Workshop ABM–CEF 2015 Pre-conference workshop on Agent-based Models in Economics and Finance Chapter 2- Introduction to NetLogo

Murat Yıldızoğlu
http://yildizoglu.info
Université Bordeaux
GREΘA (UMR CNRS 5113)

CEF 2015 Conference, Taipei

NetLogo?

- A programming platform with a graphical interface (GUI), and a meta-language dedicated to ABMs
- GUI: three panels:
  - **Interface**: an interface for interaction with the model and for representing the results (graphics, spatial structure of the agents' space, networks, etc.);
  - **Information**: an interface to give information to the users of the model;
  - **Procedures**: an interface for writing the procedures of the model (code for agents' behaviors, their interactions, and system dynamics).
- Important stuff is in the Procedures panel, while funny stuff happens in the Interface panel.

Main agent types

A model in NetLogo may contain the following main types of agents :

- the World represents graphically the complete spatial environment of the model. It can be in 2D or 3D.
- patches are the elementary spatial components of the World. Agents move over the patches, and can read the properties of patches (the patches can have variables).
- turtles are individual agents in the model. They have properties (variables), they can move in the World, they can give birth to other turtles (hatch an egg), and they can die. They can be of different *breeds* with different sets of properties (wolves, sheep, consumers, firms, etc.).
- links connect two individual agents (turtles). They can be oriented or not. (Network models are easy in NetLogo).

Other elements

A NetLogo model may also contain:

- global variables, visible by all agents;
- *agent sets* containing a list of agents of the same breed;
- specialized procedures dedicated to operations used by the model (including setup and go);
- comment lines that must all start with a semi-colon: `; This is a comment`

1 Elements of language

Lists

- Lists play a central role in NetLogo
- We delimit the elements of a list with square brackets and separate these elements with at least one space (the comma is not the separator in NL!):
  - A list of three numbers: [0.5 1 2]
  - A list containing two lists: [[0.5 1 2] [2.5 1.33 25]]
  - An heterogeneous list: [0.5 [2.5 1.33 25] ["*Paul*" "*Jacques*" "*Aliye*"]]
  - An empty list: []
- We can also build a list using the instruction *list* instruction with parentheses: `(list 0.5 1 2)`
- We refer to specific members of a list using the instruction *item*: `item 0` [0.5 1 2] → 0.5
- Other useful instructions on the lists: `replace-item, fput, lput, sentence, map, reduce, sort, sort-by`

1.1 Variables and breeds

Declaring and using variables

Variables can be declared

- In the GUI, when we create controls (a text input field, a slider, a selection combo, etc.),
- At the start of the program, as a global variable declared int he list called *globals*: `globals [ price nbAgents ]`
- As a local variable in a procedure or program or a code bloc with the instruction `let`: `let prix-initial 10.`
- As a property of a specific breed of turtle (see below).

Variables 2

- We modify the value of an existing variable using the instruction *set*:
  - `set initial-price 0.5`
  - `set my-prices` [1.5 10 25]
  - `set my-name` "*Toto*"
  - `set price ( item 0 my-prices )`
  - `set my-prices ( replace-item 1 my-prices 15 ) show my-prices` → [1.5 **15** 25]

Declaration of turtle breeds and of their properties

- Declaration of agent breeds: `breed [plural-name singular-name] breed [firms firm]`
- Declaration of individual properties of agents of a specific breed: `firms-own [profit]` each firm has a property called profit, and it can contain a specific value for each one

A variable that counts: ticks

- NetLogo models take place in discrete time, advancing step-by-step
- NL includes a counter that memorizes the number of steps (*periods*) already executed: *ticks*
- We can read and use the value of this counter using its name `let` this-period `ticks`
- We initialize its value at the start of the time using the instruction *reset-ticks* (executed in general at the end of the `setup` procedure)
- We increment its value by one using the instruction *tick* (executed in general in the `go` procedure, just after the operations corresponding to the end of a period)

1.2 Procedures

Declaration of specialized procedures

- We regroup in specialized procedures sets of instructions that we may need in different moments in the program, or have to execute them several times in a period.
- NetLogo includes two procedure types :
  - *Commands* are used to modify the values of global variables or variables given to the as parameters, without returning any result;
  - *Reporters* return a result at the end of the operations they execute. We need to save the returned value in a variable or directly use in other computations (example: *replace-item* returns the modified list).

Commands

- Their declaration starts with the instruction *to* and ends with the instruction *end* `to my-command [ argument1 argument2 ] instruction1 instruction2` ⋮ `end`
- Later in the program, we can call this command: ⋮ `my-command arg1 arg2` ⋮
- to execute its instructions using the values passed as arguments

Reporters

- Their declaration starts with the instruction *to-report* and ends with the instruction *end* `to-report my-reporter [ argument1 argument2 ] let my-result 0. instruction1 instruction2` ⋮ `set my-result ... report my-result end`

- Later in the program, we can call this reporter for setting the value of a variable using its result `set my-variable my-reporter arg1 arg2`
- `my-variable` contains now the last computed value of `my-result`

Structure of a command

$$my-prices: \quad [1.5\ 10\ 25]$$

$$\downarrow$$

$$set\ \underbrace{my-prices}_{final\ list}\ (\ replace-item\ \underbrace{1}_{element\ to\ be\ changed}\ \underbrace{my-prices}_{initial\ list}\ \underbrace{15}_{new-value}\ )$$

$$\downarrow$$

$$my-prices\ \rightarrow [1.5\ \mathbf{15}\ 25]$$

1.3 Controlling the flow of the program

Repeating instructions

- To repeat n times a set of instructions: `repeat n [ instructions ]`
- To repeat a set of instructions as long as a condition is fulfilled (it evaluates to `true`), and to stop as soon as it becomes `false`: `while [ condition ] [ instructions ]` `while [any? other turtles-here] [ fd 1 ]` The turtle that executes these instructions goes forward (fd = forward) as long as it finds another turtle on the patch where it arrives.

Setting the flow depending on conditions

- Executing certain instructions if and only if some conditions are fulfilled
- We use instructions *if* or *ifelse* :
- `if condition [instructions]` : instructions executed only if `condition` = `true`
- `ifelse condition [instructions1] [instructions2]` : `if condition = true` → instructions1 are executed; `otherwise` → instructions2 are executed.
- `let x 10 ifelse x >= 0 [show sqrt x] [print "negative number"]`

1.4 Computations with agent lists

Doing computations with agents' properties

- We can use the plural name of a breed (`firms`) to address the set of all agents of this type
- we can then ask them to execute some instructions, and each one of them will execute them in its turn, determined in a random order by NetLogo: `ask firms [ set price market-price fix-production ; execute the command that computes the output compute-profit ; execute the command that computes the profit ]`

Computations with agents – 2

- We can also easily compute statistics and aggregate indicators about agents' properties
- We execute in this case: `operator [a breed variable] of plural-breed-name`
- `[a breed variable] of plural-breed-name` returns a list containing the value of the corresponding variable for each member of this breed
  - Average profit of firms: `mean [ profit ] of firms`
  - Maximal profit of firms: `max [ profit ] of firms`
  - Also other operators can be used: `min`, `median`, `standard-deviation`, `variance`

Computations with agents – 3

- We can do any computations using a breed variable of the agents of the same breed :
- `show [ who ] of firms` => [0 3 2 1] (random order)
- `show sort [who] of firms` => [0 1 2 3]
- `show sort [who * who] of firms` => [0 1 4 9]
- `show sort [sales - costs ] of firms` => [100 110 115 150]

Computations with agents – 4

- Collecting the list of agents of the same type verifying a desired condition using `with`:
- `let champions firms with [profit > 1000]`
- Collecting the list of all ships placed on the same patch as the turtle:
- `let friends other ships-here`
- Creating an ordered list of agents given a criterium: `let increasing-price-firms sort-by [[price] of ?1 < [price] of ?2] firms`
- And execute instructions on agents selected the same order (instead of a random order): `foreach increasing-price-firms [ ask ? [ show price ] ]`

2 The common structure of NetLogo models

Main blocs of a typical NetLogo program

1. **Declarations** of global variables, agent breeds, and breed variables for each breed;
2. *setup* procedure:
   (a) **Initialization** of global variables (values read from the GUI, or fixed in the code);
   (b) **Creation** of the populations of each agent type (breed), and initialization of their breed variables;
   (c) Initialization of the ticks and of different output forms (graphics, CSV files, etc.);
3. A procedure (*go*) that groups all operations that take place in each "period" of execution of the model, and increments the period counter (*ticks*);

Other common components

- Complementary procedures that
  - manage the behavior of different types of agents;
  - collect data and statistics on these behavior, and on aggregate properties of the model;
  - update the outputs of the model (saving the data file, etc.).
- A **GUI** that contains different elements that allow to the user
  - to set the values of the parameters of the model;
  - to observe the evolution of the results from period to period.
- Comments for documenting the model.

Initialization and updating of the plots

- We use the plots to observe the evolution of important variables in real time
- Plots are created in the GUI
- NetLogo 5 automatically updates the plots:
  - using the initialization of the plots when the commands *reset-ticks* or *setup-plots* are executed in procedures *setup* and *go*;
  - and the update of the plots when commands *reset-ticks* (in *setup*), *tick* or *update-plots* (in *go*) are executed.

Help

Useful resources are:

- The documentation of NetLogo (Menu Help, User manual), and, especially,
- the Dictionary of NetLogo (Menu Help, Dictionary)
- and the Programming guide in the User manual.

Homework:

- Check the tutorials in the "Learning NetLogo" section of the NetLogo help
- Check in the NetLogo dictionary the meaning of each term indicated in blue in these slides

3 First example: A very simplistic predator-prey model

A very simple model

With Lambs running on then grass and eating the grass on the green patches, with:

- grass randomly growing on patches: with a probability of 3%, a patch becomes green;
- lambs eating the grass from the patches on which they are placed (the patch becomes black, and the lamb get some energy from grass);
- lambs moving randomly (choose a random direction, and advance one step, each step costs one unit of energy);
- lambs making a baby if they have enough energy to give birth ($birth - energy$), and the babies taking this energy from their parents;
- lambs dying when they do not have any energy left.

Step 1: Declarations and setup

- Declaration of global variables:
  - Number of lambs (`nbLambs`)
  - Quantity of grass (`grass`)
- Declaration of the agent breed, `lambs`
- Declaration of the breed variables (only one here): `energy`

Step 2: Setup

- Clear the memory and reset the ticks
- Setup patches by setting their color to green
- Create the lambs population given their initial number fixed in the GUI: `lambs-number`
- Place the lambs on random spots in the World

Step 3: Go

1. Move lambs (→ `move-lambs` procedure)
2. Let them eat the grass present at the arrival patches (→ `eat-grass`)
3. Check if some lambs are dead (→ `check-deaths`)
4. Living one reproduce (→ `reproduction`)
5. Count the current lambs population and the current quantity of grass
6. Grow grass on the patches (→ `grow-grass`)
7. Increment the period counter (`tick`)