Chapter 2: Starting agent-based modelling

This Chapter shows how to create a simple agent-based model and introduces the programming environment, NetLogo, that will be used for the models described in the rest of the book. The model simulates consumers shopping for fruit and vegetables in a produce market. The consumer agents are initially programmed to choose a market stall to purchase from at random, and then successive enhancements are made to record the cost of purchases, to stop them revisiting a stall they have previously been to, and to try to find the cheapest stalls to buy from. The program is explained in detail to show how an agent-based model is constructed and how a relatively simple model can be enhanced step by step. At the same time, many of the basic building blocks of NetLogo programming are introduced.

Keywords: NetLogo, programming agent-based models, bounded rationality, market, verification

Introduction

While it is possible just to use an agent-based model that someone else has developed, it is much better and more interesting to see what goes on ‘under the hood’, so that you can see and understand the program code that is making the model work. Learning how to program, as well as being a worthwhile exercise in its own right, will allow you to modify existing programs, for example to explore the effect of different settings and different assumptions, and eventually to build your own programs. Programming used to be a matter for experts, requiring many months of study or a degree in computer science. Fortunately, advances in programming languages and interfaces have meant that programming is becoming ever more easily accessible to those without expert knowledge. In this book, we use a programming system called NetLogo (Wilensky, 1999) that was originally designed for secondary (high) school pupils, and is still used, particularly in the United States, to teach science by means of simulations. The origins of NetLogo mean that a great deal of attention has been paid to making the NetLogo system easy to use and to understand. We will take advantage of that in this book, and we will also benefit from the fact that NetLogo is especially good for developing agent-based models. The authors describe it as a ‘multi-agent programmable modelling environment’.

NetLogo has three parts:

- a code editor to write programs,
- an interface that shows the controls to operate the program and any of a range of graphs, maps and other outputs to show what the program is doing, and
- a documentation editor that can be used to describe the program and what it is intended to do.

Each of these is independent so, for example, you run a program that someone else has written and observe what it is doing without looking at the code. NetLogo comes with a large library of pre-built sample models taken from physics, chemistry, geography and other disciplines, as well as economics, and trying these is a good way of starting to become familiar with it.

NetLogo can be downloaded from http://ccl.northwestern.edu/netlogo/ . It is open source and free, and works almost identically on Windows, Mac OS and Unix systems. You might like to download it onto your computer now and follow along as you work through this Chapter. There are other agent-based modelling environments (examples include Mason and RePast, both of which are open source, but are based on the programming language, Java, so that using them needs some prior knowledge of that language, and AnyLogic, a commercial system), but for beginners, NetLogo is at the moment the best place to start.

This Chapter explains some aspects of NetLogo, but for a fuller account, see the textbook authored by its main developer (Wilensky & Rand, in press). Other books on agent-based modelling also include introductions, for example Railsback & Grimm (2011) and Gilbert (2007). The NetLogo system itself includes an excellent tutorial and many code examples in its Model Library. There is also a NetLogo Users Group email list at https://groups.yahoo.com/neo/groups/netlogo-users/info and a StackOverflow community at http://stackoverflow.com/questions/tagged/netlogo where questions can be asked.

To illustrate how NetLogo is used, we will develop a simple model and explain the program code step by step. First let us see the model in action (the code can be downloaded from the website: *Chapter 2: Starting Agent-based modelling*).

A simple market: the basic model

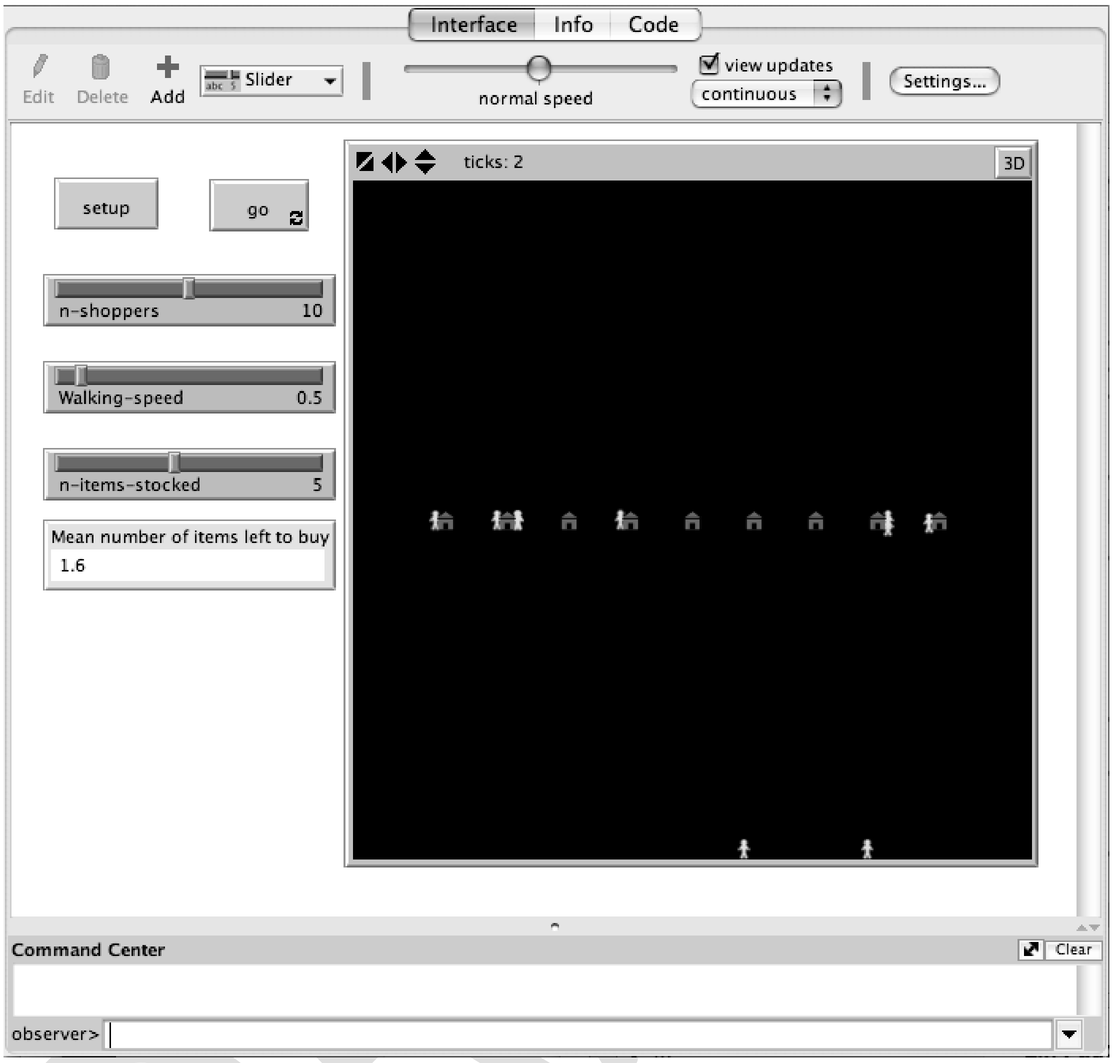
Much of this book is about modelling markets of different types. One of the simplest is a produce market, such as is often found in towns and cities selling fruit and vegetables. In this model, there are shoppers, each with a shopping list, and a number of market stalls, each selling a range of fruit and vegetables. The shoppers want to buy the items on their lists, which will differ from one shopper to the next, and may want to minimise the cost of their shopping. The stallholders sell their produce for different prices, depending on what they think the customers will pay, the prices that they paid in a wholesale market and other factors. Not all stalls sell the complete range of fruit and vegetables.

Agent-based models almost always follow a standard pattern: they are initialised with parameters that define the starting situation. Then the model is executed, to simulate the passage of time. At each step, representing some short duration (e.g. a day), each agent performs some action (or does nothing), as determined by its behavioural rules. The action can include communicating with other agents, changing the environment, moving through the environment, and many other things. The execution of the program continues, step by step, until either some programmed stopping condition is met, or the user stops the simulation manually. While the program runs, what is happening to the agents can be measured on graphs or monitors. This will become clearer as we work through the example model.

Figure 2.1 shows the interface of the model at a point midway through the simulation. The houses represent the market stalls, and the people are the shoppers. To run the model, you first press the `setup` button at the top left, which initialises the model, and then the `go` button to run it. The top slider is used to adjust the number of shoppers before the simulation begins. The slider under it, labelled `walking speed`, adjusts the speed at which the shoppers walk from stall to stall. This slider can be adjusted while the simulation runs to make the shoppers move faster or slower. The third slider sets the number of kinds of produce that traders keep on their stall. The three sliders allow the user to adjust the parameters of the model before and during each simulation run.

The bottom object is a monitor. It keeps a continually updated record of the average number of items on the shoppers' lists, that is, how many items have yet to be bought. Figure 2.1 shows the interface; clicking on the tab labelled 'Code' at the top takes you to the program itself (see Figure 2.2).

Figure 2.1 NetLogo Interface window showing the Market model after two ticks.



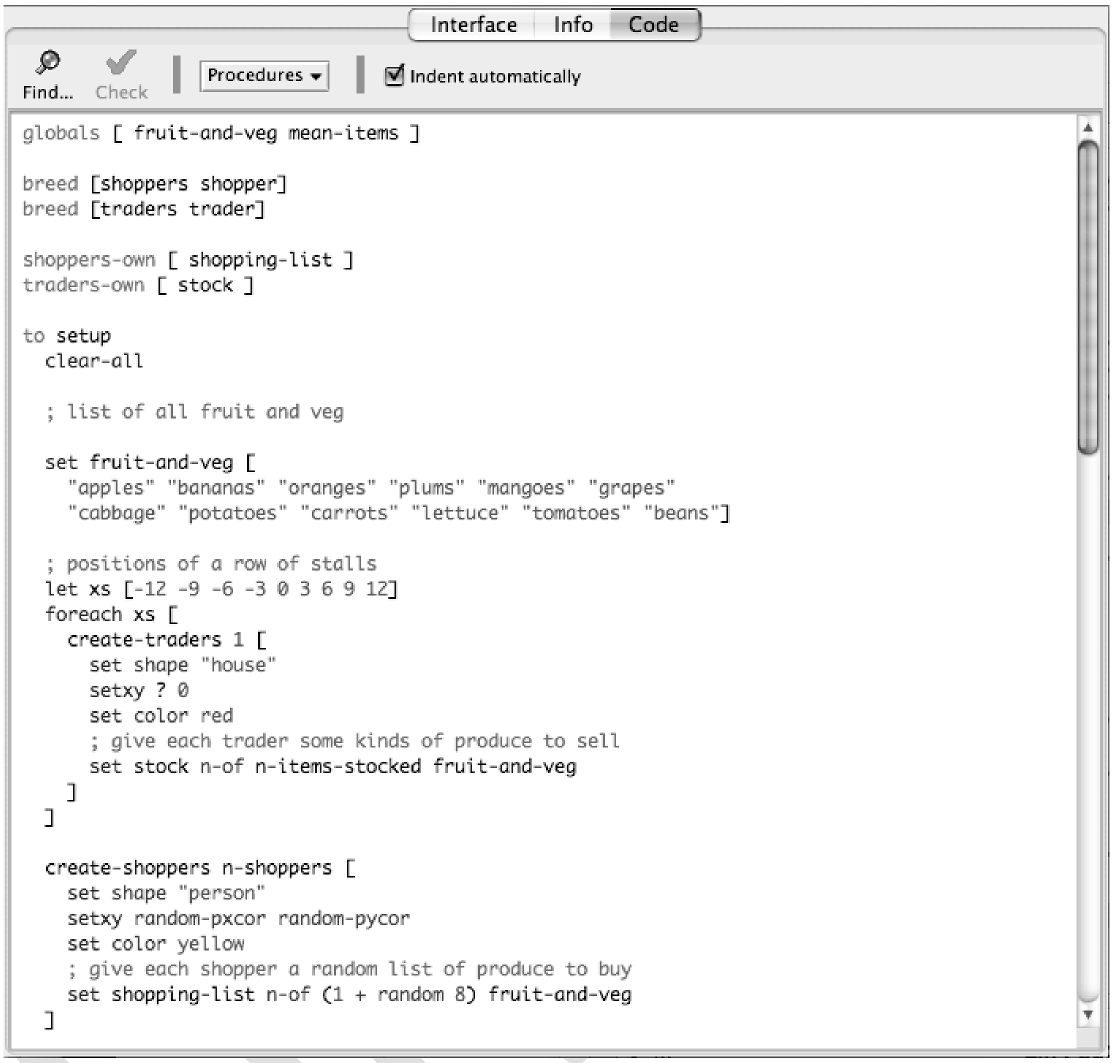
n-shoppers 10

Walking-speed 0.5

n-items-stocked 5

Mean number of items left to buy 1.6

Figure 2.2: The NetLogo Code window showing the top portion of the market model code.



```
globals [ fruit-and-veg mean-items ]

breed [shoppers shopper]
breed [traders trader]

shoppers-own [ shopping-list ]
traders-own [ stock ]

to setup
  clear-all

  ; list of all fruit and veg

  set fruit-and-veg [
    "apples" "bananas" "oranges" "plums" "mangoes" "grapes"
    "cabbage" "potatoes" "carrots" "lettuce" "tomatoes" "beans"]

  ; positions of a row of stalls
  let xs [-12 -9 -6 -3 0 3 6 9 12]
  foreach xs [
    create-traders 1 [
      set shape "house"
      setxy ? 0
      set color red
      ; give each trader some kinds of produce to sell
      set stock n-of n-items-stocked fruit-and-veg
    ]
  ]

  create-shoppers n-shoppers [
    set shape "person"
    setxy random-pxcor random-pycor
    set color yellow
    ; give each shopper a random list of produce to buy
    set shopping-list n-of (1 + random 8) fruit-and-veg
  ]
```

The basic framework

NetLogo programs have two sections: a ‘setup’ section which is executed once and which initialises the model to its starting state, and a ‘go’ section which contains code that is executed again and again as the simulation runs. In addition, at the beginning there is a header section that specifies the variables and the types of agents that will be used in the program.

The header for the example program (Box 2.1) begins by reserving two ‘global’ variables, `fruit-and-veg` and `mean-items` (line 1). The former will be used to hold a list of all the kinds of fruit and vegetable that any trader can offer for sale. The latter will in due course be used to hold the average number of items remaining on shoppers' shopping lists. When this variable had reached zero, everyone's shopping will have been completed. They are called global variables because they are accessible everywhere throughout the program.

Box 2.1: Market simulation header section.

```
globals [ fruit-and-veg mean-items ]
breed [shoppers shopper]
breed [traders trader]
shoppers-own [ shopping-list ]
traders-own [ stock ]
```

Note: the line numbers have been added; they are not part of the program code.

There are two kinds of agents in the model: shoppers and market traders. NetLogo allows you to name the types of agents as `breeds` (by analogy to the way that breeds of dog are different types of dog). One of the points emphasised in the previous chapter is that economic models need to allow for the fact that not everyone is the same; that is, to be able to deal with heterogeneity. In this simple market model each shopper has a different list of items to buy, and each trader has different kinds of fruit and vegetables in stock. So each shopper has its own shopping list and each trader, its own list of what it sells. Shoppers are provided with a variable, `shopping–list`, that will be used to store that agent's own shopping list. Similarly, each trader has a variable, `stock`, in which will be stored a list of items that the trader has for sale.

So far the code has done nothing more than to define some names, of variables and agents. Next comes the setup section of code (Box 2.2), which is executed once at the beginning of the simulation, when the user presses the setup button on the interface shown in Figure 2.1. The setup section includes code that initialises the variables defined in the header section (shown in Box 2.1) and creates the agents that will populate the model.

Box 2.2: Setup section.

```
 6  to setup
 7    clear-all
 8    ; list of all fruit and veg
 9    set fruit-and-veg [
10      "apples" "bananas" "oranges" "plums" "mangoes" "grapes"
11      "cabbage" "potatoes" "carrots" "lettuce" "tomatoes" "beans"]
12    ; positions of a row of stalls
13    let xs [-12 -9 -6 -3 0 3 6 9 12]
14    foreach xs [
15      create-traders 1 [
16        set shape "house"
17        setxy ? 0
18        set color red
19        ; give each trader some kinds of produce to sell
20        set stock n-of n-items-stocked fruit-and-veg
21      ]
22    ]
23    create-shoppers n-shoppers [
24      set shape "person"
25      setxy random-pxcor random-pycor
26      set color yellow
27      ; give each shopper a random list of produce to buy
28      set shopping-list n-of (1 + random 8) fruit-and-veg
29    ]
30    set mean-items mean [ length shopping-list] of shoppers
31    reset-ticks
32  end
```

The setup section is marked by `to setup` at the beginning (line 6) and `end` at the end (line 32). Everything in between is executed once in order from top to bottom. First (line 7) NetLogo is instructed to remove everything (agents etc.) that might have been left over from a previous run. Then the `fruit-and-veg` variable is initialised to a list of fruits and vegetables. All the lines beginning with a semi-colon are comments intended for the human reader and are ignored by NetLogo. Hence line 8 is for our information only. The next line, line 9, does the work. The square brackets here, `[` and

`]`, indicate that what lies in between is a list of items. `set` is the command that puts a value into a variable (other programming languages often use = to mean the same). So after line 9, the value of the variable `fruit-and-veg` is a long list of the names of fruit and vegetables.

Now we draw the market traders (the house shapes in Figure 2.1). First, we create another list, this time consisting of the positions of each trader on the NetLogo grid. This grid consists of squares (called `patches` by NetLogo) in a coordinate system, which is this example, runs from -16 to +16 from left to right, and -16 to +16 from bottom to top. Line 13 sets up the variable `xs` to hold the x-coordinates of each of the traders on the grid, in preparation for actually creating the trader agents. `let` is used instead of `set` because `let` does two jobs: it creates a variable and also gives the variable an initial value. `foreach` (line 14) works its way down the list of coordinates in `xs` and for each one executes the commands between the square brackets in lines 14 and 22. First the left most trader (at x-coordinate -12) is created, then the next trader is created at -9 and so on until the last one at coordinate 12. In line 15, the command `create-traders 1` generates one new trading agent and this agent is initialised using the commands in lines 16 to 20. The displayed shape of the agent is set to house icon (line 16), the house is positioned at the current x-coordinate (-12 on the first time through) using the place holder `?` to represent the current item in the `xs` list and at y coordinate 0 (i.e. on the horizontal centre line) and the colour of the house is set to red.

In line 20, the just created trader is given some stock to sell. The number of items of produce has been set as a parameter by the user by adjusting the `n-items-stocked` slider on the interface. This number is used in `n-of`, which selects a random selection of that number of items from the `fruit-and-veg` list. That concludes the first time though the loop that started at line 14. Each further pass through lines 14 to 22 creates another agent, located at successive x-coordinates taken from the list of `xs`.

This part of the code has demonstrated some basic building blocks of NetLogo programming:

- A variable can be given a value using the `set` command

- A list can be laid out between square brackets; the list can contain anything, e.g. numbers or symbols such as ‘`apple`’
- Agents are generated using the `create-`*`breed`* command. As part of the agent creation process, commands enclosed in square brackets following the `create-`*`breed`* command are executed to initialise the agent (e.g. to set its shape, position and colour)
- You can iterate down a list, one item at a time, with the `foreach` command.

These building blocks are used in many of the programs later in the book.

The next part of the setup code (lines 23 to 29) is concerned with creating the shopper agents. This follows the pattern for creating the traders. The number of shoppers is set by the slider called `n-shoppers` (see Figure 2.1). Once created, each agent is set to a ‘`person`’ shape and located at a random location on the grid (line 31). `random-pxcor` chooses a random x-coordinate somewhere on the grid, and `random-pycor` does the same in the y direction. The colour of the agent is set to yellow and finally the shopper is given a shopping list selecting items from the `fruit-and-veg`. The number of items on the shopping list is calculated as a random number. This is done by using a NetLogo reporter, which is code that returns some value. The reporter `random` returns a random integer between zero and the number given: in this case up to but not including 8. Adding 1 ensures that there is at least one item on every shopper’s list.

All the agents, both traders and shoppers, have now been created and placed on the grid. As a final step, the mean number of items on the shopping lists is calculated in line 30. Line 30 introduces further NetLogo constructs. Recall that each shopper has its own shopping list. The length of the shopping list is obtained using the reporter, `length` i.e. in this case the number of items in a list.

`shoppers` refers to what NetLogo calls an `agentset`. An agentset is, unsurprisingly, a set of agents, in this case the set of all the shoppers that were created earlier. Agentsets are very useful in NetLogo, as they are an easy way to refer to groups of agents. `of` works with agentsets and returns a list composed of what is returned by the preceding reporter (i.e. `[length shopping-list]`). So, the construct `[length` `shopping-list] of shoppers` produces a list of the lengths of the shopping lists of all the shoppers. This is fed to the `mean` reporter, which finds the average of a list of numbers. Thus, taking the command as a whole, `mean-items` is set to the value of the mean length of the shoppers' shopping lists.

Lastly, in line 31, the NetLogo clock, which counts steps or `ticks,` is reset to zero and the setup is complete. (Using the NetLogo's `tick` command is the simplest approach but you can easily devise your own time counter, which is more flexible.)

Box 2.3: The go section.

```
33 to go
34  ; for each  shopper in turn that still has something to buy
35  ask shoppers with [not empty? shopping-list] [
36    ; choose a stall
37    let stall one-of traders
38    ; go to that stall
39    face stall
40    while [ patch-here != [patch-here] of stall ]
41       [ forward 0.005 * walking-speed ]
42    ; buy everything on my shopping-list that is for sale
43    ; at this stall
44    let purchases filter [ member? ? [stock] of stall] shopping-list
45    foreach purchases [
46      ; delete the items bought from the shopping list
47      set shopping-list remove ? shopping-list
48    ]
49    ; when shopping is done, go home
50    ; (move to the edge of the grid)
51    if empty? shopping-list [ set ycor -16 ]
52  ]
53  ; calculate the average number of items on the shopping lists
54  set mean-items mean [ length shopping-list] of shoppers
55  ; if no one has anything left to buy, stop
56  if mean-items = 0  [ stop ]
57  ; count the iterations
58  tick
59 end
```

The final section of the program is the part that gets executed repeatedly, once for every time step. It is bounded by `to go` (line 33) and `end` (line 59). It is an example of a procedure; the `setup` section also consisted of a procedure. Procedures start with `to` and the name of the procedure, and end with `end`. Procedures can have any name you wish, but by convention, the initialisation is carried out by a `setup` procedure and the main part of the simulation is called the `go` procedure. Execution of the `go` procedure starts when the user presses the `go` button on the user interface (Figure 2.1).

Most of the `go` procedure consists of an `ask` command: each of the shoppers that still have some items to buy is asked to carry out some commands. When `ask` is executed, each of the agents mentioned immediately after the command, in this case all the shoppers that have items still to buy on their shopping list, are given the set of commands that follow the square brackets (lines 36 to 51). `ask` is equivalent to writing out some instructions on a sheet of paper, copying the instructions as many times as there are agents, and then asking each agent to carry out those instructions independently. In principle, the agents carry out the instructions simultaneously, but since this is difficult to implement on a single computer, they act one after another, but with the order of agents assigned randomly (and in a different order with every `ask`).

First, each shopper chooses a trader to buy from using `one-of`, which selects one agent (a trader) from an agentset (all the traders). Then the shopper changes the direction in which they are pointing on the grid until they are facing the selected trader's stall and move in that direction. The `forward` command (line 41) moves the agent in the direction it is facing. The movement continues `while` the patch on which the shopper is walking remains different from the patch on which the trader's stall is located. The speed of movement is controlled by the `walking-speed` slider on the user interface: higher values of walking speed mean that the agent moves further each time around the `while` loop. (The constant 0.005 is there to make the agents walk sufficiently slowly that their movement is visible. It may need adjustment to suit the speed of your computer).

Eventually the shopper arrives at the stall and can buy anything on its shopping list that the stall sells. (Recall that a stall doesn't necessarily stock everything that might be on the agent's shopping list). The agent is assumed to buy everything it wants that is available from the trader. This is the purpose of line 45. `filter` works along a list, item by item, and makes a new list, `purchases`, of those items in the shopping list for which a reporter (in this case `[member? ? [stock] of stall]`) is true. The item being considered is represented by the placeholder `?` . So, the reporter asks whether the item (`?`) is a member of the stall's stock list and reports true if it is, which is just what we want to happen. Traders can always meet shoppers' demand and do not run out of stock.

This may leave some items not available on this stall and therefore yet to be bought. To find which items were not bought, because they were not in the stall's stock, we take the list of purchases, work through it item by item, and remove from the shopping list those items that have been bought (lines 45 to 48). Now that the shopper has bought everything it wants that is available from this trader, it is time to check whether there is more shopping to do. If not (i.e. the shopping list is now empty), the shopper is moved to the edge of the grid to be out of the way.

That is the process for one agent. The `ask` command gets all the agents to go through this process once. Then the mean number of items left on the shoppers' lists is calculated. If everyone has done all their shopping (the mean number of items on the lists is zero), the simulation stops. If not, the simulation clock is incremented by one. That completes one step of the simulation, and the loop is repeated. On the second time around, those agents with more shopping to do will again choose a trader's stall and buy items that the trader has in stock. Eventually all the shopping will have been completed, all the shoppers will have gone home, and the simulation stops.

The `go` procedure demonstrates some common features of NetLogo programs. Most of the procedure consists of an `ask` command and inside this command is a list of the behaviour rules that the agents carry out. The actions can include movements around the environment, represented by the grid, manipulation of the agents' state (here, a shopping list) and the calculation of variables such as `mean-items`.

Enhancing the basic model: adding prices

The model described so far is intentionally rather simple, to make it easier to explain the general structure of agent-based models. As you may have noticed, it does not include any economic variables as yet. But once a basic model is working, it can be modified and enhanced with further details and, often, this is the best way to develop models: one stage at a time. Indeed, it is very important to check that a model is doing what you intended as you develop it. If this process, called verification, is left until the model is large and complicated, it may well be almost impossible to do. NetLogo makes this approach to development easy because it is simple to edit a program and see the effect of changes.

For our first enhancement, we shall add prices to the model. Each kind of fruit and vegetable will have a price set by the market trader.

The following changes and additions are needed:

```
globals [ fruit-and-veg fruit-and-veg-prices mean-items ]
…
shoppers-own [ shopping-list spent]
traders-own [ stock prices ]
```

As well as having a global variable to hold the full range of fruit and vegetables, there is a variable that stores a list of the prices of each of these (e.g. the wholesale price). These prices are arranged in the same order as the produce in `fruit-and-veg`, so for example, if oranges is the third item in `fruit-and-veg`, the price of oranges will be the third item in `fruit-and-veg-prices`. Both shoppers and traders need additions to their own variables: shoppers record the money that they spend for their fruit and vegetables in the variable `spent`, and traders have a price list containing the prices of each of the kinds of produce they have in stock.

Traders add a mark up to the wholesale prices. The mark varies between traders and is a random percentage between 1 per cent and 30 per cent. The following lines calculate the mark up and generate a list of the prices, including the mark up, arranged in the same order as the traders' list of produce in `stock`.

```
let mark-up (1 + random 30) / 100
foreach stock [
    set prices lput ((1 + mark-up) *
          (item (position ? fruit-and-veg) fruit-and-veg-prices)) prices
   ]
```

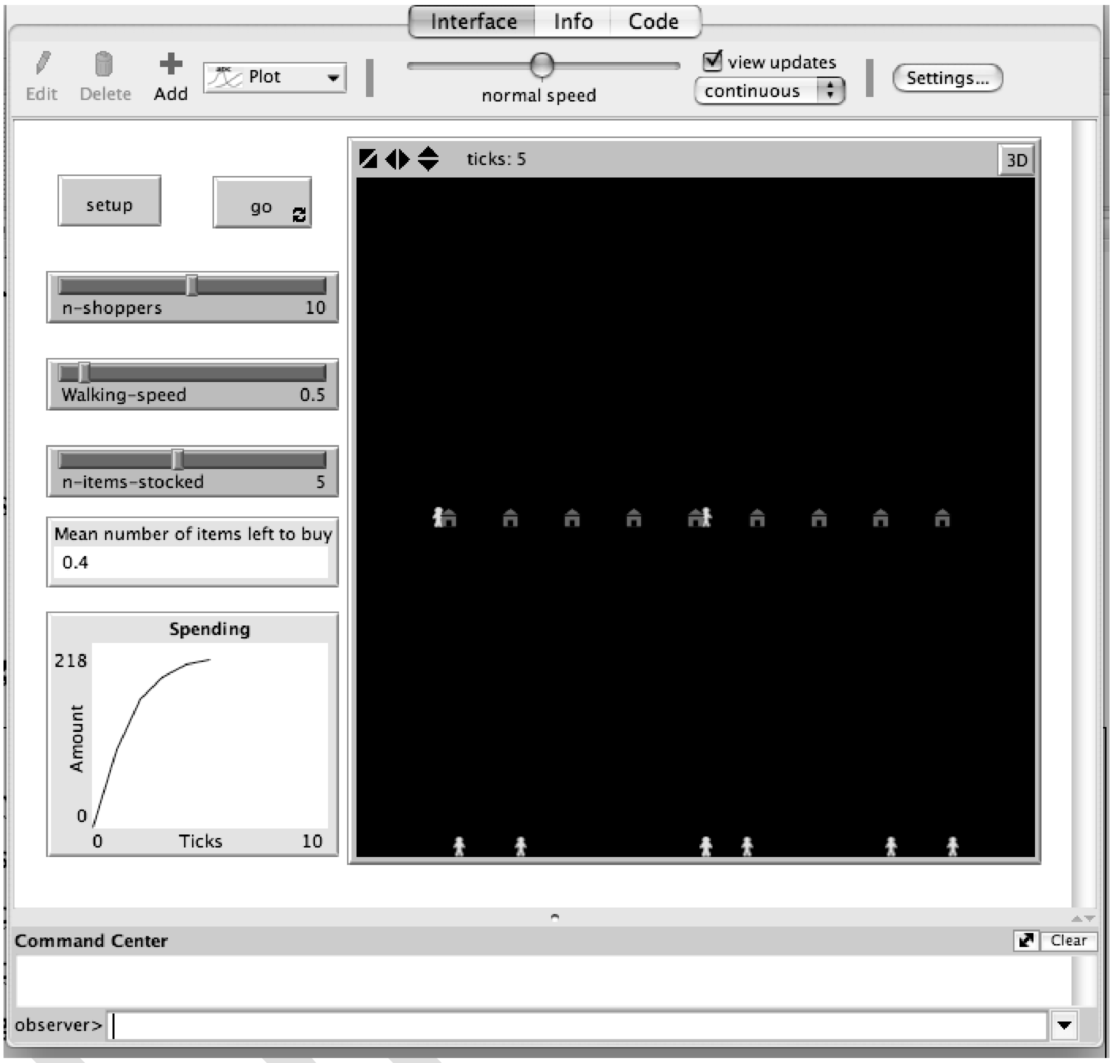
The `foreach` works down the list of stock items. For each one it puts the price of the item on the end of a list of prices using `lput`. The price of the item is determined by finding the position of the item in the list of fruit and vegetables `(position ? fruit-and-veg)`, then extracting the price found at that position in the list of wholesale prices, and adding the mark up.

```
45 foreach purchases [
       set spent spent +
               item (position ? [stock] of stall) [ prices ] of stall
46   ; delete the items that have been bought from the shopping list
47   set shopping-list remove ? shopping-list
48 ]
```

This block of code is inserted after line 45 in the `go` procedure. For each purchase ("oranges", "bananas" or whatever), the price of that purchase is looked up in the trader's pricelist and added to the sum spent by the shopper. It is assumed that the shopper buys only one unit of each item. The look up is done in two parts: first the position of the item in the trader's stock list is found. For example, if the stock list is `["apples" "bananas" "oranges" "plums" "carrots" "lettuce" "tomatoes"]` and the shopper has bought carrots, `(position ? [stock] of stall)` will yield 4 (the first position in a list is position 0). Then the `item` reporter returns the price at the corresponding position in the trader's pricelist (e.g. the price at position 4).

Finally, it would be helpful to be able to observe the amounts being spent by the shoppers as they go about their shopping. This can be done by drawing a plot of the average amount spent against time, measured in ticks. The result is shown in Figure 2.3.

Figure 2.3: The interface of the model enhanced with a plot of the average amount spent.



n-shoppers 10
Walking-speed 0.5
n-items-stocked 5
Mean number of items left to buy
0.4

This first enhancement shows how to augment a model by adding just a few lines of code, in this case to calculate the amount the shopper is spending, and how you can instrument a model by adding a plot showing how the agents' behaviour changes over time. We have also seen other list processing commands, to filter lists and remove items. There are many more such commands available in NetLogo, all of which are described in the NetLogo dictionary.

Enhancing the model: selecting traders

In this and the basic version of the model, the shoppers choose the stall they go to at random, and then if their shopping is not yet done, they choose another stall at random. The problem with this is that it is possible for the shopper to choose the same stall twice, which seems somewhat stupid. This can be avoided if the shoppers are given some memory to record where they have been so that they can then choose their next stall from those they have not yet visited.

The third version of the model does this. A new variable for each agent to store those stalls it has not yet visited is added to the setup procedure for the agentset of all the shoppers.

```
   4  shoppers-own [ shopping-list not-yet-visited spent ]
…
   23 create-shoppers n-shoppers [
   24    set shape "person"
   25    setxy random-pxcor random-pycor
   26    set color yellow
         set not-yet-visited traders
   27    ; give each shopper a random list of produce to buy
   28    set shopping-list n-of (1 + random 8) fruit-and-veg
   29 ]
```

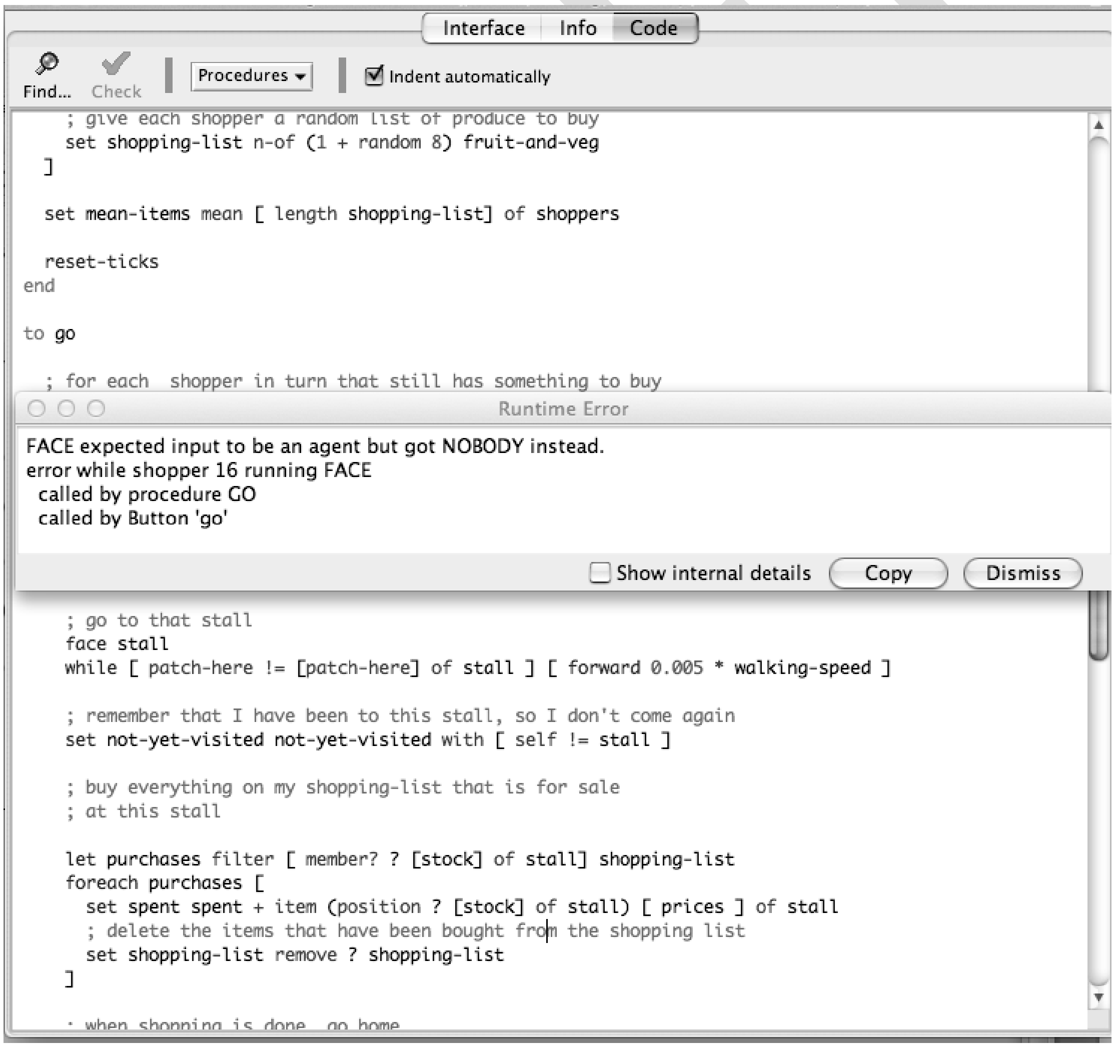
In the `go` procedure, the agent selects a stall to visit, not from every stall as before, but from the agentset of those not yet visited (line 37).

```
   36 ; choose a stall
   37 let stall one-of not-yet-visited
   38 ; go to that stall
   39 face stall
   40 while [ patch-here != [patch-here] of stall ]
   41    [ forward 0.005 * walking-speed ]
      ; remember that I have been to this stall, so I don't come again
      set not-yet-visited not-yet-visited with [ self != stall ]
```

The agent needs to remember where it has been, so there is an additional line after line 41. This line needs some explanation. We want to remove the trader whose stall the agent has visited from the `not-yet-visited` agentset. This can be done with the NetLogo primitive, `with`, which was used before to select shoppers with non-empty shopping lists (line 35). `with` is followed by a condition (here, `[ self != stall ]`): an agentset is created containing a copy of the original but including only those agents for which the condition is true. In this case, the condition uses the special name, `self`, which refers to the agent for which the condition is being tested. So the condition in square brackets can be translated as meaning ‘select only those stalls from the `not-yet-visited` agentset that are not equal to the current market stall’. While this may be difficult to follow at first, filtering agentsets in this way can be very powerful and concise.

That is all that seems to be needed to allow agents to remember where they have been and to avoid returning to the same stall again. However, under certain circumstances, the code produces an error (see Figure 2.4).

Figure 2.4 A NetLogo runtime error.



```
    ; give each shopper a random list of produce to buy
    set shopping-list n-of (1 + random 8) fruit-and-veg
  ]

  set mean-items mean [ length shopping-list] of shoppers

  reset-ticks
end

to go

  ; for each  shopper in turn that still has something to buy
```

Runtime Error

```
FACE expected input to be an agent but got NOBODY instead.
error while shopper 16 running FACE
  called by procedure GO
  called by Button 'go'
```

```
    ; go to that stall
    face stall
    while [ patch-here != [patch-here] of stall ] [ forward 0.005 * walking-speed ]

    ; remember that I have been to this stall, so I don't come again
    set not-yet-visited not-yet-visited with [ self != stall ]

    ; buy everything on my shopping-list that is for sale
    ; at this stall

    let purchases filter [ member? ? [stock] of stall] shopping-list
    foreach purchases [
      set spent spent + item (position ? [stock] of stall) [ prices ] of stall
      ; delete the items that have been bought from the shopping list
      set shopping-list remove ? shopping-list
    ]

    ; when shopping is done, go home
```

Getting errors such as this one is an almost inevitable part of programming. To find the source of the problem requires working through the program, command by command. It is part of the all-important process of verification, mentioned above. It is often useful to add commands to display intermediate values, such as the `show` command (which prints values in the command centre that is visible at the bottom of the interface, Figure 2.1)

In this case, the error arose because the agent had already visited all the stalls in the market, but still had items on its shopping list. For such an agent, the `not-yet-visited` agentset is empty – in NetLogo parlance, it contains `nobody`. This must be because the items the agent has not yet managed to buy are in fact not sold by any trader (remember that each trader sells a random selection of items, and sometimes no trader selects one of the fruits or vegetables in the `fruit-and-veg` list). So the variable `stall` has the value `nobody`, the agent tries to face in the direction of `nobody`, and NetLogo complains and halts the program.

What can we do? One possibility is to arrange matters so that all the kinds of produce are sold by at least one trader. Another is to allow the shopper to give up if it cannot find a trader to sell it the remaining items on its list. The latter seems more useful, so the following code is added after line 43:

```
42 ; choose a stall
43 let stall one-of not-yet-visited
   if stall = nobody [
     show (word "No one sells " shopping-list)
     set shopping-list []
     stop
   ]
```

Notice here the use of the `show` command, and the reporter `word`, which joins two or more things together to form one string of characters. This might print something like `(shopper 12): "No one sells [lettuce]"` in the ‘command center’.

In this third version, we have shown how to use the `with` keyword to create tailored sets of agents, and how to deal with runtime errors. In the final enhancement, we shall explain more about procedures in NetLogo.

Final enhancement: more economically rational agents

While the agents now do not keep going back to the same trader, they still behave in a rather implausible way, choosing stalls randomly. It would be more interesting if they were able to plan ahead and buy from the stalls that would give them the best price. (Recall that each trader offers its stock at its own price; there is no uniform market price in this model). The agents should view the pricelists of the traders, calculate what their shopping lists would cost if they bought from those traders and then go for the cheapest. In principle, it would be possible for agents to do this exhaustively, checking every price list, but we will also assume that the shoppers have limited time and energy and so will only check a few traders.

In this version of the model, before an agent buys anything, it tries out one or more potential purchases from traders. For example, it sees what would be the cost of buying its shopping list from trader 1, trader 10 and trader 5, and compares that with the cost of buying the same shopping list from, for instance, trader 3, trader 2 and trader 7. It does this for `n-scans` different sets of traders, selecting the set that gives the cheapest outcome. Then it actually buys what is on its shopping list from that cheapest set.

We can think of this as giving the agent some additional 'intelligence' so that it can plan ahead. Clearly, to minimise the cost of the purchases, the agent could find the prices at every stall, and engage in some calculation to minimise its outgoings (although this might be quite complicated and time consuming, since not all traders sell all the kinds of produce, so the shopper would have to find the trader with the minimum price for each item on the shopping list and go to that trader's stall). We shall assume that such an optimising strategy, which requires knowledge of every trader's prices for every kind of produce (i.e. perfect information) and potentially visiting as many traders as there are items on the shopping list (i.e. unbounded time and energy) is not possible for these agents. Instead, we make them boundedly rational: able to visit only a few stalls and to compute only a few possible permutations of where to buy what.

To implement this, we write an additional procedure. This procedure, called `search-before-buying`, will be similar to the code used before, except that the agent will just note the potential cost of its purchases, rather than actually buying. It does this a number of times (as set by a slider on the interface called `n-scans`) and returns to the main section of code a list of the stalls that yields the lowest total price of all that it has tried. The `go` procedure has then just to visit those stalls in order.

Procedures are of two kinds: those, such as the `go` procedure we have seen already, which carry out a set of commands, and those, called reporters, that compute and return a value. `search-before-buying` is of the second kind, because it returns a list of the best stalls to buy from. A reporter must start with the keyword `to-report` and must have within it the command `report` followed by some value, which it returns to the calling procedure. For example, we could write: `let route search-before-buying` and the variable `route` would then be set to the best list of stalls as calculated by `search-before-buying`.

It is often more convenient and clearer to extract distinct parts of a program and put them into a procedure. Another example is the code to look up the price of product from a stall. In the previous version of the program, this was written as `item (position ? [stock] of stall) [ prices ] of stall`. It takes a little thought to work out what that code does. It is clearer if you create a new reporter with a helpful name:

```
to-report produce-price [ produce stall ]
  report item (position produce [stock] of stall) [ prices ] of stall
end
```

The items after the procedure name in square brackets (`[ produce stall ]`) are the procedure's arguments. When the procedure is called, the appropriate values are passed to the procedure. For example, we could execute `produce-price “oranges” trader 1` and the procedure would report the price of oranges at trader 1's stall.

These two procedures are shown in Box 2.3, as well as one that reports what from a shopping list can be bought from a particular stall.

Box 2.4 Additional procedures to implement finding the cheapest stalls, under bounded rationality.

```
to-report search-before-buying
  ; see how much it would cost to purchase using n-scans sequences
  ; of traders' stalls and report the cheapest
  ; initialise cheapest with a very large number
  ; so every purchase will be cheaper
  let cheapest-price 100000
  let cheapest-route []
  repeat n-scans [
    let this-route []
    let cost 0
    let to-buy shopping-list
    let visited []
    while [ not empty? to-buy] [
      let stall one-of traders with [ not member? self visited ]
      if stall = nobody [
      show (word "Trying to buy " to-buy ", but no trader sells it.")
      set shopping-list []
      report []
    ]
    set visited lput stall visited
    let purchases buy-from-stall to-buy stall
    if not empty? purchases [
      set this-route lput stall this-route
      foreach purchases [
        set cost cost + produce-price ? stall
        ; delete the items that have been bought
        set to-buy remove ? to-buy
        ]
      ]
    ]
    if cost < cheapest-price [
      set cheapest-price cost
      set cheapest-route this-route
    ]
  ]
  report cheapest-route
end
to-report produce-price [ produce stall ]
  report item (position produce [stock] of stall) [ prices ] of stall
end
to-report buy-from-stall [ what-to-buy stall ]
  report filter [ member? ? [stock] of stall] what-to-buy
end
```

The `go` procedure has to be amended to get the best route through the market stalls and to follow that route:

```
ask shoppers [
  let route search-before-buying
  foreach route [
    let stall ?
    ; go to that stall
    face stall
```

The rest of the code is the same as before (although we can if we wish use the `produce-price` and `buy-from-stall` reporters to make the code more readable).

Running experiments

The amount of scanning that agents do to find the cheapest stalls should be correlated with a reduction in the average price that they pay: the more they search, the more likely that they will find the optimum set of prices. However, it is not easy to see this by running the model once for each of several values of the `n-scans` slider. This is because the prices of the fruit and vegetables, the mark-ups that the traders apply, the range of stock that each stall has, the shoppers' lists of what to buy and the choices of stalls to visit, all vary randomly from run to run, and all affect the average price paid. This is typical of many agent-based models. They are stochastic, with behaviour that varies randomly. The usual method of assessing such models is to run them many times and take the average of the outcomes. (How many is enough runs to yield a reliable average can be complicated to work out, but as a guide, it is useful to calculate the standard deviation of the mean and continue the runs until this is no longer reducing).

Let us assume that 100 runs is enough. We should run the model 100 times with the agents each scanning one route, find the average price of the agents' shopping lists, then repeat for another 100 times with the agents scanning for the best of two routes, and so on. This will clearly require running the model several hundred times and it would be impracticable to do this manually, pressing the setup and go buttons and calculating the average of the runs hundreds of times. Fortunately, NetLogo can do the work for us. Since both `setup` and `go` are procedures, we can write some additional code that calls these procedures and calculates the desired mean of the average cost of the shopping baskets (see Box 2.5).

Box 2.5: Code to run an experiment to test the effect of varying the number of times agents scan for the cheapest combination of stalls to purchase from.

```
to run-experiment
  set n-scans 1
  let runs-per-trial 100
  while [ n-scans <= 10 ] [
    let total-of-averages 0
    repeat runs-per-trial [
      setup
      go
      set total-of-averages total-of-averages + mean [ spent ] of shoppers
    ]
    show (word "Mean of average of cost of shopping lists over "
runs-per-trial " runs for " n-scans " scans = " (total-of-averages /
runs-per-trial))
    set n-scans n-scans + 1
  ]
  show "Finished.”
```

The `run-experiment` procedure can either be executed by typing `run-experiment` into the Command Center and pressing the Return key, or a button can be added to the interface with `run-experiment` as the command to run when the button is pressed (this is similar to the way that the `setup` and `go` procedures are run when their buttons are pressed). Code like `run-experiment` can also save the results to a file for later analysis using a statistical program, and this is the approach used in many of the examples in this book.

An alternative approach to use when many runs are needed is to restructure the program so that all the initialisation is done from the `go` command. NetLogo also provides a tool called BehaviorSpace that can run such experiments without writing code and often this is easier to use than writing code such as in Box 2.5, but it is less flexible and has not been used for this book.

Discussion

This Chapter has aimed to provide an idea of what the internals of an agent-based model looks like, as well as providing an introduction to some of the most common constructs and commands in NetLogo. Although the example we have dissected is rather simple (as it had to be in order that it could be explained within the confines of one chapter), it does illustrate the characteristics of agent-based models that were mentioned in Chapter 1. The agents were heterogeneous: the range of fruit and vegetables stocked differed between market traders and every shopper had a different shopping list. Consequently, the behaviour of each agent differed. The dynamics of the system were important: there is no notion of equilibrium in the model, but we can observe agents proceeding through time to purchase their shopping. Thirdly, the agents interacted: shoppers bought from traders. Furthermore, agents were endowed with some rationality that allowed them to attempt to optimise, but within constraints imposed by limited information and limited resources of time and effort.

At the end of each of the following chapters, there are appendices that summarise the models introduced in the chapter using a standard structure and a listing in ‘pseudo code’, a version of the program code that is closer to natural language. This Chapter does the same, to show for the example that we have described in detail what the pseudo code version looks like.

References

Gilbert, N. (2007) *Agent-based models*. London: Sage.

Railsback, S. F. & Grimm, V. (2011) *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton : Princeton University Press.

Wilensky, U. (1999) *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Online] Available at: http://ccl.northwestern.edu/netlogo/ [Accessed 31 January 2015].

Wilensky, U. & Rand, W. (in press). *An introduction to agent-based modeling: Modeling natural, social and engineered complex systems with NetLogo*. Cambridge, MA: MIT Press.

Appendix to Chapter 2: the example model – full version

Purpose: The aim of the model is to illustrate some of the basic features of NetLogo using a simple model of a fruit and vegetable market as an example.

Entities: There are two types of agents: shoppers and traders.

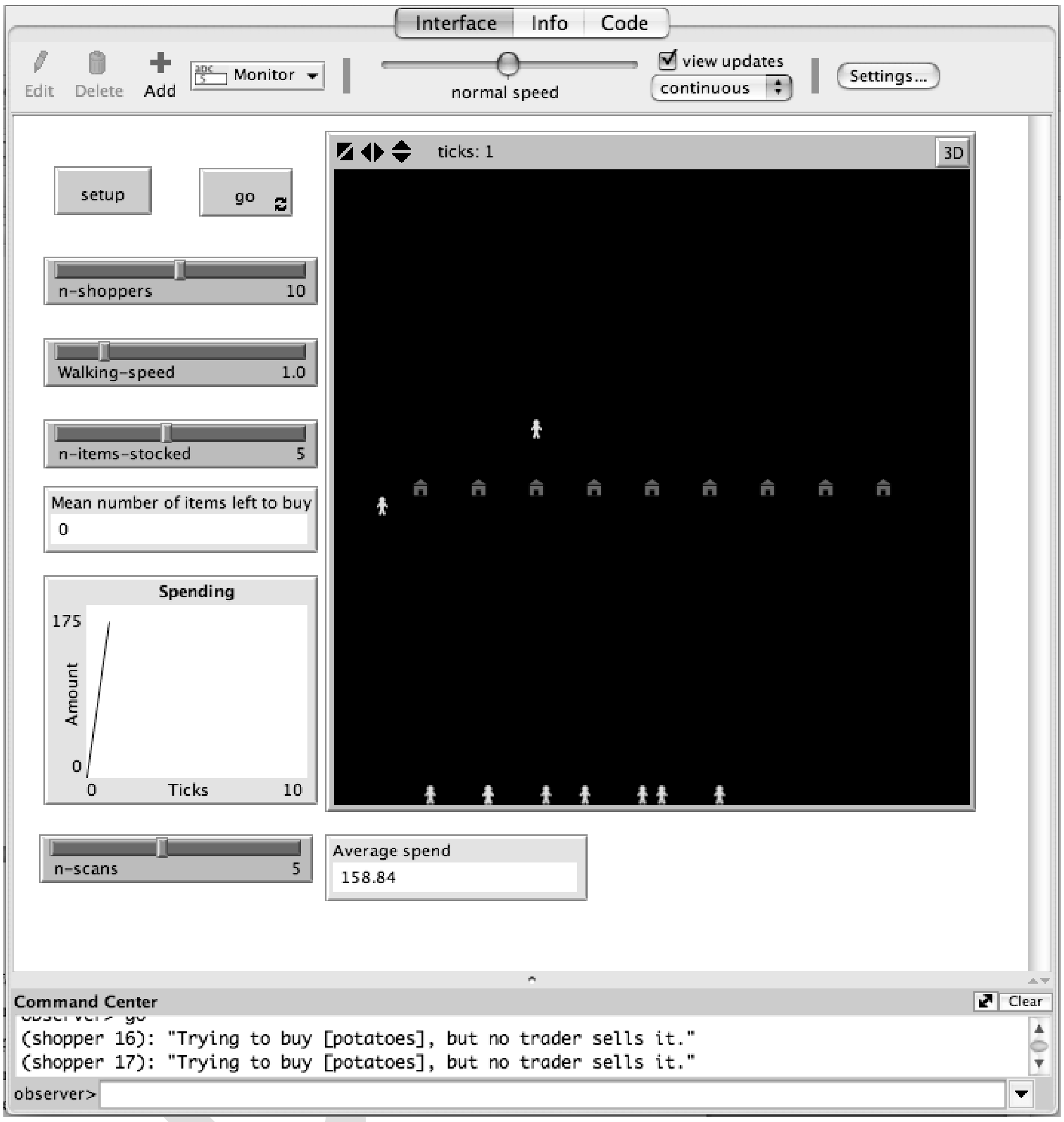
Stochastic processes: The items in the shoppers' lists, the items stocked by traders and prices charged by traders are selected randomly (within set limits).

Initialisation: For shoppers, select the number of shoppers, the speed at which they walk and the number of alternative buying options they are willing to consider. For traders, select the number of items stocked.

Output: A graph showing the amount spent over time and two reporters: the mean number of items left to buy and the average spend.

A screenshot is shown in Figure A2.1. The pseudo-code is in Box A2.1.

Figure A2.1: Screenshot of the interface of the final version of the model.



n-shoppers 10

Walking-speed 1.0

n-items-stocked 5

Mean number of items left to buy
0

n-scans 5

Average spend
158.84

```
(shopper 16): "Trying to buy [potatoes], but no trader sells it."
(shopper 17): "Trying to buy [potatoes], but no trader sells it."
observer>
```

Box A2.1: Pseudo-code for the final version of the model.

```
Set the list of 12 fruit and vegetables the traders are to sell.
Set the (wholesale) prices of the items randomly between 1 and 100.

Generate 9 agents to represent traders with the shape 'house', coloured red and placed in a line across the middle.
Allocate randomly to each trader:
        Stock: which of the 12 possible items are stocked, given that the number of items to be stocked is set by the slider.
        Prices: based on wholesale prices plus a random mark-up of between 1 and 30 per cent.

Generate the number of agents selected with the slider to represent shoppers with the shape 'person', coloured yellow, and distributed randomly.
Allocate three attributes each shopper:
        A shopping list of between 1 and 8 items, selected randomly.
        A list of traders not yet visited: initially this is all traders
        The amount spent: initially this is nil.

Calculate the mean length of the shoppers' lists.

Each shopper goes shopping by:
        Scanning the stalls for the cheapest sequence of stalls. It does this the number of times selected.
        Visiting the selected stalls:
                Ensuring that no stall is visited twice
                Recording what is bought
                Adding up how much is spent.

When the shopping list is empty, the shopper goes home.

Report the results and plot the graph.
```