Improving Execution Speed of Models Implemented in NetLogo

Steven Railsback[1], Daniel Ayllón[2], Uta Berger[3], Volker Grimm[2], Steven Lytinen[4], Colin Sheppard[5], Jan Thiele[6]

[1]*Lang Railsback and Associates, 250 California Avenue, Arcata, California, 95521, United States*
[2]*Department of Ecological Modelling, Helmholtz Centre for Environmental Research-UFZ, Permoserstrasse 15, Leipzig 04318 Germany*
[3]*Institute of Forest Growth and Computer Sciences, Technische Universität Dresden, Postfach 1117, Tharandt 01735, Germany*
[4]*School of Computing, DePaul University, 243 S. Wabash, Chicago, Illinois, 60604, United States*
[5]*International Energy Studies Group, Lawrence Berkeley National Laboratory, 1 Cyclotron Road MS 90R2121, Berkeley, California, 94720, United States*
[6]*Department of Ecoinformatics, Biometrics and Forest Growth, Büsgen Institute, University of Göttingen, Büsgenweg 4, Göttingen, DE-37077, Germany*
Correspondence should be addressed to *steve@langrailsback.com*

Journal of Artificial Societies and Social Simulation 20(1) 3, 2017
Doi: 10.18564/jasss.3282 Url: http://jasss.soc.surrey.ac.uk/20/1/3.html

Received: 22-09-2016 Accepted: 27-10-2016 Published: 31-01-2017

Abstract: NetLogo has become a standard platform for agent-based simulation, yet there appears to be widespread belief that it is not suitable for large and complex models due to slow execution. Our experience does not support that belief. NetLogo programs often do run very slowly when written to minimize code length and maximize clarity, but relatively simple and easily tested changes can almost always produce major increases in execution speed. We recommend a five-step process for quantifying execution speed, identifying slow parts of code, and writing faster code. Avoiding or improving agent filtering statements can often produce dramatic speed improvements. For models with extensive initialization methods, reorganizing the setup procedure can reduce the initialization effort in simulation experiments. Programming the same behavior in a different way can sometimes provide order-of-magnitude speed increases. For models in which most agents do nothing on most time steps, discrete event simulation – facilitated by the time extension to NetLogo – can dramatically increase speed. NetLogo's BehaviorSpace tool makes it very easy to conduct multiple-model-run experiments in parallel on either desktop or high performance cluster computers, so even quite slow models can be executed thousands of times. NetLogo also is supported by efficient analysis tools, such as BehaviorSearch and RNetLogo, that can reduce the number of model runs and the effort to set them up for (e.g.) parameterization and sensitivity analysis.

Keywords: Agent-Based Modeling, Computational Efficiency, Execution Speed, Individual-Based Modeling, NetLogo, Modeling Platforms

Introduction

1.1 Agent-based models (ABMs) have become essential tools in social (and other) sciences, and NetLogo (Wilensky 1999) is probably now the most widely used software platform for ABMs. The CoMSES Net Computational Model Library (http://www.openabm.org/models), to which this journal encourages submitting models, appears to be dominated by models implemented in NetLogo; and two recent textbooks on ABMs (Railsback & Grimm 2012; Wilensky & Rand 2015) use NetLogo as the platform. The reasons for NetLogo's popularity include its professional design and packaging, comprehensive documentation, high-level programming language with many built-in commands and data types specialized for ABMs, integrated graphical user interface, integrated tool for performing simulation experiments, and active user community.

1.2 Despite its many advantages, NetLogo has a reputation as not suitable for large or complex models. It is widely accepted that NetLogo can make model-based science efficient by greatly reducing programming effort and

making it easier to test both the software and model design. Therefore, NetLogo appears to have a reputation as especially suited for relatively simple ABMs intended mainly to communicate ideas. But for computationally intensive "serious" ABMs, there is a belief that NetLogo's execution speed is such a constraint that models will need to be re-implemented in lower-level languages (e.g., Sklar 2007; Bouquet et al. 2015; Lammoglia et al. 2015).

1.3 This belief that NetLogo is inherently unsuited for large models is not well supported (Tisue & Wilensky 2004). The belief likely originated in part with the understanding that NetLogo and the Java language it is based on are interpreted instead of compiled. This understanding is no longer as meaningful as it once was: many parts of NetLogo are in fact compiled (Sondahl et al. 2006); and modern versions of languages that run on the Java virtual machine are no longer considered seriously slower than other languages (Wikipedia 2016). Railsback et al. (2006) compared version 2.1 of NetLogo to other ABM platforms and found its execution slower, but not dramatically slower, than the fastest (RePast and MASON). That comparison was made before NetLogo was converted from an interpreted to mainly compiled language (Sondahl et al. 2006).

1.4 In fact, people deciding which platform to use for large ABMs need to be aware that NetLogo's compiler now includes a number of optimizations and clever designs (Sondahl et al. 2006; CCL (Center for Connected Learning, Northwestern University) 2016) that could often make NetLogo faster than platforms lacking such attention to execution speed. Especially, NetLogo's optimizations are likely to make it faster than simple code written in a standard programming language by an inexpert or time-constrained programmer. As an illustration, Lytinen & Railsback (2012) found version 5.0 of NetLogo about 20 times faster than the "ReLogo" element of RePast (Ozik et al. 2013) for several example models of modest complexity.

1.5 Even though NetLogo does not appear inherently too slow for serious modeling, our experience using and teaching agent-based modeling indicates that many NetLogo programs can in fact be very slow until their time-consuming "bottlenecks" are found and remedied. We encourage students to write the first versions of their software in NetLogo's natural style, using its primitives and characteristic code structures to make program statements as simple and understandable as possible. This approach allows models to be programmed and tested rapidly, but it is not uncommon for the initial software to be prohibitively slow. However, we have also found that such prohibitively slow programs can almost always be sped up, often by orders of magnitude, with a few simple changes. A few common NetLogo statements are often the bottleneck, not because the NetLogo primitives are poorly designed but because they require extensive computations that would be time-consuming in any programming language. Using the strategy and techniques we provide here, speeding up a NetLogo program usually takes, in our experience, less than a day and the changes can easily be tested to find any mistakes that were introduced. However, one technique for producing computationally efficient models, discrete event simulation, affects the entire model design and needs to be considered from the start.

1.6 Our first objective here is to describe a process for finding execution speed bottlenecks in NetLogo code and techniques that often speed up execution. We present a five-step strategy that includes ways to quantify execution speed and a set of often-effective techniques. We provide NetLogo programs illustrating some of the techniques online.

1.7 As a second objective, we also present ways of making large simulation experiments on NetLogo models feasible. Once a model has been developed, tested, and made computationally efficient, it is then analyzed via simulation experiments to understand both the model and the system it represents. We address two tools for implementing simulation experiments: NetLogo's "BehaviorSpace" experiment manager and the popular "RNetLogo" package that can run NetLogo models from the R statistical package (Thiele et al. 2012, 2014).

1.8 We assume that readers are familiar with the basics of NetLogo and therefore make reference to standard parts of NetLogo (e.g., "patches" as square grid cells; "turtles" as mobile agents; "primitives" as the programming language's built-in commands) without explaining them. We provide example code statements to illustrate the issues and solutions we describe; text in Courier font is NetLogo commands (primitives) or code. The tests we used to find and evaluate potential speed increases were all conducted in version 5.3 of NetLogo.

1.9 Example NetLogo models illustrating techniques we describe are available for download at http://www.railsback-grimm-abm-book.com/JASSS-models.html (archived at: "http://www.webcitation.org/6nFYwsz7C). References below to an "example model" refer to NetLogo files available at this site.

A Strategy for Producing Efficient NetLogo Code

2.1 We advocate the following five-step strategy for building an efficient NetLogo model. Details on conducting steps 2-5 are then provided in following sections. This strategy depends on consistent use of one or several test

scenarios, each being a carefully selected and documented model version; set of inputs (input files, parameter values); model settings such as which output files are written, the rule for when the model stops, and display settings such as when the View is updated; and hardware – the computer the tests are executed on. These scenarios might include a short test case, a typical simulation, or an especially demanding extreme case.

1. Write the initial software in NetLogo's natural style, making program statements as simple and understandable as possible. (However, one technique for making models efficient, discrete event simulation, should be considered from the start.) Test the software thoroughly to find the inevitable mistakes before proceeding further, e.g., by using methods described in Chapter 6 of Railsback & Grimm (2012). (Common programming mistakes can themselves slow down execution immensely, so code testing often produces major speed increases. However, we do not address effects of programming errors, instead focussing on improving code that has already been tested.)
2. Measure execution time of the initial software under the test scenario(s) discussed above, using NetLogo's timer.
3. Try simple speed-up methods that do not require code changes and determine whether they result in acceptable execution times. Even slow execution speeds may be acceptable if computer time is less important than the programming time needed for further speed-up steps.
4. If further improvement is needed, use NetLogo's profiler extension to identify the slow procedures.
5. Attempt to speed up the slow procedures using several techniques which do require (usually minor) changes to the program. As these methods are used, repeat the execution time measurements to determine how much improvement has been made. After each technique is implemented, test the code to eliminate any errors that were introduced by showing that it produces the same results as the initial version.

Measurement of Execution Speed and Detection of Bottlenecks

3.1 An essential step to speeding up a model is measuring execution speed so that particularly slow parts can be identified and so that attempts to speed up execution can be evaluated quantitatively. NetLogo provides two tools for doing so.

3.2 The first tool is the timer, used via the primitives `reset-timer` and `timer`. The timer primitive simply reports the clock time, in seconds, since `reset-timer` was last executed. It can be used, for example, to report the time taken for a full model run (as needed for steps 2 and 3 of the above strategy) by modifying the go procedure:

```
to go
  if (ticks < 1) [ reset-timer ]
  tick
  if (stopping-condition)
  [
    show (word "Execution finished in " timer " seconds")
    stop
  ]
  ...
```

(Our example models use this timer method.) Alternatively, the time for one execution of the go procedure can be reported:

```
to go
  reset-timer
  ...
  show timer
end
```

The `timer` primitive can also be included among the reporters used to produce output for a BehaviorSpace experiment, so the execution time is a model output. (The example model "InRadius-vs-DistanceMyself.nlogo" illustrates this method.)

When using `timer` to investigate execution speed, be aware that NetLogo code runs substantially slower the first 2-3 times it is executed after being edited (possibly due to the Java virtual machine re-compiling the revised

code and some hardware caching; Wikipedia (2016)). Therefore, measurements need to be repeated until they produce consistent results.

The profiler extension packaged with NetLogo is also an essential tool for understanding and quantifying execution speed. It can be used as an alternative to `timer` for steps 2-3 of the above strategy for producing efficient code, and is required for step 4. The profiler (thoroughly documented in the NetLogo User Manual) reports the time spent in each procedure. In the profiler's output report, look for procedures with high values of "exclusive time" (the time spent executing code within the procedure); these should be the targets of efforts to speed up. (If the "go" procedure has high exclusive time, it can be because much of the execution time is spent updating the display. Re-run the profiler with "view updates" turned off.) Figure 1 illustrates the profiler's value.

When measuring execution time, it is important to avoid misleading results due to hardware issues. Measurements used to compare versions of the code should, to be comparable to each other, be made on the same machine and in the absence of competition for computer resources (CPU availability, memory) with other software. Use tools like the Windows CPU meter or Task Manager to make sure at least one processor core and ample RAM are available just for NetLogo.

Simple Steps to Increase Speed

4.1 There are several steps that NetLogo users can take as soon as a model's program has been written and tested. While these steps do not always result in substantial speed increases, they are simple and safe to try.

4.2 First, make sure the NetLogo profiler is deactivated by commenting out the statements that use it. Profilers use up computer resources and by themselves slow a model down.

4.3 Second, check and possibly adjust settings for the NetLogo view (the two-dimensional graphical display). Although these settings generally have no effect when a model is run in BehaviorSpace with view updates turned off (as models typically are for serious simulation experiments), they can dramatically slow down performance of some models when the view is in use. Setting the view update to "on ticks" (once per time step) can speed up some models. Models that execute each tick very quickly can have their speed limited solely by the "frame rate" setting, which by default limits NetLogo to 30 ticks per second. It is essential to understand the complex effects of this setting, explained in the "View updates" section of NetLogo's programming guide.

4.4 Third, try a 64-bit version of NetLogo. Starting with version 5.3, NetLogo is distributed in both 32- and 64-bit mode. Some models run substantially faster (e.g., 30-40% faster) in 64-bit mode (which can also allow models with more agents and larger spaces to run without exceeding memory limits). However, 64-bit NetLogo does not always help: the model of Ayllón et al. (2016), in which relatively few agents execute many calculations, actually took 15% longer to execute in 64-bit compared to 32-bit versions of NetLogo 5.2.

4.5 Finally, when a model is ready for simulation experiments executed without view updates, users can comment out code statements that set agent colors and shapes that are strictly for display purposes. (Figure 1 illustrates a case where such code is a significant use of execution time.) This technique of course cannot be used for colors or shapes used as important model variables.

Techniques for Speeding Up Slow Code

5.1 For many models, the above simple steps will not provide substantial execution speed benefits, so users must then revise their code to make the slow procedures more efficient. In our experience, NetLogo programs that run very slowly usually do so because of how key primitives – especially `with` – are used. In the following subsections we discuss how to avoid these primitives or use them more efficiently. We also identify some other code revisions that can provide substantial benefits for some models. These techniques were identified from our own experience and from the NetLogo Users Group.

5.2 We remind users that all of these techniques are likely to introduce errors, which can be found by carefully comparing the modelâĂŹs results to those obtained before the revisions. Start by documenting one or several test scenarios (discussed above) and saving the output from them, including detailed output (e.g., file output generated every tick, perhaps via BehaviorSpace). Then make the code changes and, if they succeed in producing worthwhile speed increases, test whether the code still produces the same results that the original test cases did. (Whether the revised code should produce *exactly* the same results as the original can depend on whether random-seed is used to control the sequence of random numbers and whether the revisions result in subtle

A

Command Center

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                          Calls  Incl T(ms)  Excl T(ms)  Excl/calls
SELECT-TADPOLE-HABITAT         2616   16156.542   16156.542       6.176
GO                              154   74817.969   12889.784      83.700
SHADE-PATCHES                   306    9929.612    9929.612      32.450
UPDATE-HYDRAULICS-FOR           153   15782.168    8439.546      55.160
OVIPOSIT                        942    7551.801    7547.651       8.012
SELECT-BREEDER-HABITAT          942    6724.955    6724.955       7.139
TADPOLES-DEVELOP            5691330    6405.282    6268.666       0.001
TADPOLES-SURVIVE               2543       0.707    1868.812       0.735
OVI-SUITABILITY             6444513    2326.567    1292.418       0.000
DECIDE-IF-READY                1119     831.243     821.206       0.734
SAVE-EVENT                   215098     519.804     519.804       0.002
RANDOM-BERNOULLI            5748592     499.510     499.510       0.000
LOGISTIC-WITH-1-9-INPUT     1484345     472.062     472.062       0.000
UPDATE-HABITAT                  153   21083.075     387.349       2.532
EGGS-HATCH                     1722     632.265     373.905       0.217
EGGS-SCOUR-SURVIVAL         1481802    1035.965     345.046       0.000
EGG-VELOCITY                1482392     222.603     222.603       0.000
UPDATE-OUTPUT                   153      49.167      49.167       0.321
EGGS-DEVELOP                   1722       3.598       3.598       0.002
EGGS-SURVIVE                   1758       6.407       3.594       0.002
```

B

Command Center

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                          Calls  Incl T(ms)  Excl T(ms)  Excl/calls
SELECT-TADPOLE-HABITAT         2664   14259.105   14259.105       5.353
SHADE-PATCHES                   306    9903.050    9903.050      32.363
OVIPOSIT                       1104    8472.032    8470.486       7.673
UPDATE-HYDRAULICS-FOR           153   15762.314    8450.411      55.231
SELECT-BREEDER-HABITAT         1104    7362.284    7362.284       6.669
TADPOLES-DEVELOP            5360421    5908.949    5787.483       0.001
GO                              154   66053.174    5142.541      33.393
TADPOLES-SURVIVE               2570       0.185    2032.993       0.791
OVI-SUITABILITY             6444513    2351.928    1385.484       0.000
DECIDE-IF-READY                 937     719.794     717.939       0.766
RANDOM-BERNOULLI            5424564     468.455     468.455       0.000
LOGISTIC-WITH-1-9-INPUT     1484401     444.112     444.112       0.000
SAVE-EVENT                   213164     440.297     440.297       0.002
UPDATE-HABITAT                  153   21057.666     352.277       2.302
EGGS-SCOUR-SURVIVAL         1481831     968.106     301.751       0.000
EGGS-HATCH                     1755     477.385     288.853       0.165
EGG-VELOCITY                1482426     225.786     225.786       0.000
UPDATE-OUTPUT                   153      14.491      14.491       0.095
EGGS-SURVIVE                   1788       5.518       3.265       0.002
EGGS-DEVELOP                   1755       1.783       1.783       0.001
```

C

Command Center

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                          Calls  Incl T(ms)  Excl T(ms)  Excl/calls
SELECT-TADPOLE-HABITAT         2664   13255.651   13255.651       4.976
SHADE-PATCHES                   306    9879.469    9879.469      32.286
OVIPOSIT                       1104    8455.530    8454.087       7.658
UPDATE-HYDRAULICS-FOR           153   15626.440    8383.795      54.796
SELECT-BREEDER-HABITAT         1104    7361.537    7361.537       6.668
TADPOLES-DEVELOP            5360421    5797.491    5688.940       0.001
GO                              154   68244.609    5612.883      36.447
UPDATE-OUTPUT                   153    3297.369    3297.369      21.551
TADPOLES-SURVIVE               2570       0.228    1760.824       0.685
OVI-SUITABILITY             6444513    2295.923    1273.323       0.000
DECIDE-IF-READY                 937     740.823     739.137       0.789
LOGISTIC-WITH-1-9-INPUT     1484401     486.182     486.182       0.000
RANDOM-BERNOULLI            5424564     470.622     470.622       0.000
SAVE-EVENT                   213164     398.218     398.218       0.002
UPDATE-HABITAT                  153   20910.244     351.057       2.294
EGGS-SCOUR-SURVIVAL         1481831    1024.237     318.351       0.000
EGGS-HATCH                     1755     463.061     285.669       0.163
EGG-VELOCITY                1482426     223.099     223.099       0.000
EGGS-SURVIVE                   1788       4.351       2.174       0.001
EGGS-DEVELOP                   1755       1.880       1.880       0.001
```

Figure 1: Profiler reports from three runs of the frog model of Railsback et al. (2016). In (A) NetLogo's view was updated each tick, while it was not in (B) ("view updates" was turned off). As a consequence, the go procedure used almost 13 seconds (12,889 milliseconds of exclusive time) in (A) and only 5.1 seconds in (B). In (C), file output was turned on; generating over 13 megabytes of output increased the time used by the update-output procedure by only 3 seconds (3297 milliseconds exclusive time, compared to 49 in A). The cost of file output is small, presumably due to hardware caching. In all runs, `shade-patches` was among the most time-consuming procedures; it is for display purposes only and could be turned off for simulation experiments, saving 10 seconds per model run.

changes such as affecting the order in which agents execute actions.) Debug the code changes by resolving the differences between the new and test output.

Make Agent Filtering Statements Efficient – Or Eliminate Them

6.1 NetLogo's primitives for filtering (subsetting) agentsets are extremely useful for writing simple, clear code statements such as `ask turtles with [ not happy? ] [ find-new-spot ]` (from the NetLogo library's Segregation model). The `with` primitive examines an agentset (in this example, all turtles) and creates a new agentset containing those members of the first agentset that meet a criterion (here, their variable `happy?` has a value of false). Other examples of primitives that examine an agentset and create a subset of it are `in-radius`, `max-n-of`, `max-one-of`, and `with-min`. These primitives are often combined in particularly powerful, but computationally demanding, statements such as `min-n-of 10 (turtles with [size < 5]) [distance myself]` (which reports the 10 nearest turtles with size less than five). The ability to write such statements is a key feature of NetLogo, making it easy to identify agents with particular characteristics even as those characteristics change during a simulation.

6.2 These filtering primitives are both commonly used and often slow. When with or related primitives are used to filter a large agentset – e.g., all the turtles or patches in a large model – they must perform the calculations necessary to evaluate the subsetting criterion for each member of the agentset. When we look for ways to speed up a NetLogo code, we often start (after using the profiler to identify the slow procedures) by searching for statements that use `with`. We have experienced numerous models in which just reprogramming such statements has dramatically decreased execution time, often by several orders of magnitude.

6.3 The following subsections describe ways to make filtering statements faster or to avoid them.

By using global agentsets

6.4 When a subset of patches or turtles is used repeatedly in a model, it can be saved as a global variable instead of being re-created many times via `with` or other filtering primitives. For example, the coffee farm model of Railsback & Johnson (2011, 2014) represents land uses via patch colors: forest is green, shade-grown coffee is grey, sun-grown coffee is yellow, etc. The model often uses information for just one land use type, for example the number of birds in shade-grown coffee patches. Such information could be obtained via `count turtles-on patches with [pcolor = grey]`. It is far faster, though, to create global variables that each contain the agentset of patches of each land use type. These variables are initialized in the setup procedure via statements such as:

```
set forest-patches patches with [pcolor = green]
set shade-coffee-patches patches with [pcolor = grey]
set sun-coffee-patches patches with [pcolor = yellow]
```

6.5 Then the statement to count birds on shade coffee becomes `count turtles-on shade-coffee-patches`, which does not require NetLogo to look at all patches and identify the ones with grey color and, therefore, is far faster.

6.6 This global variable technique is especially efficient in the coffee farm model because the land use types (patch colors) do not change during a simulation. However, it can still be very efficient even if the agentsets contained in the global variables do change, which requires updating the global-variable agentsets. If the model represented conversion of forest land to coffee production (some patches have their color changed from green to yellow), then the global variables would need to be updated. Usually this would be done by simply repeating the statements used to create them in the first place. As discussed below, this kind of change can be error-prone and should be tested carefully.

6.7 Our example model "With-vs-global-vars.nlogo" illustrates the use of global variables to avoid filtering primitives. With the NetLogo view updates set to "on ticks", the version using patches `with [pcolor = yellow]` executed 100 ticks in 25 seconds. The version using a global variable for yellow patches executed in 4.7 seconds, a 5X decrease in execution time. With view updates turned off, the difference was much more dramatic, a decrease of almost 300X in execution time.

By using agentsets as patch or turtle variables

6.8 This technique is similar to the use of global variables to hold agentsets, but works when each patch (or turtle) needs to use an agentset that differs among patches (or turtles) but does not change over time (or changes rarely, compared to how often the agentset is used). In our example model "With-vs-agent-vars.nlogo", turtles move each tick to a randomly chosen patch that is green and within a radius of 10 patches. The natural way for

a turtle to identify the potential movement destinations is via the statement `patches in-radius 10 with [pcolor = green]`. This statement, however, requires NetLogo to examine many patches to determine if the radius and color criteria are met. (`in-radius` is cleverly designed to examine only the patches within a square defined by the radius.) Instead, during model setup, patches can be given a variable `green-destinations` that is initialized via:

```
ask patches
    [set green-destinations patches in-radius 10 with [pcolor = green]]
```

6.9 Then turtles on blue patches would use `green-destinations` as their set of potential destinations. In the example model, this change decreased execution time by 1.4X with the view updated on ticks and by 49X with view updates off.

6.10 Another example is the wild dog model described in Sect. 16.4.2 of Railsback & Grimm (2012). In this model, "packs" are a NetLogo "breed"; each represents a group of dogs. A pack could refer to the dogs that belong to it via `dogs with [my-pack = myself]`, where `my-pack` is a dog variable that identifies its pack; but it is faster for each pack to have a variable `pack-members` that is an agentset of all the dogs belonging to it. This agentset is updated when new dogs are born or older dogs leave the pack. (The alternative of modeling the dog-pack relation using links is discussed below.)

By using local agentsets

6.11 When filtering primitives like with are used more than once within a procedure to obtain the same subset, it is often much faster to create a local variable that holds the subset. A common example is when we ask a subset of turtles or patches to do something, but that subset can sometimes have no members. In our example model "With-vs-local-vars.nlogo", red patches are rare. When we ask turtles to do this:

```
move-to one-of patches in-radius 20 with [pcolor = red]
```

6.12 we will get a run-time error whenever there are no red patches within a radius of 20. A solution is to tell the turtles to do something else (move to one of the green patches, which are common) if there are no red patches:

```
ifelse any? patches in-radius 20 with [pcolor = red]
[
  move-to one-of patches in-radius 20 with [pcolor = red]
]
[
  move-to one-of patches in-radius 10 with [pcolor = green]
]
```

6.13 However, a (usually) faster solution avoids the second use of with by instead creating a local agentset:

```
let red-destinations patches in-radius 20 with [pcolor = red]
ifelse any? red-destinations
[
  move-to one-of red-destinations
]
[
  move-to one-of patches in-radius 10 with [pcolor = green]
]
```

6.14 In the example model, this technique reduced execution time from 36 seconds to 23 seconds for 100 ticks (view updating had little effect).

6.15 To use this technique, search slow procedures for any filtering statements that are used more than once to produce exactly the same subset of agents. (But be aware that the same statement may not produce the same results later if the agents have changed.) If such statements are found, use let to introduce a new local variable that holds the subset of agents, and then use that local variable instead of repeating the filtering statement. (But also consider whether a global or turtle/patch variable might be more appropriate.)

By using the table extension

6.16 In some models, each agent has a unique value of some variable, and we want to find the agent that has a specific value of that variable. For example, the frog model of Railsback et al. (2016) uses a unique patch variable cell-number: each patch in the model space has its own value of cell-number. Input data for such patches are referenced by cell number instead of by patch coordinates: e.g., the input data lists the cell number and ground elevation of each patch. Therefore, when setting up the patches from the input data file, we need to find the patch that has each cell number so we can set its elevation. The natural way to do this in NetLogo is:

```
ask patches with [cell-number = the-input-cell-number]
     [set elevation the-input-elevation]
```

where `the-input-cell-number` and `the-input-elevation` are local variables with values read from the input file.

6.17 In cases like this when it is necessary to find the patch (or turtle) with a specific value of some patch (or turtle) variable, and that variable is unique (no two agents have the same value of it) and static, it can be much faster to use NetLogo's table extension instead of the `with` primitive. In our example with cell numbers, the table extension can be used to make a table linking cell numbers with patches. If we create a table (called `cell-patch-table`) with cell numbers as keys, finding the patch that has a cell number equal to the variable `the-input-cell-number` then becomes `table:get cell-patch-table the-input-cell-number`. In the frog model, reading habitat variables from a file into each of 42,121 patches took 98 seconds using patches with and only 2.3 seconds using the table extension.

Avoid Unnecessary Re-Initialization

7.1 Complex NetLogo models that have especially slow setup procedures can sometimes be programmed in a way that allows multiple model runs and BehaviorSpace experiments to be conducted without repeating the slow parts of setup and, as an additional benefit, allows BehaviorSpace to control global variables that are not on the Interface. The frog model of Railsback et al. (2016) is an example: setting up the model's World requires reading in over 60 variables and building two lookup tables for each of over 40,000 patches. However, these habitat variables and lookup tables are static and do not change during or among simulations.

7.2 It is very important to realize, before implementing this technique, that it does not allow BehaviorSpace experiments to run on multiple processors: each experiment must run on only one processor. Multiple processors can be used simultaneously only by opening multiple copies of the model in separate instances of NetLogo.

7.3 The speedup technique is to separate model initialization into two stages that contain the procedures that do and do not need to be repeated to re-run the model after the initial setup. Here, we refer to `setup` as the procedure that is used first to create the world and completely initialize the model, and reset as a procedure that re-initializes only those parts of the model that need to be reset between model runs. The trick is for `reset` to clean up and re-initialize things like turtles without using the primitive `clear-all`, which erases everything. The `setup` procedure can be organized like this:

```
to setup  ; a global procedure to fully initialize the model, used once.
  clear-all
  set-parameters ; the procedure that sets all global variable values
  build-world ; the procedure that reads in patch variables and sets up the World
  reset  ; the procedure that re-initializes the model between runs
end
```

7.4 The reset procedure can then be organized like this:

```
to reset ; a global procedure to re-initialize the model between runs
  ; check to make sure setup has been executed
  if some-global-parameter  = 0
    [ error "You ran reset before running setup!" ]
  ; manually clean up without clearing the World
  ask turtles [ die ]
  clear-output
  clear-all-plots
  reset-ticks
```

```
  ; now create turtles and do everything else needed to initialize the model
  ; including re-setting non-static global variables
  ...
end
```

7.5 With this organization, `setup` must be executed once at the very beginning, but new model runs can be initialized by only running `reset`. BehaviorSpace experiments can use reset as the "setup commands", as long as setup is manually executed before the experiment is started and only one processor is used.

7.6 When model initialization is organized this way (but not in the standard way, with all initialization in setup, BehaviorSpace experiments can safely vary parameters that are not on the Interface. BehaviorSpace changes parameter values before it executes its "setup commands", so with a standard setup organization the value given to a parameter by BehaviorSpace would be overwritten if the same parameter was also set in setup. With the reset organization shown here, global parameter values are initialized during setup, not in the reset procedure that we tell BehaviorSpace to use as its setup command. Therefore, parameter values set by BehaviorSpace are not overwritten.

Use State Variables Instead of Links

8.1 NetLogo's "links" are objects that track a relationship between two turtles, often used to represent networks: a turtle can use primitives such as `my-links` to identify and interact with other turtles it is linked to. However, the same link relationship can be modeled using turtle state variables instead of links: a turtle can have a state variable that is simply an agentset of the other turtles it is linked to. Such state variables must be updated by turtle procedures that are typically more complex and error-prone than simply creating or removing a link.

8.2 The wild dog model code in Sect. 16.4.2 of Railsback & Grimm (2012) uses this state variable approach. For example, each dog has a state variable for the pack it belongs to, and each pack has a variable containing an agentset of all the dogs that belong to it. As dogs leave their original pack and create new packs, the code must carefully update these variables. We have re-implemented this model using links instead of state variables: each pack creates a link between itself and each dog that belongs to it, and dogs that leave a pack destroy their link to it. The code using links is simpler and less error-prone than the original. However, the version using NetLogo links takes over six times longer to execute. (This model contains little other than managing links among dogs and their packs, so the cost of creating and destroying links dominates its execution speed.)

8.3 This experience indicates that models in which many links are created and destroyed could potentially be sped up considerably by using state variables instead of links. The state-variable approach is more error-prone, so we recommend such models first be built using the simpler link approach and then, if necessary, converted to the state-variable approach and carefully tested against the original code.

Try Alternative Statements

9.1 There are often more than one way to code a particular function in NetLogo, and sometimes one way is substantially faster than another. One common example is an agent (turtle or patch) searching for other agents within some distance of itself. The most natural way to program this search is using the `in-radius` primitive, e.g., `turtles in-radius search-radius`. However, the same search can also be coded as `turtles with [distance myself <= search-radius]`. In our example model "InRadius-vs-DistanceMyself.nlogo", we found `in-radius` to outperform `distance myself` when the radius was small and the number of agents being searched large, while `distance myself` was much faster when the search radius was larger. The trout model of Ayllón et al. (2016), with approximately 1000 trout agents searching a few hundred patches, ran approximately four times faster using the `distance myself` approach, even though this search is only a small part of the trout behavior each tick. (The speed difference between `in-radius` and `distance myself` has been discussed on the NetLogo user forum, with some users finding extremely large differences in execution speed between the two alternatives.)

9.2 Another example of speed differing among statements that produce exactly the same results is the order in which boolean conditions appear in logical statements using and and or. For example, the statements `turtles with [(distance myself < 20) and (color = red)]` and `turtles with [(color = red) and (distance myself < 20)]` produce exactly the same subset, but one may be faster than the other, depending on

how many turtles there are, where they are, and how many are red. The differences between these statements in execution time are because they change whether NetLogo (a) first excludes turtles beyond a radius of 20 and then checks the color of the remaining ones, or (b) first excludes all the non-red turtles and then checks the distance of the red ones.

9.3 Differences in execution speed among alternative statements that perform the same function are difficult to predict. Our advice is that if profiling shows a procedure to be particularly slow, then simply experiment with alternative code statements to see if they make a difference.

Efficient Model Design: Discrete Event Simulation as an Alternative to Time Steps

10.1 Typical NetLogo models use time step simulation, in which all model actions are called from the "go" procedure, which is executed once each tick. In some models, many agents do nothing on many ticks, so the code must include conditional statements that determine whether or not each agent should execute some procedure on the current tick. For example, a model of vehicle traffic must use a very short time step (a few seconds or minutes) to capture the movement of vehicles being driven. But at any time, most vehicles are parked and not moving at all; hence, the NetLogo code must ask each vehicle whether it is currently moving and then decide whether it should do anything. Over many agents and many ticks, just checking which agents should do something can become a major computational burden.

10.2 Discrete event simulation (DES) is an alternative to the time step method of organizing how events are scheduled in a simulation. DES is widely used (there are many books and specialized software platforms for DES), but not supported well by NetLogo. With DES, there is a global schedule that keeps track of simulated time, and events are added to the schedule with a specific time to be executed. Each event consists of a particular agent or set of agents, the procedure they are to execute, and the simulated time at which they are to execute it. The modeler can think in more natural terms about what procedures the agents should execute when (e.g. "have agent X execute procedure Y at time Z", or "have agent X execute procedure Y at Z minutes from now"). The scheduler then executes the appropriate procedures on the appropriate agents at the appropriate time in chronological order. There is no longer a tick-by-tick progression in the model; instead, actions are executed according to the schedule and the "dead" time between successive actions is skipped. (If convenient, some actions can happen at regular ticks while others are scheduled at times on or between ticks.) DES is therefore most useful for models where agents spend a lot of time idle despite it being knowable when they need to act next. With DES, each action is performed only when needed, with no conditional testing and very little overhead.

10.3 A second benefit of DES is that it avoids complications associated with the assumption of time step simulation that multiple events happen simultaneously once per tick. For example, if agents are competing for a resource (e.g., parking places), the order in which they arrive at a location may matter very much. Typically, if multiple agents attempt to take an action or access a resource at the same tick, careful attention must be paid to the ordering to avoid artificial bias (which is why the ask primitive randomizes the order in which agents execute an action). With DES, events can happen at any time, not just at time steps, making the order of events purely chronological and more an outcome of the model than an artifact of the simulation style. In some cases, this leads to more natural logical flow of the model and can avoid difficult-to-diagnose bugs or biases in results.

10.4 The time extension for NetLogo (https://github.com/colinsheppard/time/); available via the "Extensions" link from the NetLogo home page) was designed in part to facilitate DES in NetLogo programs. It includes primitives to link NetLogo's ticks to a specific time interval (i.e., so each tick represents one day, or one week, or 3.7 seconds), to schedule actions (as NetLogo "tasks") at specific simulation times or to be repeated whenever a specific amount of time has passed, and to label output in time units.

10.5 If a NetLogo model addresses a problem for which DES seems more natural and efficient than conventional time-step simulation, then it can be designed from the start to use the time extension to schedule actions instead of (or in addition to) the standard "tick" approach. A model of automobile use in Delhi, India, was implemented using DES in NetLogo with the time extension (Sheppard et al. 2016b,a), and was several orders of magnitude faster than a pilot version using standard time steps.

Efficient Execution of Simulation Experiments

11.1 The goal of model design, implementation, testing, and the previous speed-up steps is to reach the point where

the model is ready to be analyzed and applied to a scientific question. Typical analyses include parameterization (calibration), sensitivity and uncertainty analyses, and analysis of scenarios that address research questions (Railsback & Grimm 2012). These analyses typically require simulation experiments that use many – often, hundreds or thousands – of model runs. Without efficient ways to implement such experiments, analysis of large and complex models can be impractical.

11.2 One key to making standard kinds of model analysis practical is to use efficient analysis designs, which are widely discussed in the simulation literature. The "BehaviorSearch" tool (Stonedahl & Wilensky 2013) includes several sophisticated algorithms (e.g., genetic algorithms and simulated annealing) for fitting parameter values in NetLogo models; BehaviorSearch should be an easy and efficient approach to parameterization for many complex NetLogo models. Thiele et al. (2014) make several other well-known techniques for reducing the computational burden of model analysis available via RNetLogo (discussed below). Those techniques include Latin hypercube sampling for uncertainty analysis and the "Morris screening" approach to sensitivity analysis. We strongly recommend that NetLogo users find and use such efficient analysis methods but do not discuss them further here.

11.3 Instead, we discuss two ways of executing simulation experiments and computational techniques for making them more efficient.

Executing BehaviorSpace Experiments on a High-Performance Cluster

12.1 NetLogo's BehaviorSpace tool is designed to automate the setup and execution of simulation experiments. BehaviorSpace can perform multiple model runs in parallel, with the number of parallel runs up to or even exceeding the number of processor cores in the computer. Therefore, the execution time for a large BehaviorSpace experiment is usually limited by how many cores are available.

12.2 The use of a High-Performance Computing (HPC) cluster, typically a large massively-parallel computing system with many processor cores and a shared storage system connected together via a fast network, allows BehaviorSpace to execute large numbers of model runs simultaneously and makes extensive simulation experiments feasible even for very slow models. In one example, Ayllón et al. (2016) report parameterization and sensitivity analysis experiments that each required thousands of runs of a model that can take hours to days per run, made possible via HPC and BehaviorSpace. In another example, Sheppard et al. (2016b,a) used HPC to run their vehicle model millions of times to optimize electric vehicle charging station locations in Delhi. Many universities and research laboratories have HPC clusters, and some commercial "cloud computing" services offer free trial access to clusters.

12.3 Using NetLogo on a HPC cluster is not necessarily difficult, especially if administrative support is available. Doing so requires installing NetLogo and Java Runtime Environment so they are accessible to all nodes; and a batch file that specifies the path to the NetLogo and Java directories, the path to and name of the NetLogo file, and the BehaviorSpace experiment to run and its options (e.g., output file name and format). (It is important to use the "table" output format because the alternative "spreadsheet" format stores all results in memory and thus can consume all available memory.) In a HPC cluster, BehaviorSpace experiments are run in "headless" mode, that is, without any graphical user interface. Documentation and tools for using NetLogo headless and on an HPC cluster are available in the BehaviorSpace section of the NetLogo User Manual and by searching on-line.

Efficient Use of RNetLogo Model Analysis: Design of Experiments

13.1 RNetLogo is a package for running NetLogo models from the popular R statistical software (Thiele et al. 2012, 2014). RNetLogo can, for example, send NetLogo a set of input values and then receive and analyze model results; this allows R programs to set up, execute, and analyze NetLogo simulation experiments. Here, we provide a few tips for making the link between R and NetLogo computationally efficient.

13.2 First, set the Java options in R to values that configure the Java Runtime Environment most efficiently for running NetLogo, e.g., by making adequate memory available. These options are set via a statement such as (the exact statement may depend on the operating system):

```
options(java.parameters=c(" server"," Xmx1300m"))
```

13.3 Second, keep in mind that R is a vector-oriented language and handles vectors more efficiently than individual values. Therefore, using RNetLogo in a way that uses vector operations to send data to and from R will be much faster and more stable than mass calls with single values.

13.4 When using the RNetLogo commands `NLGetAgentSet` and `NLGetAgentPatches`, if possible use the list return option via `as.data.frame=FALSE` in combination with `agents.by.row=TRUE` or `patches.by.row=TRUE`, instead of receiving the data from NetLogo as a data frame.

13.5 Finally, keep in mind that R programs can be executed in parallel by using, for example, the "parallel" package. This capability allows parallel execution of RNetLogo experiments similar to that discussed above for BehaviorSpace experiments.

Conclusions

14.1 NetLogo is widely recognized as an efficient platform for agent-based simulation, in the sense that it allows modelers, including both beginners and experienced ones, to move rapidly through the design, programming, and testing stages and on to using models for analysis and developing scientific understanding. However, NetLogo is not widely recognized as an efficient platform in the sense of providing low (or at least reasonable) execution times for working models. Potential users who hear NetLogo's reputation as being too slow or limited for big models, or who perhaps look only at the very simple examples that (understandably) dominate NetLogo's built-in models library, may be discouraged from selecting NetLogo as a platform for large scientific models. Choosing another platform, especially those requiring programming in a base language such as C++ or Java, comes at a high cost: programming will take much longer, mistakes will be harder to find and hence more expensive, the tools necessary for testing and understanding models (graphical and interactive displays, experiment managers like BehaviorSpace) must be developed, and – unless the code is designed as cleverly as NetLogo's primitives appear to be – the result may turn out to actually be slower than NetLogo.

14.2 There is now sufficient evidence that NetLogo is neither inherently slow nor incapable of handling large and complex models. Our previous experience comparing ABM platforms (Railsback et al. 2006; Lytinen & Railsback 2012), while confirming the general understanding in computer science that it is not simple or straightforward to say which platform or programming languages are faster or slower, indicates that NetLogo is not dramatically slower at executing models than other popular platforms are. Perhaps the best evidence that NetLogo is suitable for large scientific models is that many such models have now been successfully implemented and analyzed extensively in NetLogo (Table 1).

Model and citation	Computational challenges	Largest simulation experiments (approximate number of model runs per experiment)	Techniques used to increase execution and analysis speed
InSTREAM-GEN trout model (Ayllón et al. 2016)	Up to 30,000 agents make complex calculations for each of many habitat patches, daily for 100-year runs	Calibration (2000), global sensitivity analysis (14,000)	Using `distance myself` instead of `in-radius`; HPC; Latin hypercube sampling in calibration; Morris screening and Sobol's variance decomposition method in sensitivity analysis
Coffee farm model (Railsback & Johnson 2011, 2014)	~ 1800 agents selecting habitat among 40,000 patches, up to 720 times per day	Parameter uncertainty analyses (2500)	Global agentsets for habitat patch types; parallel execution via BehaviorSpace
Frog breeding model (Railsback et al. 2016)	>300,000 habitat patches, >100,000 agents	Parameter sensitivity analyses (2500)	Table extension to relate cell numbers to patches; `reset` procedure to minimize re-initialization; local agentsets to reduce use of `with`
Electric vehicle charging station model (Sheppard et al. 2016b,a)	Simulating movement of 10,000 vehicles on 1000s of km of road continuously through a day	Optimization analyses (3000)	Discrete event simulation; HCP
Savanna model (unpublished reimplementation of model by Jeltsch et al. (1996)	Dispersal of $\sim 50,000$ seeds within a limited radius		Using patch variables containing an agentset of other patches within the dispersal radius

Table 1: Examples of large models successfully implemented in NetLogo

14.3 There is actually at least one good reason to prefer NetLogo for large models: the set of tools for running simulation experiments efficiently. The convenience of using BehaviorSpace or RNetLogo to run simulations in parallel, on either a desktop computer or HPC cluster, means that large simulation experiments can be underway and finished in less time than they would be in many other platforms. Efficient model analysis packages such as BehaviorSearch (Stonedahl & Wilensky 2013) and the RNetLogo recipes of Thiele et al. (2014) can let parameterization and sensitivity analysis be completed with fewer model runs and far less effort than required for platforms lacking such tools.

14.4 Our overall advice to beginners in implementing ABMs in NetLogo is to not worry about execution speed while writing and testing the program, but to then consider the strategy and techniques provided in this article before running analyses that use possibly thousands of simulations. The initial focus in NetLogo programming should be on writing code that is easy to understand and test. Then, with experience, NetLogo users can quickly learn to avoid inefficient programming. But even experienced NetLogo programmers continue to find new ways of increasing execution speed, making it valuable to check the on-line resources for NetLogo programmers, including NetLogo's User Group, Stack Overflow, and Github sites, for solutions to specific speed issues. The reason for this is that most speed issues are context-dependent: the solutions we describe here are often, but not necessarily always, efficient; some techniques help in some situations and hurt in others; and unique models are likely to have execution limitations unlike those of other models. Therefore, the most important thing to learn is to use the NetLogo timer and profiler extension to carefully check where a program is slow and whether modifications, often trial and error, actually make it faster.

Acknowledgements

Many of the methods presented here were discovered in the short courses we teach at Technische Universität Dresden and Humboldt State University; we thank the course participants who inspired and contributed to their development. We also thank three helpful reviewers.

References

Ayllón, D., Railsback, S. F., Vincenzi, S., Groeneveld, J., Almodóvar, A. & Grimm, V. (2016). InSTREAM-Gen: Modelling eco-evolutionary dynamics of trout populations under anthropogenic environmental change. *Ecological Modelling*, *326*, 36–53

Bouquet, F., Chipeaux, S., Lang, C., Marilleau, N., Nicod, J.-M. & Taillandier, P. (2015). Introduction to the agent approach. In B. A., C. Lang & N. Marilleau (Eds.), *Agent-based Spatial Simulation with NetLogo. Volume 1. Introduction and Bases*, (pp. 1–28). London: ISTE Press

CCL (Center for Connected Learning, Northwestern University) (2016). *NetLogo compiler architecture*

Jeltsch, F., Milton, S. J., Dean, W. & van Rooyen, N. (1996). Tree spacing and coexistence in semiarid savannas. *Journal of Ecology*, (pp. 583–595)

Lammoglia, A., Josselin, D. & Marilleau, N. (2015). Some propositions to find optimal conditions to simulate a flexible transport using an agent-based model. *Cybergeo: European Journal of Geography*

Lytinen, S. L. & Railsback, S. F. (2012). The evolution of agent-based simulation platforms: A review of NetLogo 5.0 and ReLogo. In *Proceedings of the fourth international symposium on agent-based modeling and simulation*. Citeseer

Ozik, J., Collier, N. T., Murphy, J. T. & North, M. J. (2013). The ReLogo agent-based modeling language. In *Simulation Conference (WSC), 2013 Winter*, (pp. 1560–1568). IEEE

Railsback, S. F. & Grimm, V. (2012). *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton, NJ: Princeton University Press

Railsback, S. F., Harvey, B. C., Kupferberg, S. J., Lang, M. M., McBain, S. & Welsh Jr, H. H. (2016). Modeling potential river management conflicts between frogs and salmonids. *Canadian Journal of Fisheries and Aquatic Sciences*, *73*(5), 773–784

Railsback, S. F. & Johnson, M. D. (2011). Pattern-oriented modeling of bird foraging and pest control in coffee farms. *Ecological Modelling*, *222*(18), 3305–3319

Railsback, S. F. & Johnson, M. D. (2014). Effects of land use on bird populations and pest control services on coffee farms. *Proceedings of the National Academy of Sciences*, *111*(16), 6109–6114

Railsback, S. F., Lytinen, S. L. & Jackson, S. K. (2006). Agent-based simulation platforms: Review and development recommendations. *Simulation*, *82*(9), 609–623

Sheppard, C. J., Gopal, A. R., Harris, A. & Jacobson, A. (2016a). Cost-effective electric vehicle charging infrastructure siting for Delhi. *Environmental Research Letters*, *11*(6), 064010

Sheppard, C. J., Harris, A. & Gopal, A. R. (2016b). Cost-effective siting of electric vehicle charging infrastructure with agent-based modeling. *IEEE Transactions on Transportation Electrification*, *2*(2), 174–189

Sklar, E. (2007). Netlogo, a multi-agent simulation environment. *Artificial Life*, *13*(3), 303–311

Sondahl, F., Tisue, S. & Wilensky, U. (2006). Breeding faster turtles: Progress towards a NetLogo compiler. In *Proceedings of the Agent 2006 conference on social agents, Chicago, IL*

Stonedahl, F. & Wilensky, U. (2013). *BehaviorSearch*

Thiele, J. C., Kurth, W. & Grimm, V. (2012). RNetLogo: An R package for running and exploring individual-based models implemented in NetLogo. *Methods in Ecology and Evolution*, *3*(3), 480–483

Thiele, J. C., Kurth, W. & Grimm, V. (2014). Facilitating parameter estimation and sensitivity analysis of agent-based models: A cookbook using NetLogo and R. *Journal of Artificial Societies and Social Simulation*, *17*(3), 11

Tisue, S. & Wilensky, U. (2004). NetLogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*, vol. 2004, (pp. 7–9). Chicago, IL

Wikipedia (2016). *Wikipedia*

Wilensky, U. (1999). *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University

Wilensky, U. & Rand, W. (2015). *An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo*. Cambridge, MA: MIT Press