Introduction to NetLogo

3.1. Introduction

NetLogo is a programming environment which allows for the construction and exploration of agent-based models. Developed at the Center for Connected Learning, the software currently draws from StarLogoT[1], which is available for Mac OSX, and StarLogo[2], which was developed at MIT's Media Laboratory. It is the latter that has had the greatest influence on the programming language used by NetLogo, known as Logo[3], which was itself inspired by the Lisp programming language family. The history of Logo allows for a partial understanding of NetLogo's philosophy.

3.1.1. *A little history*

When NetLogo is presented in workshops, the first advantage expressed is that of how easy it is to pick up and use. This stems both from its graphical interface and the programming language used,

Chapter written by Frédéric AMBLARD, Eric DAUDÉ, Benoît GAUDOU, Arnaud GRIGNARD, Guillaume HUTZLER, Christophe LANG, Nicolas MARILLEAU, Jean-Marc NICOD, David SHEEREN and Patrick TAILLANDIER.

1 http://ccl.northwestern.edu/cm/StarLogoT/.

2 http://education.mit.edu/starlogo/.

3 http://el.media.mit.edu/logo-foundation/index.html.

which is known as Logo. This language was created in 1967 by a collaborative effort between Wallace Feurzeig and Seymour Papert. Papert was largely inspired by the constructivism of Jean Pigaet with whom he had worked several years before. With the advent of the first computers in the 1980s, researchers began to ask themselves about their utility within an educational context: how were computers to be used to enable the teaching of dynamic and complex worlds? Computers needed to be a medium that bridged the difference between the learner's need for knowledge and the world to be explored, which implied a need for a language permitting the learner and the computer to communicate. However, none of the programming languages available at the time were adapted for use by young learners. It is from the observation of this fact that the premise for Logo came about, a language that was intuitive for the learner and close to natural language as a result of being interactive, modular and flexible.

Logo is an interpreted programming language, which means that each line (containing a particular command) inputted by the user is immediately executed. These commands are interpreted by Logo as an order (e.g. the `jump 10` command results in the turtle moving by 10 steps), which will send back an error message if the command cannot be carried out. The language is modular, which means that commands can be grouped to form more complex sequences that can be made into new terms and are combined to form the complete program. This modularity allows for the construction of large projects. Finally, Logo is flexible as it does not require the direct input of the figures used. Type assignment is done based on the data used in the instructions. Even though this might be slightly perturbing for those used to other programming languages, the choice to not require direct input was made as it is closer to the way that non-programmers think. This choice is also found in other programming languages such as Caml[4] or Python[5].

There is only one single syntactic rule used in Logo: that of prefix notation. A command must always be placed before any eventual

4 http://caml.inria.fr/caml-light/index.en.html.

5 https://www.python.org/.

variables. Thus, the `jump` command followed by the variable `10` which makes the turtle move by 10 steps is written as follows: `jump 10`. To this rule is added that of the left–right analysis of instructions. Therefore, for the `jump random sqrt 4` command, the evaluator begins by reading the `jump` command, which receives the single variable `random`, which receives the single variable `sqrt`, which, in turn, receives the single variable 4. In practice, this leads to first executing the square root of 4, with the other instructions being put on hold, while the value of their respective variables is being calculated. Next, the `random` command is executed with the value 2 as a variable. Finally, the `jump` command is executed resulting in a jump of a number of steps corresponding to the result given by the `random` command with 2 as its input variable.

3.1.2. *Purpose of the chapter*

This chapter aims to help readers new to programming to discover the NetLogo modeling and simulation platform in an educational manner. The final aim is for such an individual to be able to write their own models and simulate them within NetLogo by the end of the chapter. So as to accomplish this, the reader will be instructed in the development of a simple model which will be expanded upon later. This first experience of developing a multiagent model is simplistic but remains nonetheless complete and enables us to understand and grasp the main concepts and techniques useful for defining models and simulations.

The remaining chapter is organized into five sections. In the first section, we will see that, using its metamodel, the NetLogo platform is particularly well-adapted for representing spatial phenomena. In the second section, we will present the tool interface used in NetLogo. In the third section, we will develop a simple model step by step, while referring to the metamodel viewed previously. The fourth section introduces the interaction model used for the behaviors of the agents. Before concluding, we will put forward a brief presentation of the additional functionalities offered by NetLogo.

3.2. Metamodel of NetLogo

The NetLogo platform corresponds to a simulation approach said to be “in time-discrete intervals”, which means that it makes a collective group of entities evolve in successive time intervals of equal length[6]. The corresponding modeling approach therefore consists of identifying the entities that are to be incorporated into the model and then defining the behavior of each one across each time interval. This approach is centered on the entities involved, otherwise known as agents. NetLogo’s metamodel identifies three different types of entities which can be modeled: 1) the environment: this is a rectangular space modeled in the form of a regular grid of n x m square tiles (*patches*); 2) the mobile agents (*turtles*): these move within the environment and interact with it and each other; 3) the links: these are dynamically created inbetween the *turtle* agents.

Finally, there is a specific agent known as the *observer*, which exists outside of the model. Its role is to control and monitor the execution of the simulation. This agent creates all the entities within the model (*patches*, *turtles and link*s) and controls their simulated behavior.

NetLogo proposes a particularly useful functionality for the manipulation of agent-based models. It allows us to give instructions to groups or subgroups of agents, or *agentset*s. As a result, it is possible to collectively control all of the *turtles* as well as to pick a smaller subset and give it instructions that are not followed by the remaining agents. So as to do this, it is possible to create species or *breeds* which can be manipulated as groups of agents.

3.2.1. *Patches*

The environment is a rectangular space made up of a grid of n x m squares that are known as *patches*. Each *patch* corresponds to a square of the grid, within which movement is impossible. Each *patch* also has a corresponding position within the two-dimensional (2D) space of the

6 In NetLogo, a unit of time is known as a `tick`.

environment. All *patches* are also autonomous agents with their own state and behavior that is independent of that of the surrounding *patches*.

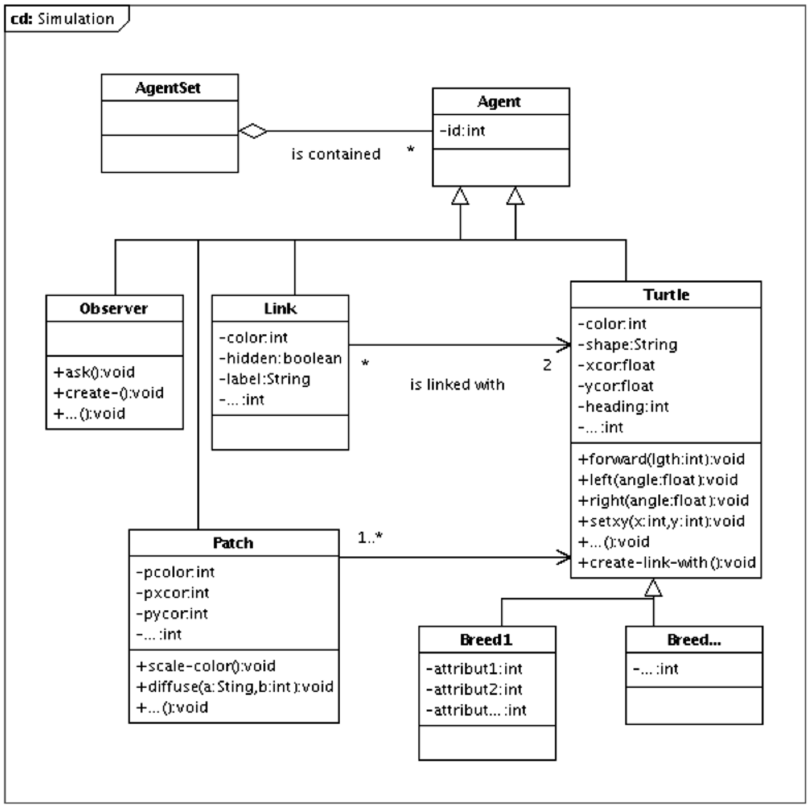


Figure 3.1. *A NetLogo model's corresponding metamodel*

3.2.1.1. *Topology of the environment*

The modeler has the possibility, by use of the interface, to choose the topology of the environment, by determining whether the grid should be horizontally or vertically continuous. To do this, all that is needed is to edit the large black square on the *Interface* tab. Making the grid horizontally continuous means that the first and last columns of *patches* are effectively next to each other. Therefore, two *patches* belonging, respectively, to the first and last columns of a single line become neighbors. In the same manner, making the grid vertically continuous means that the first and last lines of *patches* become adjacent. By default, the environment is continuous along both axes, which corresponds to a toroidal topology. When the environment is only continuous along a single axis, its topology is a cylinder, either horizontally or vertically oriented. If the environment is not continuous along either axis, then its topography is that of an enclosed, 2D square.

This same interface also allows the user to divide the simulation space into *patches* of different sizes, effectively changing the resolution used. It must, however, be noted that all *patches* must be rectangular in shape. If the modeler requests a higher number of *patches* along a particular axis, then the space is not sectioned into smaller segments but instead, the size of the environment is increased along the axis in question. This leads to a simulated environment that is rectangular in shape.

3.2.1.2. *Patch variables*

The state of each *patch* is defined using a certain number of predefined variables to which any number of additional variables can be added.

The most commonly used predefined variables are the `pcolor` variable, which defines the color of the *patch* and the `pxcor` and `pycor` variables, which, respectively, define its *x* and *y* coordinates. Since the position of the *patch* is fixed within the grid, its coordinates can only be read, not edited. On the other hand, their color can be modified. Other than their position and color, *patches* can be linked with a text label (`plabel`) to which a numerical value or string can be added, and whose color can also be defined (`plabel-color`). The size of the *patches* and the size of their labels can only be indirectly modified via use of the interface.

3.2.1.3. *Patch primitives*

Patches can be manipulated using primitives defined within NetLogo. Below are some of the most commonly used primitives:

– `neighbors`: enables access to its neighbors;

– `distance`: returns the distance between the agent that called the function and another agent given as a variable;

– `sprout-<breeds>`: creates a certain number of agents of the `breeds` species on the patch that called the command;

– `diffuse`: this command is a little particular as it is a primitive of the *observer*. It allows for the spreading of variables to its neighbors; etc.

3.2.2. *Turtles*

The *turtles* are the mobile agents in the simulation. They are designed to move around the environment and therefore on and across the *patches*. They are spatially located within the environment and are visible on the grid. The *turtles* can view their environment and the other agents within it. They have an action capacity, a characteristic that is essential for an agent. It is possible to define more detailed agent types than the simple *turtle*s with the keyword `breed`. They can then be assigned specific attributes and a unique behavior.

3.2.2.1. *Turtle variables*

Just as with the *patches*, the state of each *turtle* is defined using a certain number of predefined variables to which any number of additional variables can be added. The *turtles* share certain variables with the *patches* such as color (`color`) as well as having their own location coordinates on the *patch* grid (`xcor` and `ycor`). Equally, labels can be assigned to them, as well as the color of these (`label` and `label-color`). The `size` variable allows us to modify the *turtle's* size and `who` returns the *turtle's* identity (`id`).

3.2.2.2. *Turtle primitives*

Turtles can be manipulated using primitives defined within NetLogo. Here again, some of the most commonly used variables are listed below:

– `distance`: returns the distance between the *turtle* that called the function and another *turtle* given as a variable;

– `die`: kills the *turtle*;

– `hatch`: creates a given number of *turtles* that are daughters of the selected turtle. The children are created identical to the mother and are placed at the same location as it;

– `forward`: moves the *turtle* forward a given number of steps;

– `move-to`: the *turtle* moves to the location of an agent given as a variable;

– `left, right`: allows the *turtle* to turn left or right, respectively, by a given number of degrees;

– `<breeds>-here`: returns a group of agents containing the *turtles* that are on the patch of the agent that called the command; etc.

3.2.3. *Links*

The *links* are also agents but they have the particular function of linking two *turtles* together. The *turtles* are then known as nodes. The *link* is clearly represented by a line linking the two *turtles*. As a result of this, *links* are not located on the *patch* grid. There are two types of *links*: directed and undirected. Just as with the *turtles*, the modeler can define their own *link* types.

3.2.3.1. *Link variables*

As with the other agents, the state of each *link* is defined using predefined variables to which any number of additional variables can be added. Again, certain variables described earlier can also be used: `color`, `label` and `label-color`. Also useful are `end1` and `end2` which describe the nodes at either end of the *link*.

3.2.3.2. *Link primitives*

Links can be manipulated using primitives defined within NetLogo. Once again, some of the most commonly used primitives are listed below:

– `create-links-to, create-links-from, create-links-with`: these are different ways of creating *links*;

– `link-with`: returns the *link* between the *turtl*e calling the function and another *turtle* given as a variable;

– `my-links`: returns a list of all the undirected *links* that are connected to the *turtle* calling the function;

– `link-neighbors`: returns a group of agents. It contains all the *turtles* found at the other end of *links* connected to the *turtle* calling the function;

– `my-in-links`: returns a group of agents. It contains all the directed links leading to the *turtle* calling the function;

– `my-out-links`: returns a group of agents. It contains all the directed links leaving from the *turtle* calling the function; etc.

3.2.4. *The observer*

The *observer* is located outside the simulation space and controls and monitors its progress. It allows for the sending of instructions to all the agents in the simulation. It is the link between the user and the agents.

3.3. The NetLogo software interface

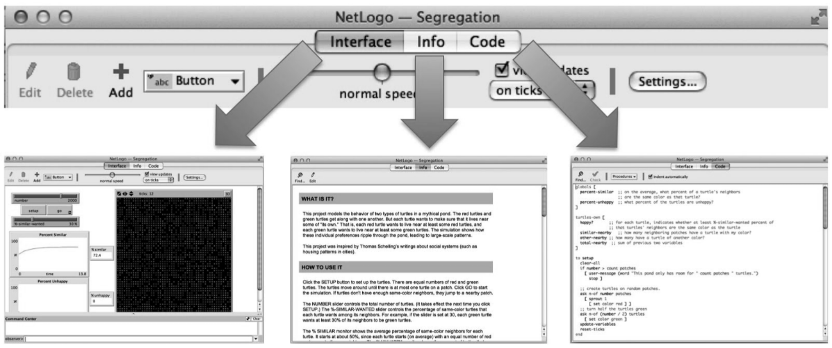
The NetLogo interface is centered around three tabs: *Interface*, *Information* and *Code*. This allows for rapid switching between the development and simulation aspects of a model, and therefore allows for an incremental modeling process which alternates between development and testing phases.

It is these three tabs (Interface, Info and Code) that the user is presented with upon first launching the NetLogo software application, all of which have specific roles within the modeling process.

The *Info* tab is used to document the model. NetLogo provides a basic framework which can be modified (with the *edit* button) to be perfectly suited to the needs of the modeler. The description can, for example, be an Overview, Design concepts, Details (ODD) description of the model. It should be noted that this documentation is saved along with the model and will, as such, be transmitted along with it.

The *Code* tab mainly contains an editable text field within which the modeler writes their NetLogo code. Notably, it contains a *Check* button which is activated whenever the file is saved. It allows for the detection of syntactic mistakes in the NetLogo code entered in the text field. This check also occurs automatically whenever the user switches between tabs. If a mistake is discovered, a yellow band appears above the tab. The tab also contains a scrollable list that contains the different procedures contained in the model, for quick and easy access.

Finally, the *Interface* tab contains the graphical interface of the simulator. When a new model is created, a default environment is displayed. This environment can then be easily modified by the modeler, who can add control elements to the simulation (such as *buttons*), elements which allow for control more than various input values of the simulation variables (*sliders*, *switches*, *choosers*, etc.) or to display returned values and other simulation indicators (*monitors*, *plots*, etc.). At the lower end of the *interface,* tab can be found the *Command Center*, which contains the console that displays all of the messages produced during the simulation. However, this is not the extent of its utility; the user may also choose to use it to execute NetLogo code on the spot, which most notably allows for the testing of a currently running simulation. They may also choose within what context this code is executed, determining which specific entities (*observer*, *turtles*, *patches* or *links*) are to be affected by it.



WHAT IS IT?

This project models the behavior of two types of turtles in a mythical pond. The red turtles and green turtles get along with one another. But each turtle wants to make sure that it lives near some of "its own." That is, each red turtle wants to live near at least some red turtles, and each green turtle wants to live near at least some green turtles. The simulation shows how these individual preferences ripple through the pond, leading to large-scale patterns.

This project was inspired by Thomas Schelling's writings about social systems (such as housing patterns in cities).

HOW TO USE IT

Click the SETUP button to set up the turtles. There are equal numbers of red and green turtles. The turtles move around until there is at most one turtle on a patch. Click GO to start the simulation. If turtles don't have enough same-color neighbors, they jump to a nearby patch.

The NUMBER slider controls the total number of turtles. (It takes effect the next time you click SETUP.) The %-SIMILAR-WANTED slider controls the percentage of same-color turtles that each turtle wants among its neighbors. For example, if the slider is set at 30, each green turtle wants at least 30% of its neighbors to be green turtles.

The % SIMILAR monitor shows the average percentage of same-color neighbors for each turtle. It starts at about 50%, since each turtle starts (on average) with an equal number of red

```
globals [
  percent-similar  ;; on the average, what percent of a turtle's neighbors
                   ;; are the same color as that turtle?
  percent-unhappy  ;; what percent of the turtles are unhappy?
]

turtles-own [
  happy?       ;; for each turtle, indicates whether at least %-similar-wanted percent of
               ;; that turtles' neighbors are the same color as the turtle
  similar-nearby   ;; how many neighboring patches have a turtle with my color?
  other-nearby ;; how many have a turtle of another color?
  total-nearby  ;; sum of previous two variables
]

to setup
  clear-all
  if number > count patches
    [ user-message (word "This pond only has room for " count patches " turtles.")
      stop ]

  ;; create turtles on random patches.
  ask n-of number patches
    [ sprout 1
      [ set color red ] ]
  ;; turn half the turtles green
  ask n-of (number / 2) turtles
    [ set color green ]
  update-variables
  reset-ticks
end
```

Figure 3.2. *The NetLogo software interface, with details of each of the three tabs*

3.4. Step-by-step creation of a simple model

A NetLogo program is a sequence of procedures which permits the simulation of the behaviors of created objects (agents and environment) during their execution. As seen in the previous chapter, a multi-agent system (MAS) is characterized by a static architecture (the environment and species) and individual behaviors which determine the dynamics of the system, the interactions therein and, as a result, can show emergent behavior. The dynamics of the system are largely determined by the initial situation which can be calibrated in order to receive interesting results.

Therefore, the development of a NetLogo model is centered around defining the following elements:

1) the model's structure, which involves defining the model's global variables, the structure of the environment and the species within it;

2) an initial state, with variables defined in the user interface;

3) the behaviors of the agents and environment using procedures;

4) the model's outputs.

Figure 3.3 proposes a suggested setup of a NetLogo model, which is as follows:

– expressing and defining the structure of the world to be simulated, of the global variables and species;

– defining an initial state described within a set of procedures with the `setup` prefix (of which there should be one per species). These procedures are called in by a general procedure conventionally named as `setup`;

– defining the agents' behaviors;

– defining the agents' lifecycle and of the environment within a set of procedures with the go prefix (again, there should be one per species). These procedures are called in by a general procedure conventionally named go;

– defining procedures that will be used for the model's outputs, notably in the form of charts and/or tables.

COMMENT 3.1.– The NetLogo language can be expanded by means of extensions (e.g. for the manipulation of geographical information systems (GIS) data). It is mandatory to declare the extensions used when creating a program, prior to defining global variables and species. The program can then be organized in a logical manner, based on the temporal order in which various elements will be executed when running the program. This begins with the initialization procedure, followed by the actions of the various agents, the sequencing within each iteration of the simulated world and finishes with the outputs, such as charts, tables or other graphical representations of data.

In the case of a large model, it is possible to split the model into several files. The `includes` primitive adds new tabs which allow for quick access to any additional files included in the model. This primitive must link to files in the NetLogo file extension format (.nls).

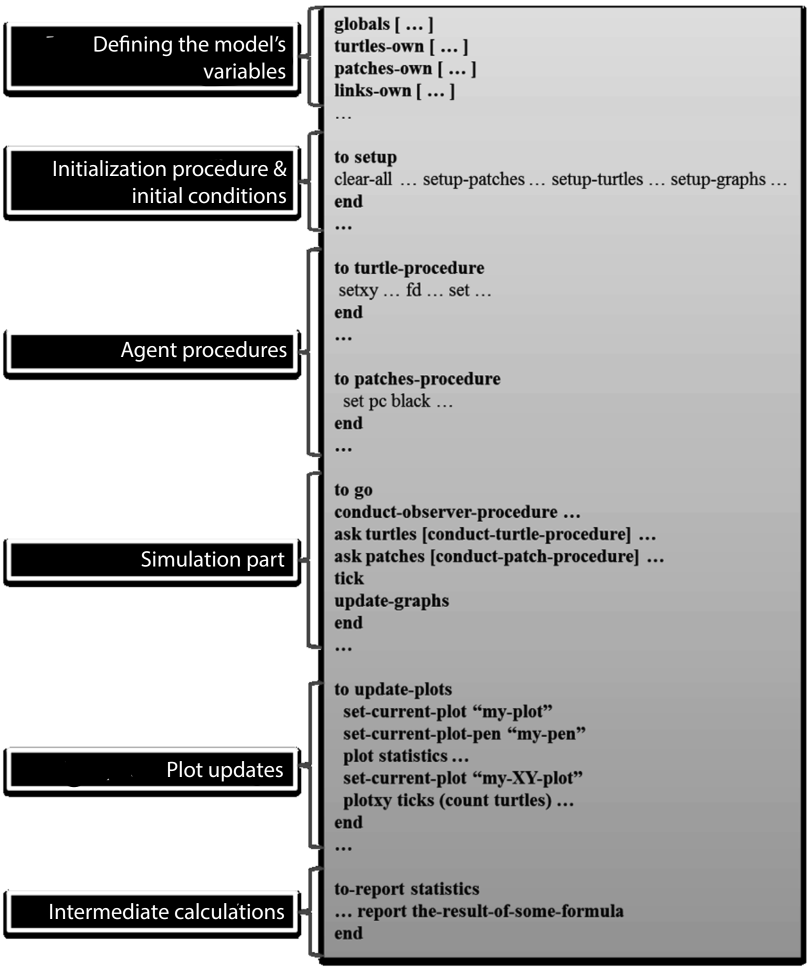
3.4.1. *Creating the structure of the world and defining its initial state*

The modeler's first task is to create and initialize the world which will function as an environment for the simulation's agents. This is done both with the use of NetLogo's graphical interface and by writing code.

3.4.1.1. *Defining the simulation space*

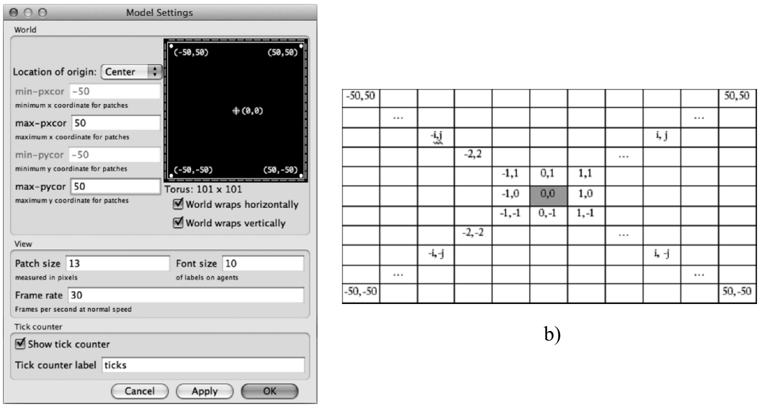
NetLogo allows for the creation of a single modeling and simulation space per case study. It is this space that can be edited from the *Model Settings* window (Figure 3.1). The topology of the simulated area (*World wraps*), the origin of the x and y coordinates (*location of origin*) and its dimensions (`pxcor` and `pycor`) are selected from this window. In the example of Figure 3.4(a), the origin $(0, 0)$ is placed in the center of the space, its topology is toroidal and the space is made up of 10,201 *patches* spread over 101 lines and 101 columns.

A *patch* is characterized by its position in the simulation space which is identified from the coordinate system chosen. Figure 3.4(b) represents the layout of the environment's *patches* in the form of a matrix; the *patch* $(-4, 5)$ corresponds to the cell with coordinates $x = -4$ and $y = 5$ in the matrix.


Defining the model's variables
Initialization procedure & initial conditions
Agent procedures
Simulation part
Plot updates
Intermediate calculations

```
globals [ … ]
turtles-own [ … ]
patches-own [ … ]
links-own [ … ]
…

to setup
clear-all … setup-patches … setup-turtles … setup-graphs …
end
…

to turtle-procedure
 setxy … fd … set …
end
…

to patches-procedure
  set pc black …
end
…

to go
conduct-observer-procedure …
ask turtles [conduct-turtle-procedure] …
ask patches [conduct-patch-procedure] …
tick
update-graphs
end
…

to update-plots
  set-current-plot "my-plot"
  set-current-plot-pen "my-pen"
  plot statistics …
  set-current-plot "my-XY-plot"
  plotxy ticks (count turtles) …
end
…

to-report statistics
… report the-result-of-some-formula
end
```

Figure 3.3. *Structure of a NetLogo model*


Model Settings

World

Location of origin: Center

min-pxcor -50
minimum x coordinate for patches

max-pxcor 50
maximum x coordinate for patches

min-pycor -50
minimum y coordinate for patches

max-pycor 50
maximum y coordinate for patches

Torus: 101 x 101

☑ World wraps horizontally

☑ World wraps vertically

View

Patch size 13
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

☑ Show tick counter

Tick counter label ticks

-50,50										50,50
	...								...	
		-i,j						i, j		
			-2,2				...			
				-1,1	0,1	1,1				
				-1,0	0,0	1,0				
				-1,-1	0,-1	1,-1				
			-2,-2				...			
		-i,-j						i, -j		
	...								...	
-50,-50										50,-50

Figure 3.4. *a) World configuration interface and b) representation of the environment in matrix form*

3.4.1.2. *My first procedure*

3.4.1.2.1. Creation of a `setup` button

The only way for a user to launch a simulation, that is to say, to execute any of the model's procedures, is to activate a button within the interface. A right click in any empty space on the interface will bring up a menu containing the various different elements that can be added, from which the user can make their choice. Start by creating a single button, which will prompt a window to open with options for its configuration (Figure 3.5).

The *Commands* text input field allows the modeler to describe the actions to be executed when the button is activated, which is written in NetLogo code. Buttons are most commonly used to call a procedure from the model. For this to happen, the procedure name must be written within this field. In Figure 3.5, clicking on the button launches the `setup` procedure (which we will define in the following paragraph). The button's text appears in red if there are any errors detected in the code assigned to it or within any procedures the button may call. The name of the button corresponds to what is written in the *Display name* field, and will default (if said field is empty) to the text in the *Commands* field.

A procedure is always executed in a specific context (that of the *observer*, the *turtles*, the *links* or the *patches*). For example, executing a procedure in the context of a *turtle* allows for the usage of *turtle*-specific variables (e.g. `color`) within the procedure, the variables that do not perhaps apply to *patches* (which have a `pcolor` variable, but not a `color` one), *links* or the *observer*. The scrollable *Agent* list allows the user to define within which context the *Commands* code is executed: if *patches* is selected, then the code is applied to all the *patches* in the model.

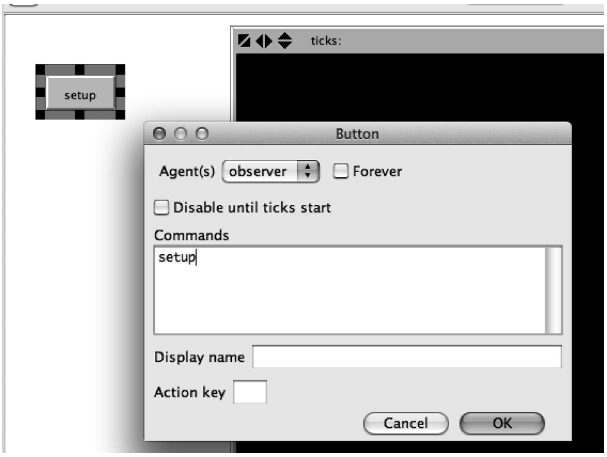

Agent(s): observer
Forever: unchecked
Disable until ticks start: unchecked
Commands: setup
Display name:
Action key:

Figure 3.5. *Button creation*

3.4.1.2.2. The `setup` procedure

Once the `setup` button has been created, the modeler must write the corresponding `setup` procedure. A procedure always has the same form in NetLogo:

```
to nom_procedure
     [NetLogo code instructions or
     calls of procedures defined within the model]
end
```

We want to write a procedure called `setup` which displays the message “model initialization” in the *observer*.

```
to setup
     show "model initialization"
end
```

COMMENT 3.2.– The instruction `show` displays (in the *observer*) the message entered as a variable. This message can be a character string (which is written in between inverted commas), a numerical value or a color, such as `show "my message"`, or `show 3.14`.

A `setup` procedure also always has the same form in NetLogo:

– it begins by setting the simulation back to its initial state, due to the `clear-all` primitive: the values of all variables are set to their defaults (parameters, *patch* variables), all *turtles* and any remaining *links* are killed and all output displays are cleared;

– it initializes the global variables and the state of each *patch* and creates and initializes the different agents;

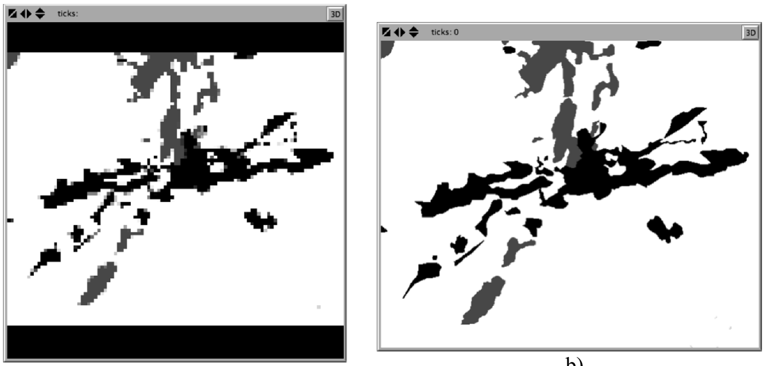
– it resets the tick counter and sets the output displays to their values at the initial state (`reset-ticks` primitive).

```
to setup
     clear-all
     [global variable initialization]
     [agent creation and initialization]
     reset-ticks
end
```

3.4.1.2.3. Initializing the environment: loading an image

One of the tasks of the `setup` procedure is to initialize the environment and therefore to define the initial state of all *patches*. In the example we have been using throughout the text, we will start by initializing the environment to represent the studied region. So as to accomplish this, we have a map of land use in the subregion of Maroua at our disposal, in the form of a raster image named as landuse87NBB.BMP. This image has a resolution of 523 by 424 pixels. Each type of land use is characterized by a particular color. So as to import the data available within this image, we use the `import-pcolors` primitive. This command scans the image, resizes it to the scale of the environment and then assigns each patch the color of the corresponding pixel (`pcolor` variable of the *patches*). Figure 3.6(a) displays the result of importing an image into an environment with a lower resolution, that is to say a lower number of *patches* than there are pixels contained in the image and Figure 3.6(b) shows the result of importing an image of equal resolution to that of the environment.

```
to setup
      ...
      show ‘‘Model initialization’’
      import-pcolors ‘‘landuse87NBB.BMP’’
      ...
end
```



a) b)

Figure 3.6. *Importing a same image into an environment of a) 50 patches by 50 patches and b) of 523 patches by 424 patches*

3.4.1.3. *Diversifying the world*

By default, NetLogo only recognizes three agent types in its simulation models: *patches*, *turtles* and *links*. Yet, it is often useful, or even necessary to be able to create different agent types with specific attributes. In NetLogo, new agents are created using the keywords `breed` (for new *turtle* types) and `directed-link breed` or `undirected-link breed` (for new *link* types). These keywords are assigned by several variables: the name of the new agent group (in the form of an *agentset*) of this species as the first variable and the name of an individual agent of the species as the second variable[7].

As presented in the Unified Modeling Language (UML) diagram of our example, we need to create two different new *turtle* agent types: mosquitoes and humans. The term *mosquito* is therefore the name of the species, and the term *mosquitoes* indicates the collective group of all existing *mosquito* agents. Furthermore, to represent the infection class that links two agents, we have created an additional *link* type: *infection*.

```
breed [mosquitoes mosquito]
breed [humans humain]

directed-link-breed [infections infection]
```

The defining of species also creates global variables: for example, `mosquitoes` will be able to be used within the model and will contain all agents of this species (which would, for example, be useful for ordering all the mosquito agents to move). The term will also be able to be used with primitives: for example, `create-<breeds>` is used to create mosquito agents by replacing `<breeds>` by `mosquitoes`: `create-mosquitoes 50` will create 50 mosquitoes. Many other primitives can adopt this ability, such as `<breeds>-at` and `<breeds>-on`.

It should be noted that each agent type created in this fashion possesses any variables that the current *turtles* own (such as `xcor, ycor, color, heading`, etc.). Similarly, new link types will be given the variables of existing *links*. Nonetheless, it is possible to give additional variables to both the basic agent types in NetLogo (*patch*es, *turtles* and *link*s) and any and all new species created by the modeler. Importantly, when a variable is added to the *turtle* agent group, this variable will also be added to any other existing *turtle* species.

7 Conventionally, the first variable is the pluralized form of the second variable (mosquitoes/mosquito or wolves/wolf as in the classic predator-prey model in the NetLogo model library.

The modeler can add new variables to the *patches*, *turtles* or *links* as well as all other breeds, by using the keywords `patches-own`, `turtles-own`, `links-own` or `<breeds>-own`:

```
patches-own  [pvar1 pvar2 ...]
turtles-own  [var1 var2 ...]
links-own    [lvar1 lvar2 ...]

<breeds>-own [bvar1 bvar2 ...]
```

This command, which will be executed before any other event in the procedure, allows for the creation of additional variables belonging to the *patches* (`pvar1, pvar2`, etc.). Each agent that is created (whether a *patch*, *turtle*, *link* or other species) is as a result given an instance of these variables which can be locally modified, independently from the other agents.

In the example that we have been studying, we wanted all the agents (mosquitoes and humans) to have an infectious state. Therefore, we add an `is-infected?` variable to the *turtles*, and as a result, to all species (in this case, humans and mosquitoes) so as to avoid having to add this variable to both species manually. Nonetheless, the mosquitoes and humans each have their own personal variables: the mosquitoes have the variable `is-infected-external?` which specifies whether a mosquito has been infected by another agent during the simulation or if it was a source of the epidemic, infected prior to the launch of the simulation.

```
turtles-own    [is-infected?]
humans-own     [house work begin-work end-work]
mosquitoes-own [is-infected-external?]

patches-own    [location-home?]

infections-own [date generation]
```

3.4.1.4. *Populating the world*

We will now create and initialize the different agents which we want to be present in the model. When the agents are created with the `setup` procedure, the `create-<breeds>` primitive must be used[8]. This primitive creates new agents of this breed, the number of which is given as a variable, and calls the commands in between the brackets for each of the agents, respectively.

```
create-<breeds> number [
    [grouped commands which apply to all the
    newly created agents]
]
```

These commands usually serve to initialize each of the agents' variables. To modify the value of an *existing* variable (or, more generally speaking, of an individual variable), the modeler uses the `set` primitive which gives the variable (given as a variable of the primitive) the value of an expression:

```
set variable expression
```

The `set` command can only be used for existing variables. It is sometimes useful to define local variables, that is to say variables which exist only for the current procedure, such as to store the results of intermediate calculations, for example. So as to accomplish this, the modeler must use the `let` primitive, which will both create a new variable (given as a variable of the primitive) and assign it the value of the following expression:

```
let variable expression
```

For example, the following code (inspired by our running example) allows for the modeler to write a procedure `init-humains` which creates 50 human agents and gives them each a position, a `color`, a `size` (inherited from the *turtles*) and the `is-infected?` and `home` variables.

8 When creating a new *turtle* breed, the primitive `hatch-<breeds> number [commands]` must be used; on the other hand, if a new *patch* form is being created, the primitive `sprout-<breeds> number [commands]` must be used.

```
to init-humans
     let list-houses patches with [locationHome?]
     create-humans 50
     [
        set home one-of list-houses
        setxy [pxcor] of house [pycor] of house
        set size 24
        set is-infected? false
        set color green
    ]
end
```

For each of the 50 human agents who are created, the program initializes the `size` variable (set to 24), `is-infected?` (set to `false`), which means that the agent was not infected at the start of the simulation, and `color` (set to `green`). To initialize the `home` variable, we begin by selecting all the *patches* which are houses and can be homes:

```
let list-houses patches with [locationHome?])
```

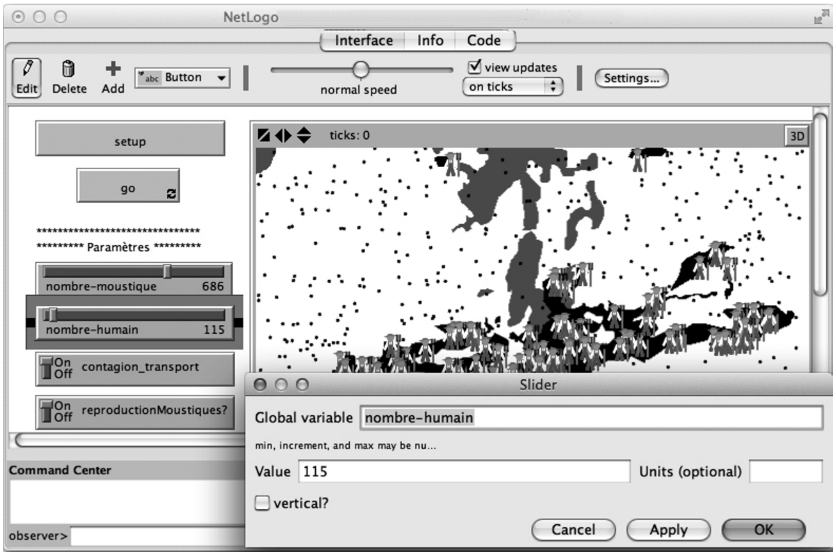
and we store the list of these *patches* within the `list-houses` variable. The `with` variable carries out a search within an *agentset* (a list of agents) so as to put together a new *agentset* containing all the agents that satisfy the expression run in the variable that is to its right (in this case `[locationHome?]`, which is to say that their `locationHome?` variable has the value `true`).

Then, we initialize the `house` variable with respect to one of the *patches* of this *agentset* with the `one-of` primitive.

The `setxy` primitive assigns a value to the agent's position variables (`xcor, ycor`). Here, we give the agent the coordinates of its `home` variable as it is initial position: the `of` primitive returns the value of the variable given as a variable to the left of the agent that is placed on the right (`[pxcor] of home`).

3.4.1.5. *Influencing world creation: modifiable variables*

The modeler may consider that the initial number of *human* agents should be a variable of the model. This may be so that it can be modified by the user during the simulation or so that it can be modified so as to explore the model in full detail. In such cases, the modeler has the option of adding an element to the interface, such as a *slider*, so as to make the variable manually modifiable (Figure 3.7).



view updates ☑ on ticks
normal speed
Settings...
ticks: 0
****
********* Paramètres *********
nombre-moustique 686
nombre-humain 115
On Off contagion_transport
On Off reproductionMoustiques?

Global variable nombre-humain
min, increment, and max may be nu...
Value 115 Units (optional)
☐ vertical?

Figure 3.7. *Creating a slider that controls the value of the* `number-human` *variable*

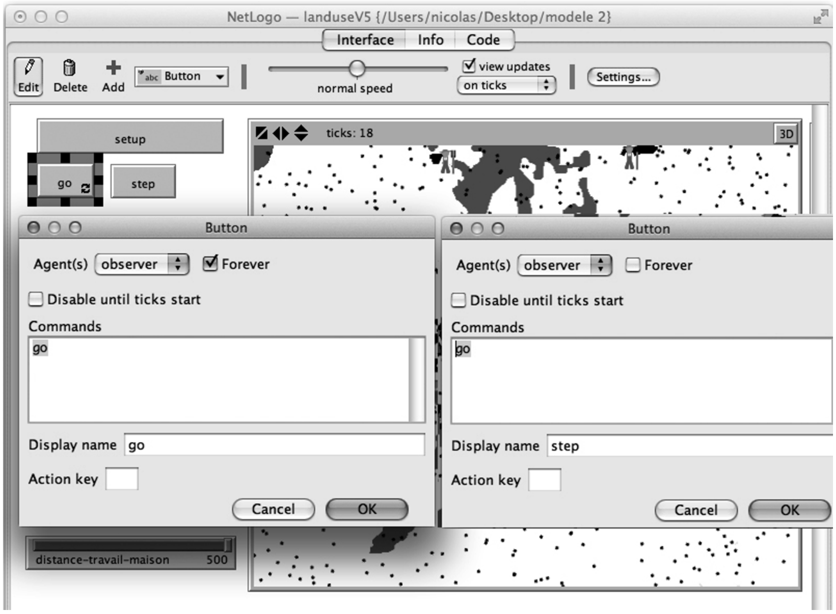
Adding a *slider* to the interface means that a new global variable must be created and initialized (within the *Global variable* field), which will be able to be used within the model, for example, for the creation of as many new agents as the user should want.

```
to init-human
    ...
    create-humans number-human [ .... ]
    ...
end
```

3.4.2. *Introducing environmental behaviors*

3.4.2.1. *Simulation lifecycle*

The simulation's lifecycle is managed by a procedure usually named as go (see Figure 3.8). It successively describes the behavior of the environment and the agents for each time frame (*tick*). This procedure is called into action by clicking on a button or by somehow modifying another element of the interface.



Button

Agent(s) observer [✓ Forever]

Disable until ticks start

Commands

go

Display name go

Action key

Cancel OK

Button

Agent(s) observer Forever

Disable until ticks start

Commands

go

Display name step

Action key

Cancel OK

Figure 3.8. *Illustrating different button modes*

Two different execution modes of the model can be identified with the interface: a step-by-step mode (*step* button) and an automatic mode (*go* button) are available. To define a step-by-step mode, a simple button that calls the go procedure is created within the interface. Every time that this button is clicked, a single *tick* passes. The automatic mode requires the creation of a similar button that also calls the go procedure. However, in this case, the check box next to *forever* must be ticked.

When this button is first clicked, it remains activated and calls the go procedure at regular intervals, until clicked again.

The go procedure activates the behavior of the grid's cells (*patches*) and the different agents (*mosquitoes* and *humans*). In practice, this means that the go-patches, go-mosquitoes and go-humans procedures are called and executed. The go procedure finishes with the ticks command, which ends the current *tick* and incrementally increases the *tick* counter.

```
to go
   go-patches
   go-mosquitoes
   go-humans   ; calling the go-humans procedure
               which models the human
               agents' behavior
   ticks       ; passing to the next tick end

to go-humans
   ask humans
   [
           ... ; behavior of a human
   ]
end
```

COMMENT 3.3.– The ask command is one of the most important available in the NetLogo language. It asks all of the agents contained within a list (list_of_agents) given as its variable to execute the commands defined within the section enclosed by square brackets ([...]). In other words, this command can be read as saying: for *EACH* agent contained within list_of_agents, *EXECUTE* the instructions contained within the bracketed section.

```
      ask listofagents
      [
  ... ; commands detailing all actions
      ; to be carried out by an agent.
      ]
```

3.4.2.2. *Environmental behaviors*

Introducing a behavior within the environment serves to model the phenomena which apply at the spatial level rather than at the level of the individual, for example, the spreading of a forest fire, the dispersion of a virus by winds or the pollination of a space. In the case of the example we are using, the environmental behaviors enable us to model the proliferation of mosquitoes.

In an environment represented in grid form (as in NetLogo's case), giving it dynamic behaviors effectively models a phenomenon of the studied system that is composed of a simulated environment within which the *turtles* evolve. As a result, each *patch* agent within the simulation space has a specific behavior associated with it. Furthermore, each *patch* behaves individually by evolving its internal state and acting within the world.

The behavior of the *patches* is usually controlled by a procedure that is assigned this job alone. It is known as `go-patches` by convention. This procedure analyses all of the *patches* and executes each of their respective behaviors. So as to do this, it uses the `if` and `ask` operators as well as the other operators associated with *patches*, as listed in the documentation.

In our running example, the mosquito proliferation is periodic if and only if this behavior is activated by the `reproductionMosquitoes?` variable that is present on the interface. In this case, a conditional order is used.

In the NetLogo language, the *if-then* command is different from the *if-then-else* command. The `if` command executes the instructions located in the associated section if the condition is satisfied. The `ifelse` command adds an alternative section which is executed when the condition is not satisfied.

```
...
 ; Previous instructions
 if condition
```

```
  [
  ... ; group of instructions to be executed if
          condition is satisfied
      ; (condition = true).
  ]
;  Following instructions
...
```

```
...
 ; Previous instructions
ifelse condition
 [
 ... ; group of instructions to be executed if
          condition is satisfied
      ; (condition = true).
 ]
 [
 ... ; group of instructions to be executed if
         condition is not satisfied
     ; (condition = false).
 ]
; Following instructions
...
```

In NetLogo, a condition is a Boolean expression which has a `true` or `false` value. Such an expression is made up of, among other things: 1) equality operators (=), inequality operators (<>) and relational operators (>, <, <=, >=) between different variables (numeric or alphanumeric); 2) binary Boolean operators such as `and` or `or` and 3) the unary operator `not`.

As we have seen before, the mosquito proliferation is periodic. It is also a function of an activation variable within the interface which allows for the activation of the mosquito proliferation phenomenon. As such, the activation condition of the cell behavior `(if (reproductionMosquitoes?) and (ticks mod timeStep = 0))` is split into two parts:

– `reproductionMosquitoes?`, a variable defined within the interface as part of a *switch* element (interface element allowing for a choice between only two options);

– `ticks mod timeStep`, where `ticks` is a NetLogo variable that indicates the number of *ticks* that have passed since the beginning of the simulation and `timeStep` is an interface variable that determines the number of *ticks* between two proliferation events. When `ticks` is a multiple of `timeStep`, the result of the `ticks mod timeStep` calculation is equal to 0.

COMMENT 3.4.– `mod` gives the remainder of a euclidean division between two whole numbers. For example, `10 mod 8` returns 2.

When combined, the `ask` command and the `with` operator allow for the assigning of a specific behavior to a reduced agent population depending on the modeling hypotheses. In our example, two hypotheses are adopted: 1) only the humid areas (in blue), where stagnating water is present, allow for the proliferation of mosquito larvae – (`pcolor = blue`); 2) mosquito proliferation can only take place if they are locally present, that is to say when their number is 1 or greater – (`count moustiques-here >= 1`).

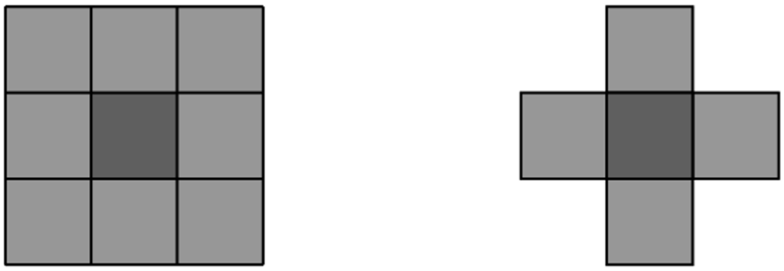


Figure 3.9. *Cells selected (in light gray) with the neighbors and neigbors4 commands*

Each selected cell creates a new mosquito agent and places it somewhere in its surroundings. Initially, a cell is randomly chosen (with the `one-of` operator) from the adjacent cells: whether the 8 that surround the selected cell (`neighbors` operator), or whether the 4 that share a side with the currently selected cell (`neighbors4` operator). Then, a mosquito agent is created with the `sprout-mosquito` command.

The `sprout-<breeds>` and `hatch-<breeds>` commands allow for the creation and initialization of *turtles* of a given species in the same way as the `create` command (as discussed in section 3.4.4.1). However, these three commands distinguish themselves by the context in which they are used: `create` is used in the general context by a procedure called by the interface or by the user; `hatch` can only be used for *turtles* and `sprout` can only be used for *patches* (see example below). The `sprout-<breeds>` and `hatch-<breeds>` commands are nonetheless used in the same way as the previously discussed `create-<breeds>` command:

```
ask patches [
   sprout-<breeds> number [
   ... ; set of instructions which apply to each agent
       ; created for its initialization]
    ]
]

OR

ask <breeds> [
   hatch-<breeds> number [
   ... ; set of instructions which apply to each agent
       ; created for its initialization]
    ]
]
```

In our example, the newly-created mosquito is placed in the center of the `myNeighbor` cell by the following command:

```
setxy [pxcor] of myNeighbor [pycor] of myNeighbor
```

Finally, the complete `go-patches` procedure is as follows:

```
to go-patches
  if (reproductionMosquitoes?) and (ticks mod
       timeStep = 0)
  [
  ask patches with [(pcolor = blue) and count
       mosquitoes-here
                       >= 1)]
  [
    let myNeighbor one-of neighbors
     sprout-mosquitoes 1
    [
      setxy [pxcor] of myNeighbor [pycor] of myNeighbor
      set size 5
      set shape "butterfly"
      set isInfected? false
      set isInfectionExternal false
      set color black
      ]
   ]
 ]
end
```

3.4.2.3. *Turtle agent behaviors*

Defining *turtle* behavior is done in the same way as behaviors are added to *patches*: a procedure `go-<breed>` is usually implemented for each species. Apart from the commands detailed in the previous section, a *turtle*'s behavior will often use movement commands to enable the agent to move within the environment.

A *turtle* agent is considered to be a self-supporting entity that has an x, y position (within the `xcor` and `ycor` variables) and an orientation (`heading`). This direction is represented by an angle α in degrees measured between the *y*-axis and the tangential trajectory of the agent (its speed vector).

So as to move, the *turtle* agents perform commands which modify their position and orientation. These are limited to:

– `forward` Δ or `fd` Δ $\rightarrow$ moves the turtle a distance of value Δ (forward motion);

– `back` Δ or `bk` Δ $\rightarrow$ reverses the turtle a distance of value Δ (backward motion);

– `move-to` Ω $\rightarrow$ moves the turtle to the center of an agent Ω;

– `setxy` xy $\rightarrow$ moves the turtle to the coordinates given by x and y;

– `left` α or `lt` α $\rightarrow$ turns the turtle to the left by an angle of α in degrees (anticlockwise rotation);

– `right` α or `rt` α $\rightarrow$ turns the turtle to the right by an angle of α in degrees (clockwise rotation);

– `face` Ω $\rightarrow$ makes the turtle face an agent Ω;

– `facexy` xy $\rightarrow$ make the turtle face the coordinates given by x and y.

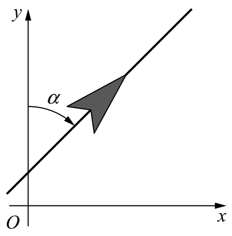


Figure 3.10. *An agent's orientation is defined by the angle α between the* y*-axis and the agent's trajectory (this angle is given in degrees, and in a clockwise direction)*

In our example, the mosquitoes have a movement behavior consistent with Brownian motion, which is made up of a rotation and a forward movement. The `lt random 360` command rotates the chosen agent to the left (`lt`) by an angle α chosen randomly from the domain $\alpha \in [0;360[$. Then, the `fd 1` command moves the agent forward by 1 unit.

```
to go-mosquitoes
  ask mosquitoes
  [
    lt random 360
    fd 1
    bite
  ]
end
```

Many other commands exist, most notably `die`, which instantly kills the agent (removing it from the simulation). We invite the reader to discover these within the relevant documentation when they further experiment with modeling.

3.4.2.4. *Visualization*

A multiagent simulation can produce a multitude of results. It is useful for the modelers to be able to analyze and intelligently synthesize these so as to display them within the interface in the form of monitors, graphs or 2D maps.

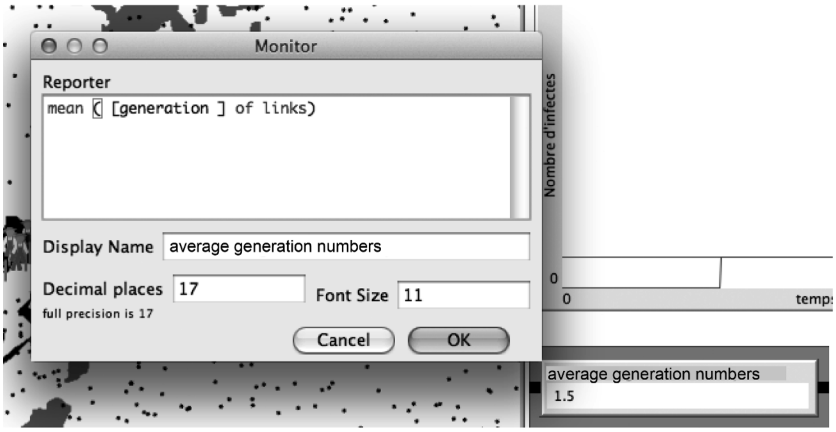
3.4.2.4.1. 2D display of the simulated world

Displaying the simulation within a 2D graphical interface is obligatory and cannot be removed. The display can be resized and there is an option to visualize the multi-agent systems (MAS) in pseudo-3D. However, the interface and its use are heavily restricted because of its uniqueness and due to the fact that it does not offer different viewpoints of the simulated world without specifically programming these in. If this were to be the case, special procedures would have to be created to change the *patch* colors as well as the size and color of the *turtles* and *links*, based on their internal states.

NetLogo distinguishes between two different display modes: 1) the first is known as *on tick* mode, which updates the interface on each *tick*; 2) the other is known as *continuous* mode, which updates the interface independently of the model's execution, which can lead to the display refreshing inconsistently. While developing and testing the model, it is advisable to use *on tick* mode, which will allow for a rigorous visualization of the model. The *continuous* mode can then used while exploring a complete, finished model, as it considerably reduces the amount of time needed for the simulation.

3.4.2.4.2. Displaying alphanumeric results

A monitor (*Monitor* in NetLogo's interface) is an alphanumeric section of the interface which displays a variable or the result of an expression (*reporter*). This type of display is very useful for obtaining aggregated results about the *turtles*, *patches* and *links* during a simulation.



Reporter
```
mean ( [generation ] of links)
```
Display Name: average generation numbers
Decimal places: 17
full precision is 17
Font Size: 11

average generation numbers
1.5

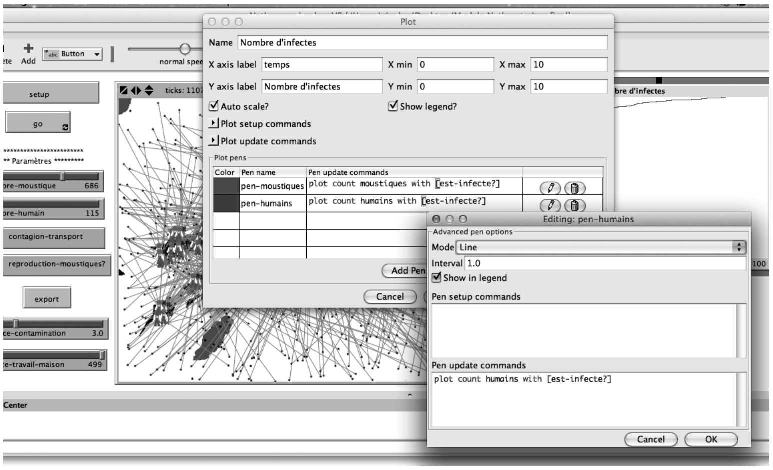
Figure 3.11. *The configuration interface used for monitors*

Figure 3.11 shows a monitor named "average generation number". Via the `mean([generation] of links)` expression, this monitor calculates the average of the values of the `generation` variable that the *link* agents possess, which effectively calculates the average distance between contaminated individuals and individual who was the source of the contamination.

3.4.2.4.3. Displaying results in graphical form

A graph (called a *plot* in NetLogo's interface) is a part of the interface which follows one or more variables that evolve with the progression of the simulation. This type of display resembles the monitor, but with the added factor of time at the expense of precision.

Several graphs may be used within the same model. Each shows one or more series of points (which is known as *pen*s in NetLogo) displayed in an orthonormal plane bounded by axes with set coordinates (X min, X max and Y min, Y max).



Name: Nombre d'infectes
X axis label: temps — X min: 0 — X max: 10
Y axis label: Nombre d'infectes — Y min: 0 — Y max: 10
✓ Auto scale? ✓ Show legend?
Plot setup commands
Plot update commands

Plot pens

Color	Pen name	Pen update commands
■	pen-moustiques	plot count moustiques with [est-infecte?]
■	pen-humains	plot count humains with [est-infecte?]

Editing: pen-humains
Advanced pen options
Mode: Line
Interval: 1.0
✓ Show in legend
Pen setup commands
Pen update commands:

```
plot count humains with [est-infecte?]
```

Figure 3.12. *The configuration interface used for graphs*

For each series, the user must define a display mode (line, point or bar), the interval between two values along the *x*-axis, an expression that sets the value of the first point and a second expression that defines the value of each of the subsequent points to be graphed for each *tick* of time.

It is possible to modify a graph with programming code. This feature gives the user many more possibilities than NetLogo's graphical interface. The following primitives are used along with the `do-plot` procedure to redefine the manner in which a graph will update:

– `set-current-plot name_graph` → selects a graph (`name_graph`) to be modified;

– `set-current-plot-pen nane_series` → selects a series (`name_series`) of the previously selected graph that is to be modified;

– `plot y` → add a point to the previously selected series. This point is situated at the *y* coordinate of the previous point to which is added the value of the interval as defined in the configuration interface;

– `plotxy x y` $\rightarrow$ adds a new point at the x, y coordinates of the previously selected series.

Initially, the `do-plot` procedure selects a graph:

`set-current-plot "number of infected"`
and then, a series: `set-current-plot-pen "pen-mosquitoes"`.

Finally, the `plot count mosquitoes with [isInfected?]` command calculates the number of infected mosquitoes and updates the series. The two previous commands (`set-current-plot-pen` and `plot`) update the second series named as `pen-humans`.

```
to go
   go-patches
   go-mosquitoes
   go-humans
   do-plot ticks; passing to next tick
end
to do-plot
   set-current-plot "number of infected"
   set-current-plot-pen "pen-mosquitoes"
   plot count mosquitoes with [isInfected?]

  set-current-plot-pen "pen-humans"
  plot count humans with [isInfected?]
end
```

In the case of the running example, two series need to be updated on the same graph.

3.5. Agent–agent and agent–environment interactions

3.5.1. *Agent–agent interactions*

One of the advantages of multiagent simulation, notably compared to microsimulation, is that it is possible to model interactions between different agents. In NetLogo, interactions take the form of commands and "questions" (`ask` commands) that certain agents can make of others. By respecting the principle aim of encapsulation in this manner, the agents are not allowed to directly modify the variables of other agents within the model. So as to do this and as such to implement an interaction, an agent (`a`) must ask another agent (`b`) to do something by using the `ask` primitive that has previously been introduced.

As such, within our example, a random human agent that would destroy all of the mosquitoes would appear in the following form:

```
ask one-of humans
   [
       ask mosquitoes [die]
   ]
```

The `one-of` command allows for the random selection of a single element of a particular group (in this case, `humans`). The corresponding program would be translated as follows: ask one of the humans to ask each of the mosquitoes to die.

It should be noted that this question takes the form of an order in the sense that the agent who receives it does not have the possibility to refuse to carry it out or discuss its terms. It is an imperative demand which enters the communications in the form of a message written in object-oriented language.

So as to have a wider scope of expression of the interactions, it may be necessary to reference each of the agents concerned: the asker (`a`) and the executer (`b`). NetLogo allows this by using the keywords `self` and `myself`. The `self` term corresponds to the selected agent (who is the executer in our case) and `myself` corresponds to the agent from the superior level (who is the asker in our case).

In the case of the infection of a human by a mosquito in our model, the mosquitoes are hence asked to carry out the `attack` procedure:

```
to go-mosquitoes
   ask mosquitoes
    [
    ...
    attack
    ]
end
```

The `attack` procedure is therefore carried out by a mosquito. In an object-focused view, `attack` would be a method belonging only to the mosquito. As such, it can be said that the corresponding code will be carried out from the point of view of the latter. To carry out its attack, the mosquito begins by putting together a list (an *agentset*) of the humans who surround it (at a distance lesser than the `distance-contamination` variable).

```
to attack
   let humansNear humans with [distance myself <
       distance-contamination]
```

In this case, `myself` refers to the mosquito which is currently carrying out the procedure and the `humansNear` group will contain all of the humans at a lesser distance than the `distance-contamination` variable relative to the mosquito in question.

The attack itself then translates to a question posed by the mosquito to all nearby uninfected humans to let it infect them (as long as the mosquito itself is infected, naturally) and will be written as follows:

```
to attack
    ;;context of a mosquito agent
    ask humansNear
    [
      ;;context of a human agent
    if (not [isInfected?] of self) and [isInfected?]
        of myself
    [
```

While this code is being carried out, we will be changing context, passing from the context of the mosquito agent to that of the human agent. This will cause the evolution of the agents that the `self` and `myself` primitives refer to.

In the context of the mosquito agent (before the `ask humansNear` command), `self` refers to the mosquito agent which is currently executing the code and `myself`, which by default should refer to the agent which asked the formerly mentioned mosquito to execute the procedure, actually refers to nothing at all (`nobody`), since the command was called by the model at a upper level.

In the context of the human agent (after the `ask humansNear` command, which asks each of the human agents in the `humansNear` group to execute the subsequent code), `self` refers to the human agent that is currently executing the code and `myself` refers to the mosquito agent which asked it to do so.

To practically illustrate the references detailed above, the human and mosquito agents which both own a Boolean (true/false) `isInfected?` variable allowing us to store whether the agent is infected or not, `[isInfected?] of self` corresponds to the `isInfected?` variable of the human agent (`self` refers to the current context) and `[isInfected?] of myself` corresponds to the `isInfected?` variable of the mosquito agent. Finally, `myself` refers to the context of the upper level.

The `if (not [isInfected?] of self) and [isInfected?] of myself` command can, as such, be translated as follows: "if the currently selected human agent is not infected and the mosquito is infected, then...".

Let us briefly return to the process that puts together the `humansNear` group:

```
to attack
     let humansNear humans with [distance myself <
         distance-contamination]
```

Please note the reference to `myself`. In this situation, the code: `[distance myself < distance-contamination]` is executed in the context of a human agent and `myself` refers to the mosquito agent of the upper level that is currently executing the attack procedure.

3.5.2. *Structuring interactions as part of a social network composed of links*

Originally, NetLogo was more of a platform designed to simulate spatial phenomena, as shown by the numerous examples available in the Models Library. Nonetheless, since social networks were being more and more often used in multiagent simulations to model interactions between individual agents, NetLogo satisfied the demand for this functionality by introducing *links* in its 4.0 version. As shown in Figure 3.1, the *links* take the shape of special agents which link two *turtles* and can be selectively given directionality. When directed, link creation takes on the form of NetLogo's common syntax (`create-link-to`, `creation-<breed>-to`, `create-link-from`), which are commands that link the agent calling for the creation of the link to the agent given as a variable:

```
if any? humans-near with [is-infected?] [
    become-infected
    ask one-of humans-near with [is-infected?]
    [
       create-infection-to myself
       [
             create date ticks
       ]
    ]
]
```

The *infection* species is a special class of *link* (or a *link* type created in the context of this model), `create-infection-to myself` creates a link between the agent executing the function and the agent of the upper level (`myself`), with the direction of the link being specified by the use of `from` or `to` (a *link* is created going toward or away from an agent).

3.5.3. *Adjacency*

When we wish to simulate spatial interactions between entities, such as in the case of a prey-predator model, the notion of adjacency is extremely important: a predator may only eat its prey when immediately adjacent to it. Identifying which entities are adjacent to an agent depends upon the type of topology of the environment (adjacency is not the same on a grid as it is within a network). In the environment defined within NetLogo (a continuous environment coated in a grid made up of *patches*), there are two ways in which adjacency can be calculated. The first solution consists of selecting the subgroup of agents that is at a distance less than a given threshold (e.g. a perception threshold):

```
ask turtles with [distance myself
       < threshold-perception]|.
```

The second solution consists of using, perhaps less precisely, the *patches* which pave the environment upon which exist the *turtles*.

The keyword `here` will allow for the grouping of all the *turtles* which are found on the *patch* which is calling the (`ask turtles-here[...]`) command. Also, the `neighbors` keyword gives access to all the patches adjacent to the current `patch`. Therefore, to access any *turtles* which might be found on adjacent *patches* (and as such, within proximity) all that is necessary is to ask which *turtles* are found on adjacent *patches*, as follows: (`ask turtles-on neighbors`).

The same principle has been kept for the *links* with which it is essentially needed to access some or all of the neighbors linked to a specific agent, and as such `ask one-of link-neighbors` gives access to each of the agents (which are selected in a random order) that are linked to the chosen agent.

Despite the very intuitive implementation of these different functions, identifying the types of functions that can be used or combined across the three main agent types (*turtles*, *patches* and *links*) can be difficult. Once each of these three different approaches has been well identified and understood, NetLogo allows for these functions to be easily combined together.

3.6. Introduction to NetLogo's additional functionalities

As well as the basic model editing and simulation execution tools presented in this chapter, NetLogo provides a collection of additional tools useful for modelers. Additionally, numerous extensions have been created for NetLogo, which serve to extend its language with the aim of integrating new objects into its simulations (such as GIS or networks).

3.6.1. *The Behavior space tool*

Accessible from the *Tools* menu, the *Behavior space* tool allows for the quick setup of experimental test designs to be performed on the model. Global variables and parameters can be defined with all their possible values for their exploration. It then executes an exhaustive experimental design and is able to save any interesting variables or indicators within a .csv file, to be studied using appropriate tools at a later date. Despite the limited capacity of the *Behavior Space*, it is often necessary to use it to launch quick and simple test designs that can help with the detection of any eventual bugs before using more powerful tools whose primary purpose is to study models in great depth (such as *OpenMole*[9]). The study of a model and the usage of the *Behavior Space* are described in the following section.

3.6.2. *Multiplayer (HubNet)*

HubNet is a NetLogo mode dedicated to the management of individual, remote interfaces. This mode is particularly useful for the implementation of serious participatory games where each actor connects, via a local network, to a same simulation within which they play a particular role. HubNet modifies the model based on a particular viewpoint, structured by its own interface which contains a personalized

9 http://www.openmole.org/.

view of the simulation as well as any specific actions available to that player. The user interface is graphically constructed by a instructive interface (*HubNet Client Editor*) but user management is carried out within NetLogo via specific primitives such as `hubnet-send-message`. Participative simulation is further detailed in section 6.5.3.

3.6.3. *Dynamic systems*

While multiagent modeling concerns itself with the individual behaviors of agents, modeling based on dynamic systems is instead focused on the global behaviors of the agent population. The *System Dynamics Modeler* extension (menu *Tools* then *System Dynamics Modeler*) allows for the representation of systems in which there are few entities that effectively represent groups of individuals. Values that are associated to entities evolve thanks to the interactions between these entities. This evolution is commonly represented by a system of ordinary differential equations. Four basic elements allow for the construction of a dynamic system diagram: stocks, variables, flows and links.

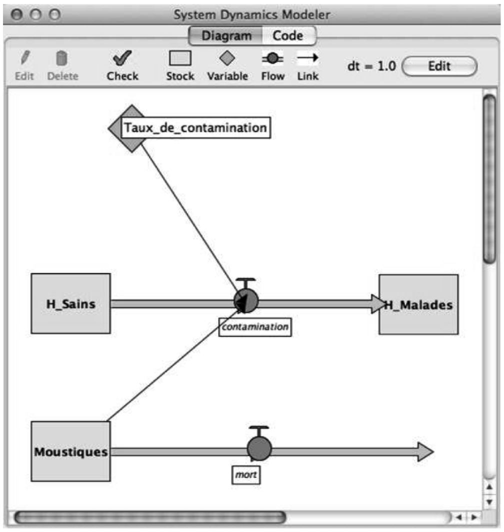


Figure 3.13. *The configuration interface used for modeling dynamic systems*

A stock represents an aggregate, such as a population of healthy humans, a population of contaminated humans or a population of mosquitoes. A flow represents a quantity that passes from one stock to another and of which the volume is controlled by a tap. For example, the flow between the stock of healthy humans and the stock of contaminated humans is a function of the interactions between the stock of healthy humans and the stock of mosquitoes. A variable can represent a constant or an equation that depends on other variables. For example, a variable representing the infection rate is added to the interaction between the stock of healthy humans and the stock of mosquitoes. Finally, a link allows for a value resulting from a stock or a variable to be available (known) to several entities of stocks or variables. As a result, the flow between the two human stocks depends on the stock of mosquitoes. This module, as well as modeling based on equations, is presented in greater detail in section 5.5.

3.6.4. *Introduction to models in 3D environments*

3D, or the use of a 3D space for the movement of agents, allows for the modeler to have access to many improvements with respect to display and interaction (made realistic and immersive, superimposing additional information throughout the simulation, etc.). However, the introduction of 3D into a model does not have as its only goal to make the environment more realistic, it also allows for a more intuitive and immersive study of the model.

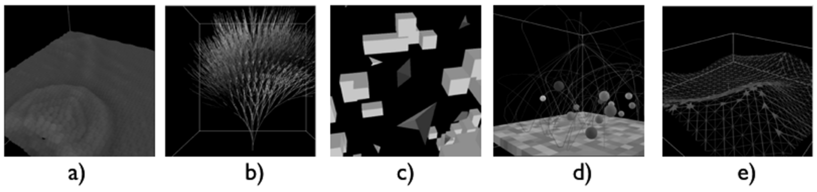


Figure 3.14. *Example of 3D models in NetLogo3D: a) a water drop falling on a solid surface, b) a 3D fractal tree, c) termites, d) a bouncing ball and e) turtles evolving upon a 3D surface. For a color version of the figure, see www.iste.co.uk/banos/netlogo.zip*

NetLogo allows for a 3D world to be easily defined using NetLogo3D (an application bundled with NetLogo). The environment in a 3D world possesses a `width`, a `height` and a `depth`. The *patches* become cubes with an additional coordinate `pzcor`. Now the *turtles* also possess three Cartesian coordinates `xcor, ycor` and `zcor` as well as an orientation defined by three variables (`heading, pitch` and `roll`). The viewpoint from which the user sees the world corresponds to the location and orientation of the *observer*, which faces an initial point as defined by the `face` or `facexyz` commands. Its position is defined by the `setxyz` command. An agent may move within the 3D world using the `follow, follow-me, ride` and `ride-me` primitives. It is also possible to change the viewpoint of the simulation by using the `watch` and `watch-me` primitives. Finally, it is possible to import 3D shapes into the environment with the `load-shapes-3d` primitive.

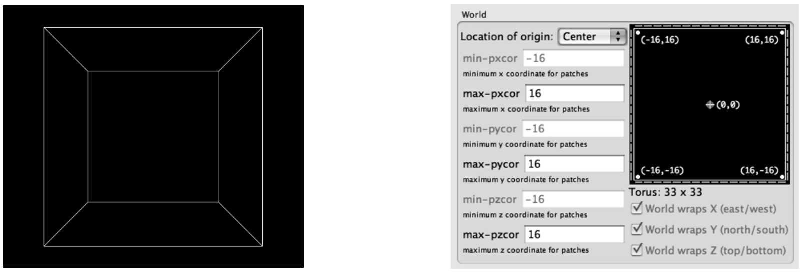

World
Location of origin: Center
min-pxcor -16
minimum x coordinate for patches
max-pxcor 16
maximum x coordinate for patches
min-pycor -16
minimum y coordinate for patches
max-pycor 16
maximum y coordinate for patches
min-pzcor -16
minimum z coordinate for patches
max-pzcor 16
maximum z coordinate for patches
Torus: 33 x 33
[✓] World wraps X (east/west)
[✓] World wraps Y (north/south)
[✓] World wraps Z (top/bottom)

Figure 3.15. *An environment modeled in NetLogo3D*

When NetLogo3D is launched, the world is represented in the shape of a cube. The *Model Settings* add limiting coordinates in the third dimension (`max-pzcor` and `min-pzcor`), which are added to the *x* and *y* coordinate information available previously (`max-pxcor`, `min-pxcor`, `max-pycor` and `min-pycor`).

It should be noted that a model written in NetLogo can often be opened with NetLogo3D and will then be displayed on a 2D plane. The opposite is scarcely true, as the 3D primitives and variables are not recognized by NetLogo.

3.6.5. *Geographical information systems*

When the heterogeneous nature of environmental data is an important element affecting the dynamic of a multiagent system, then the use of the GIS extension (`extensions [gis]` at the beginning of the main block of code) is most useful.

Various raster file formats, such as ascii `grid` (.asc and .grd) and vector shapefiles (.shp), can be read with the use of the `gis:load-dataset "name.(shp l asc)"` primitive. The included operations consist of reading the data, defining its coordinate system and defining or executing operations upon this data. It is therefore possible to import a group of values by *patch*, by points, by lines or by polygons.

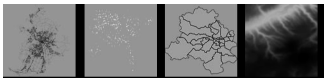


Figure 3.16. *Examples of importing geographical data. For a color version of the figure, see www.iste.co.uk/banos/netlogo.zip*

In a general sense, the information needed to be able to project the values resulting from a GIS layer into NetLogo are the [minimum-x maximum-x minimum-y maximum-y] coordinates of the GIS layer and the [min-pxcor max-pxcor min-pycor max-pycor] coordinates of the NetLogo space. It is possible to use one of the GIS extensions' primitives to transfer these values, as follows:

– in a predefined domain, with:

```
gis:set-transformation gis:envelope-of name_of_the_layer
 [min-pxcor max-pxcor min-pycor max-pycor]
```

– in the entire domain, with:

```
gis:set-world-envelope gis:envelope-of name_of_the_layer
```

A group of primitives can then be used to perform operations of this data such as selection based on a variable's value, the calculation of polygon centroids, the intersection between two entities with `gis:intersects? A B`, or A and B, can be of the *VectorDataset* type, *VectorFeature*, *turtle*, *link*, *patch*, *agentset* or list, or even to assign the values of a variable to *patches* or to *turtles*.

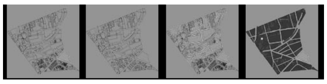


Figure 3.17. *Display of different layers of information. For a color version of the figure, see www.iste.co.uk/banos/netlogo.zip*

Finally, it is also possible to put together .shp data files from the *turtle*, *patch* and *link* data so as to read a work with them in GIS. For example, the `gis:turtle-dataset turtles` primitive returns all the values of the agentset's variables. These are then modified and saved with the use of the `gis:store-dataset` primitive. The following command:

```
gis:store-dataset gis:turtle-dataset turtles ‘‘centroid’’
```

allows for the *turtles* which happen to be centroids of polygons to be exported within a .shp file. In this manner, NetLogo data can be transformed into a format that can be read by a GIS such as QGIS (previously known as QuantumGIS).

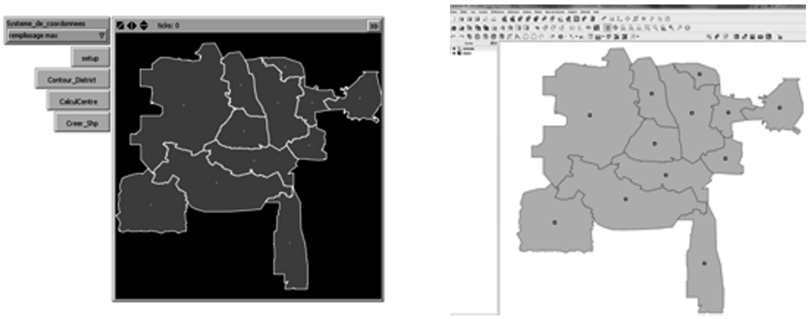


Figure 3.18. *Comparing the display of a shapefile in NetLogo and QGIS*

3.6.6. *Algorithms in graphs*

As mentioned in this section and the previous sections, agents are entities that interact with each other regardless of adjacency, as well as the environment that surrounds them. It is possible to display a representative graph of these interactions, in which agents are represented by vertices and the interactions are shown by edges. Relationships between the various elements may be expressed or implied. For example, if we were modeling a city with roads and intersections, the relationships are clearly stated with weights associated with the edges that can be distances or traffic density (maximum and/or current) which cannot be exceeded. Equally, if the model contained drains and sewers, the limit to be exceeded would be a flow rate assigned as before to the edges. On the other hand, the relationships between a model's entities may be implied and therefore depend on properties associated with a particular agent, which will vary greatly over the course of the simulation. For example, it is possible to model relationships such as those based on affinities between agents, such as musical tastes, belonging to a same company, friendships via social networks, coworkers, etc. Here, again, it is possible to create graphs that are the result of focusing on a particular relationship present during the model's simulation. It is also possible to combine several of these into yet another graph.

Once a graph has been created from the model, it can be useful to extract information from it to better analyze the model. In this situation, the graph has the only purpose of clearly displaying a particular property of the model. In fact, this display of data is very broad, as any data can be displayed on a graph, provided that it makes sense. This ranges from tubular representation to the representation of lists and all unorganized relationships as well as the absence of relationships.

When a graph has been put together and correctly displays the chosen data, it is possible to benefit from a large amount of reading around the study of graphs. The first use of a graph goes back to the 18th Century. Léonhard Euler showed the inhabitants of Königsberg – who had invited him there in 1736 – that it was not possible to cross all of the bridges in the town once and only once. He set down the conditions that a graph must respect for this challenge to be solved. Hence, such a graph is today referred to as an Eulerian graph. His results were published in 1741, but without a proof. It was Hierholzer who eventually published the proof in 1873. The origins and rise of modern graph theory are attributed to Claude Berge, who developed his ideas on the topic in the late 1950s. It was the advances in computational techniques which helped to highlight the utility of graphs for a great number of fields, as automatic computer processing made them a viable tool for data analysis, which was not the case when all graphing had to be done manually. As a result of this, a great number of algorithms were developed to help exploit the numerous properties of graphs, whether on a literal or theoretical level.

Among the algorithms, which immediately appear to be of use for the exploitation of a multiagent model, are the following: Dijkstra, Bellman-Ford and Floyd-Warshall's algorithms, for finding the shortest path; Bellman-Ford's algorithm, for finding the longest; Tarjan's algorithm, for finding the strongly connected components of a graph;), Ford–Fulkerson's algorithm, for calculating the maximum flow in a flow network; the Hungarian algorithm, a combinatorial optimization algorithm; the Traveling Salesman algorithm (if one passes once through each edge) or Eulerian algorithm (passing only once through each vertex), used for pathing and routing problems; and Prim and Kruskal's algorithms, for finding minimum spanning trees in weighted graphs.

Equally, it is possible to use a particular type of graph to display social networks, these are known as small-world networks. They allow us to show relationships between nodes, with very dense areas in some places. They are based on the concept that everyone is linked to all others in some fashion, whether through common likes or dislikes, or any other property of social relations, and that this link is very short. Milgram carried out the first experiment of its kind in the United States (published in 1967 amid high criticism) during which envelopes were handed from person to person, from the sender to the recipient. One of the letters took only 4 days to arrive at destination. Within the same school of thought, mathematicians created a number, the Erds number, which gives the "collaborative distance" between any researcher to the Hungarian mathematician Paul Erdös, measured from co-authorship of scientific papers. For example, if I were to publish a paper with Paul Erdös, I am at a distance of 1 from him, yet if I publish instead with one of his co-authors, I am at a distance of 2, and so on. The same calculation was made to find the average distance between individual Facebook users. Thus, the study of the interactions between actors in a social model can be beneficial for its analysis in a multiagent simulation.

Finally, graphs are excellent at displaying classifications of agents based on their intrinsic or acquired properties. They allow for agent populations with common or similar characteristics to be clearly visible, which, in turn, means that the main groups and subgroups of a simulation can be easily identified, which may not have been possible beforehand, as the creators of the model will not necessarily have been able to predict which groups would emerge as the largest. As well as being able to follow the evolution of a model during its simulation, the capacity for illustrating the emergence of new agent categories is another benefit for a study of a multiagent simulation. As a result of this brief presentation on the uses of graphs in the context of multiagent simulations, it is quite clear that a comprehensive understanding of their functionality is important for the creation and analysis of multiagent models. This is the case at the simulation level, by influencing agent behavior based on the decisions they may make during the course of a simulation. This is also true during analysis, where they can be used to explain the reasoning behind the creation of a simulation, such as a study of the impacts of roadworks on urban journeys or a study of emergent behaviors outside city sports facilities.

3.6.7. *NetLogo dictionary and abbreviated commands*

All of NetLogo's commands are documented on the software's website, in the *NetLogo dictionary*[10]. The site allows most notably to identify which instructions can be used in the context of the *observer* (👁), of a *turtle* (🐢), of a *patch* (▚) or of a *link* (🔗).

The NetLogo dictionary also introduces an abbreviated notation for the main commands. Both forms of a command (full and abbreviated) are exactly the same in terms of functionality. For example:

– `create-turtles` can be abbreviated to `crt`;

10 http://ccl.northwestern.edu/netlogo/5.0/docs/dictionary.html.

– `create-turtles 10 [ ... ]` is equivalent to `crt 10 [ ... ]`.

3.7. Conclusion

This chapter introduced the basics of the NetLogo language by the use of our running example. While not comprehensive, it gives the reader the basics needed to build their first models. It also allows them to understand our example, as well as the various models supplied with the NetLogo software. Yet, it is only practice that will help the reader to master the language and use it without external help.

Now that the model has been built and tested, and that the modeler has “played” with their model, that is to say that they have changed it by modifying its basic variables, a more in-depth analysis of simulations is necessary (notably, to carry out a sensitivity test). This is the aim that the following chapter has been given, by describing the *Behavior Space* tool that was briefly introduced previously.