2006-11

Using NetLogo in the Data Farming Environment

Koehler, Matthew

Using NetLogo in the the Data Farming Environment

Matthew Koehler
The MITRE Corporation[1]

NetLogo is a freely available agent-based modeling environment being developed by Northwestern University's Center for Connected Learning (ccl.northwestern.edu/netlogø). NetLogo is an excellent environment for creating simpler or smaller-scale agent-based models or prototyping more complex models. NetLogo's strengths include using a very easy to learn and flexible scripting environment, a GUI interface that handles all the necessary code for you, and a section dedicated to documenting your model, and a very large sample model library with very good documentation. The down side of NetLogo is that you must create all functionality you desire to have in the model, which can be time consuming if you have a great deal of complicated behaviors. Furthermore, NetLogo is written in Java and its scripting language is only semi-compiled (some primitives are compiled into Java byte-code, other primitives are interpreted), which can lead to some performance issues if your models is very large or involves a great deal of computation. Finally, NetLogo is compatible with the Data Farming and cluster computing methods and tools created by Project Albert and its collaborators.

Structure & Features of NetLogo

NetLogo contains three basic types of entities within it: turtles (agents), patches (the landscape), and the observer. Turtles can be subdivided into different classes (called breeds). All of these entities can run code and interact with each other and with other types of entities. Variables can be assigned globally, all entities having access to them, or specifically (in which case only the specified group has access to that variable).

This structure gives modelers a great deal of flexibility when creating a model. Different types of agents can have a common set of variables as well as unique sets used to create specific behaviors. For example: all agents that move around on the ground could have a common set of variables used for movement, however, they could all have unique variables associated with other capabilities. Discretizing the landscape into autonomous regions (patches) all of which can execute code and maintain a unique internal state presents many opportunities to the modeler. For instance, one can import a .bmp image file (perhaps created with GIS data) which the patches can use to set internal parameters. Then the agents moving around on the patches can query the patches for the values of the parameter and then behave appropriately.

The NetLogo user's interface is also fairly flexible and very easy to use. You can create sliders, buttons, switches, and monitors with drag-and-drop convenience. Plots can also be created very easily; however, they will require a few lines of code in the procedures also. NetLogo can also print output to a window on the user interface or print to a file. Finally the entire NetLogo state (the value of all parameters and the states of all agents, patches, etc.) can be saved and reloaded.

Although the NetLogo scripting language is not extensible per se, NetLogo can be setup to access external Java programs so one could create external programs to handle particularly computationally intensive procedures or create NetLogo models that update themselves based upon a web service or database call. NetLogo also has the ability to act as a server and receive input from Texas Instrument calculators or other computers. This gives modelers the ability to create human-in-the-loop models. The next section contains details on how to create a NetLogo model that is compatible with the Data Farming Environment (DFE). Full detail can be found in Koehler (2005).

Flow of the DFE with NetLogo

The general flow of the system is as follows: 1. create a NetLogo model following a set of conventions; 2. parse the NetLogo file into an input XML file; 3. using the XStudy tool, pick the sliders, choosers, or switches that will be varied during the runs; 4. use Old McData (OMD) and Condor to kick-off the runs and collect the data (this can de done on a single machine or multiple machines). All of the software is written in Java and should work on any machine

[1] The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

with a Java Virtual Machine. This system, though not perfected, is robust enough to handle the pressure of workshop demands—including thousands of runs done remotely on clusters in different countries. We have successfully run Netlogo in two different cluster computing environments: the Maui High Performance Computing Center and on a cluster maintained by the Singapore Defense Science Organization. The system is capable of handling any sort of experimental design from full factorial to Nearly Orthogonal Latin Hypercube. Furthermore, OMD has post-processing capabilities that can be used with evolutionary programming algorithms and other types of user defined algorithms to create a more dynamic study.

In the following discussion we will examine the conventions necessary when putting together a Netlogo program, as well as general instructions for the use of the other software used for the multiple runs; however, it is assumed the reader is familiar with Condor. The software discussed in this paper is, or soon will be, available on SourceForge. Alternatively, the software is available from the authors. Condor is available from its developers at: http://www.cs.wisc.edu/condor/. NetLogo is available from its developers at: http://ccl.northwestern.edu/netlogø.

Setting up the NetLogo Model

The current system requires certain features within the NetLogo model.[1] These requirements will be discussed below. These requirements have minimal impact on the structure of the program and on the speed of execution and are designed to allow an external Java program to start the model, set parameter values (sliders, choosers, and switches), start and end a run, and collect output data (both end of run and time series). In general, the wrapper starts Netlogo and loads the model, and then it tells Netlogo to iterate a certain number of times. At the end of the requisite number of iterations, output data is collected and the Netlogo run is terminated.

Global Variables

The model needs three global variables: **`stopped`**, **`filename`**, and **`clock`**. These are used by the external program to run NetLogo, keep track of output data, and allow the modeler to control the behavior of their model separately from the Java wrapper.

The `Setup` Procedure

First, the NetLogo model must have a procedure called **`setup`** to instantiate the model and to prepare the output files. At a minimum it will need the following lines of code:

```
to setup
set clock 0
set stopped false
setup-file
end
```

Every time the model is run it will be in a newly started instantiation of NetLogo; therefore, one is not required to set variables (unless they need to be something other than zero). However, you may want to clear values and set others so that you will know exactly how the model is starting up. If you do clear values DO NOT use the command **`clear-all`** or **`ca`**. If you want to clear values use commands such as **`clear-turtles`**, **`clear-patches`**, **`clear-all-plots`**, **`clear-output`** and then manually set the variables. If you use **`clear-all`** you will set the variable **`filename`** to 0. This will cause problems later on, when the output from all the runs is collected because all the files will have the same name. The batch version of NetLogo is run by a Java program that will set certain parameters, among those is **`filename`**. Once NetLogo is started, the Java program will call the **`setup`** procedure. If **`setup`** then resets the value of **`filename`** Condor and OMD will have trouble keeping track of the output files because they will all have the same name. A more comprehensive version of the setup procedure that includes resetting of values is below:

```
to setup
ct
cp
clear-output
clear-all-plots
;; manually set all variables
set clock 0
set stopped false
setup-file
end
```

The setup-file procedure is very short and could be called from within the setup procedure. It is recommended to keep them separate for clarity. A sample of this procedure is below:

```
to setup-file
 ifelse filename = 0
 [file-open "Your_BackUp_Name_Here.csv"]
 [file-open filename]
end
```

[1] It should be noted that this system was created in 2005; the current version of NetLogo may require slight changes to the code described herein. Please contact the author with questions.

This procedure allows you to run the NetLogo program inside the cluster computing environment or in the standard NetLogo program for testing purposes. This works because it checks to see if the variable **filename** has been set by the Java wrapper program. If it has not been set by the Java wrapper, it will open a default file of your choosing.

The Go Procedure

All models must also have a **go** procedure. The **go** procedure is a little different than the usual NetLogo program. First of all, the procedure must be called "go." Second, the wrapper runs the NetLogo program by asking it to step a certain number of times. Due to this structure, it is important to "protect" your runtime code by nesting it inside an **if** statement that returns true if stopped is false. Sample code for the **go** procedure can be found below:

```
to go
set clock clock + 1
if not stopped
  [
  ;;runtime code goes in here

  if 'stop condition is true'
        [do-file-print close-files set stopped true]
  ]
end
```

By nesting the runtime code inside the **if** statement, the wrapper can run the model any number of times without any potential damage to the output after the stop condition is met. For example, if you have set up the wrapper to run your model 6000 times but you have a stop condition that is triggered at time step 3500, the wrapper will continue to tell your model to step another 2500 times. If you generate output every time step and do not protect it, then you will end up with another 2500 lines of output. As your stop condition could be triggered at different times it could be very difficult to fix your data post run. It is also important to segregate any end-of-run printing procedures from the file close procedure. Once the wrapper is done stepping the NetLogo program, it will tell the program to **close-files**. Therefore, you must have a procedure in your program that is called **close-files**. If this procedure includes anything other than file closing code, it may cause a problem as it will be run anytime files are closed. If you close files anytime the stop condition for your model is true, then any other code will be run every time the wrapper steps your program once the stop condition is met (this is not an issue if you protect your runtime procedures in the aforementioned **if** statement and make the **close-files** procedure exclusively devoted to closing files). However, this does require that your model have a stop condition that will be triggered at least one time step before the wrapper ends the run because the wrapper will simply stop telling the program to step and then call the **close-files** procedure. Sample code for the **do-file-print** and **close-files** procedures can be found below:

```
to do-file-print
file-print "output goes here"
end

to close-files
file-close-all
end
```

Also, there is no post-processing currently associated with NetLogo runs, so if you want something in the output file, such as input parameters, you must write it there in the program (in something like the **do-file-print** procedure). This file will be a single line if you are only collecting end of run data. If, however, you are collecting time series data, this file may be very large.

The above represents all the requisite code for a NetLogo program to set it up for cluster computing. Now, part of the utility of cluster computing is being able to run a model many times with different parameter values. The system we have developed can run NetLogo programs many times and change parameter values. However, the parameters that will change need to comport with a set of standards. First, they must be sliders, choosers, switches, etc. and, therefore, must appear in the "Interface" tab of the NetLogo environment. Second, these parameters may not contain any special characters like ?, %, $, *, and so on. Third, they may be set to numeric values only--no strings. For example, a chooser with the values: High, Medium, and Low would not be acceptable. The chooser should have values such as: 1, 2, and 3 which could then be mapped to High, Medium, and Low in the procedural part of the NetLogo model. This does not preclude other parameters from taking on any values you wish and having special characters in their name…these standards only apply to parameters values you wish to change in an automated fashion.

References

Koehler, M. and S. Upton, B. Tivnan, Clustered Computing with NetLogo and RepastJ: Beyond Chewing Gum and Duct Tape. Proceedings of Agent 2005. Argonne National Lab, Chicago, IL. (2005).

Wilensky, U. NetLogo. http://ccl.northwestern.edu/netlogø. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. (1999).