ]
        [ infect-here / (1 - protect-here * efficacy) ]
        if random-float 1 < prob [ set state "I" ]
      ]
    ]
  ]
end

```

**Turtles know about their location**

Any turtle is able to access the values of the patch owned variables for the patch where the turtle is located. This access is automatic by simply using the variable name, in exactly the same way as if the turtle owned the variable.

The monitors demonstrate the counterintuitive results that arise in complex system behaviour. A naive approach would expect that the proportion of protected people becoming infected would be lower than the proportion of unprotected people becoming infected, potentially by a factor equivalent to the efficacy of the protective behaviour. With further reflection, however, it is apparent people near the epidemic front are more likely to have adopted protective behaviour and also have greater exposure to the epidemic. Therefore, it is possible that incidence is actually higher in the protected population, making it appear that the protective behaviour increases the risk of becoming infected. Agent-based models can be used to understand such counterintuitive results (Barbrook-Johnson et al., 2017).

The tutorial model is now complete. The final version is displayed at Figure 6.1. Note that the

widgets have been organised into related groups. The group headings are Note widgets.

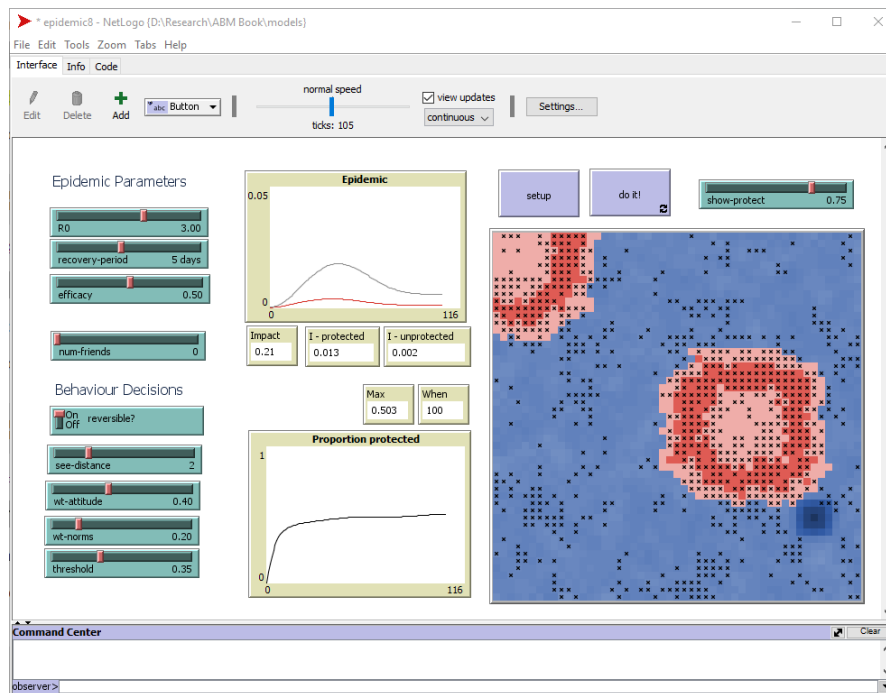


Figure 6.1: Screenshot of the interface after completing version 8 of the model. Markers indicate patches where at least 75% of the simulated people are protected.

6.5 Tutorial Progress Check

Model versions 5 to 8 have each introduced an enhancement to the model's features. Very few additional primitives have been used, the features are instead delivered by using previously discussed primitives in new ways. Model 5 substantially decreased the time to construct the network with better algorithm design. Model 6 added a turtle breed to improve the visualisation. Model 7 extended the range of perception through some minor modifications to the a patchset. Finally, model 8 allowed the people to become infected by accessing the patch based epidemic.

Throughout the tutorial, the instructions have been gradually becoming less detailed. At this point, you should be comfortable with a range of good programming practices such as adding code gradually, checking syntax, interpreting and resolving some syntax errors, initialising variables, looking up the *NetLogo* dictionary for useful primitives, saving your model regularly and running logic tests. You should also understand what many important *NetLogo* primitives actually do, and familiar with common tasks such as adding widgets and letting *NetLogo* know what variables to create.

6.5.1 *NetLogo* language

New *NetLogo* keywords explained in the model versions in this chapter are:

- turtles: `show-turtle`, `shape`, `color`
- agentsets: `in-radius`, `all?`

Bringing it all Together

The main tutorial is now complete. This chapter summarises some of the key ideas presented and points to further materials to improve your understanding of agent-based modelling and *NetLogo*. It also brings together related ideas that are spread throughout the tutorial. The structure of the tutorial, gradually building a model from start to finish, means that concepts are introduced in the order that they are needed. This chapter, however, is structured thematically, linking together pieces of information that may have been presented far apart.

There are three key skills needed to build an agent-based model. You need to be able to design the model, to understand and describe the entities and behaviour to be represented. You need to use good programming practices to implement the model, so that the model runs quickly and accurately implements the design. Finally, you need to be familiar with the programming language, in this case *NetLogo*, to translate the intent into instructions. The first two of these skills are required even if you are not personally writing the code, they are necessary to facilitate communication with the programmer.

7.1 Model Design

An agent-based model is a simulation, representing one or more processes with a computer program. That model takes an agent-centric perspective, representing individual entities taking actions in accordance with their own characteristics and taking into account their own situation.

An agent-centric perspective is critical where there is both heterogeneity and interaction. If the individuals are different in ways that impact on their behaviour, modelling individuals allows that variation to be represented. This contrasts with methods that model the behaviour of an ‘average’ individual and therefore treat them all as identical. Individuals may interact with each other, choosing their action based on their social environment. Or they may interact with the physical environment, responding to local conditions such as resource availability and altering the environment by their actions. This enables different agents to behave differently in the same situation and the same agent to act differently in different situations.

This approach is particularly beneficial when modelling complex system behaviour that is driven by the interactions between entities. An agent-based model is able to generate that behaviour from the actions of the simulated individuals rather than attempting to develop rules that represent directly the overall behaviour. That is, the behaviour the rules of the agent-based model replicate the theoretical understanding of the behaviour that occurs in the real world system.

While there is no standard set of questions that will lead to a good model design, there are key features that should be considered. They relate to the representation; identifying the processes, agents, environmental features, and the characteristics and rules that govern interactions. It is important to simplify as much as possible to assist in design and interpretation, but features that are important in the system behaviour must be retained for the model to be useful.

Some potential questions to stimulate a model design are listed at Table 7.1, together with how they apply to the tutorial model. Many other considerations may also be relevant, but these should be sufficient to start any consideration or discussion.

Table 7.1: Broad design questions for developing the model.

Aspect	Question	Tutorial model
Process	What actions or repeated decisions?	Adopt/drop protective behaviour.
	What state changes without action?	Epidemic transmission. Disease progression.
	What does theory say about these processes?	Behaviour change (Theory of Planned Behaviour, Health Benefits Model, Social Cognitive Theory) and SIR epidemic theory.
Agents	What entities take actions?	People.
	What personal characteristics affect action?	Attitude.
	Any other entities need to be included?	No.
	What characteristics of those entities are relevant?	Not applicable.
	What characteristics change over time automatically?	None.
Spatial	Does location change the way in which the process operates?	Yes.
	What location specific resources (including information) affect action?	Population, viral load.
	How do resources change over time automatically?	SIR equations.
Social	Do other entities change the actions of individuals?	Yes.
	How are agents aware of each other?	Friendship and norms.
AA interaction	What is the interaction between agents?	Pass information about risks, observe norms.
AE interaction	How do the agents affect the environment?	Protective behaviour reduces infections.
EA interaction	How does the spatial environment influence actions?	Infections (viral load) perceived as threat.
EE interaction	How do the characteristics of a location impact on other locations?	Epidemic transmits spatially.

This list of questions identifies the key features of the model but significant effort is needed to create a full model design. In the tutorial, the design and operationalisation was presented as the model was constructed, together with the *NetLogo* implementation of that operationalisation.

In a real project, a relatively detailed design should be specified before any coding. It is not sufficient to specify what is to be included in the model, but also how it is to be included. For example, follow up questions could include:

- for agent attributes and spatial resources, what information is available about how the attributes are distributed?
- for interactions, what is the specific rule or equation that governs the influence and what information and parameters are required to determine the outcome of that rule?

These questions focus on the process that is to be represented by the model. One important element in model design is distinguishing between the intialisation and the actual simulation. A model starts at a specific point in time, so the initialisation represents a static view of the world

(as if time had been paused). Typical tasks in the initialisation include creating the agents and allocating appropriate variable values. These are conceptually different than the simulation of the process as time progresses.

The model design is agnostic with respect to the particular software in which the model is to be implemented. That is, it should be expressed with an agent-centric representation and relevant rules, but not with *NetLogo* keywords. One way to approach this is to imagine yourself as the agent (each type separately) taking action and think about the choices facing you and the information you require to decide an action. This consideration should be based on theory and data, including narratives and other qualitative information. The goal of the thought experiment is to encourage the agent-centric perspective required for the model specification instead of the perspective of an observer.

Other aspects of the design concern the interface. The input widgets should allow the user to control the primary features of different scenarios related to the research question. Outputs provide the user with information that is relevant to the research question, but also communicate what is occurring within the simulation.

Finally, the design should also consider broader aspects of how the model is to be used. This includes the resolution and accuracy required to answer the research question, how to check whether the model meets the accuracy requirements, and what experiments are to be conducted. These issues are outside the scope of the tutorial, but are covered in other chapters.

7.2 Good practice in programming

The key good practices in programming agent-based models have been repeatedly emphasised throughout the tutorial; incremental development combined with continual testing. The syntax check only detects problems with the way that the *NetLogo* code has been constructed, it does not detect errors in translating the model design into code. Such errors can be very difficult to detect in an agent-based model because it is not obvious whether unusual behaviour is due to an error or the complexity of the represented behaviour. The smaller the change, the easier it is to detect these subtle errors.

Each new piece of code should be the smallest change that can be made with which the model will run. It may also be appropriate to program a simpler version of the intended behaviour, and test that, before amending the code to implement more complex behaviour. For example, if you need to have only certain agents take some action, this can be separated into selecting the appropriate agents and then adding the action. Some simpler action (like changing size or colour) could be used to ensure the selection is correct.

The tutorial used many different ways to check whether implemented code worked as expected. Critical to all these methods is to know what to expect, in advance so that your thinking is not affected by the outcome of the test. Test methods demonstrated were:

- Monitors and plots of key variable values and their distribution;
- Colours and other visualisation to highlight key behaviour;
- Inspect windows to verify variable values and changes to those values each tick;
- `print` and related statements to verify variable values within a procedure;
- Running specific procedures from the Command Center to check equations are properly implemented.

A real model requires far more extensive testing than demonstrated for the tutorial. The tutorial sanity checks are testing only ‘face validity’, does the code appear to work as intended?

Another programming practice demonstrated throughout the tutorial is code readability. Factors that contribute to readability include: spacing and indenting, meaningful variable and procedure names, comments (including section headings), and modularisation to keep each procedure relatively short.

Modularisation is common to many programming languages. Each block of code has a specific task, and can call other blocks of code in a nested structure. In *NetLogo*, the blocks of code are referred to as procedures, and are delimited with `to` and `end` or with `to-report` and `end`. By convention, the main procedures are given the names ‘setup’ and ‘go’. These are (usually) triggered by a button, and the code section of the button settings simply calls the relevant main procedure. Each procedure is comprised of calls to other procedures or primitives from the *NetLogo* language. There is little conceptual difference between a procedure and a primitive; they are simply names for a set of computer instructions, but some are written by the *NetLogo* developers and built into the language (primitives) while the others are written by the model developer (procedures). Both procedures and primitives are able to use arguments as inputs. Those arguments form the syntax that *NetLogo* expects when the procedure or primitive is called in the model code.

Finally, good algorithm design is important in making the model efficient, so that it runs as quickly as possible. This becomes critical as the number of agents increases. Using the model will involve multiple simulations, and any delay in a single run has a substantial impact on the model’s long term usefulness.

7.3 *NetLogo* language

The third thread of the tutorial is *NetLogo*, which is both a programming language and a modelling environment. The *Code* tab holds the keywords that define the model (such as `breed` and `turtles-own`) and the procedures comprised of commands and reporters that instruct the model’s operations. This code tab is automatically linked to the *Interface* tab that supports widgets for model input and output and other model infrastructure such as spatial coordinates.

The tutorial uses only a fraction of the *NetLogo* commands and reporters. However, most of the primitives not covered are for specialised tasks and may never be required. The major exception is those dealing with agent movement (see Chapter 8), which is a common feature of agent-based models. Another important topic is lists, which are essential when the order of agents or values matters. For example, if a turtle needs to remember just the most recent five interactions with other turtles, those interactions must be stored in list form so that the oldest can be identified and deleted to make room for the most recent.

The relatively small set of simple commands can be used to generate rich behaviour. The core functionality applies `ask` to an agentset defined with conditions; this is the structure that implements heterogeneity of characteristics or circumstances with an agent-centric perspective. Conditional application applies to selecting agents to act using `with`, and to choosing the actions of selected agents using `if` (see Figure 7.1).

Agentsets are comprised of one type of model entity. The World is divided into `patches`, a regular grid of squares that each cover a unit square of the coordinate system. The entities representing individuals are `turtles`, and `links` represent relationships between a pair of `turtles` (see Figure 7.2). Both turtles and links have `breed` statements to create different varieties. The variables available to each entity are declared in a `patches-own`, `turtles-own` or `links-own` statement, with different breeds able to declare different sets of variables. These variables are created with the entity to which they belong.

One of the strengths of *NetLogo* is that any command that makes sense for multiple entity types is identical for all entity types. For example, `ask` iterates through an agentset of patches, turtles or links, without any need to specify the type. *NetLogo* also intelligently parses code and determines the entity type to which it applies or expected context.

Entity owned variables are one of three types of variables in *NetLogo*. Variables are simply names to store and retrieve values. The values that can be stored by variables include text strings, numbers, agentsets, boolean values (`true` and `false`) and lists. Global variables are available to all model entities (including the observer) and exist for as long as the model is running. These are created either with input widgets on the *Interface* tab or by being named in the `globals` list. Agent variables store information about the specific patch, turtle or link to which it belongs.

```

patches-own [test?]

to testme
  ask patches [set test? one-of [true false]]
  ask patches
  [ ifelse test?
    [ set pcolor red ]
    [ set pcolor blue ]
  ]
end

patches-own [test?]

to testme
  ask patches [set test? one-of [true false]]
  ask patches with [test?] [ set pcolor red ]
  ask patches with [not test?] [ set pcolor blue ]
end

```

Figure 7.1: Each of these code snippets are complete models that set the colour of patches to red or blue based on a randomly allocated patch variable. These demonstrate the two approaches to conditional code with agentsets, testing each agent (*ifelse* within *ask*, upper) and selecting relevant agents in the agentset construction (*with*, lower).

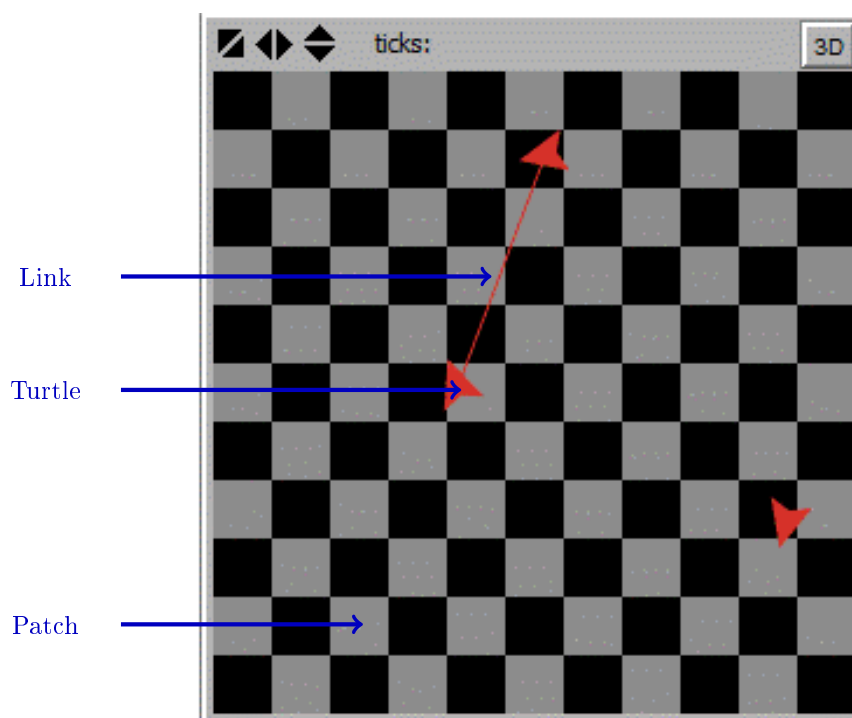


Figure 7.2: The three types of model entities available in NetLogo; patches for space, turtles for individuals, and links for relationships between individuals.

There are multiple variables with the same name, each assigned to a particular model entity. An entity is able to access its own variables automatically, and turtles can access the variables belonging to the patch where it is located. Accessing a different entity's variables requires the

`of` reporter, to specify which entity's variable is to be accessed. Finally, local variables are temporary storage that exist only within the code block in which they are created. They are used, for example, to store the results of a short piece of code to make the model more efficient by avoiding repeating that code or to make the model code more readable to humans. Local variables are created with a `let` statement and the values of all types of variables are updated with a `set` statement.

Interaction is an essential element of agent-based modelling, so any modelling platform must allow agents to perceive characteristics of other agents. In turn, this requires a concept of 'me' and 'them', as well as the capacity to access the variable values of other agents. In *NetLogo*, `self` refers to the agent currently in focus, the one to which the instruction is being applied. In contrast, `myself` refers to the agent issuing the instruction. The other key primitive in bridging the agent perception gap is `of`, which is used to specify the information that is known to some agent (or agentset) and is to be perceived by the instructing agent.

One of the reasons that *NetLogo* requires only a small language to implement agent-based models is that many required features are built into the modelling environment, rather than requiring explicit coding. These automatic features include time, space, and the mutual awareness of the code and the interface widgets.

Time is managed by an internal time step counter. That counter is initialised with `reset-ticks` (typically the final command in the *setup* procedure) and advanced with `tick` (typically the final command in the *go* procedure). The tick counter can be observed by any model entity with the `ticks` reporter.

Space is dealt with extensively in Chapter 8, as the coordinate system and related concepts such as heading are more salient in models where turtles move. The key spatial concept in the tutorial is `neighbors`, the agentset of patches surrounding the patch or turtle in focus. The `in-radius` primitive extends this spatial neighbourhood further out.

A complete list of *NetLogo* keywords used in the book appears as a separate index. The following list is the most important keywords that are used in the tutorial, which should become very familiar for any *NetLogo* modeller:

- procedures: `to to-report end`
- creating entities: `breed sprout`
- variables: `globals patches-own turtles-own let set`
- agentsets: `ask n-of max-n-of with`
- time and space: `tick neighbors`
- interaction: `myself of`
- mathematical and logical operators

7.4 Where to from here?

Congratulations, that's the main tutorial all done! The obvious question is what to do next? There is definitely benefit in redoing the tutorial; now you have a framework, you will learn much more deeply.

However, I would recommend working through some other materials first to gain a different perspective. At the very least you could work through the tutorial available on the *NetLogo* website, which will also expose you to some of the movement primitives. In addition, look at some of the models in the *Model Library* and see how they work. Note that these models may use the abbreviated form of some of the primitives. Look up any unfamiliar primitives in the *NetLogo* dictionary. Many of these library models have suggestions for extensions that you could try and do. Then do the moving turtles tutorial at Chapter 8 before optionally redoing this main tutorial.

The next step is to design and construct your own model. During coding, try and follow the sequencing of the tutorial. Create the agents and their physical world before trying to code the interactions and behaviour. Remember to make the behaviour as simple as possible and then amend it once the simple behaviour works correctly.

At this point, you can start working with some more advanced resources. Many of these are listed at the *NetLogo* website. You should also dip into the remaining chapters of this book. The most appropriate order will depend on your modelling project and your background.

There are two books that I would recommend to any agent-based modeller. Despite its name, *Agent-Based and Individual-Based Modeling: A Practical Introduction* by Railsback and Grimm (2011) is thorough and rigorous and comprehensively covers a range of important topics. While it can be difficult for a complete novice, it uses *NetLogo* to build the example models and should be accessible after completing this tutorial. A more theoretical perspective is taken in *Agent-Based Models* by Gilbert (2008). If there is a strong spatial element to your model or you are intending to work with GIS data, you are likely to find *Agent-Based Modelling & Geographical Information Systems* (Crooks et al., 2018) particularly useful.

A further two books are particularly good for stimulating ideas about what model to build: *Modeling Behavior in Complex Health Systems* by Keane (2013), and *An Introduction to Agent-Based Modeling: Modeling natural, social and engineered complex systems with NetLogo* by Wilensky and Rand (2015). Both use *NetLogo* extensively and also have downloadable models available.

Finally, there are several resources that take a different approach, focusing on the *NetLogo* language itself and how it is structured. The *NetLogo Dictionary* and the Programming Guide are included in the *NetLogo User Manual* (online, linked from the Help menu). Once you are reasonably comfortable with *NetLogo*, it is very useful to browse through the dictionary and to read the Guide thoroughly. These resources will help you become aware of some of the other primitives and gain a deeper understanding of how to think in *NetLogo*. This, more formal, programming language approach is also taken in *NetLogo: A Modeling Tool* by Garcia Vázquez and Sancho Caparrini (2016).

Turtles on the Move

An important part of many agent-based models is the mobility of turtle agents. For example, livestock must move around grazing areas in land use models, cars or pedestrians must follow routes and interact with each other in traffic models, and random movement can be used to control interactions for economic or social transactions. This chapter uses a mix of a small tutorial model to introduce the main *NetLogo* concepts and primitives for mobile turtles and examples drawn from the Library Models to demonstrate how to achieve some more advanced model features.

8.1 Mobility Tutorial Setup

There are three key concepts to understand for effective use of mobile turtles: the co-ordinate system that identifies location; heading or direction; and moving. The mobility tutorial is designed to demonstrate the basics of each of these concepts. This tutorial progresses much faster than the main tutorial. Only new primitives will be explained and instructions about creating widgets and creating or amending procedures are brief. There are also only limited reminders of practices such as saving the model and iteration and testing.

8.1.1 Model Design

The model is a fairly straightforward chasing game. Each turtle will choose another turtle to chase and will chase it until it captures it. Of course, the turtles being chased are also chasing other turtles. In addition, turtles will be able to detect a chaser that gets too close, and will move away from that chaser as a higher priority than chasing their prey. To make it more interesting, there will be obstacles that stop a turtle from simply following its target. See Table 8.1 for a translation of this game into responses to the design questions (at Table 7.1).

The general approach to developing this model will follow that of the main tutorial model. Start with creating the world and agents, and then add in the agent behaviour. Consistent with good programming practices of iteratively adding complexity, the chasing behaviour will be implemented (and tested) before any obstacles are included.

8.1.2 Setting up the Model

Following the general approach of the main tutorial model, the first steps are to establish the model and initialise the spatial environment. There are several elements that are common to establishing any *NetLogo* model, setting up the World topology, the *setup* and *go* buttons, and their equivalent procedures. These are contained in Snippet M1 (page 105), which can be considered a type of template.

This code includes the standard commands used within the 'setup' and 'go' procedures. It is an alternative starting template to that used in the tutorial model, where the procedures were

Table 8.1: Applying the broad design questions to the chasing game.

Aspect	Question	Tutorial model
Process	What actions or repeated decisions?	Chase and capture.
	What state changes without action?	None.
	What does theory say about these processes?	Move toward target.
Agents	What entities take actions?	Players.
	What personal characteristics affect action?	Speed.
	Any other entities need to be included?	No.
	What characteristics of those entities are relevant?	Not applicable.
	What characteristics change over time automatically?	None.
Spatial	Does location change the way in which the process operates?	Yes.
	What location specific resources (including information) affect action?	Obstacles to avoid.
	How do resources change over time automatically?	Not applicable.
Social	Do other entities change the actions of individuals?	Yes.
	How are agents aware of each other?	Selected target, avoid any close chaser.
AA interaction	What is the interaction between agents?	Chase target, move away from chaser.
AE interaction	How do the agents affect the environment?	Not applicable.
EA interaction	How does the spatial environment influence actions?	Cannot move through obstacles.
EE interaction	How do the characteristics of a location impact on other locations?	Not applicable.

Snippet M1: Establish model

1. Open *NetLogo* (if closed) or create a new model (if open).
2. Amend world settings to 81 by 81 patches (40 in max) and 7 patch size (see Figure 2.1 for hints).
3. Adjust the window size and World location.
4. Create a button labelled 'setup' to call the procedure named 'setup'.
5. Create a button labelled 'step' to call the procedure named 'go' with Forever not checked.
6. Create a button labelled 'go' to call the procedure named 'go' with Forever checked.
7. Enter the following code to create the 'setup' and 'go' procedures.
8. Check the code (green tick) and Save the model (no further reminders).

```

;-----
; MAIN CONTROL PROCEDURES
;-----

to setup
  clear-all
  reset-ticks
end

to go
  tick
end

```

created without any content. The tutorial model approach is clearer because `clear-all` is added when model entities are created, and `reset-ticks` and `tick` when time is introduced, so there is a connection between the new code and the process being implemented. However, adding these standard commands at the start of a new model can be a good habit.

The code can be tested by pressing the *setup* button and then either the *step* or *go* button. The tick counter should advance. It's not a complete test as the `clear-all` could be missing without affecting the model.

The next step is to create the physical environment. However, in the simple version of the behaviour, without obstacles, there is nothing to create. The final initialisation step is to create players, but that first requires an understanding of how *NetLogo* implements location.

8.2 Co-ordinates

The co-ordinate system within *NetLogo* was introduced at section 3.9. All locations are specified with two co-ordinates (*NetLogo 3D* is outside the scope of this book) in the common cartesian arrangement of an *x*-axis with values that increase to the right and a *y*-axis that increases upwards. The intersection of these axes is referred to as the origin. Locations to the left or below the origin have negative values for their *x* and *y* co-ordinates respectively.

Each turtle agent is located at some point in this two dimensional space. The co-ordinates are stored in the variables `xcor` and `ycor` (*x* and *y* axes respectively), which are automatically created for each turtle. These co-ordinates change as the turtle agent moves (for example, by moving forward some amount). Similarly, setting these variables to specific values moves the turtle agent to the specified location.

Each patch occupies one unit square in the two dimensional space, centred on integer values of the co-ordinates. The patch boundaries are included in the patch on the left and bottom, and are excluded at the top and right (see Figure 8.1). The patch co-ordinates are stored in the

variables `pxcor` and `pycor` (x and y axes respectively). As patches do not move, these variables can be accessed but not set. Those values are also used to identify patches, so `patch 0 0` is the patch centred on the origin.

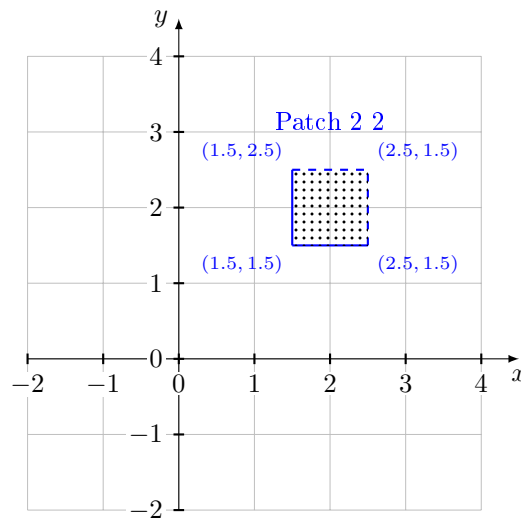


Figure 8.1: Overlay of patches with the co-ordinates. The limits of the patch with `pxcor` of 2 and `pycor` of 2 are displayed. Note that the solid lines at the bottom and left indicate that these co-ordinates are within the patch (eg `xcor = 1.5`, `ycor = 1.5`) and the dashed line at the top and right indicate that these are within the neighbouring patch (eg `xcor = 2.5`, `ycor = 2.5`).

Model settings (see Figure 2.1) control the number of patches in the World and the location of the origin. The only requirement is that `patch 0 0` is somewhere within the World. These settings also control wrapping, whether the edges of the World are borders that cannot be traversed, or whether turtles simply pass through the edge of the World and re-enter at the opposing edge.

8.2.1 Creating players

Having established the environment, the game needs turtle agents to chase each other. From the design, these turtle agents are players that need two variables: speed and the player they are chasing. That player is to be randomly assigned, the simplest approach. These requirements are implemented in Snippet M2 (page 107).

At this point, the first new primitives are required. The main tutorial model had each patch `sprout` the turtle agents, creating the appropriate number of people in the centre of each patch. For this model, the observer will create the turtle agents with `create-turtles`.

They will be randomly located throughout the world using `setxy`, which specifies the x and y co-ordinates, and those co-ordinates are randomly selected with `random-xcor` and `random-ycor`. Turtle agents have several built-in variables that are created automatically with the turtle. As well as the variables for the co-ordinates (`xcor` and `ycor`), turtles have a `color` and `size`. As for `sprout`, the code contained in the square brackets that is part of the `create-turtles` command is run for each turtle as it is created. That code is used to `set` the in-built variables.

Note that the targets are not assigned until all players have been created. If you tried to assign as each player is created, there would be two problems. The first player would have no potential targets, which would generate an error. Also, players created earlier would have more opportunities to be selected as a target and therefore introduce a bias into the model.

Press the `setup` button to initialise the game. Open an inspect window for a player and check that it has a target to chase and that the speed is some value up to the maximum speed on the slider.

Snippet M2: Create players

1. Create a slider for a global variable named num-players, from 0 to 100 by 1.
2. Create a slider for a global variable named max-speed, from 0 to 5 by 0.5.
3. Create a breed of turtles for players with variables named 'speed' and 'target', as shown in the code.
4. Create the setup-players procedure.
5. Add a call to setup-players in the setup procedure.

```

breed [players player]
players-own
[ speed
  target
]

;-----

to setup-players
  create-players num-players
  [ setxy random-xcor random-ycor
    set color blue
    set size 1.5
    set speed random-float max-speed
  ]
  ask players
  [ set target one-of other players
  ]
end

```

setxy

setxy <desired-x-co-ordinate> <desired-y-co-ordinate> sets the location of a turtle by specifying the co-ordinates. It is a single primitive version of separately setting the co-ordinates with a combination of **set xcor** <desired-x-co-ordinate> and **set ycor** <desired-y-co-ordinate>. See also turtle built-in variables **xcor** and **ycor**.

8.3 Direction

Look at the model after running the *setup* procedure, each player (blue dart shape) is pointing in some direction. This direction is stored in a built-in variable named 'heading', which is the number of degrees clockwise from pointing upwards.

In exactly the same way as other variables, a turtle's heading can be **set** to a specific number, such as 90 to point to the right. But there are other ways to control direction, such as turning **right** or **left** by a specific amount, or facing a model entity or specified co-ordinates. For example, open an inspect window for one of the players and enter **right 90** into the code area at the bottom of the window, the value of the **heading** variable will increase by 90 and the player will turn.

Note that the naming of these commands reflects the agent-centric perspective of *NetLogo* and agent-based modelling. Turning right or left is unambiguous from your own perspective. However, if somebody else asks you to turn right, the question then arises as to 'whose right' to turn. The reason that headings can be described as clockwise from pointing upwards is that the *NetLogo* World has a fixed perspective to a model user, a turtle with heading of 0 points upward.

and there is no way to get behind the computer screen and see the co-ordinate system from the other side.

What happens if **heading** is increased to more than 360 or decreased to less than 0? The value simply adjusts to the equivalent within that range by adding or subtracting 360 as required (that is, **heading** is technically heading modulo 360).

8.3.1 Choosing a heading

Clearly, if the chasing player is to capture their target, the chaser must move toward the target. The most straightforward approach is to face towards the target and then move forward. This is also the most natural representation of human behaviour, if you are chasing someone, you would face them rather than run sideways while facing a different direction.

However, players are also being chased, potentially by multiple other players. Highest priority is to face away from the closest chaser, and only face the target if there are no chasers in threatening distance. This is implemented with Snippet M3 (page 108).

Snippet M3: Choose a heading

1. Create a slider for a global variable named 'warning', from 1 to 10 by 1.
2. Add the choose-direction procedure.
3. Amend the go procedure to call choose-direction within an **ask**, as shown.

```

to go
  ask players
  [ choose-direction
  ]
  tick
end

;-----

to choose-direction
  set color blue
  let targetme (players in-radius warning) with [target = myself]
  ifelse any? targetme
  [ set color red
    face min-one-of targetme [ distance myself ]
    set heading heading + 180
  ]
  [ face target
  ]
end

```

The player first checks whether there is any nearby player chasing them. If so, they **face** the closest chaser and then add 180 to their **heading**, which faces them in the opposite direction. If there is no close chaser, they **face** their target.

face

The primitive **face** instructs the turtle agent to change its heading so that it is facing the specified model entity, a patch or turtle agent. See also **facexy**.

As well as **face**, this code introduces the **distance** primitive, which reports the distance between the asking turtle agent and some other agent (in the official *NetLogo* sense, so it may be a patch).

It is commonly used with `min-one-of` to find the closest turtle agent or patch that meets some requirement. In this case, it finds the closest of all the turtles targeting the asker.

distance

The `distance` primitive reports the distance to the specified turtle agent or (centre of) patch. See also `distancexy` to find the distance to a specific location referenced by x and y co-ordinates.



Finding the closest

The construction `min-one-of <candidates> [distance myself]` is used to find the member of the specified agentset that is closest to the asking turtle agent.

Turtle context procedures were shown in Figure 4.1, but were not required in the main tutorial. Here, the `go` procedure has `ask players` calling `choose-direction`, which is a turtle procedure. That is, *NetLogo* is already taking the player's perspective when the code enters the `choose-direction` procedure. The reason that this approach is used in this model is to ensure that each player makes its full move before moving on to the next player.

Consider what would happen if the `go` procedure called an observer context procedure to choose directions and then an observer context procedure to move (with `ask players` for each procedure). Every `ask` iterates through the agents. If a player chooses its direction, and then one of the chasers moves within the warning distance before the target has moved, then the target player may move toward a close chaser.

The code can be tested by pressing the *setup* then *step* buttons. Each player will initially choose a random direction, but then rotate to point to its target. You should see this rotation, and may see a turtle change to red. Inspect a player to identify its target and then check players that it is facing to verify that one of them is the target.

8.4 Movement

The most common primitive used in moving is `forward` (sometimes abbreviated to `fd`), which instructs the turtle to move in whatever direction it is facing. The distance it moves is given in units of the co-ordinate system. For example, if a turtle with a heading of 90 turtle located at the origin (given by `xcor` of 0 and `ycor` of 0) was asked to `forward 1`, it would move to the location with `xcor` of 1 and `ycor` of 0. If, however, the turtle has a diagonal heading, then one unit of distance is not enough to cross the patch.

Other primitives allow a turtle to look at the patch they would move onto if they either moved forward to the next patch or moved some arbitrary heading and distance. Turtles are also able to move in a direction that increases or decreases the value of a specified variable.

8.4.1 Chasing the target

The final step to implement the basic behaviour is for the player to move forward, since it is facing the correct direction. The speed variable controls how far it can move; speed is simply distance per unit time and each tick in a *NetLogo* model a unit of time.

However, if the chasing player simply moved by the amount available due to its speed, it could overshoot the target. Instead, the player needs to move at maximum speed if the target is too far away, but only as far as the target if it is close. This is implemented in Snippet M4 (page 110).

Run the model, you may need to slow it down (speed slider marked (4) at Figure 1.1) to see the players chasing each other. A screenshot is displayed at Figure 8.2.

Snippet M4: Move toward target

1. Add the move procedure.
2. Amend the go procedure to include a call to the move procedure after the player has chosen their direction.

```
to move
  let intended-move min (list speed distance target)
  forward intended-move
end
```

forward

The primitive `forward` instructs the turtle agent to move in the direction it is heading by the specified distance. If the World boundaries do not wrap and the requested forward movement would take the agent past the boundary, it stops on an edge patch. It is sometimes seen in code in its abbreviated form, `fd`.

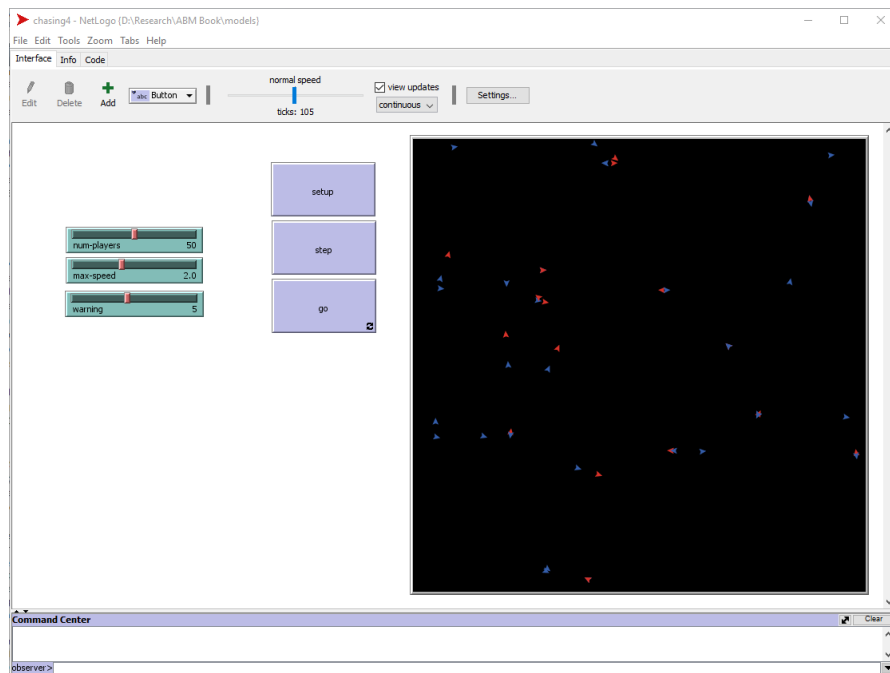


Figure 8.2: Screenshot after implementing movement with Snippet M4 (page 110). The game settles into a pattern of some clusters of players and other players following each other around the World.

The chase never ends because nothing happens when a player catches its target. One resolution is that the caught player is removed from the game. Any players who were targeting that player then need to choose a new target, but only the player who caught it should be given credit for the capture.

Consider what is required to implement this behaviour before amending the code, as described in Snippet M5 (page 111). Basic output widgets are also included, you could optionally add plots over time or histograms instead of simply reporting current counts and averages.

The critical line in this snippet is `if distance target < 0.005 []`. If a chaser gets very close to its target then the target is considered to be captured. This is an example of using a tolerance to avoid subtle errors arising from precision limits in the way that computers represent numbers

Snippet M5: Capture target

1. Add a variable for players named 'caught'.
2. Explicitly initialise variable 'caught' to 0 as the player is created.
3. Add the capture procedure.
4. Amend the go procedure to include a call to the capture procedure after the player has moved.
5. Add monitor for number of players.
6. Add monitor for average speed of players.
7. Add monitor for average number caught by players.

```

to capture
  if distance target < 0.005
    [ let my-target target
      ask target [die]
      ask players with [target = my-target]
      [ set target one-of other players
        ]
      set caught caught + 1
      set size size + 0.5
    ]
end

```

(referred to as floating point errors).

die

A turtle or link that is asked to **die** removes itself from the model.

Now what happens when you run the model? It still doesn't end but that's because the last two players are targeting each other. Any time one player gets close enough to almost catch the other, the warning triggers and they run away instead. While a pair of final players is the most common outcome, some simulations will end with 3 players in a chase and others will have two pairs in separate chases.

In a real situation, you would need to consider how to determine whether the simulation has finished so it stops. One option is to add a global variable for the number of players still playing. At regular intervals (such as 100 ticks), that variable is checked against the current number of players. If it's the same, then stop the simulation and, if not, replace the variable value. For the tutorial, manual stopping is fine. In addition, adding obstacles may change the outcome because it will be harder for the target to move away.

At this point the model can be considered a full agent-based model, even though it is very simple. Players are individually represented and are 'doing something' through time. Players interact with each other and those interactions influence the behaviour, by chasing and running away. Even without obstacles, the physical environment has an influence because it represents distance. There are several heterogeneous aspects of the players; location, target and speed.

8.5 Adding Obstacles

Before adding obstacles, more detailed thought is required about the model design. What is meant by an obstacle? In this model, players will be able to see over an obstacle (that is, they still know how to face their target), but cannot move across it. What is the effect on behaviour? If the player cannot move in its desired direction, then it will instead move in a random direction

without an obstacle. Eventually such random choices combined with the target moving should allow the chaser to move around the obstacle.

The next step is to create these obstacles in the physical environment. This can be implemented with a boolean variable indicating whether or not a patch is an obstacle, and the obstacles should be visible. The density of the obstacles is to be controlled by the user. These requirements are implemented in Snippet M6 (page 112).

Snippet M6: Create obstacles

1. Create a slider for a global variable named `prop-obstacles`, from 0 to 0.2 by 0.05.
2. Create a patch variable named `'obstacle?'`.
3. Add the `setup-patches` procedure.
4. Add a call to the `setup-patches` procedure in the `setup` procedure (before setting up players).

```
patches-own
[ obstacle?
]

;-----

to setup-patches
  ask patches
  [ ifelse random-float 1 < prop-obstacles
    [ set obstacle? true
      set pcolor yellow
    ]
    [ set obstacle? false
      set pcolor white
    ]
  ]
end
```

This snippet introduces the conditional code structure of `while`. The code inside the square brackets runs repeatedly until the condition returns `false`. Randomly placing the players runs the risk of having them start on a patch that is actually an obstacle. The `while` loop continues to move the player to a new random location until it is on a patch that is not an obstacle.

while

The full syntax is `while [<condition>] [<instructions>]`. The instructions are repeated until the condition fails. Note that this repetition is conceptually different from the repetition arising from repeated calls to the `go` procedure, because the ticks clock does not advance.

You can test this code by pressing the `setup` button and observing the location of players. No player should start on an obstacle (yellow patch). Find a player that appears to be on patch and open an inspect window to check its co-ordinates.

The final step is to have the player avoid an obstacle. This is more difficult than it seems as turtle agents cannot actually 'see' if there's an obstacle somewhere along their intended path. There are several potentially helpful *NetLogo* primitives available, of which `patch-ahead` is the most promising. This looks at the patch that is ahead of the asking turtle (that is, in the direction of the turtle's heading) some specified distance.

What distance should be checked? The first option is to check the distance intended to be moved, but an obstacle may have ended before that distance. Another option is to check integer points of distance, such as 1, 2 and so on, because each patch occupies one unit square of space. While attractive, this option may not detect a patch whose corner crosses the intended path if that corner is smaller than one unit across. But the idea is sensible, the resolution just needs to be finer than a full spatial unit. The finer the resolution, the smaller the corners that can be detected. This is the approach that is implemented with Snippet M7 (page 113).



Sight lines

There is no concept of perception along a line but *NetLogo* can be instructed to examine specific points either as absolute positions (x and y co-ordinates) or as a relative position from some anchoring location (heading and distance, or change in x and y co-ordinates). Perceiving along a path must be done by iteratively checking individual points on that path.

If the player detects an obstacle, it instead randomly selects a neighbouring patch that is not an obstacle and moves onto (or toward) that patch. To avoid potential problems of there being an obstacle beyond this safe neighbour, the moved distance is limited to 1.

Snippet M7: Avoiding obstacles

1. Create the check-path procedure.
2. Amend the move procedure.

```

to move
  let intended-move min (list speed distance target)
  forward intended-move
  ifelse check-path? intended-move 0.1
  [ forward intended-move ]
  [ face one-of neighbors with [not obstacle?]
    forward min (list 1 intended-move)
  ]
end

to-report check-path? [#how-far #resolution]
  let safe? true
  while [safe? and #how-far > 0]
    [ if [obstacle?] of patch-ahead #how-far
      [ set safe? false
      ]
      set #how-far #how-far - #resolution
    ]
  report safe?
end

```

The check-path? procedure implements checking at multiple points. If you can't remember the difference between a command (**to**) and reporter (**to-report**) procedure, review section 3.8, and look at section 4.3 for a reminder about the square brackets following the procedure name. The way it checks is that it looks the full distance first and then gradually checks closer. As soon as an obstacle is found, the **while** code block exits and the potential path is rejected.

The model is now complete. The final version is shown at Figure 8.3. You can test the obstacle avoidance by increasing the obstacle density and using the *step* button to move forward one

tick at a time. Look at a player that is moving toward an obstacle to check that it does not pass through. Alternatively, add a monitor that counts the number of players on patches with obstacles and check that it remains 0.

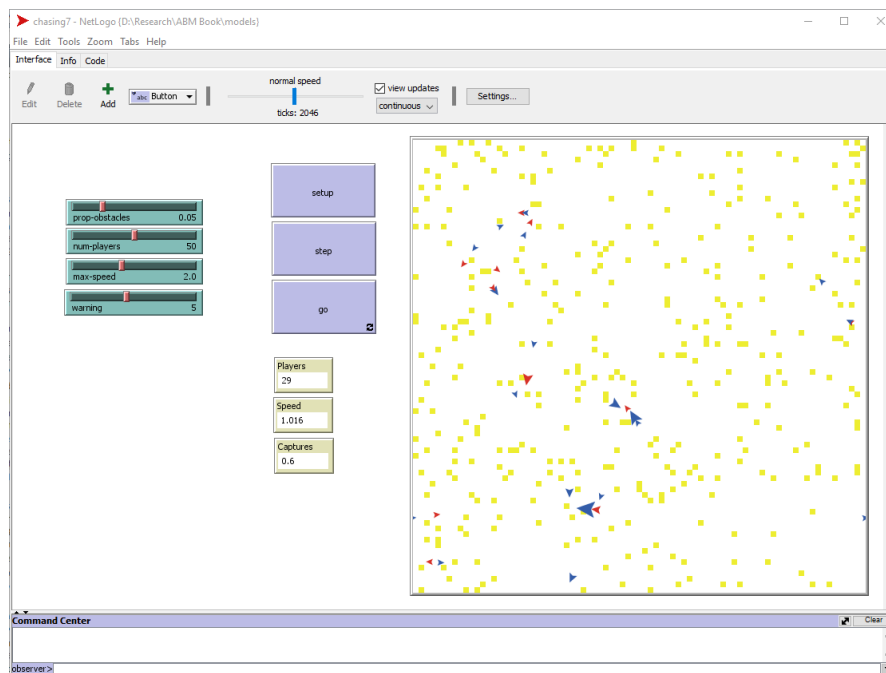


Figure 8.3: Screenshot after completing the chasing game model. Players (blue darts) avoid obstacles (yellow patches) while chasing each other. Player size indicates the number of captured targets.

8.6 Comments

NetLogo has many features built in to support mobile turtle agents. The most fundamental of these is the x and y co-ordinate system, which defines all locations.

A turtle's location is specified by its `xcor` and `ycor`, the co-ordinate values. A turtle occupies a point only, regardless of the size of the icon used to display the turtle agent. Further, co-ordinates are essentially continuous numbers, so the probability of two turtles being at the same location is negligible unless one turtle is explicitly moved to the other turtle.

One consequence of this arrangement is that *NetLogo* does not have any built in method to detect that two turtles are colliding, overlapping icons is not equivalent to occupying the same location. If you wish to check for overlaps, you will need to write your own code to calculate the co-ordinates of the edges of the two icons and check whether the edge of one icon is inside the space occupied by the other icon.

A patch occupies a unit square of the co-ordinate space, spanning from a half smaller (inclusive) to a half larger (exclusive) of the integer co-ordinate value. *NetLogo* understands the concept of 'here' to mean 'on this patch'.

With a co-ordinate system in place, directions can be defined by reference to that co-ordinate system. Headings in *NetLogo* are given by a number from 0 to 360 (using modulo arithmetic) with 0 pointing directly up the world and 90 pointing to the right. Headings can be specified explicitly using a number, or indirectly by asking a turtle to `face` a model entity or to change its `heading` by some value.

Distances and motion are also defined by reference to the co-ordinate system. The distance between two points is simply the difference between the co-ordinates. Motion is simply changing location by some distance.

The *NetLogo* Models Library contains several models that you can examine to help consolidate the concepts and primitives involved in mobile turtle agents. The *Ants* model was briefly discussed in section 1.7 as an example of agents interacting with their environment. You can now revisit the model to understand how the interaction is implemented. This model is particularly interesting because it uses the `uphill` primitive to instruct ant agents to move in the direction that increases the value of the specified patch variable. As traffic simulation is a common use of mobile agents, you may also wish to examine the *Traffic Grid* model in the library, which uses the co-ordinate system and headings extensively to restrict cars to roads. Note that library models may use the abbreviated form of *NetLogo* commands, such as `fd` for `forward`.

The library also contains Code Examples to demonstrate coding techniques to achieve particular objectives. This includes an alternative approach to the line of sight problem. Instead of standing at one point and checking at other points along the path, the library model hatches a turtle to move forward gradually, checking the patch where it is located and then dying when it reached the required distance.

Bibliography

- Abdou, M., L. Hamill, and N. Gilbert (2012). *Designing and Building an Agent-Based Model*, pp. 141–165. Dordrecht: Springer.
- Aristotle (2011). *Aristotle's Metaphysics*. Green Lion Press. Translated by Joe Sachs.
- Badham, J., S. Elsayah, J. H. Guillaume, S. H. Hamilton, R. J. Hunt, A. J. Jakeman, S. A. Pierce, V. O. Snow, M. Babbar-Sebens, B. Fu, P. Gober, M. C. Hill, T. Iwanaga, D. P. Loucks, W. S. Merritt, S. D. Peckham, A. K. Richmond, F. Zare, D. Ames, and G. Bammer (2019, jun). Effective modeling for integrated water resource management: A guide to contextual practices by phases and steps and future opportunities. *Environmental Modelling & Software* 116, 40–56.
- Barbrook-Johnson, P., J. Badham, and N. Gilbert (2017). Uses of Agent-Based Modeling for Health Communication: the TELL ME Case Study. *Health Communication* 32(8), 939–944. PMID: 27435821.
- Bass, F. M. (1969, jan). A new product growth for model consumer durables. *Management Science* 15(5), 215–227.
- Crooks, A., N. Malleson, E. Manley, and A. Heppenstall (2018). *Agent-Based Modelling and Geographical Information Systems*. SAGE Publications Ltd.
- Diekmann, O. and J. A. P. Heesterbeek (2000). *Mathematical Epidemiology of Infectious Diseases: Model Building, Analysis and Interpretation*. Wiley.
- García Vázquez, J. C. and F. Sancho Caparrini (2016). *NetLogo: a modeling tool*.
- Gilbert, N. (2008). *Agent-Based Models*. Quantitative Applications in the Social Sciences. Los Angeles: Sage Publications.
- Hammond, R. A. and R. Axelrod (2006, dec). The evolution of ethnocentrism. *Journal of Conflict Resolution* 50(6), 926–936.
- Hegselmann, R. (2017). Thomas C. Schelling and James M. Sakoda: The intellectual, technical, and social history of a model. *Journal of Artificial Societies and Social Simulation* 20(3).
- Keane, C. (2013). *Modeling Behavior in Complex Public Health Systems: Simulation and Games for Action and Evaluation*. Springer.
- Kermack, W. O. and A. G. McKendrick (1927). A contribution to the mathematical theory of epidemics - i. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 115A(772), 700–721.
- Knuth, D. E. (1984, feb). Literate programming. *The Computer Journal* 27(2), 97–111.
- Lotka, A. J. (1925). *Elements of Physical Biology*. Baltimore, MD: Williams & Wilkins Co.
- Luke, D. A. and K. A. Stamatakis (2012, April). Systems science methods in public health: Dynamics, networks, and agents. *Annual Review of Public Health* 33(1), 357–376.

- Macal, C. M. and M. J. North (2010, sep). Tutorial on agent-based modelling and simulation. *Journal of Simulation* 4 (3), 151–162.
- Miller, J. H. and S. E. Page (2007). *Complex Adaptive Systems*. University Press Group Ltd.
- Newman, M. (2010). *Networks: an introduction*. Oxford University Press.
- Railsback, S. F. and V. Grimm (2011). *Agent-based and Individual-based Modeling: A Practical Introduction*. New Jersey: Princeton University Press.
- Rogers, E. M. (2003). *Diffusion of Innovations* (5th ed.). Simon & Schuster.
- Sakoda, J. M. (1971, jan). The checkerboard model of social interaction. *The Journal of Mathematical Sociology* 1 (1), 119–132.
- Schelling, T. C. (1971, jul). Dynamic models of segregation. *The Journal of Mathematical Sociology* 1 (2), 143–186.
- Smith, J. M. (1964). Group selection and kin selection. *Nature* 201 (4924), 1145.
- Volterra, V. (1926). Variazioni e fluttuazioni del numero d’individui in specie animali conviventi. *Memoria della Reale Accademia Nazionale dei Lincei* 2, 31–113.
- Weaver, W. (1948). Science and Complexity. *American Scientist* 36, 536–544.
- Wilensky, U. (1999). NetLogo <http://ccl.northwestern.edu/netlogo/>. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.
- Wilensky, U. and W. Rand (2015). *An Introduction to Agent-Based Modeling: Modeling natural, social and engineered complex systems with NetLogo*. Cambridge, MA: MIT Press.

NetLogo Keywords

List of all the *NetLogo* commands, reporters and other keywords presented in this book. See the *NetLogo* Dictionary for all available keywords, their definitions and short examples. Some keywords also have short versions, but such abbreviations are not used in the tutorial. The full words are easier to read and more natural to remember.

and, 77	let, 26, 36	random, 29
any?, 85	link-neighbors, 84	random-float, 70
ask, 51	links, 72	repeat, 39
breed, 32	list, 85	report, 55
clear-all, 35	max-n-of, 48	reset-ticks, 45, 54
comment (;), 11, 22	max-one-of, 48	round, 47
count, 48	min-n-of, 48	scale-color, 36
create-links-with, 83	min-one-of, 48	self, 73
create-turtles, 32, 33	my-links, 82	set, 46, 47
die, 111	myself, 60, 67	setxy, 107
diffuse, 39	n-of, 47	show-link, 84
distance, 109	neighbors, 59	sprout, 33
end, 21, 55	neighbors4, 59	tick, 45, 53, 54
face, 108	not, 77	ticks, 45, 54
floor, 33	of, 26, 56, 57, 83	to, 21
forward, 110	one-of, 47, 60	to-report, 55
globals, 27	or, 77	turtles, 72
hatch, 33	other, 82	turtles-here, 76
hide-link, 84	patch-set, 73	turtles-on, 74
if, 49, 50	patches, 72	turtles-own, 32
ifelse, 49, 50	patches-own, 28	type, 85
ifelse-value, 49, 50	pcolor, 37	while, 112
	print, 85	with, 74
	pxcor, 19	
	pycor, 19	

Index

- agent-centric, 4, 10, 44, 49, 51, 107
- agentset, 71
- breed, 31, 34
- call, 21
- co-ordinates, 19, 58, 105
- code
 - spacing, 31, 41
- colours, 35
- Command Center, 37, 47, 85
- comment, 11, 21, 23, 29, 41
- complexity, 9
- conditional, 49, 73
- context, 7, 66
- context menu, 20, 23, 34
- debug
 - context menu, 37
 - inspect, 34
 - sanity check, 27–29, 35, 38, 47, 49, 55, 68, 69, 73, 83, 85
 - syntax, 23, 24
- design, 78
- epidemic, 11, 16, 43
- false, *see* logic
- floating-point, 111
- forever, 23
- initialise, 22, 32
- interface, 73
- keyword, 5, 10, 17, 55
- list, 56, 83
- logic, 57, 73, 75, 77
- mathematics, 30
 - precedence, 30
- model, 3
- models, classic
 - diffusion, 11, 13, 44, 61
 - predator-prey, 9
 - segregation, 13
- models, NetLogo Library
 - Ants, 14, 115
 - Ethnocentrism, 14
 - Line of Sight, 115
 - Rabbits Grass Weeds, 7, 22, 24, 36
 - Segregation, 13
 - Traffic 2 Lanes, 14
 - Traffic Grid, 115
 - Virus on a Network, 11
- operationalisation, 15
- patches, 19
- pop-up menu, *see* context menu
- position, *see* co-ordinates
- precedence, 31
- primitive, *see* keyword
- probability, 70
- procedure, 21, 55, 69
 - call, 23, 28
- SIR, *see* epidemic
- time, 51
- true, *see* logic
- turtles, 31
 - attributes, 32
 - creating, 32
- variables, 24, 45, 98
 - boolean, 26, 57, 73
- verification, *see* debug
- widgets, 7, 22, 50, 61
 - button, 22
 - monitor, 27, 61

note, 94
output, 85
plot, 61
position, 62

slider, 52
switch, 74
World, 34