An Introduction to NetLogo

Alessio Netti

University of Bologna

April 11, 2018
Outline

1. First Steps in NetLogo
2. Basic Programming in NetLogo
3. A Bit of (Useful) Functional Features
What is NetLogo?

NetLogo is a programmable modeling environment for simulating natural and social phenomena.

- NetLogo was written in 1999 by Uri Wilensky, and it was designed for modelling complex systems developing over time:
  - You can give instructions to hundreds or thousands of agents all operating independently;
  - Observe the emergence of global behaviors;
  - Connect micro-level behaviors of individuals and macro-level patterns emerging from interactions.

- NetLogo is based on the programming language Logo (Seymour Papert, MIT), which is a simplified dialect of LISP;

- It runs on the Java Virtual Machine, so it works on all major platforms.
Other Features

- Extensive documentation and tutorials;
- Models Library, a large collection of pre-written simulations covering topics in:
  - Biology and medicine;
  - Physics and chemistry;
  - Mathematics and computer science;
  - Economics and social psychology.
- Free and open source;
- Fully programmable;
- Double precision floating point math;
- First-class function values (aka tasks, closures, lambda);
- And much more.
The NetLogo Environment

- **Simulation** performed in **Interface**;
- **Information** contains the documentation;
- **Code** contains... the code;
- The black window is the world.
World and Agents

- In NetLogo the world is intrinsically **discrete**.
- The world is a **grid** whose basic regions are called **patches**.

![Cartesian coordinate; Each patch is a 2 x 2 grid cell; Each patch is identified by the coordinate at its center.](image-url)
World and Agents

- The world is inhabited by agents called **turtles**;
- Turtles are movable entities within the world;
- Turtles have a **position** and a **heading**.

- The **position** of a turtle is given as grid coordinates;
- The **heading** is expressed in degrees (0…360), where 0 is north, 90 est etc....
World and Agents

Turtles are movable entities within the world.
World and Agents

Remark

We said that in NetLogo simulations are essentially discrete: the world (the space) is discrete (grid, patches, ...). Time is discrete too.
Turtles

- Turtles have several attributes, e.g.
  - `shape` here is the default one (other options e.g. `person`);
  - `size` refers to the size of a turtle with respect to the patch size (default is 1).
Turtles

- Turtles can be **visible** or **hidden**;
- In NetLogo turtles have a special properties **hidden?**. This property has a boolean value: true if the turtle is hidden, false otherwise.
Properties

- who
- heading
- xcor
- ycor
- shape
- size
- color
- hidden?

Unique id for each turtle in NetLogo
The Observer

- When you open NetLogo the world is empty (no turtle): you have to create turtles;
- There is a special agent, called the **Observer** that take care of this kind of actions.
- The observer has statements or **commands**.
You can now modify properties of \texttt{turtle 0}. 

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\texttt{who} & & & & & & \\
\texttt{heading} & & & & & & \\
\texttt{xcor} & & & & & & \\
\texttt{ycor} & & & & & & \\
\texttt{shape} & & & & & & \\
\texttt{size} & & & & & & \\
\texttt{color} & & & & & & \\
\texttt{hidden?} & & & & & & \\
\hline
\end{tabular}
Some Commands

- The observer gives commands to turtles and patches.
  - `create-turtles 1`: creates one turtle (who is `turtle 0`) in the centre of the world;
  - `inspect turtle whoID`: shows properties of `turtle whoID`;

- The `ask` command specifies command to be run by turtles or patches: `ask somebody [ setOfComands ]`
  - `ask turtle whoID [ set color red ]`: set the color of `turtle whoID` to red;
  - `ask turtles [ set color red ]`: set the color of all turtles to red;
  - `ask patch 2 3 [ set pcolor green]`: set the color of the patch with center (2,3) to green (note that we used `pcolor`).
Remark
In an ask command, the actions to be executed must be consistent with the agent we are asking to execute those very actions.

```
ask turtle whoID [ set color green ]
;; this makes sense
ask turtle whoID [ create-turtles 1 ]
;; this doesn’t make sense
```
Moving Turtles

- We can move turtles forward and backwards with the commands `forward <distance>` and `back <distance>`:

  ```
  ask turtle 0 [ forward 2 ]
  ```
Moving Turtles

- Turtles can turn left and right: `left <degree>`, `right <degree>`:

  ```
  ask turtle 0 [ right 90 ]
  ```
Outline

1. First Steps in NetLogo
2. Basic Programming in NetLogo
3. A Bit of (Useful) Functional Features
Introduction

- To make interesting simulations we need to be able to write programs and functions implementing agents’ actions and interactions;
- The programming language used is essentially Logo (procedural programming);
- You can write programs in the ‘code’ tab.
- We will be able to cover only few aspects of programming in NetLogo. We will see procedures, variables, tasks and basic data structures in NetLogo. For what concerns the latter, we focus on lists. We will then finish with a quick overview of (some of the) functional features of the language.
Instructions tell agents what to do. We can divide instructions according to three criteria:

- Whether they are built into NetLogo (primitives) or whether they are implemented by the user (procedure);
- Whether the instruction produces an output (reporters) or not (commands);
- Whether the instruction takes inputs or not.

The main reference for what we are going to see is the NetLogo Dictionary (file:///Volumes/NetLogo-5.3.1/NetLogo%205.3.1/docs/index2.html).
Procedures (Commands)

- We distinguish between **commands** and **reporters**;

  - **Commands** are procedures that do not output (only side effects):

    ```
    to <commandName>
    <commands>
    end
    ```

  - **Example:**

    ```
    to go
    clear-all ;; reset the world
    create-turtles 100 ;; create 100 turtles
    ask turtles [ forward 1 ] ;; all turtles move fd of 1
    end
    ```
Procedures (Inputs)

- Procedures can take inputs:
  \[
  \text{<procedure>} \ [ \ <\text{par1}> \ <\text{par2}> \ldots \ <\text{parn}> \ ].
  \]
  For commands we have

  \[
  \text{to} \ \text{<commandName>} \ [ \ <\text{formalParameters}> \ ]
  \\
  \text{<commands>}
  \\
  \text{end}
  \]

- Example

  \[
  \text{to} \ \text{createNumTurtles} \ [ \ \text{num} \ ]
  \\
  \text{create-turtles} \ \text{num}
  \\
  \text{end}
  \]
Procedures (Reporters)

- **Reporters** are procedures that compute a result and report it:

  ```
  to-report <reporterName> 
  <body> 
  report <reportValue> 
  end 
  ```

- **Example**

  ```
  to-report double [ num ] 
  report 2 * num ;; note that there is no parentheses 
  end 
  ```
Procedures

Remark

Note the particular use of spaces between square brackets and the lack of parentheses.

For what concerns the former, NetLogo is much more flexible than Logo (but we will still be quite strict using spaces). For the latter, this is possible thanks to the evaluation strategy used by NetLogo.

We will see more about that later 😊.
Procedures

Style Guide

There is no official NetLogo style guide (the NetLogo Models Library is stylistically fairly consistent, so it can serve as a guide by example). Nonetheless, there are good habits:

- Use camel-case instead of hyphenated long names, beginning with a lower-case letter i.e. write myProcedure rather than my-procedure (hyphenated names are a convention in Lisp derived languages);
- Do not use underscores in names;
- Name command procedures with nouns and reporter procedures with verbs;
Procedures

Style Guide

- Identify procedure context with a comment:

```netlogo
to move [ num ] ;; turtle procedure
  right num
  forward 1
end
```

Remark

NetLogo is **case insensitive**, so case conventions are purely for reader convenience.
Variables

In NetLogo variables are used to store values. We can divide variables in:

- **Global variables**: they have only one value. Every agent can access it;

- **Agent variables** (e.g. turtle variables): each agent has its own value for each agent variable. There are both built-in and user-defined agent variables.

- **Local variables**: variables that are defined and used only in the context of a particular procedure or part of a procedure.
Global Variables

- Global variables can be read and set by any agent at any time (by default global variables are set to zero);

- Global variables are declared with the keyword `globals`:

  ```netlogo
  globals [ <variableNames> ]
  ```

  E.g.

  ```netlogo
  globals [ score counter ]
  ;; declare the global variables score and counter
  ```

- To set a global variable we use the keyword `set`:

  ```netlogo
  set <variableName> <value>
  ```
Agent Variables

- Agent variables can be both built-in and user-defined;
- Built-in turtle variables are color, xcor, heading etc.
- To define a new agent variable we use the constructor

```
<agentType>-own [ <variableName> ]
```

where agent types own are turtles-own, patches-own, links-own. E.g.

```
turtles-own [ energy ] ;; each turtle has its own energy
```
Agent Variables

- Each agent has direct access to its own variables, both for reading and setting.

- Other agents can read and set a different agent’s variables using `ask`:

  ```
  ask turtle 0 [ show color ] ;; print turtle’s 0 color
  ask turtle 0 [ set color blue ]
  ;; turtle 0 becomes blue
  ```

- An agent can also read a different agent’s variables using `of`:

  ```
  show [ color ] of turtle 0 ;; print turtle’s 0 color
  ```
So far we have asked either to **single** agents to do something 
\( \text{ask turtle 0 [ doSomething ]} \) or to **all** agents to do 
something \( \text{ask turtles [ doSomething ]} \);

We can also ask to specific sets of agents to execute actions. 
This is done via the so-called **agentsets**;

An agentset is nothing but a set of agents. An agentset can 
contain either turtles, patches or links, but not more than one 
type at once;

An agentset is not in any particular order. In fact, it’s always 
in a random order. And every time you use it, the agentset is 
in a different random order.
Excursus on Agentsets

- There are several primitives to construct agentsets. What is really interesting is that you can construct agentsets that contain only some agents (e.g. all the red turtles);
- You can also filter agentsets by a particular attribute:

```
ask one-of turtles [ set color green ]
;; make a randomly chosen turtle turn green
patches with [ pxcor < 3 ]
;; create the agentset of patches with x coordinate smaller than 3
to changeColor ;; let is explained in the next slide
let green-turtles turtles with [ color = green ]
;; create the agentset of green turtles
ask green-turtles [ set color red ]
;; make green turtles red
end
```
Local Variables

- A local variable is defined and used only in the context of a particular procedure or part of procedure;
- To create local variables we use the keyword `let`;

```
let <variableName> <value>
```

- E.g.

```
to-report sumOfThree [ num1 num2 num3 ] ;; report num1 + num2 + num3
let numTemp num1 + num2 ;; store num1 + num2
report numTemp + num3 ;; report the sum
end
```
Aside: Minimal Program Structure

Every (well-written) NetLogo program should be divided in at least three parts:

- **Variable part**: it contains the declaration of global and agent variables (in this order);
- **Setup part**: this part consists of a single procedure setup that initializes global variables, create agents and does other setup operations;
- **Go part**: this part consists of the main procedure go that implements one cycle of the simulation.
Logo is a dynamically-typed language;

We have the following basic types:

- **Numbers**: floating point numbers;
- **Booleans**: true or false;
- **Lists**: ordered, immutable collections of objects;
- **Strings**: immutable sequences of characters; create with double quotes.

We have the usual operators:

- numerical: +, -, /, ^;
- relational (for comparisons): >, >=, =, !=, <, <=;
- logical: and, or, xor, not.
Basic Types

Remark

- All numbers are floating points...remember you are working with approximations!

- The value of $0.1 + 0.2$ is $0.30000000000000004$, so the value of $0.1 + 0.2 = 0.3$ is false, and the value of $0.1 + 0.2 > 0.3$ is true.

- Note again the lack of parentheses: this requires special care in using spaces. NetLogo distinguishes between expressions like $-3$ and $-3$ (this is the ‘correct’ one).
Aside: Conditionals and Loops

- We can use booleans for conditional code execution.

```
if ( <boolCondition> ) [ <commands> ] ;; note parentheses
ifelse ( <boolCondition> )
[ <commands4true> ]
[ <commands4false> ]
ifelse-value ( <boolCondition> )
[ <reporter4true> ]
[ <reporter4false> ]
```

- Example:

```
if ( random-float 1 < 0.5 ) [ show "heads" ]
;; random float num return a floating number between
;; 0 and num
```
Aside: Conditionals and Loops

Example:

```netlogo
ifelse ( random-float 1 < 0.5 )
[ show "heads" ]
[ show "tails" ]
```

NetLogo also provides the `ifelse-value` primitive, which allows condition determination of a value.

```netlogo
ask turtles [
set color ifelse-value ( wealth < 0 ) [ red ] [ blue ]
]
```
NetLogo offers several looping constructs. One is `loop`:

```
loop [ <commands> ]
```

The list of commands `<commands>` is repeatedly executed (potentially forever);

We can use `stop` to exit a loop: if one of the commands eventually calls `stop`, you will exit the loop.

```
loop [ ifelse ( counter > 100 )
[ stop ]
[ set counter counter + 1 ]
]
```
Aside: Conditionals and Loops

- We can also use `repeat` for bounded iteration:

\[
\text{repeat } \langle \text{num} \rangle \ [ \langle \text{commands} \rangle ]
\]

The list of commands \(<\text{commands}>\) is executed exactly \(<\text{num}>\) times.

Remark

For most of the tasks we will encounter loops are not needed. In NetLogo, we usually program a single cycle/step of the simulation (via the procedure `go`), and then use a forever button in order to repeat that cycle forever. We can click again on the forever button to exit the loop.
Lists

- Lists are containers of elements in a fixed order. In NetLogo lists are:
  - Immutable,
  - Ordered,
  - Potentially heterogeneous.

- We can construct lists with the `list` constructor (note parentheses):

  ```
  ( list ) ;; empty list
  ( list 0 "one" whatever ) ;; list of three items
  ```

  The usual square brackets notation can be used as well.
Lists

- Usual lists’ functions are built-in in NetLogo (see Dictionary), e.g.

  ```
  sentence list1 list2 ;; concatenates list1 and list2
  ```

- Lists are interesting also because they allow to see some basic functional style in NetLogo.

Remark

Lists are immutable: `sentence 11 12` returns a `new` list, which is nothing but the concatenation 11 and 12.
Creating lists with n-values

- **n-values** takes as inputs the number of values to produce, and a **reporter task** (next slides). The output is a list, where each item is a function of its index.

- That is, we can imagine a reporter task to be a unary function `fun`. Then **n-values num fun** simply constructs the list 
  
  \[
  \text{[fun 0, fun 1, \ldots, fun (num - 1)]}
  \]

  (this is NOT NetLogo notation).

- Compare **n-values num fun** with 
  
  map fun [0 .. num - 1]
Outline

1. First Steps in NetLogo

2. Basic Programming in NetLogo

3. A Bit of (Useful) Functional Features
Tasks and Higher-Order Functions

- **n-values** is a first example of a basic functional features in NetLogo. It is a procedure that takes another procedure as input, that is **n-values** looks like an higher-order procedure;

**Remark**

An higher-order function/procedure is a function/procedure that can take functions/procedures as arguments.

- Properly speaking, (Net)Logo is not an higher-order language: we cannot pass procedure as actual parameters to other procedures;
- Nevertheless, we can easily simulate an higher-order behaviour using tasks;
- Tasks are essentially procedures regarded as values.
Tasks

- Like procedures, tasks store code for later execution, but unlike procedures, tasks are values and thus can be passed around like any other value.

- Tasks are declared via the keyword `task`:

```
task [ <body> ]
```

- Analogously to procedures, we have command tasks and reporter tasks.

- The formal arguments of a task are written as ?1, ?2, ?3, etc (if a task has only one formal argument, the latter can be written as ?).

- ? is a special variable that cannot directly be set by the user.
Tasks

- A task `myTask` with arguments `?1, . . ., ?n` intuitively corresponds to the procedure `myTask [ ?1 ?2 . . ?n ]` regarded as a value (cf. functional abstraction);

- A command task is used to run code without returning a value. To evaluate the procedure ‘described’ by a task we use the primitive `run`:

```
globals [ stack push ]

to setup
  set stack [] ;; set the stack to be empty
  set push task [ set stack lput ? stack ]
  ;; set push to be the function that
  ;; given x sets stack to x :: stack (append)
end
```
Tasks

- Then \((\text{run} \ \text{push} \ 1)\) pushes a 1 on our stack stack.
- Note that \text{run} requires parentheses. In the previous example \((\text{run} \ \text{push} \ 1)\) works (NetLogo understands quite a lot of programs, even if these are not as strict and formal with parentheses as we required) but \text{run push 1} fails.
- The parentheses determine what is considered to be an input to the task.
Tasks

- A **reporter task** is a task that returns a value;
- We can evaluate the procedure ‘described’ by a reporter task with the primitive `runresult`.

```plaintext
let square task [ ? * ? ]
print ( runresult square 5 )
```

- Again, note that `runresult` requires the presence of parentheses.
A Tiny Bit of Functional Style in NetLogo

- Having tasks we can simulate higher-order procedures. This allows us to write some NetLogo programs in functional style.
- Benefits of functional style are well-known, and using it to write (and think about) your NetLogo simulations can make your code much more elegant and compact.
- We only see a couple of constructs, namely map, filter and reduce (fold).
An Example

;;; Call a payoff a function from numbers to numbers.
;;; We want to define an update function that takes a payoff
;;; p and returns a new payoff p’ such that for all n
;;; p’ n = (p n) + 1
;;; Easy: we model payoffs as tasks taking one argument
;;; (which is exactly what they are!)

to-report update [ payoff ]
report task [ ( runresult payoff ? ) + 1 ]
end

to-report apply [ payoff argument ]
report ( runresult payoff argument )
end

;;; Thus: apply (update p) n = (apply p n) + 1 ;}
Map, Filter and Reduce

- These are basic constructions that will make your life easier;
- **map** takes a reporter-task, a list, and returns a new list whose elements are the result of the reporter-task applied to each element of the list:

  \[
  \text{map} \ <\text{reporter-task}> \ <\text{list}>
  \]

  ;; informal notation: \( \text{map} \ f \ [x_1 \ldots \ x_n] = [f \ x_1 \ldots \ f \ x_n] \)

- **filter** takes a predicate i.e. a boolean-valued reporter-task, a list, and returns a new list obtained from the first one by filtering the elements that satisfied the predicate:

  \[
  \text{filter} \ <\text{predicate}> \ <\text{list}>
  \]

  \[
  \text{filter} \ [ \ ? \ < \ 3 \ ] \ [ \ 2 \ 3 \ 4 \ 1 \ ] ; ; \text{returns} \ [ \ 2 \ 1 \ ]
  \]

  ;; note: we do not need to use the keyword task
Map, Filter and Reduce

- **reduce** (aka fold) takes a reporter-task and a list: it reduces the list from left to right using the given task, resulting in a single value. If the list has a single item, that item is reported. It is an error to reduce an empty list.

```
reduce <reporter-task> <list>

;; reduce [ ?1 + ?2 ] [ 1 2 3 4 ] => ((1 + 2) + 3) + 4
;; returns the minimum xs
```
Evaluation in (Net)Logo

- We saw some basic features of the (Net)Logo programming language. You have noticed the strange use (better, the almost complete absence) of parentheses.

- One could think that (Net)Logo implicitly associates to the left or to the right:

\[
a \ b \ c \ ;; = a \ (b \ c). \ Problem: \ update \ payoff \ arg
\]

\[
a \ b \ c \ ;; = (a \ b) \ c. \ Problem: \ sentence \ sentence \ xs \ ys \ zs
\]
Evaluation in (Net)Logo

- But what happens then?
- The evaluation of e.g.

```
sentence sentence xs ys zs
```

works as follows:
The evaluator proceeds from left to right: it reads sentence and computes its ariety: 2. It now proceeds trying to saturate sentence with two values.

It reads the second sentence. Proceeds as in previous point, giving priority to saturating the second sentence.

It reads xs. This is a value, and it is used to saturate sentence. To completely saturate sentence another value is needed.

It reads ys. Proceeds as in previous point. It has completely saturated the second sentence, so it can evaluate it, obtaining a value. Priority goes back to the first sentence and the value obtained from the evaluation of sentence xs ys is used to saturate (the first) sentence.

It reads zs. Since it is a value it is used to completely saturate sentence. This is now evaluated and the value obtained is returned.
Concluding Remarks

- There are lots of things we did not see. Examples are breeds and synchronization (but we are going to see the latter in examples). You will likely need to study some of these in the future.

- NetLogo is really easy to use, and you can learn it in a few hours (as in acquiring not-too-deep skills in writing simulations). Moreover, there are a lot of tutorials on the web.

- As usual, the more you practice the more you learn.
A good starting reference is the NetLogo homepage: https://ccl.northwestern.edu/netlogo/.
There you can find links to several references (ranging from books to university classes).

Going through the official documentation while practicing with the language is always a very good way to learn.

Finally, you can also find several online video tutorials.

These slides are based on those created by Francesco Gavazzo for the past iterations of the CSNS course.