NetLogo
A Basic Introduction

Alan G. Isaac

American University

2016-02-10

Overview

1. What Is NetLogo?
   - Background
   - NetLogo: A First Look
   - NetLogo Resources
2. Basic Concepts
   - Basic Concepts: Command Center
   - Basic Concepts: Patches
   - Basic Concepts: Turtles
   - Basic Concepts: Agentsets
   - Basic Concepts: Global Variables
   - Basic Concepts: Procedures
   - Basic Concepts: Plotting
   - Basic Concepts: Lists
   - Advanced Concepts: Breeds and Links
   - Models Library

Goals

After mastering this basic introduction, you will be able to:

- briefly describe the history of the NetLogo programming language
- describe key NetLogo features
- use the NetLogo Command Center to do calculations
- understand the use of NetLogo's `Interface`, `Info`, and `Code` tabs
- know how to find help in the NetLogo Dictionary
- use the Models Library to find & experiment with existing models
- explain the intent and usefulness of the following example models: Party and Wolf-Sheep Predation

What Is NetLogo?

programming language

- procedurally focused
- derived from Logo (a dialect of Lisp)

agent-simulation toolkit

- language includes many primitives for agent-based modeling
- good GUI support, including real-time charts & visualizations

agent-based modeling environment

- spatially located agents (2D or 3D)
  - stationary agents (patches)
  - mobile agents (turtles)
- other agents
  - observer (global command interpreter)
  - links (relationships between turtles)

Logo

- member of Lisp family
- invented in the late 1960s by a group of scientists led by mathematician, computer scientist and educator Seymour Papert
- substantial graphical capabilities
- designed for learning: intended to be usable by children ("no threshold, no ceiling" [harvey-1993-web])

More history can be found at the Logo Foundation: `http://el.media.mit.edu/logo-foundation/what_is_logo/history.html`

Logo Turtle

- 1969 Seymour Papert's turtle http://cyberneticzoo.com/wp-content/uploads/22-turtle2-x640.jpg
- by early 1970s, the Logo drawing object was represented as a turtle.
- move the turtles around on the screen with Logo commands.
- mobile NetLogo agents are still called turtles, for this reason

More history from Reuben Hoggett: http://cyberneticzoo.com/cyberneticanimals/1969-the-logo-turtle-seymour-papert-marvin-minsky-et-

More history from Cynthia Solomon http://logothings.wikispaces.com/

About NetLogo

- Related to Logo and StarLogo
    - Resnick's StarLogo added multiple agents and patches
- designed by Uri Wilensky and Seth Tisue (Center for Connected Learning and Computer-Based Modeling at Northwestern University)
    - adds agentsets and concurrency to StarLogo
    - written in Java and Scala, but does not require any knowledge of these (and provides its own, private Java installation)
- free download and (as of version 5) open source

References: [wilensky.resnick-1999-jset], [tisue.wilensky-2004-wp]

Compared to other Logos: `http://ccl.northwestern.edu/netlogo/docs/programming.html#syntax`

NetLogo Limitations

- limited support for reading or writing binary files
- limited support for object-oriented programming (e.g., no inheritance or data hiding)
- command center is not a full-fledged interpreter (e.g., it does not support declaration of new global variables or procedure defintions)

NetLogo GUI Features

- simple GUI addition of sliders and switches (for parameter setting)
- simple GUI addtion of charts and monitors (for display of simulation data)
- GUI automatically displays spatially located agents (patches and turtles)
- save complete state of the world (in a format that can easily be re-opened or parsed with other software)
- export the contents of the graphics window as an image
- turn the images into a movie

NetLogo Models Library

A key attraction of NetLogo for teaching is that it ships with an extensive vetted collection of models.

Advanced NetLogo Features

- supports 3D modeling
  - patches become cubes
  - see examples in Models Library
- ships with important language extensions (e.g., `array`, `table`, `matrix`, `nw`, and `csv`),
- supports user-written extensions
- export and import data in standard formats (including bitmap imports, via an extension)
- scriptable with Java - provides an application programmers interface (API) requiring just a little Java

Putting NetLogo Models on the Web

`http://ccl.northwestern.edu/netlogo/docs/applet.html`

SaveAs > Applet

- Java Virtual Machine (JVM) included (solves problems: old versions work!)
- applet can read and write files on your computer, but not on your users' computers
- however, this is a deprecated technology, and the replacement (NetLogo Tortoise) is still in the works.

Goals

After mastering this section, you will be able to:

- explain the role of the 3 tabs in the NetLogo GUI
- make basic use of the Command Center
- explain how NetLogo uses the terms ‘agent’
- find and use the NetLogo Dictionary

Tabbed User Interface

Upon start up, you will notice three tabs. In NetLogo 5 these are named as follows.

Interface: the GUI for the model;
includes the Command Center, where we can enter NetLogo commands.

Info the documentation editor; provides an area to store and edit the model documentation

Code the code editor; provides an area to view and edit the model code
(Syntax highlighting is added when code is displayed in the `Code` tab, but this is not part of the code.)

First Look: Command Center

Command Center output area, and a command line.

To the left of the command line your should see the observer> prompt. (If not, click it and choose that prompt.) Later we will explain why it is so named. Enter the following lines at the NetLogo command line. (We will explain the synatx later.)

```
print "Hello World!"
print 1 + 1
```

Notice that each command and its result appear in the output area (usually, above the command line).
To recall the same command to the command line, use the up arrow key to scroll through your command history.

First Look: Patches

Patches are:

- automatically created at start up
- laid out in a rectangular grid
- centered at integer locations
- immobile
- identified by a fixed location (e.g., `patch 0 0`)

Visual display:

- colored, usually black by default
- displayed in the model's *Graphics Window*. (so the Graphics Window will iniitally look like a black square).

Basic Interaction with Patches

Enter the following at the NetLogo command line to talk to one patch. (We will explain the synatx later.)

```
ask patch 0 0 [set pcolor red]
```

The NetLogo keyword `patches` refers to the entire “agentset” of all patches. If we `ask patches` to do something, every patch will do it, one after the other. E.g.,

```
ask patches [set pcolor one-of [white blue]]
```

You should see the patches change color.

> Exercise: Turn all patches black.

Randomness

Randomness is common in agent-based modeling. Here is a short selection of NetLogo examples:

`ask`	iterates through an agentset in random order
`one-of`	reports a random choice from an agentset or list
`random` *n*	produces a random integer in $[0..n)$
`random-float` *x*	produces a random number in $[0..x)$

random-seed

In the “old days”, one fetched the random numbers for an experiment from a book of random numbers.

The process: open the book arbitrarily to pick a starting page, column, and row, and then read off the numbers sequentially from that point.

Nowadays the whole process is implemented on a computer, using a pseudo-random number generator (PRNG). Setting `random-seed` is like picking the page column and row.

Replicability requires that each simulation must specify its seed. Usually you will do this in you *setup* procedure.

First Look: Turtles

Turtles are:

- not automatically created at start up
- drawn on top of the patches
- centered at floating-point locations within the dimensions of the world
- mobile (can change location)
- identified by order created (e.g., `turtle 0`)
- have a `heading` attribute (direction for movement)

Visual display:

- colored, usually random by default
- displayed in the model's *Graphics Window*, on top of the patches

Basic Creation of Turtles

Enter the following command at the NetLogo command line.

```
create-turtles 10
```

This will create 10 turtles, all at the default location `(0,0)`. You should see ten turtles appear as arrowheads (the default shape) at the center of the Graphics Window,

Basic Interaction with Turtles

After creating your turtles, enter the following command at the NetLogo command line. (We will explain the synatx later.)

```
ask turtle 0 [forward 3] ;interact with one turtle
```

Here `turtle 0` is the first turtle that you created. (This is explained below.) The NetLogo keyword `turtles` refers to the entire “agentset” of all turtles. If we `ask turtles` to do something, each turtle will do it, one after the other. E.g.,

```
ask turtles [forward 3] ;interact with all turtles
```

You should see each turtle move in the direction it is pointed. (That direction is called its `heading`.)

NetLogo Agents

A NetLogo agent is able to process commands and own variables. There are four basic types of NetLogo agents.

observer
: the global command processor (interpreter environment)

patch
: stationary agent
  - square divisions of the world over which turtles move
  - attributes often represent environmental properties

turtle
: mobile agent (can move across patches)
  - people often mean “turtle” when they say “NetLogo agent”
  - can be subtyped (by using `breed`)

link
: embodies a relationship between turtles
  - can be directed or undirected
  - can be subtyped (`directed-link-breed` or `undirected-link-breed`)

Documentation: `http://ccl.northwestern.edu/netlogo/docs/programming.html#agents`

What is a NetLogo Agent?

NetLogo uses the term “agent” more broadly than is typical in the ABS literature. A NetLogo agent can execute commands and “own” variables. (We will say that global variables are owned by the observer.)

- most modelers do not refer to the equivalent of the observer as an agent,
- most modelers do not refer to the equivalent of a link as an agent,
- many modelers do not refer to the equivalent of a patch as an agent,
- the most common ABS use of the word ‘agent’ corresponds to a NetLogo turtle (or sometimes a patch)

Commands and Reporters

In NetLogo, we ask agents (e.g., patches or turtles) to run commands or reporters.

- command: tells an agent to do something, but returns no value
- reporter: tells an agent to calculate a value and return it for further use

User-defined reporters and commands are called *procedures*.

User Manual

The core NetLogo resource is the NetLogo User Manual.
`https://ccl.northwestern.edu/netlogo/docs/`
This includes tutorials, reference guides, and the NetLogo Dictionary.

NetLogo Dictionary

- alphabetical list of primitives
- also grouped by type

The NetLogo Dictionary lists NetLogo's primitives: reporters and commands that are built into NetLogo.
NetLogo comes with a copy of the Dictionary:

```
``Help > NetLogo Dictionary``
```

Make the NetLogo Dictionary your constant companion when working with NetLogo.
If NetLogo restricts the type of agent that can run the primitive, an icon showing the allowable context appears in the Dictionary.

Guides

Interface Guide `http://ccl.northwestern.edu/netlogo/docs/interface.html`

- world settings: size, topology (wrapping vs. reflecting)
- menu items, controls

Programming Guide `http://ccl.northwestern.edu/netlogo/docs/programming.html`

Other NetLogo Resources

- discussion group: `http://groups.yahoo.com/group/netlogo-users/`
- StackOverflow: `http://stackoverflow.com/questions/tagged/netlogo`
- homepage includes mail lists, documentation, and example models `http://ccl.northwestern.edu/netlogo/`
- onlines articles and papers, e.g.,
  - `http://ccl.northwestern.edu/papers/`
  - `http://ccl.northwestern.edu/papers/netlogo-outside.html`

Other Useful Resources

- The NetLogo Models Library `Social Science` models
- The NetLogo Models Library `Code Examples`
- TurtleZero: `http://www.turtlezero.com/`
- Modeling Commons: `http://modelingcommons.org`
- OpenABM: `https://www.openabm.org/`
- StackOverflow `http://stackoverflow.com/questions/tagged/netlogo`

Download, Install, and Run (Windows version)

- Download
  - Go to: `http://ccl.northwestern.edu/netlogo/`
  - Download link on the left
  - Save to desktop or accept the default
- Install
  - Double-click on downloaded file to install (NetLogo4.1Installer.exe)
  - Accept all defaults
- Run (Windows version)
  - Start > All Programs > NetLogo > NetLogo 4.1

Goals

After mastering this section, you will be able to:

- use the NetLogo Command Center to do calculations
- manipulate patches and turtles from the command center
- distinguish observer context, patch context, and turtle context

Introduction to the Observer

- NetLogo has a kind of central planner, known as the “observer”.
- The “observer” manages your simulations by responding to your commands.
- When you ask the observer to execute a command, we say that the command is executed in **observer context**.
- Some commands can only be executed by the observer.

Using NetLogo as a Calculator

Try entering the following in the Command Center, in the order presented. (Not all will execute successfully!)
Make sure you are in observer context.

```
print 2+2
print 2 + 2
PRINT 2 + 2
let x -2
print x
let x -2 print x
let x -2 print -x
let x -2 print (- x)
let x -2 set x (x + 1) print x
```

Calculator: Lessons Learned

- Arithmetic operators must be surrounded by white space.
- We define local variables with the `let` command. (We do *not* use the equals sign for assignment!)
- NetLogo is case-insensitive
- The local variables we define with `let` are local to their code block. (Each line we enter in the command center executes as a separate code block.)
- Although `-2` is a literal number, we must use `(- x)` to produce the additive inverse of the variable `x`.
- *after* we have defined a local variable with `let`, we can change its value with `set`

Booleans

NetLogo has boolean primitives, `true` and `false`. These can be combined with the boolean operations `not`, `and`, and `or`.

`not` <bool> report `true` if bool is `false`; report `false` if bool is `true`;

<bool01> `and` <bool02> report `true` if bool01 and bool02 are `true`; otherwise report `false`

<bool01> `or` <bool02> report `true` if bool01 or bool02 are `true`; otherwise report `false`

<bool01> `xor` <bool02> report `true` if bool01 or bool02 (but not both) are `true`; otherwise report `false`

Comparisons

Comparisons (e.g., >, >=, <, <=, =, ! =) produce a boolean result. Try entering the following commands at the NetLogo command line:

```
print (1 + 2 = 3)
print (1 + 2 <= 3)
print (1 + 2 >= 3)
print (1 + 2 != 3)
print (1 + 2 < 3)
print (1 + 2 > 3)
```

The first three commands should print `true`; the last three should print `false`.

Repeated Execution of Code Blocks

The repeat command allows you to repeat a command block as many times as you wish. http://ccl.northwestern.edu/netlogo/docs/dictionary.html#repeat

If you want to repeatedly execute a block of commands a known number of times, use repeat. For example:

```
let x 0 repeat 50 [set x (x + 1)] print x
```

Note how we use brackets to delimit a command block.

Explain why the observer prints 50 if you enter this code in the Command Center.

Question: What happens if we move the closing bracket to after print x?

Question: What happens if, after executing the above code, we enter print x at the NetLogo command line?

Limitations of the Command Center

The command center is not a full-blown interpreter.

You cannot:

- define procedures
- introduce new global variables
- introduce new local variables for use in more than one input line

However, you can use any global variables or procedures that you define in the `Code` tab. (See below.)

Context in NetLogo

- context: who is acting
- default context: the observer
- change context with `ask`:

  `ask turtle 0 [ ... ]` sets context to turtle 0

  `ask turtles [ ... ]` sets context to turtles
- agents can ask other agents to do things

  `ask turtles [ ask links [ ... ] ... ]`

Context in the Command Center

- at the bottom of the command center is the NetLogo command line, where you can enter commands
- click on the arrow to the left of the command line, and you will see the different "contexts" in which commands may be executed
- there are four types of NetLogo "agent": observer, patches, turtles, and links

`observer` context give instructions to the observer; (use `ask` to instruct the observer to pass commands to agents)

`patches` context give instructions to patches

`turtles` context give instructions to turtles

`links` context give instructions to links

The command center defaults to "observer context": commands are sent to the observer.

The patches Context in the Command Center

To the left of the NetLogo command line, you will see the “command context” (*observer* is the default) and an arrow to change it. Change the command context from `observer` to `patches`. We can now directly ask all patches to do things. E.g.,

```
set pcolor random-float 140
```

Context at the Command Center

At the *Command Center*, switch to `turtles` context. Now the commands you enter are given to all turtles. Essentially, instead of having to type:

```
ask turtles [<commands>]
```

this allows you to simply type:

```
<commands>
```

You may still talk to a single turtle at the Command Center, even though you are in the turtle context at the time:

```
if (who = 0) [set color red]
```

If you switch to the observer context, you could enter this same request as:

```
ask turtles [if (who = 0) [set color red]]
```

However, the observer context offers a simpler (and much more efficient) phrasing:

```
ask turtle 0 [set color red]
```

Goals

After reading this section you will be able to

- explain what patches are and describe some of their built-in attributes.
- explain what patch coordinates are and describe how they relate to the NetLogo world size
- explain how to get and set attributes for any patch
- explain how to add user-defined attributes to patches

Patches

Patches are

- stationary NetLogo “agents” (i.e., they have a fixed location)
- present at NetLogo start up (i.e., they are not created by user code)

Patch Attributes

Patches have a small number of pre-defined attributes. These are described in the documentation:

Documentation: `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#builtinvariables`

Patch Monitor

Patch attributes can be inspected by opening a patch monitor. E.g., to see all the bulitin attributes:

- go to the NetLogo `Interface` tab
- at the command line, enter `inspect patch 0 0`

The `inspect` command opens a graphical “patch monitor”.
The monitor displays current attribute values. (These are dynamically updated if they change.)
To close the turtle monitor enter `stop-inspecting patch 0 0` at the command line. (Or click the monitor’s `close` button.)
If inspect a patch in a new NetLogo window (no model), you will see the builtin patch attributes. However a model may declare additional attributes.

Patch Coordinates

Each patch has unique integer coordinates, which designate the center of the patch.

`pxcor`	the x-coordinate of the patch
`pycor`	the y-coordinate of the patch

Immutable Attributes

Patches are stationary agents; they cannot move. (The location of a patch never changes.) So the pxcor and pycor attributes are *immutable*: they cannot be changed.

It is not currently possible to add immutable attributes to patches.

Attribute Getting

Any agent can directly access its own attributes.

```
ask patch 0 0 [show pxcor]
```

Note when you enter these commands at the Command Line, the output specifies the agent who is doing the showing. This is a difference between `show` and `print`.
In this case, we see that `patch 0 0` does the showing.

Attribute Getting (of)

We can also use the NetLogo keyword `of` to access agent attributes. http://ccl.northwestern.edu/netlogo/docs/dictionary.html#of

Example: let `mypatch` be a patch.

```
show [pxcor] of patch 0 0
```

Note the mandatory brackets.

The same value is show, but in this case, we see that `observer` does the showing.

Patch Coordinates and World Size

The extent of the patch coordinates constitute the extent of the NetLogo world.

- the patch coordinates are integer values, where `min-pxcor` <= `pxcor` <= `max-pxcor` and `min-pycor` <= `pycor` <= `max-pycor`
- `world-width` reports `max-pxcor - min-pxcor + 1`
- `world-height` reports `max-pycor - min-pycor + 1`

You can think of `max-pxcor`, `min-pxcor`, `world-width`, and `world-height` as observer attributes.

```
world-width * world-height = count patches
```

Mutable Attributes

Other patch attributes are *mutable*: they can be changed.
The most commonly changed built-in attribute is `pcolor`. In the `Interface`, each patch displays a color, determined by its `pcolor` attribute.

Attribute Setting with ask

To reset an attribute value, we must `ask` an agent to reset the value. Most often, we tell the observer to `ask` the agent.
Example:

```
ask patch 0 0 [set pcolor red]
```

Note the mandatory brackets, which delimit a code block. (But the attribute name is **not** separately bracketed.)

Named Colors

Some colors have names, including `red`, `orange`, `yellow`, `green`, `blue`, and `violet`. Naturally `black` and `white` are also named “colors”.

Documentation: `http://ccl.northwestern.edu/netlogo/docs/programming.html#colors`

Patch Labels

Patches can be labeled; they have two mutable label attributes.

`plabel` the patch label

`plabel-color` the color of the patch label

Example:

```
ask patch 0 0 [
  set plabel "origin"
  set plabel-color yellow
]
```

User-Defined Attributes

We can add patch attributes with NetLogo's `patches-own` keyword.
http://ccl.northwestern.edu/netlogo/docs/dictionary.html#patches-own
Example:

```
patches-own [income wealth]
```

The `patches-own` keyword cannot be used at the command line. Enter it in the `Code` tab, above any procedure definitions.
Note: in spatial models, we may use patch attributes to represent the state of the "environment", including resources and environmental obstacles (e.g., food supply, rivers).

Patch-Related Commands

Documentation: `https://ccl.northwestern.edu/netlogo/docs/dictionary.html#patchgroup`

Goals

After reading this section you will be able to

- explain what turtles are and describe some of their built-in attributes.
- explain what turtles coordinates are and describe how they relate to patch coordinates
- explain how to get and set attributes for any turtle
- explain how to add user-defined attributes to turtles

Turtles

Turtles do not exist by default!
We can create turtles with the `create-turtles` command. The following will create 10 turtles:

```
create-turtles 10
```

We can also ask existing turtles to `hatch` new turtles, or patches to `sprout` turtles.
We can ask newly created turtles to immediately execute a code block. The following creates 10 turtles and immediately moves each to a random location.

```
create-turtles 10 [setxy random-xcor random-ycor]
```

Turtles are mobile agents: they can change location.

Creating Turtles

- use `create-turtles <number>` to ask the observer to create turtles
- use `sprout <number>` to ask a patch to create turtles
- use `hatch <number>` to ask a turtle to create turtles

```
crt 10   ;; `crt` is short for `create-turtles`
ask patch 0 0 [sprout 1]
ask turtle 0 [hatch 1]
```

Note: when displayed graphically, turtles are painted in the order created, so when turtles overlap the one created last will appear on top.

Turtle Attributes

Turtles have a number of pre-defined attributes. These are described in the documentation:

Documentation: `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#builtinvariables`

Turtle Monitor

Turtle attributes can be inspected by opening a turtle monitor. E.g., to see all the bulitin attributes:

- go to the NetLogo `Interface` tab
- create at least one turtle
- at the command line, enter `inspect turtle 0`

The `inspect` command opens a graphical “turtle monitor”.

The monitor displays current attribute values. (These are dynamically updated if they change.)

To close the turtle monitor enter `stop-inspecting turtle 0` at the command line. (Or click the monitor’s `close` button.)

In the `Interface` tab, you can also open a turtle monitor with a mouse click. (Windows: right-click. Mac: cmd+click.)

The who Number

Each turtle has a single immutable attribute: its `who` number

- `who` is a unique, immutable integer id
- `who` numbers start at 0 and are incremented for each turtle created
- we can use `who` to determine which turtles receive a command.

```
ask turtle 0 [set color red]  ;; set turtle 0's color
```

Comment: use of `who` is seldom needed in NetLogo programming.

Turtle Coordinates Are Mutable

Each turtle has unique real-valued coordinates, which designate its location in the Graphics Window.

`xcor`	the x-coordinate of the turtle
`ycor`	the y-coordinate of the turtle

Since turtles are mobile agents, `xcor` and `ycor` are mutable attributes.

Turtle Coordinates

- turtles are located anywhere in the world, not just at a patch center
- more than one turtle may be at the same location, so a turtle is not uniquely identified by its coordinates (`xcor`, `ycor`);
- the turtle coordinates are floating-point values, where:

```
min-pxcor - 0.5 <= xcor <= max-pxcor + 0.5
min-pycor - 0.5 <= ycor <= max-pycor + 0.5
```

Turtles Are Mobile

Turtles can change location (move across patches).

```
ask turtle 0 [jump 10]
ask turtle 0 [forward 10] ;; in unit increments
ask turtle 0 [jump -10]
ask turtle 0 [back 10] ;; in unit increments
ask turtle 0 [move-to patch 0 0]
ask turtle 1 [move-to turtle 0]
```

Note: code blocks (i.e., collections of commands) are bracket delimited.

Some Other Mutable Attributes

`color` the turtle color

`size` the visual-display size of a turtle relative to a patch

`label` the turtle label

`label-color` the color of the turtle label

More Mutable Attributes

- heading (in degrees, 0 = up, 90 = right, defaults to random)
- shape, label, label-color
- breed (used like a sub-type)
- hidden? (boolean)
- pen-size, pen-mode ("up", "down", or "erase")

Attribute Getting

Any agent can directly access its own attributes.

```
ask turtle 0 [show xcor]
```

We can identify a turtle by its `who` number:

```
ask turtle 0 [show who]
```

As with patches, we can also do attribute access with the keyword `of`:

```
show [xcor] of turtle 0
show [who] of turtle 0
```

Attribute Setting with ask

Example:

```
ask turtle 0 [set color red]
```

We can add new turtle attributes (in `Code` tab) with `turtles-own`
`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#turtles-own`

Attribute Access: A Turtle's Patch

For the most part, only the agent that "owns" an attribute can find (or change) its value.
An exception is that a turtle can directly access the attributes of its patch. For example,

```
create-turtles 10 [setxy random-xcor random-ycor]
ask turtle 0 [show pxcor]
```

We might expect to ask a turtle to show the color of its patch as follows:

```
ask turtle 0 [show [pcolor] of patch-here]
```

This works, but recall that turtles have direct access to the attributes of their patches. It is therefore simpler to write:

```
ask turtle 0 [show pcolor]
```

A Turtle Knows Its Patch

We can ask a turtle to set its patch attributes.
Example: although `pcolor` is a patch attribute, not a turtle attribute, we can use it to directly ask a turtle to change the color of the patch it is on:

```
ask turtle 0 [set pcolor green]  ;; sets *patch* color
```

Turtles in the Command Center ...

```
clear-all               ;; restore defaults (e.g., no turtles)
create-turtles 24       ;; create 24 turtles
ask turtles             ;; ask each turtle ...
  [forward 10]          ;;  to move fd 10 units
ask turtle 0            ;; ask the first turtle ...
  [set color red        ;;  to turn red and ...
   set pcolor green]    ;;  turn its patch green
```

Turtles Context

In the turtles context, our commands are given to each turtle:

```
fd 5              ;; move forward 5 units
rt 180            ;; turn right 180 degrees
pd fd 5 pu        ;; pen-down, move fd 5, pen-up
set hidden? true  ;; turn invisible
set pcolor white  ;; change patch color to white
```

Try this in turtles context: what happens?

```
ask turtle 0 [fd 1]
```

User-Defined Attributes

We can add turtle attributes with NetLogo's `turtles-own` keyword.
http://ccl.northwestern.edu/netlogo/docs/dictionary.html#turtles-own
Example:

```
turtles-own [income wealth]
```

The `turtles-own` keyword cannot be used at the command line. Enter it in the `Code` tab, above any procedure definitions.

Introduction to Agentsets

We have already met these two agentsets: `patches` and `turtles`.

- an unordered collection of agents
- traverse (in random order) with `ask`
- order is randomized on each use!

Examples:

```
ask patches [set pcolor black]
```

Producing Patch Sets

`patches`	all patches
`no-patches`	empty agentset
`n-of <num> patches`	a random subset of `<num>` of the patches
`patches with [<criterion>]`	all patches satisfying a criterion

The patches Agentset

Use `patches` to report an “agentset” that contains all patches.
Use `ask` on an agentset to have each agent execute commands. For example, in the *observer* command context, try the following:

```
ask patches [ set pcolor random-float 140 ]
```

Question: what would happen if you executed the same command in the *patches* command context.
Note: the patches are asked in random order to execute the commands; one finishes before the next starts.
Note: use `clear-patches` (or `cp`) to restore patch attributes to their defaults.

Random Subsets via n-of

Use `n-of` to create a random subset of an agentset.

```
ask n-of 5 patches [set pcolor white]
```

Here `n-of 5 patches` is an agentset that is a random subset (of size 5) of all patches.

Selection is without replacement. An error will be raised if we try to create a subset bigger than the original set.

Criterion-Based Subsets via with

Use `with` to filter any agentset on some boolean criterion. The result is a subset of the original agentset, which contains only the agents satisfying the criterion. E.g.,

```
ask patches with [ pxcor < -5 ] [ set pcolor white ]
```

Here `patches with [ pxcor <= -5 ]` is an agentset that holds all the patches at or to the left of `-5` in our world.

```
ask turtles with [ xcor <= -5 ] [ set color orange ]
```

Here `turtles with [ xcor < -5 ]` is an agentset that holds all the turtles at or to the left of `-5` in our world.
Exercise: Suppose we start with all patches black and all turtles blue, and then run the two commands above. Explain why there may still be some blue turtles on white patches. How could you reuse `pxcor` to avoid this?

Producing Patch Sets Related to a Turtle

A turtle knows its patch, so it can produce any patch set that its patch can produce. E.g.,

```
ask turtle 0 [show neighbors4]
```

Producing Patch Sets Related to a Patch or Turtle

`other patches` the agentset of all other patches

`neighbors` or `neighbors4` nearest 8 (or 4) neighboring patches

`patches in-radius <number>` patches that are near enough

`patches at-points [<points>]` patches at the relative locations given by `<points>`

Neighborhoods

Since patches are stationary, each has a fixed set of adjacent patches.

`neighbors` the agentset of 8 surrounding patches

`neighbors4` the agentset of 4 abutting patches

Patch neighbors

We can access neighborhoods like a patch attribute, using the `of` keyword:

```
clear-patches
ask [neighbors] of patch 0 0 [set pcolor white]
ask [neighbors4] of patch 0 0 [set pcolor red]
```

Patches in-radius

Produce agentsets of nearby patches with `in-radius`. Omit the asker from this agentset with `other`.

```
clear-patches
ask patch 0 0
  [ask patches in-radius 1.5
    [set pcolor blue]]
ask patch 0 0
  [ask other patches in-radius 1.5
    [set pcolor red]]
```

Patches at-points

Produce agentsets of patches a given relative positions with `at-points`.

```
clear-patches
ask patch 5 5
  [ask patches at-points [[-1 -1] [1 1]]
    [set pcolor blue]]
```

Note: we specified a list of *offset* pairs, relative to patch 5 5.
http://ccl.northwestern.edu/netlogo/docs/dictionary.html#at-points

Producing Turtle Sets

`turtles` all turtles

`no-turtles` empty agentset

`n-of <num> turtles` a random subset of `<num>` of the turtles

`turtles with [<criterion>]` all turtles satisfying a criterion

`other turtles` the agentset of all other turtles

`turtles in-radius <number>` turtles that are near enough

`other-turtles-here` other turtles on the same patch

`neighbors` or `neighbors4` nearest 8 (or 4) neighboring patches

Commands for Manipulating Agentsets

`any? <agentset>` report `true` if agentset is not empty `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#any`

`max-one-of <agentset> [<reporter>]` report an agent that the reporter assigns a maximum value `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#max-one-of`

`min-one-of <agentset> [<reporter>]` report an agent that the reporter assigns a minimum value `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#min-one-of`

`one-of <agentset>` report a random agent (or report `nobody` if the agenset is empty) `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#one-of`

Example:

```
ask one-of patches [set pcolor yellow]
```

self

An agent can refer to itself with the `self` reporter. However, it is usually redundant to do so.
For example, `[pcolor] of self` can be written as `pcolor`.
However, we can use `self` to create a list from an agentset. For example,

```
[self] of patches
```

reports a list of all the patches, in random order. This is occasionally useful.

myself

Recall that `self` names current askee (e.g., an agent that is “ask”-ed).
In contrast, `myself` names current asker (e.g., if an agent asks another).

```
ask patch 0 0 [
  ask neighbors [set pcolor [pcolor] of myself]
]
```

Exercise: what is the result of the following command? Explain in detail.

```
ask patches [set pcolor black]
ask patches [
  ask neighbors4 with [pcolor = [pcolor] of myself] [
    set pcolor red
  ]
]
```

Goals

- learn how to declare global variables
- learn about dynamic typing
- use `set` to change the value of global variables
- disinguish between declared globals and interface globals

Declaring Global Variables

Global variables cannot be created in the Command Center. Instead, you can declare your global variables at the top of the `Code` tab, using the `globals` keyword. For example:

```
globals [x y z]
```

Currently (NetLogo 5.3), declared global variables are automatically given an initial value of 0.
Global variables are available everywhere in your program. Any agent can access the value of any global variable.

Changing the Value of a Global Variable

Any agent can run the set command to change the value of any global variable. For example, at Command Center, the observer can set the value of any global variable.

- start NetLogo
- In the Code window enter globals [x]
- switch to the Interface tab, and enter the following at the command line

```
set x (x + 1)
print x
```

Because x is global, we do not need to enter print x at the same time as our set command. The value of a global variable is always available to any agent.

Dynamic Typing

Unlike some other languages, we do not have to specify what kind of values a variable will have. That is determined at runtime (i.e., while your program runs). We say that NetLogo variables are “dynamically typed”.

- start NetLogo
- In the `Code` window enter `globals [x]`
- in the `Interface` tab, enter the following in the Command Center

```
print x set x true print x set x "true" print x
```

In the Command Center’s history window, you should see:

```
0
true
true
```

Note that the second of these represents the boolean value `true`, while the first of these represents the string `"true"`. These are different values, but `print` gives them the same representation.

Randomly Setting the Value of a Global Variable

The command `random-float 1` generates a random number between `0` and `1`. Here we randomly set the value of a global variable.

- start NetLogo
- In the `Code` window enter `globals [x]`
- switch to the `Interface` tab, and enter the following in the Command Center

```
set x (random-float 1)
print x
```

Remember: we use `set` to change the value of a variable.

Documentation: `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#random-float`

Repeatedly Setting the Value of a Global Variable

The `repeat` command repeatedly executes a command block. Here we increment the value of a global variable 50 times.

- start NetLogo
- In the `Code` window enter `globals [x]`
- switch to the `Interface` tab, and enter the following in the Command Center

```
repeat 50 [set x (x + 1)]
print x
```

Remember: we use `set` to change the value of a variable.

Documentation: `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#repeat`

Interface Globals

There is an alternative way to introduce global variables in a NetLogo model. In the `Interface` tab, we can add sliders, switches, choosers, or input boxes. We usually use interface globals for model parameters that we wish to experiment with.

slider	sets numeric global to value in a range
switch	sets boolean global
chooser	set global to value in a fixed list
input box	set global to value input by user

Goals

- understand the basic structure of a reporter procedure
- understand the defintion of a pure function
- understand the basic structure of a command procedure
- understand the similarities and differences between commands and reporters
- understand the basic structure of a NetLogo program

Procedures

We have learned how to enter commands in the Command Center. Now we learn how to bundle commands together as a *procedure*, which is a collection of commands.

We use procedures to make our code more readable and more maintainable. We build a NetLogo program from a collection of procedures. Many of our procedures will bundle code we want to reuse in multiple settings.

Procedures must be stored under the `Code` tab. They cannot be created in the Command Center. However, once created, they can be used in the Command Center.

Procedures can be of two types: a reporter procedure returns a value, and a command procedure does not.

Reporter Procedures: First Steps

Reporter procedures collect together a sequence of commands that we can then execute using the procedure name, and they return a value for subsequent use. The structure of a reporter is:

```
to-report <reporter-name>
  <reporter-body ...>
  report <value>
end
```

That is, use the NetLogo keywords `to-report` and `end`, introduce a name for your reporter, and place your code in the reporter body. To specify the value to return, use the NetLogo keyword `report`. Example:

```
to-report fairCoinFlip
  report one-of [0 1]
end
```

Reporters and Functions

NetLogo uses the term ‘reporter’ for any procedure that returns a value.

In other languages, the equivalent of a NetLogo reporter procedure is often called a *function*, and the reported value is often called a *return value*.

In mathematics, a function has an input (from its domain) and produces an output (in its codomain). The output is called the image of the input under the function.

Functions: First Steps

A univariate function accepts one input and returns a determinate output. NetLogo allows us to name an input argument for a reporter in brackets after the reporter name. Here is the syntax for a univariate function.

```
to-report <function-name> [<input-name>]
  <reporter-body ...>
  report <output-value>
end
```

In sum, create a reporter with one input argument, and `report` an output value. Example:

```
to-report xsq [#x]
  report (#x * #x)
end
```

Pure Functions

A function is called “pure” if any given input always produces the same output (and there are no side effects). These are the kinds of functions you know from your math classes.

The `xsq` example above is a pure function.

If a procedure changes the value of a global variable, that is a side effect. The procedure is not a pure function.

Randomness vs Purity

When possible, your procedures should be pure functions.
However, impurity can be very convenient. Consider our very simple coin flipping procedure above. The input never changes. (There is no input argument.) But the output can be different each time the procedure is called.

Note

When `fairCoinFlip` is called, the output depends on the state of the random number generator. This state is a hidden input to the procedure, but the inputs to a pure function must all be explicit.
Calling the function changes the state of the random number generator. This invisible change is a side-effect of calling the procedure.

Command Procedures: First Steps

A command procedure does not return a value. The whole purpose of a command procedure is to have side effects. The structure of a command procedure is:

```
to <procedure-name>
  <procedure-body ...>
end
```

That is, use the NetLogo keywords `to` and `end`, introduce a name for your command procedure, and place your code in the procedure body.
In other languages, the equivalent of a NetLogo command procedure is often simply called a procedure. Because the purpose of a command procedure is to change things rather than to compute a value, we usually name a command procedure with a verb.

Command Procedures: Example 1

Enter the following in the Code area:

```
to randomize-pcolor  ;;patch procedure
  set pcolor one-of [red white blue]
end
```

At the command line enter (at the observer prompt):

```
ask patches [randomize-pcolor]
```

You should see your patches change colors.

Note

Only patches have a pcolor attribute. For this reason, this procedure should be run in a patch context. It is a convention to note this in a comment, as we did above.

Command Procedures: Example 2

Procedures can access any global variable. Enter the following in the `Code` area:

```
globals [x]

to increment-x
  set x (x + 1)
end
```

Return to the Command Center and execute:

```
print x
increment-x
print x
```

The `increment-x` procedure has changed the value of the global variable `x`.

Reporter Procedures and Global Variables

Enter the following under the `Code` tab.

```
globals [x]

to-report x-incremented
  report x + 1
end
```

At the command line enter:

```
print x
print x-incremented
print x
```

Note that this reporter does not change the value of the global variable. Nevertheless it is not a pure function, because its output depends on a global variable (rather than just on its inputs).

A First Program

```
globals [x]

to setup
  clear-all
end

to go
  set x (incremented x)
end

to-report incremented [#x]
  report (#x + 1)
end
```

At the command line enter:

```
setup
go
print x
```

Plotting

We add plots to our model using NetLogo's `Interface` tab.

Add a plot in the `Interface` window by right-clicking where you want your plot located, and fill in the resulting dialogue.

`http://ccl.northwestern.edu/netlogo/docs/programming.html#plotting`

Note

In the Code tab, you can manipulate plots using the name you give them at creation.

Exercise: Basic plotxy

In the `Interface` tab, add a plot named `plot01`. Remove the default pen-update commands.
At the command line, enter the following:

```
plotxy 0 0
plotxy 1 1
plot-pen-up
plotxy 2 2
plot-pen-down
plotxy 3 3
```

Note that we only draw lines between the points when the plot pen is down. (We only use the default plot pen here; you may use multiple pens in each plot.)

Exercise: Basic plot

Start a new instance of NetLogo.
In the `Interface` tab, add a plot named `plot01`. (Remove the default pen-update commands; we will come back to these later.)
At the command line, enter the following:

```
plot 0 plot 1
plot-pen-up plot 2
plot-pen-down plot 3
```

By defaut, the plot pen is down. Calling `plot` moves the plot pen to the next point, but we only draw lines between the points when the plot pen is down.

Note

We change the x-axis value by `1` each time we call `plot`. (You can control this with `set-plot-pen-interval`.)
We only use the default plot pen here; it is possible to use multiple pens in each plot.

Exercise: Basic Plot of a Global Variable

Start a new instance of NetLogo.
In the `Interface` tab, add a plot. (Remove the pen update command.)
In the `Code` tab, do the following:

- copy our first program (above)
- at the end of the `setup` procedure, add the command `plot x` (to plot the initial point).
- at the end of the `go` procedure, add the command `plot x` (to plot a new point).

In the Command Center, run your `setup` procedure, and then run your `go` procedure 100 times.

Exercise: Basic Plot of a Patch Variable

Start a new instance of NetLogo.

Repeat the previous exercise, with the following changes.

Do not declare any global variables. Instead, declare a patch variable named x.

Change the plot in setup and go to the following line:

```
plot [x] of patch 0 0
```

Your go procedure should change slightly, since x is now a patch variable. (Remember to use ask.)

Once again go to the Command Center, run your setup procedure, and then run your go procedure 100 times.

Exercise: Basic Plot of a Turtle Variable

Start a new instance of NetLogo.
Repeat the previous exercise, with the following changes.
Do not declare any global or patch variables. Instead, declare a turtle variable named `x`.
To your `setup` procedure you now need to add the creation of a turtle.
Change the plot commands in `setup` and `go` to:

```
plot [x] of turtle 0
```

Your `go` procedure should change slightly, since `x` is now a turtle variable. (Remember to use `ask`.)
Once again go to the Command Center, run your `setup` procedure, and then run your `go` procedure 100 times.

Code in Plots

NetLogo allows you to place your plotting code within the plot itself, instead of in the `Code` tab.

When you create a new plot in the `Interface` tab, you can open a text box for entering `Pen setup commands`. This code is run each time `setup-plots` is called. (The `reset-ticks` command calls `setup-plots`.)

You will also see a text box for entering `Pen update commands`. This code is run each time `update-plots` is called. (The `tick` command calls `update-plots`.)

Note

Starting with NetLogo version 5, you must call `reset-ticks` before you can call `ticks`.

Revision: Basic Plot of a Global Variable

Start a new instance of NetLogo. In the `Interface` tab, add a plot.
Click on the edit icon for the default pen. Change the pen update command to `plot x`.
In the `Code` tab, do the following:

- copy our first program (above)
- at the end of the `setup` procedure, add the command `reset-ticks` (which will call `setup-plots` **and** `update-plots`)
- at the end of the `go` procedure, add the command `tick` (which will call `update-plots` to plot a new point).

In the Command Center, run your `setup` procedure, and then run your `go` procedure 100 times.

Note

Because `reset-ticks` calls `update-plots`, we do *not* need to add a pen setup command of `plot x`.

Introduction to Lists

A list is an ordered collection of objects.

NetLogo lists are immutable. You cannot change an existing list, but you can use a list as the basis of a new list.

List Creation

If we want to make a list using any variable names, we must use the `list` primitive:

```
let a 0 print (list a 1)
```

If our list will only contain constants (or nothing at all), we can alternatively use brackets around our space-separated constants.

```
[0 1 2 3] ;a list of constants
[]        ;an empty list
```

While not required, it is conventional in NetLogo to use the short-cut bracket notation for list creation, when possible.

Accessing List Items

report first item in list:

```
first <list>
```

report last item in list:

```
last <list>
```

report the *n*-th item of list:

```
item n <list>
```

zero-based indexing:

```
first <list> = item 0 <list>
```

New Lists from Old: Shorter Lists

report all but first item, as a new list:

```
butfirst <list>
```

report all but last item, as a new list:

```
butlast <list>
```

report all but *n*-th item, as a new list:

```
remove-item n <list>
```

zero-based indexing:

```
remove-item 0 <list> = butfirst <list>
```

Since list are immutable, the results of these operations are *new* lists.

New Lists from Old: Longer Lists

prepend value to list, producing a new list:

```
fput <value> <list>
```

append value to list, producing a new list:

```
lput <value> <list>
```

concatenate lists, producing a new list:

```
(sentence [0 1] [2 3])
```

Since list are immutable, the results of these operations are *new* lists.

Breeds

- new breeds can be declared in the declarations section of a script:

  ```
  breed [ thieves thief ]
  ```

- a breed is a bit like a “subtype” of turtle: it has all the attributes of turtle, plus any new attributes declared for the breed. E.g.,

  ```
  thieves-own [ skill known? ]
  ```

- turtles and links come with a `breed` attribute, which can be accessed or set.

  ```
  show [ breed ] of turtle 0
  ask turtle 0 [ set breed thieves ]
  ```

- when turtles are displayed in the NetLogo graphics window, breeds are painted in the order declared, so when turtles of different breeds overlap, the breed declared last will appear on top
  http://ccl.northwestern.edu/netlogo/docs/programming.html#breeds

Links

A link establishes a relationship between two turtles. This relationship may be “two-way” (undirected link) or “one-way” (directed link).

Links have attributes. (Inspect a link to see these.)

It is possible to declare link breeds.

Basic Link Creation

create an undirected link between *t1* and *t2*:

```
ask t1 [create-link-with t2]
```

create a directed link from *t1* to *t2*:

```
ask t1 [create-link-to t2]
```

create a directed link from *t2* to *t1*:

```
ask t1 [create-link-from t2]
```

Link Breeds

Link breeds must be declared as either directed or undirected. These breeds may own variables. (See the Link Breeds Example in the Models Library.)

```
directed-link-breed [unis uni]
undirected-link-breed [bis bi]
bis-own [weight]
...
ask turtle 0 [create-uni-to turtle 1]
ask turtle 0 [create-bis-with other turtles]
```

Goals

After mastering this section, you will be able to:

- find and make use of the Models Library
- explain the intent and usefulness of the following models in the Models Library: Traffic (Basic), Party, Segregations

First Look: Models Library

File menu items (top left)

- `File > Models Library`
- models have been vetted, except for those marked “unverified”
- double-click model name or icon to open
- start with `Info` tab
    - says what the model does and how to experiment with it
- Experiment by interacting with GUI, which typically includes
    - sliders (manipulate to set model parameters)
    - buttons
        - `Setup` (press to set up the simulation)
        - `Go` (press to run the simulation; press again to stop it)

Basic Models

Experiment with some basic models from the Models Library:

- Traffic Basic
- Party
- Segregation

Models Library Example: Traffic Basic

- File > Models Library > Social Science > Traffic Basic
- basic illustration of emergence:
  - cars move forwards
  - emergent traffic jams move backwards

Models Library Example: Party

- File > Models Library > Social Science > Party
- two “types” of people (e.g., men and women)
    - otherwise indentical (common attributes)
    - crucial parameter: `tolerance`
- *question:* how does tolerance affect group formation?
- closely related to the Schelling segregation model

Models Library Example: Segregation

- File > Models Library > Social Science > Segregation
- Wilenski's implementation of the Schelling segregation model
    - two types of people (e.g., geeks and jocks)
        - otherwise indentical (common attributes)
        - crucial parameter: percent-similar-wanted

Quick Load

Hint: you can use the command line to load a model from the Models Library by entering three underscores and (part of) its name. For example, entering `___party` will load the party model.

Reload

Enter `__reload` at the command center to reload a model from disk.

This is useful for resetting sliders and other widgets to their original values.

https://github.com/NetLogo/NetLogo/wiki/Unofficial-features

Models Library: Life

Models Library

- Computer Science > Cellular Automata > Life

use of patches to implement cellular automaton

- just patches (cells); no turtles
- cells have two states: alive and dead
  - Dark = cell alive, background color = dead

Transition rule

- dead cell with 3 or more live neighbors comes to life
- live cell with 2 live neighbors continues to live
- live cell with less than 2 live neighbors dies

Models Library: Heat Bugs

- `File > Models Library > Biology > Heatbugs`
- famous biological model
- each bug radiates a little heat
- bugs move if they are too hot or cold

Compare to the ’Collectivities’ model.

Models Library: Party

Models Library

- Social Science > Party (famous in another form)

implements the Schelling segregation model

- two types of people (e.g., men and women)
  - otherwise the same attributes

Key parameter:

- how “comfortable” are these agents being in a local minoriy
- adjustable parameter (slider)

Discovery

- Even a slight preference can lead to complete segregation

Models Library: Small World

- Models Library > Networks > Small Worlds
- famous: six degrees of separation
- new tools
    - uses links
    - path along links from one agent to another

Setup

- initalize links in a ring (each agent links to an agent each side)

key outcome concerns the average path length for all pairs of agents

- even a few random new links substantially reduces the average path length
    - potentially significant for organizational design
    - potentially significant for disease transmission

Models Library: El Farol

- Models Library > Social Science > ElFarol
- another version: `http://www.markgarofalo.com/ABS/ElFarol/ElFarolBarProblem.html`

Readings: [garofalo-2006-wp], [wilensky.rand-2007-jasss]

Models Library: Predation

File > Models Library > Biology > Wolf Sheep Predation

- use of breeds (wolves, sheep)
- more interesting variant: patches grow grass that sheep eat (click grass? Switch to On)
- boom-and-bust cycles common (with sheep and wolves negatively correlated)

Models on NetLogo site

- URL: ccl.northwestern.edu/netlogo/ > Community
- possible problems if models are written for earlier versions of NetLogo

Models on NetLogo site

Miller & Page Applause Model `http://luis.izqui.org/models/standingovation/`

Readings: [miller.page-2004-complexity]

References

[garofalo-2006-wp] Garofalo, Mark. (2006) "Modeling the 'El Farol Bar Problem' in NetLogo". Dexia Bank Belgium . http://ccl.northwestern.edu/papers/netlogo-outside.html

[harvey-1993-web] Harvey, Brian. 1993. Symbolic Programming vs. Software Engineering -- Fun vs. Professionalism -- Are These the Same Question?.

[miller.page-2004-complexity] Miller, John H, and Scott E Page. 2004. The Standing Ovation Problem. *Complexity* 9, 8--16.

[tisue.wilensky-2004-wp] Tisue, Seth, and Uri Wilensky. (2004) "NetLogo: A Simple Environment for Modeling Complexity". Center for Connected Learning and Computer-Based Modeling . http://ccl.northwestern.edu/papers/netlogo-iccs2004.pdf

[wilensky.rand-2007-jasss] Wilensky, Uri, and William Rand. 2007. Making Models Match: Replicating an Agent-Based Model. *Journal of Artificial*

Legalities