NetLogo Tutorial Series: Core Concepts

Nicholas Bennett
Grass Roots Consulting
nickbenn@g-r-c.com

July 2010

Copyright

Copyright © 2010, Nicholas Bennett. “NetLogo Tutorial Series: Core Concepts” by Nicholas Bennett is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. Permissions beyond the scope of this license may be available; for more information, contact nickbenn@g-r-c.com.

Acknowledgments

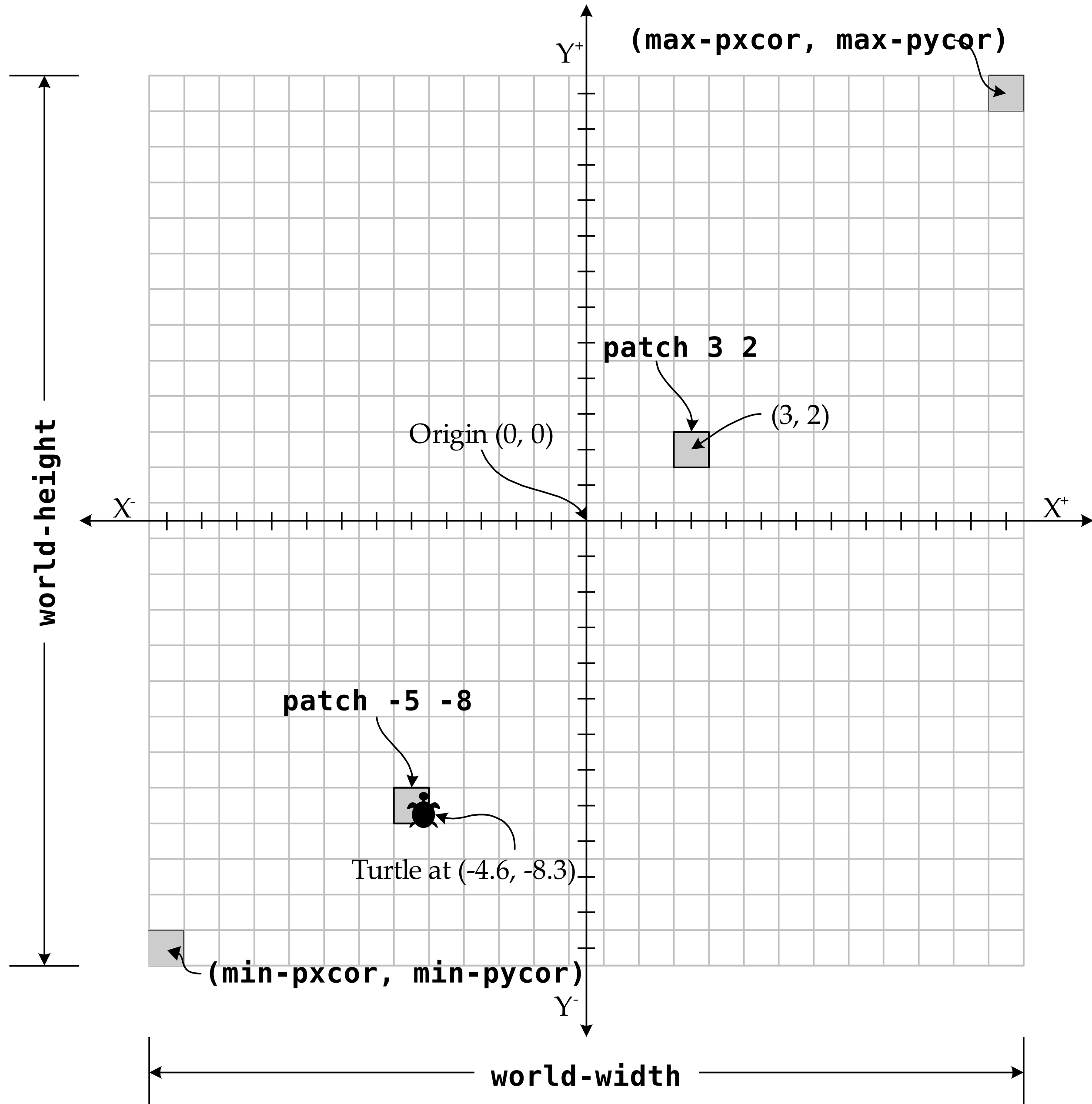
Development of this curricular material was funded in part by:

- Santa Fe Institute Summer Internship/Mentorship (SIM) program for high school students;
- New Mexico Supercomputing Challenge;
- Project GUTS.

Participants in the above programs have also provided invaluable feedback on earlier versions of this material.

The NetLogo Coordinate System

In building NetLogo models it's important to understand the coordinate system used by NetLogo. This diagram, and the explanations that follow, illustrate some important points to remember:



1. Like the Cartesian coordinate system traditionally used in algebra, analytic geometry, and calculus, the NetLogo world has X and Y axes. The center of the coordinate system is the origin (which is usually – but not always – located in the physical center of the NetLogo world, as well), where X and Y have values of zero (0).

2. Overlaid on the coordinate system is a grid of *patches* (1 X 1 squares). Each patch has a color, and an optional label; a NetLogo program can also define additional variables for a patch.

3. The center of a patch is a point in the coordinate system with integral X and Y values; these coordinates are used to refer to the patch. For example, **`patch 3 2`** in the diagram is a square with its center at (3, 2); this square is the region where $2.5 \leq X < 3.5$ and $1.5 \leq Y < 2.5$. (We can also refer to patches with floating point coordinates; they'll be rounded to integers as necessary.)

4. A patch's coordinates are always integer values, but that's not necessarily the case for a turtle. In the diagram, there's a turtle located at (-4.6, -8.3), which is on the patch centered at (-5, -8). Though a turtle may be drawn so that it looks like it is on two or more patches at once, the turtle's center point is what matters: this center point is treated as the actual location of the turtle, and the patch containing that center point is considered to be the patch on which the turtle is standing.

5. The user can change the width or height of the NetLogo world at any time; because of this, NetLogo programs should generally not assume fixed world dimensions, unless absolutely necessary. Fortunately, NetLogo programs can always use **`world-width`** and **`world-height`** to get the current dimensions of the world.

6. The patches on the extreme right-hand side of the NetLogo world have an X value of **`max-pxcor`**; those on the top of the world have a Y value of **`max-pycor`**. Similarly, **`min-pxcor`** and **`min-pycor`** are the X and Y coordinates (respectively) of the patches on the extreme left-hand side and bottom (respectively) of the NetLogo world. These variables are related to the overall size of the world, as follows:

   `world-width = (max-pxcor - min-pxcor) + 1`

   `world-height = (max-pycor - min-pycor) + 1`.

NetLogo Angles and Directions

All angles in NetLogo are specified in degrees, and directions are based on compass headings, with 0° being "up" (i.e. north), 90° being towards the right (i.e. east), etc.

When instructing a turtle to face a particular direction, we can do so by setting the heading of the turtle to the desired compass heading, or by telling the turtle to turn right or left by the number of degrees required to orient the turtle as desired. We can also instruct a turtle to face another agent by specifying the second agent in a **`face`** command, rather than computing the compass direction or turn angle required.

NetLogo Topology

Notice that we can specify that the NetLogo world should wrap horizontally, vertically, both horizontally and vertically, or not at all. When wrapping is turned on horizontally (for example), a turtle moving off the right edge of the world will reappear on the left edge, and vice versa. If horizontal wrapping is not enabled, a turtle will be unable to move off the right or left edge.

1. What is the logical "shape" of the NetLogo world, if wrapping is turned on horizontally, but not vertically?
2. What is the shape of the NetLogo world, if wrapping is turned on vertically, but not horizontally?
3. What is the shape of the NetLogo world, if wrapping is turned on both vertically and horizontally?

Programming in General: Teaching the Computer

Although computers (more precisely, the processors inside computers) are capable of manipulating data very efficiently, and though modern processors include floating-point processing units that can perform impressive arithmetic, trigonometric, and logarithmic calculations, they're also simple-minded: they're generally incapable of performing most tasks the average user would consider meaningful – *until they're taught to do these meaningful tasks*. We teach computers to do this through programming: encoding an algorithm (a procedure for completing a task or solving a problem) into a form that the computer can understand, for which it will take specified inputs, and from which it can present a meaningful result as output.

Fortunately for us, virtually every modern, commercially-available computer comes with millions of lines of these algorithmic instructions already written, and preloaded on hard drives, programmable memory chips, etc. These instructions make up the operating system (which lets us create and manipulate files), drivers (which tell the computer how to connect to and make use of hardware devices – e.g. video display adapters, disk drives, printers, external memory devices), and applications (special files which can be executed on demand by the user, for more specific functionality). We can augment this further by installing or writing new programs for the computer to execute; when we do this, we're literally teaching the computer to perform new tasks.

Some computer programs are instruction translators: they allow programmers to write new programs, without them having to understand much of the internal workings of the computer; these translators then convert the instructions the programmers have written into a form that the computer can execute. NetLogo is one such translator: it allows us to write programs in a specialized language, which we use to describe the behaviors of agents; NetLogo then converts these programs (NetLogo models) into a form the computer can execute[1], without us having to know anything about how that conversion takes place. Nonetheless, we can still think of the NetLogo models we write as being sets of instructions that we teach to the computer; perhaps more usefully, we can think of our task, when building NetLogo models, as being that of teaching NetLogo itself.

---

1 This is actually a slight over-simplification. NetLogo is built on top of Java, so it converts models into instructions which the Java Virtual Machine (JVM) can understand. As the model runs, the JVM translates those into instructions the computer hardware can execute directly.

Programming in NetLogo

We give instructions to NetLogo in three main ways:

- We can type instructions in the **Command Center** (usually located at the bottom of the **Interface** window). These instructions are executed as soon as we press the *Enter* key – but they don't become part of what we're teaching NetLogo (i.e. the program[2] we're writing). In other words, we can use the **Command Center** to instruct NetLogo to peform actions it already knows how to do, but we can't use it to teach NetLogo new capabilities.
- Some instructions can be included in buttons (and to a lesser extent, monitors) in the user interfaces we create. This functionality is most often used to connect the buttons we create to the new capabilities that we've taught NetLogo in our program.
- Finally, and most importantly, when we write instructions in the **Procedures** window, we're creating a NetLogo program (which includes one or more *procedures*), and teaching NetLogo to do something new. What we write in the **Procedures** window isn't executed immediately, but becomes part of what NetLogo knows how to do (as long as the program is loaded). We can invoke this new functionality through buttons and monitors in the user interface, by typing commands in the **Command Center**, or by referring to one or more of the new procedures in other code we write in the **Procedures** window.

When you teach another person a procedure for completing some task, you might begin by saying: “To do X, first do A, then do B,” and so on. Teaching NetLogo to perform some task is very similar: we use the keyword **`to`**, followed by the name of the task, and then the set of instructions that make up the procedure; finally, we indicate that there are no more instructions for this task by ending the procedure with the keyword **`end`**.

---

2 “Model” and “program” are often used interchangeably when talking about NetLogo. Here, we'll distinguish between the two by using “program” to refer to the contents of the **Procedures** page, and “model” to refer to the combined contents of the **Interface**, **Information**, and **Procedures** page – i.e. the procedures, the user interface, and any embedded user information and documentation.

For example, the following procedure teaches NetLogo how to draw a square (more precisely, how a turtle draws a square; see “Different Types of Agents”, below):

```
to draw-circle
  pen-down
  repeat 4
  [
    forward 10
    right 90
  ]
  pen-up
end
```

Note that there are hyphenated words in this example, just as there are in the “The NetLogo Coordinate System“, above. Though this isn't allowed in most programming languages, it's valid and common in Logo dialects, and there are a number of built-in procedures (such as **`pen-down`** and **`pen-up`**) with hyphenated names. However, while procedure and variable names can include hyphens – as well as many other punctuation symbols – they can't include spaces.

Now that we've written the **`draw-square`** procedure, we can invoke it by name in the **Command Center**, in a button, or in another procedure.

Most programming languages support two fundamentally different kinds of procedures: those that modify the state of the system, but don't compute and return some information as a result; and those that compute and return a result (these may or may not also modify the state of the system). The procedure above is an example of the former: it modifies the heading and position of an agent, but doesn't compute and return a result. In NetLogo, we can also write *reporter procedures* (usually called simply *reporters*), which return results. For example, the following reporter computes and returns the square of a specified number:

```
to-report square [input-value]
  report (input-value * input-value)
end
```

The reporter syntax has two main differences from the syntax of a regular procedure:

1. The definition of a reporter begins with **`to-report`**, instead of **`to`**.
2. The command **`report`** is used to return a value.

Note that input parameters can be included in the definitions of procedures and reporters, using square brackets after the procedure or reporter name.

Different Types of Agents

There are four types of agents in NetLogo; each is capable of different kinds of following different kinds of instructions, and each serves a different purpose in a NetLogo model:

1. **Observer** – There's always exactly one of this kind of agent; we can think of this as being NetLogo itself. This agent is not displayed on the NetLogo world, but it is the only agent that can perform certain global operations in a model (e.g. **`clear-all`**).

2. **Patches** – These are stationary agents, and there's exactly one such agent per square in the grid of the NetLogo world. They cannot be displayed with different shapes, but the color of a patch can be modified.

3. **Turtles** – These are agents that can move about the NetLogo world independently of other agents, and can be displayed with different shapes and colors. Any code that instructs the agent to move can only be used with turtles.

4. **Links** – These are agents which connect one turtle to another. There are no instructions to move links directly; a link moves when one or both of the turtles at the endpoints move. (A link can also be configured as a *tie*, where motion of one endpoint turtle will force movement of the other endpoint turtle.) Links can be directed or undirected: with undirected links, we don't recognize the link as coming from one turtle to another, but simply that it is between the two; a directed link, on the other hand, is always *from* one turtle *to* another.

Links and turtles are the only agents that can be created or destroyed by the instructions contained within the model itself. Also, links and turtles are the only agents that can be organized into breeds.

Turtles can interact with other turtles by reading the attributes of those turtles, or by asking those turtles to execute instructions; they can also interact with patches in the same ways. Patches can interact with turtles, and with other patches. Links generally interact with their endpoint turtles, but they can also be made to interact with other links, turtles, and patches. The observer can ask turtles, patches, and links to perform specified operations. On the other hand, turtles, patches, and links cannot interact directly with the observer, in the sense that they can't ask the observer to perform any actions. However, models have global variables (some predefined by NetLogo, and others we can define in our sliders and program code); in general, patches, links, and turtles can modify the values of these variables – and the observer's actions may be affected by such changes.