GEOG 5160/6160 Lab 08

Simon Brewer

1/27/2018

## Lab 08: Netlogo and GIS

## Introduction

In this lab, we will look at the GIS extension for Netlogo using two examples, one with raster data and one with vector data. The examples given are not full agent-based models but designed to demonstrate how this GIS extension works.

### Objectives

* Understand how to incorporate extensions in Netlogo models
* Use the GIS extension to load raster and vector data
* Place and move agents along a network

### Netlogo extensions

Netlogo comes with a set of standard extensions, which provide extra functionality. The basic set includes extensions for sound, matrices/arrays, robotics, and GIS. A full list can be found on the Netlogo website, under the [Extensions](http://ccl.northwestern.edu/netlogo/docs/) link. Other extensions, written by the Netlogo community can be found [here](https://github.com/NetLogo/NetLogo/wiki/Extensions), and include network functions, cluster analysis and video [the network extension is now standard in version 5].

To use an extension in a model, add the extensions keyword at the beginning of your code, before declaring any breeds or variables. The name of the extension(s) that you wish to use should be included in brackets. For example, to load the GIS extension:

extensions [ gis ]

The extensions are expected to be in Netlogo’s root directory, but you can also add them to the same directory as the model code.

In a model, functions from extensions are specified by giving the extension name, followed by a colon, followed by the function name. For example, the function to load a GIS data set is load-dataset, and to use this in code you would add:

set elevation gis:load-dataset "dataset.asc"

As a reminder, the basic components of a Netlogo model are turtles, patches and globals. The *turtles* are Netlogo’s term for an agent, such as the butterflies; the *patches* are the landscape, such as elevation values; *globals* represent the whole system and are used for any external variables.

### GIS data in Netlogo

Using the GIS extension, Netlogo can load raster data from ESRI ASCII gridfiles, and vector data from ESRI shapefiles. It is important to note, however, that Netlogo still considers two different world definitions: the model and the GIS world. The extension acts to project the GIS data into the model world, and so allow comparison or interaction between geographical features and agents.

## Example 1: Raster data

### Loading raster data

In the first example, we’ll used the GIS extension to load a standard ESRI ASCII raster grid data. From the File menu in Netlogo, create a new Netlogo program, and use File/Save save this as *elevation.nlogo* in an appropriate folder. If you close Netlogo at any point, you can reopen this model, by finding the model file in this folder, and double-clicking on the icon.

Download the data (*Lab08.zip*) for the lab and put the folder named “data” in the same folder in which you’ve saved your model.

Now we start by entering the extensions and state (general) variables. Click on the “Code” tab (this is the “Properties” tab in older versions of Netlogo) and insert the following code:

extensions [ gis ]  
globals [ elevation ]

This will load the GIS extension and create a global variable called elevation, which will store our DEM data. Start by adding the following commands to:

* load in elevation data
* set the GIS world extents to match this file
* color the patches according to elevation height.

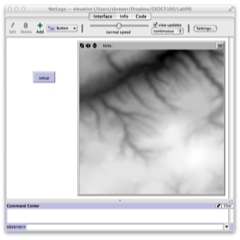
Note that all the GIS functions have the prefix gis::

to setup  
 clear-all  
   
 set elevation gis:load-dataset "data/local-elevation.asc"  
 gis:set-world-envelope gis:envelope-of elevation  
 gis:paint elevation 0  
   
end

This code starts by clearing all previously calculated variables (for re-running a model). The next line loads the data contained in the ArcInfo ascii grid file “local-elevation.asc”, into a variable called “elevation”.

The next line sets the GIS world to match the file extents. An important thing to note here is that this does not change the patch definition, but acts as a projection between the patches and the coordinates of the file. We will look at how to sample patches later, which should help clarify this. Finally, we “paint” the elevation data as gray scale, and with a transparency (alpha) of 0.

Now return to the Interface tab, and add a setup button from the [Button] drop down menu. Set the command for this button to ‘setup’ to link to our procedure. Save your model, then click the setup button. The following landscape should appear.



To check that the world has been setup correctly, click in the white box at the bottom of the Netlogo window (next to observer >), and enter the following code:

show gis:world-envelope

And it will print out the current extents according to the GIS projection layer. [The Command Center window is useful for checking the values of agents and patches during a model run.]

### Getting patch values from raster layers

Though we set the GIS world extent to our raster layer, the raster cells still do not correspond to the patches in the model. If you type the following at the Command Center, it should tell you that the model world is only 33 patches in either dimension

show world-width  
show world-height

We can use the raster layer to assign a value to our patches. This can be done using the GIS function gis:apply-raster. Add the following line to the setup procedure:

gis:apply-raster elevation pelev

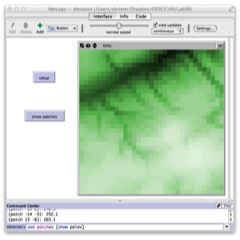
And add the patches-own statement to define ‘pelev’ as a patch variable, after the ‘globals’ statement at the top:

patches-own [ pelev ]

Now save the model and run. This will apply elevation values to each patch, but you won’t see any difference. To show the patch-based elevation, add the following procedure to your code (after the set-up procedure).

to show-patches  
 clear-drawing  
 ask patches [  
 set pcolor scale-color green pelev 210 270   
 ]  
end

Add a new button to the interface with the command ‘show-patches’ to link to this procedure. If you now click [setup] then [show-patches], this will display the patches with elevations derived from the raster layer.



To redisplay the original dem raster layer, add a new button called (‘show-elev’) which should be linked to the following procedure in your Netlogo code:

to show-elev  
 clear-drawing  
 gis:paint elevation 0  
end

The show-patches and show-elev buttons will allow you to toggle between the two layers.

You’ll notice that the default Netlogo world size results in a very coarse elevation layer, when looking at the patches. Try experimenting with changing the number of patches from the [Settings] button on the interface tab. Increase max-pxcor and max-pxcor to 64, and decrease the patch size to 3. Save the model, and re-run the setup, then click [show-patches]. You should see a much smoother representation of elevation in the model world.

### Using agents with raster layers

Agents in our model can interact with the raster layer in two ways. The first is by using the patches, and allowing the agents to ‘sense’ the elevation in a given patch, as we did with the butterfly model. The second method is to allow the agents to sense the raster layer directly, which will be demonstrated next.

We will add a single turtle (agent) and allow it to move in a straight line across the world. At each step, it will get the elevation from the raster layer, and we can track this to plot an elevation profile representing the turtle’s travel.

Add the following code in the setup procedure, after the gis:apply-raster statement:

crt 1  
[  
 set color red  
 set size 4  
 pen-down  
   
 set telev gis:raster-sample elevation self  
]

We have seen most of this code in previous models. This creates a single turtle that is red and of size 4. The pen-down records a trace of the turtles movement. The last line in the creation procedure sets a variable telev as the value of the raster layer elevation at the position of self, i.e. that turtle.

We need to define telev as a turtle variable, so add the following to the variable definitions at the top:

turtles-own [ telev ]

Save the model and check that the code looks ok. To check that this has worked, you can use the command center to get the telev value for our single turtle (turtle 0):

ask turtle 0 [ show telev ]

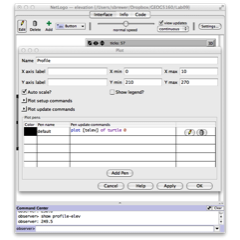
Now we need to add a go procedure to allow the turtle to move. Add the new ‘go’ procedure after your setup procedure:

to go  
 ask turtle 0 [  
 fd 1  
 set telev gis:raster-sample elevation self  
  
 ]  
 tick  
end

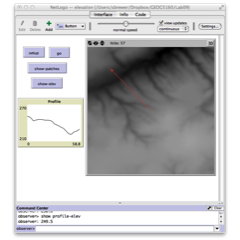
All this does is to move turtle 0 forward by one patch length at each time step, and set the turtle’s elevation variable (telev) to the elevation at the new location from the raster layer. We then increase the tick counter by one. Add the command reset-ticks at the end of your setup procedure to reinitialize the counter for each run.

Now go back to the interface tab, and add a [go] button to the interface. Finally add a plot to the interface from the [Button] drop-down menu, or by right-clicking on the interface window. In the plot dialog window change the y-limit of the plot to 210 (min) and 270 (max). In the pen update commands window add the following:

plot [telev] of turtle 0



Click [Ok] to save the plot. Check and save the model, then run the model for several steps. You should see something like the following:



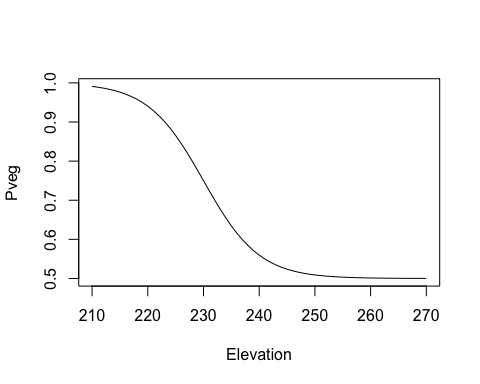
Remember that the turtle is initialized with a random heading, so you will see movement in a different direction.

### Using spatial information

As a last extension to this model, we will use the elevation information to create a map of vegetation, which could be used, for example, in a fire model. Previous iterations of this model have used a purely random distribution of vegetation, through which a fire spreads. Here, we will use the elevation to control the presence/absence of trees in each patch. We’ll assume that these trees occur preferentially at low elevations. To start, we need a function which relates elevation to probability of presence. For this we use the following model:

The parameters control the form of the function, here, we set these to

* (controls the low end probabilities)
* (the midpoint of the sigmoid curve)
* (the slope of the curve)



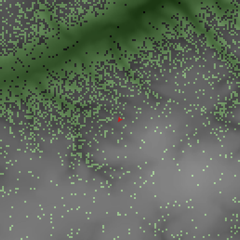
In the Netlogo model, add a new procedure to grow the trees:

to grow-trees  
 let a 0.99  
 let b 240  
 let c -5  
 ask patches [ set pcolor black ]  
 ask patches [  
 let pveg (1 - a) + (a / (1 + exp( (b - pelev ) / c) ) )  
 if random-float 1 < pveg [  
 set pcolor green  
 set tree? true  
 ]  
 ]  
end

The code does the following:

* Set the three parameters as local variables
* Reset all patch colors to black
* Loop through all patches and:
  + Estimate the probability of presence (pveg)
  + If a random number (0-1) is lower than this probability, assign trees and change color

Now add a button to the interface to call this procedure, save, setup and click this button, and you should get a landscape that is similar to the following:



## Example 2: vector data

For our second example, we will use a shapefile of roads, and use this to create a network over which agents move. The file *roadsclip.shp* is a small shapefile covering part of the Avenues.

Create a new model in Netlogo called ‘RoadModel.nlogo’. We will add the usual skeleton model to Netlogo, including the GIS extension, and a variable to hold the road layer:

extensions [ gis ]  
globals [ roads ]

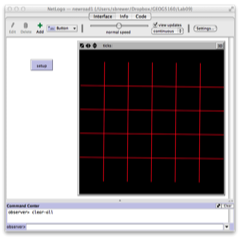
Now add a setup procedure that will load and display the vector layer:

to setup  
 clear-all  
 set roads gis:load-dataset ("data/roadclip/roadclip.shp")  
 gis:set-world-envelope gis:envelope-of roads  
  
 gis:set-drawing-color [255 0 0]   
 gis:draw roads 2  
 end

Most of this should be fairly straightforward after the previous example. Line-by-line, this procedure:

* Loads the road shapefile into Netlogo, stored with the variable name roads
* Sets the GIS world extent to the limits of the shapefile
* Sets the vector color to red (in RGB triplets)
* Draws the vector layer with a line thickness of 2

Check and save your model, then add a [setup] button to the interface, linked to this procedure. Once the setup runs, the following screen should appear:



### Getting patch values from vector layers

The gis:intersecting function can be used to find the intersection between any Netlogo object (turtles, patches) and a vector layer. Here we will use it to find those patches that intersect with the road vector layer, and assign them a variable (is-road?), defining them as roads (or not). Add this variable as a patch variable at the top:

patches-own [ is-road? ]

And now add the following in the setup procedure:

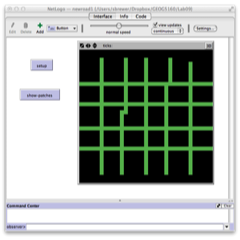
ask patches [ set is-road? false ]  
ask patches gis:intersecting roads  
 [ set is-road? true ]

We first initialize all patches as having the ‘is-road?’ variable set to false. Then we loop through and reset this to true for any patch that intersects with a road.

To show this, we can add a modified version of the ‘show-patches’ procedure used above. Create a new procedure:

to show-patches  
 clear-drawing  
 ask patches with [ is-road? ]  
 [  
 set pcolor green  
 ]  
end

And add a button on the interface tab that links to the show-patches procedure. Run the setup, and click the new button to see how the roads look as patches.



As before, the representation depends on the patch resolution of the Netlogo model world. If you change this from the settings menu, then you will get a higher-resolution set of roads shown by the patches.

The gis:intersecting function will work will all types of vector data, including points, lines and polygons. For polygons data, however, a faster method is to use the function gis:apply-coverage, where each patch takes the value of the polygon that makes up the majority of the patch’s area.

### Converting the shapefile to a network

There are several ways to get an agent to move across our street network. We could simply use the street patches as designed in the previous step, but a more useful way is to convert the shapefile into a network. Networks consist of a set of ‘nodes’ and ‘links’ that connect these. We can start by creating the nodes at the vertices of the streets (i.e. at the start and end of each street segment). The code to do this is relatively complex. We’ll start with a short description of what this does, then look at the code, and finally go through it line-by-line.

The vector layer consists of set of features (line segments), each of which has a set of vertices (the beginning and end). We need to loop through all features, extract the vertices, and then convert these to nodes. To do this, we need to project their coordinates (in GIS world) to the model world. Finally each pair of nodes will need to be linked to produce a full topological network in the model world.

Each node will be created as a breed of turtle, so add the following after the ‘patches-own’ statement at the top:

breed [ nodes node ]

Now create a new procedure to extract the vertices from the vectors:

to setup-links  
 set-default-shape nodes "circle"  
 foreach gis:feature-list-of roads [  
 foreach gis:vertex-lists-of ? [  
 let n1 makenode first ?  
 let n2 makenode last ?  
 ask n1 [create-link-with n2]  
 ]  
 ]  
end

Line-by-line:

* Begin the procedure
* Set the default shape for a node to a circle
* Loop through all the features in the roads layer
* Loop through all the vertices for a given feature from the road layer. The features are processed one at a time by the previous loop. The ? represents the feature that is being processed by this loop.
* Create a node (n1) from the first vertex. The ? now represents the current set of vertices
* Create a node (n2) from the second (last) vertex
* Make a link (a special type of turtle) between the two nodes
* End the procedure

The question mark is a very useful code shorthand. It is a temporary variable that represents the most recently created value. As we loop through the features, each feature is passed to the next command by the ?. In the second (inner) loop we go through the sets of vertices, and now the ? represents the set of vertices associated with the feature selected in the outer loop.

The two lines where the nodes are created use a function called makenode. This does not exist yet, so we will have to create it as a function. Add the following procedure:

to-report makenode [ vertex ]  
 let location gis:location-of vertex  
 let x item 0 location  
 let y item 1 location  
 let n nodes with [xcor = x and ycor = y]  
   
 ifelse not any? n [  
 create-nodes 1  
 [  
 set xcor x  
 set ycor y  
 set size 4  
 set node-name self  
 ]  
 ]  
 [  
 set node-name one-of n  
 ]  
 report node-name  
end

Line-by-line

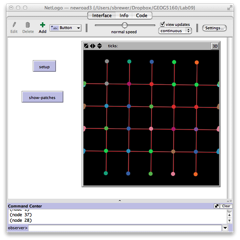
* Start the procedure. Here we use a report procedure, as we need to remember the name of the nodes as we go along, in order to connect pairs together.
* Get the GIS location of the vertex, and store this in a temporary variable (location)
* Create a variable x that stores the GIS easting from location projected into the model world
* Create a variable y that stores the GIS northing from location projected into the model world
* Create a variable n which has the number of nodes at the current location (this is to avoid creating a node where one already exists)
* The ifelse statement checks to see if any nodes are already there
* If there are no existing nodes, we create a new node, and set its coordinates to x and y (and size to 4)
  + We also use a variable to store the node name (to be used in creating the links)
  + The next two brackets close the create-nodes and first part of the ifelse statement
  + The next bracket opens the else part of the ifelse
* If there is a pre-existing node, so we simply set node-name to this existing node rather than create a new one
* We report back the node name
* And finish the procedure

The variable node-name is a global variable, and needs to be added to the declared variables in globals at the top.

Finally, add the following line to the end of the setup procedure, to call the procedure setup-links as part of the setup.

setup-links

Check and save your model. Go back to the interface tab, cross your fingers and hit [setup]. The resulting screen should look similar to this (if you changed your model settings to create a larger world):



### Adding agents to the network

The final step in this exercise will be to add an agent which will travel over the network. We will create a new breed ‘cars’ for this, so add the following to the definitions at the start of your code:

breed [ cars car ]

Cars will need to have two variables, one which records the node it is travelling from, and one for the node it is traveling to. In the definitions section add after breed:

cars-own [   
 origin-node   
 dest-node   
]

Also add two variables called p1 and p2 to the globals parameter set. These will be used to help define the travel direction of the car. We need to do three things:

* Set up the car so it is on the network and traveling along a link (we will do this by getting it to travel between linked nodes)
* Move the car a certain distance at each time step
* Add a go procedure

We will start with setting up the cars. Add the following new procedure:

to setup-cars  
 set-default-shape cars "car"  
 ask one-of nodes [  
 set p1 self  
 set p2 one-of link-neighbors   
 ]  
   
 create-cars 1  
 [  
 move-to p1  
 set size 6  
 set origin-node p1  
 set dest-node p2  
 face dest-node  
 ]  
end

Add a call to this procedure in setup. Add setup-cars to the end of your setup procedure (after setup-links).

While long, this code is fairly straightforward.

* Set the shape of the agents, then pick a random node.
* Set two temporary variables to the current node (p1) and a random choice of one of the linked neighboring nodes (p2).
* Create a single car agent
* Move this to p1
* Set the cars ‘origin node’ to p1
* Set the cars ‘destination node’ to p2, and set the car to face p2

Now we add a new procedure to move the cars:

to move-cars  
 let dxnode distance dest-node  
 ifelse dxnode > patch-size  
 [  
 fd 1  
 ]  
 [  
 move-to dest-node  
  
 ask dest-node   
 [  
 set p1 self  
 set p2 one-of link-neighbors   
 ]  
 set origin-node p1  
 set dest-node p2  
 face dest-node  
 ]  
end

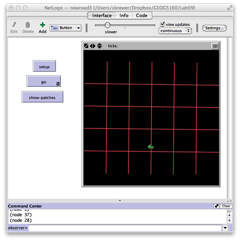
In this procedure, the cars will be moving one patch length each time step.

* We start by calculating the remaining distance to the destination node, to stop the cars from overshooting intersections.
* If this is greater than the cars next movement (patch-size), we move forward by one.
* If not, we move to the node (effectively stopping at that intersection).
* At that node we then re-use the code from the previous section, setting p1 to the current node, and p2 to a randomly chosen linked node
* We then reset the origin-node and dest-node variables for that car and turn toward the new destination

Finally, add a ‘go’ procedure:

to go  
 ask cars [  
 move-cars  
 ]  
end

Link this to a button on the interface, check, save your model, then set it up and run it. You will probably need to slow the model speed in order to see the progress made by the car. It may also take some time to process all the nodes for the dataset, so be patient with your setup procedure.



As with the raster model, there are many ways in which this might be extended to include more realistic driving behavior.

## Assignment

Use a word document to record your answers and output. Assignments, to include both the word document and .nlogo script file for the model, should be submitted to Canvas before class on Monday, XXXXXX XX. Ensure your assignment has been saved using the following naming convention: Lab08\_lastname\_script and Lab08\_lastname\_report.

1. Make a new version of the elevation model that uses a new DEM layer. For this, you will need to find your own DEM dataset, extract a DEM from the region, and save it as an ESRI ASCII grid. You can find DEM datasets for Utah [here](http://gis.utah.gov/data/elevation-terrain-data/10-30-90-meter-elevation-models-usgs-dems/) or any other place of your choosing. Update the procedure for generating trees on patches, dependent on their elevation to account for the new range of elevations. You can do this be changing the value of the parameters in the code, or by adding sliders for these (the most important one to change is , which defines the midpoint of the elevation range). Provide the model code and a screenshot displaying the output, together with a description of which region you have chosen [2]
2. There are many ways in which the road model described above might be extended to include more realistic driving behavior. One of the most useful would be to replace the random turning with movement toward a specific destination. Describe briefly how this might be achieved. What variables and processes might you want to include? How would the cars/agents be aware of a destination? You do not need to create this model, but you should give an outline of how this would work as pseudocode (see [this page](http://www.wiley.com/college/busin/icmis/oakman/outline/chap05/slides/pseudo.htm) for an example of pseudocode).[3]