The NetLogo 5.3.1 User Manual

Table of Contents

What is NetLogo?

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It was authored by Uri Wilensky in 1999 and has been in continuous development ever since at the Center for Connected Learning and Computer-Based Modeling.

NetLogo is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of "agents" all operating independently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from their interaction.

NetLogo lets students open simulations and "play" with them, exploring their behavior under various conditions. It is also an authoring environment which enables students, teachers and curriculum developers to create their own models. NetLogo is simple enough for students and teachers, yet advanced enough to serve as a powerful tool for researchers in many fields.

NetLogo has extensive documentation and tutorials. It also comes with the Models Library, a large collection of pre-written simulations that can be used and modified. These simulations address content areas in the natural and social sciences including biology and medicine, physics and chemistry, mathematics and computer science, and economics and social psychology. Several model-based inquiry curricula using NetLogo are available and more are under development.

NetLogo can also power a classroom participatory-simulation tool called HubNet. Through the use of networked computers or handheld devices such as Texas Instruments graphing calculators, each student can control an agent in a simulation. Follow this link for more information.

NetLogo is the next generation of the series of multi-agent modeling languages including StarLogo and StarLogoT. NetLogo runs on the Java virtual machine, so it works on all major platforms (Mac, Windows, Linux, et al). It is run as a desktop application. Command line operation is also supported.

Features

- System:

- Free, [open source](#)
- Cross-platform: runs on Mac, Windows, Linux, et al
- International character set support
- Programming:
  - Fully programmable
  - Approachable syntax
  - Language is Logo dialect extended to support agents
  - Mobile agents (turtles) move over a grid of stationary agents (patches)
  - Link agents connect turtles to make networks, graphs, and aggregates
  - Large vocabulary of built-in language primitives
  - Double precision floating point math
  - First-class function values (aka tasks, closures, lambda)
  - Runs are reproducible cross-platform
- Environment:
  - Command center for on-the-fly interaction
  - Interface builder w/ buttons, sliders, switches, choosers, monitors, text boxes, notes, output area
  - Info tab for annotating your model with formatted text and images
  - HubNet: participatory simulations using networked devices
  - Agent monitors for inspecting and controlling agents
  - Export and import functions (export data, save and restore state of model, make a movie)
  - BehaviorSpace, an open source tool used to collect data from multiple parallel runs of a model
  - System Dynamics Modeler
  - NetLogo 3D for modeling 3D worlds
  - Headless mode allows doing batch runs from the command line
- Display and visualization:
  - Line, bar, and scatter plots
  - Speed slider lets you fast forward your model or see it in slow motion
  - View your model in either 2D or 3D
  - Scalable and rotatable vector shapes
  - Turtle and patch labels
- APIs:
  - controlling API allows embedding NetLogo in a script or application
  - extensions API allows adding new commands and reporters to the NetLogo language; open source example extensions are included

Copyright and License Information

How to reference

If you use or refer to NetLogo in a publication, we ask that you cite it. The correct citation is: Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

For HubNet, cite: Wilensky, U. & Stroup, W., 1999. HubNet. http://ccl.northwestern.edu/netlogo/hubnet.html. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

For models in the Models Library, the correct citation is included in the "Credits and References" section of each model's Info tab.

Acknowledgments

NetLogo license

Commercial licenses

Commercial licenses are also available. To inquire about commercial licenses, please contact Uri Wilensky at uri@northwestern.edu.

NetLogo User Manual license

The NetLogo User Manual by

Open source

The NetLogo source code is hosted at https://github.com/NetLogo/NetLogo. Contributions from interested users are welcome.

Third party licenses

Scala

Much of NetLogo is written in the Scala language and uses the Scala standard libraries. The license for Scala is as follows:

```
Copyright (c) 2002-2011 EPFL, Lausanne, unless otherwise specified. All rights
reserved.

This software was developed by the Programming Methods Laboratory of the Swiss
Federal Institute of Technology (EPFL), Lausanne, Switzerland.

Permission to use, copy, modify, and distribute this software in source or
binary form for any purpose with or without fee is hereby granted, provided
that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice,
     this list of conditions and the following disclaimer.
  2. Redistributions in binary form must reproduce the above copyright notice,
     this list of conditions and the following disclaimer in the documentation
     and/or other materials provided with the distribution.
  3. Neither the name of the EPFL nor the names of its contributors may be
     used to endorse or promote products derived from this software without
     specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

MersenneTwisterFast

For random number generation, NetLogo uses the MersenneTwisterFast class by Sean Luke. The copyright for that code is as follows:

```
Copyright (c) 2003 by Sean Luke.
Portions copyright (c) 1993 by Michael Lecuyer.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
```

Colt

Parts of NetLogo (specifically, the random-gamma primitive) are based on code from the Colt library (http://acs.lbl.gov/~hoschek/colt/). The copyright for that code is as follows:

Quaqua

NetLogo uses the Quaqua Look and Feel library, which is Copyright (c) 2003-2005 Werner Randelshofer, http://www.randelshofer.ch/, werner.randelshofer@bluewin.ch, All Rights Reserved. The library is covered by the GNU LGPL (Lesser General Public License). The text of that license is included in the "docs" folder which accompanies the NetLogo download, and is also available from http://www.gnu.org/copyleft/lesser.html .

JHotDraw

For the system dynamics modeler, NetLogo uses the JHotDraw library, which is Copyright (c) 1996, 1997 by IFA Informatik and Erich Gamma. The library is covered by the GNU LGPL (Lesser General Public License). The text of that license is included in the "docs" folder which accompanies the NetLogo download, and is also available from http://www.gnu.org/copyleft/lesser.html .

MovieEncoder

For movie-making, NetLogo uses code adapted from sim.util.media.MovieEncoder.java by Sean Luke, distributed under the MASON Open Source License. The copyright for that code is as follows:

JpegImagesToMovie

For movie-making, NetLogo uses code adapted from JpegImagesToMovie.java by Sun
Microsystems. The copyright for that code is as follows:

JOGL

For 3D graphics rendering, NetLogo uses JOGL, a Java API for OpenGL, and Gluegen, an
automatic code generation tool. For more information about JOGL and Gluegen, see
jogamp.org/. Both libraries are distributed under the BSD license:

This software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.

ASM

For Java bytecode generation, NetLogo uses the ASM library. It is distributed under the following license:

Copyright (c) 2000-2010 INRIA, France Telecom. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Log4j

For logging, NetLogo uses the Log4j library. The copyright and license for the library are as follows:

Copyright 2007 The Apache Software Foundation

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

PicoContainer

For dependency injection, NetLogo uses the PicoContainer library. The copyright and license for the library are as follows:

Copyright (c) 2003-2006, PicoContainer Organization All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the PicoContainer Organization nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY

Pegdown and Parboiled

For the Info tab, NetLogo uses the Pegdown and Parboiled libraries.

The copyright and license for Pegdown are as follows:

The copyright and license for Parboiled are as follows:

What's New?

The following is a condensed history. Detailed release notes are on GitHub.

For help running models made in old versions, see the Transition Guide.

Version 5.3.1 (February 2016)

Feature Changes

- Mathematica Link is now included and has been tested to work with Mathematica 10.
- A link to Introduction to Agent-Based Modelling has been added to the "Help" menu.

Extension Changes

- The gogo extension now prompts the user for the location of Java upon opening. This version of Java is used to launch the gogo hid daemon.
- The correct version of the network extension is now bundled, which will open properly
- Bundles the cf extension, which adds match, case, and select primitives.

Bugfixes

- Corrects a bug where turtles wrapping around a torus-shaped world with pen down would sometimes cause NetLogo to loop infinitely.
- Fixes a bug where buttons would appear to remain pressed after a right click on Mac.
- Fixes a bug where pressing the right mouse button while dragging would confuse the mouse-down? primitive.
- Fixes agent type-checking of tasks (bug appeared in 5.2.1).
- link-neighbor? now returns true if and only if the neighbor is connected through an undirected link.
- Documentation fixes for my-links and mean primitives.

Version 5.3 (December 2015)

Feature Changes

- Java 8 is now bundled with all versions of NetLogo, this removes the need for a separate Java 6 installation on Mac OS X
- Separate 32-bit and 64-bit versions are available for both Windows and Linux
- The Windows installer is now an msi instead of an exe
- javax.media.opengl is no longer supported in Java 8, it has been replaced by com.jogamp.opengl
- Mathematica Link is not distributed due to Java version changes

Extension Changes

- A minor update to the `nw` extension makes `nw:weighted-path-to` behave as documented.

- Most extensions should continue to work without change unless they rely on javax.media.opengl, which was renamed in the updated version of JOGL.

Version 5.2.1 (September 2015)

Extensions

- An included Arduino extension for use with Arduino boards

New features

- New file menu item to export models to NetLogo Web

Bug fixes

- BehaviorSpace output type preference is now remembered
- Output widget font is now saved at the proper size when zoomed
- Reporter tasks are now evaluated in variable context
- `runresult` arguments now only get evaluated once
- The last used directory is now remembered on Linux
- Whitespace is now stripped from models when saving

- New Sample Models: Artificial Anasazi, Bacteria Food Hunt, Bacteria Hunt Speeds, BeeSmart - Hive Finding, Bug Hunt Disruptions, Bug Hunt Environmental Changes, Bug Hunt Predators and Invasive Species - Two Regions, Hydrogen Diffusion 3D, Lennard-Jones, Paths
- There are 46 new models in the new IABM Textbook folder: Turtles Circling Simple, Ants Simple, Heroes and Cowards, Life Simple, Simple Economy, 4 DLA extensions, 4 El Farol Extensions, 4 Fire Extensions, 4 Segregation Extensions, 5 Wolf Sheep Extensions, Agentset Efficiency, Agentset Ordering, Communication-T-T Network Example, Preferential Attachment Simple, Random Network, Traffic Basic Adaptive, Traffic Basic Utility, Traffic Grid Goal, Spread of Disease, Voting Component Verification, Voting Sensitivity Analysis, Arduino Example, Disease With Android Avoidance HubNet, Example HubNet, Run Example, Run Result Example, Simple Machine Learning, Simple Viral Marketing, Ticket Sales, Sandpile Simple
- New models in the new Alternate Visualizations folder: Ethnocentrism - Alternative Visualization, Flocking - Alternative Visualizations, Heat Diffusion - Alternative visualization, Virus - Alternative Visualization, Virus - Circle Visualization
- Promoted models: Honeycomb, Minority Game
- Many other bugfixes and upgrades

Model changes

- Improved Sample Models: Altruism, Ant Lines, Artificial Anasazi, Cooperation, Daisyworld, Divide the Cake, Heat Diffusion, Hydrogen Diffusion 3D, Lennard-Jones, N-Bodies, PD Basic Evolutionary, Sandpile, Robby the Robot, Segregation, Simple Kinetics 2 and 3, Traffic Grid, GridLock HubNet, GridLock HubNet Alternate
- Revised Curricular Models: 4 Block Stalagmites, 4 Block Two Stalagmites, Bug Hunter Competition HubNet, Fish Spotters HubNet, Ising, Tijuana Bordertowns, Urban Suite - Tijuana Bordertowns,
- Revised Code Examples: Lottery Example, self Example, Network Import Example
- Revised IABM Textbook models: Simple Economy, Fire Simple Extension 2 and 3,

Segregation Simple, 1, 2 and 3, Agentset Efficiency, Preferential Attachment Simple, Random Network, Traffic Basic Adaptive, Run Example, Traffic Grid Goal,
- New IABM Textbook model: Traffic Basic Adaptive Individuals

Version 5.2.0 (April 2015)

Extensions

- An included CSV extension to read and write CSV files
- An included Palette extension to map different values to colors
- The previous gogo extension has been removed. It has been replaced with two different gogo extensions, gogo and gogo-serial. The extensions work with different hardware. The gogo extension works with the new style HID-interface gogo boards and the gogo-serial extension works with the old-style serial-interface gogo boards
- Upgrade to the network extension including the following changes:
    - Support for many more file types, including GEXF, GDF, GML, Pajek NET, UCINET DL, and Netdraw VNA
    - Fixed bugs with GraphML support
    - Improved documentation

New features

- New primitive `netlogo-web?` added to test whether you are currently running in NetLogo Web
- New primitive `behaviorspace-experiment-name` added allowing you to get the name of the currently running experiment
- New primitive `stop-inspecting` to stop inspecting agents
- New primitive `stop-inspecting-dead-agents` and menu item to stop inspecting dead agents
- `__includes` keyword now enables the Includes button when given an empty
- hooks added for extensions to write custom log messages

Bug fixes

- `user-one-of` will now signal an error earlier when provided an empty list of choices
- `hsb`, `extract-hsb`, and `approximate-hsb` have been updated to represent true hsb conventions
- new deprecated primitives `__hsb-old`, `__extract-hsb-old`, and `__approximate-hsb-old` work as the old hsb primitives did when older models using the hsb primitives are opened in NetLogo 5.2, they will be auto-converted to use the deprecated primitives
- `extract-hsb` -- fixed bug where it didn't work correctly on rgb lists

Model changes

- New Sample Models: Artificial Anasazi, Bacteria Food Hunt, Bacteria Hunt Speeds, BeeSmart - Hive Finding, Bug Hunt Disruptions, Bug Hunt Environmental Changes, Bug Hunt Predators and Invasive Species - Two Regions, Hydrogen Diffusion 3D, Lennard-Jones, Paths
- There are 46 new models in the new IABM Textbook folder: Turtles Circling Simple, Ants Simple, Heroes and Cowards, Life Simple, Simple Economy, 4 DLA extensions, 4 El Farol Extensions, 4 Fire Extensions, 4 Segregation Extensions, 5 Wolf Sheep Extensions, Agentset Efficiency, Agentset Ordering, Communication-T-T Network

Example, Preferential Attachment Simple, Random Network, Traffic Basic Adaptive, Traffic Basic Utility, Traffic Grid Goal, Spread of Disease, Voting Component Verification, Voting Sensitivity Analysis, Arduino Example, Disease With Android Avoidance HubNet, Example HubNet, Run Example, Run Result Example, Simple Machine Learning, Simple Viral Marketing, Ticket Sales, Sandpile Simple

- New models in the new Alternate Visualizations folder: Ethnocentrism - Alternative Visualization, Flocking - Alternative Visualizations, Heat Diffusion - Alternative visualization, Virus - Alternative Visualization, Virus - Circle Visualization
- Promoted models: Honeycomb, Minority Game
- Many other bugfixes and upgrades

Version 5.1.0 (July 2014)

- bundle new network extension
- File menu includes recently opened files
- deprecate applets
- support retina display on new MacBooks

Version 5.0.4 (March 2013)

- upload models to the Modeling Commons

Version 5.0 (February 2012)

- features:
  - open source (GPL license; source code online at https://github.com/NetLogo/NetLogo)
  - new license for Sample Models and Curricular Models is Creative Commons Attribution-NonCommercial-ShareAlike
  - international characters (Unicode) supported throughout application
  - GUI is localized in Spanish, Russian, and Chinese (volunteer translators wanted)
  - rich formatting and images in Info tabs using Markdown
  - plotting code goes inside plots instead of in code tab
  - authorable model speed (target frame rate setting)
  - buttons optionally disable until ticks start
  - translucent colors in 3D view and NetLogo 3D (for RGB colors only)
- language changes:
  - "tasks" store code to be run later
    - aka first-class functions, closures, lambda
    - new primitives: `task`, `is-command-task?`, `is-reporter-task?`
    - these primitives accept tasks: `run`, `runresult`, `foreach`, `map`, `reduce`, `filter`, `n-values`, `sort-by`
  - improved overall list performance (many operations take near-constant time instead of linear time)
  - you must use `reset-ticks` to start the tick counter before using `tick` or `tick-advance`
  - new primitives `setup-plots` and `update-plots`
  - new primitive `sort-on` lets you say e.g., `sort-on [size] turtles`
  - new primitive `error` causes a runtime error
  - `random-normal` rejects negative standard deviations
- HubNet:
  - activities can run headless
  - new primitives `hubnet-clients-list`, `hubnet-kick-client`, `hubnet-kick-all-clients`

- `hubnet-set-client-interface` no longer required
- other fixes and changes:
  - tabs renamed to Interface/Info/Code
  - Command Center allows reporters, adds `show` command automatically
  - NetLogo 3D uses .nlogo3d suffix, not .nlogo
  - `import-world` is much faster
  - `startup` no longer runs headless or in background BehaviorSpace runs
  - fixed 3D rendering bug where small turtles were too bright
  - Mac app runs in 64 bit mode by default
  - upgraded Windows installer
  - GoGo extension getting-started experience now smoother on all platforms
- models:
  - new Sample Models: Sandpile, Lightning, Osmotic Pressure, Robby the Robot, Preferential Attachment 3D
  - new Curricular Models: Bug Hunt Consumers, Bug Hunt Predators and Invasive Species, Plant Speciation, epiDEM Basic, epiDEM Travel and Control, Connected Chemistry Atmosphere
  - new Code Examples: Info Tab Example, GoGoMonitorSimple

Version 4.1.3 (April 2011)

- matrix extension
- `behaviorspace-run-number`

Version 4.1 (December 2009)

- parallel BehaviorSpace
- controlling API allows embedding
- automatic code indenter
- searchable Models Library
- translucent colors
- mini-views in agent monitors
- `resize-world`, `set-patch-size`
- bitmap, QuickTime extensions
- individualized HubNet client views
- browser-based HubNet client

Version 4.0 (September 2007)

- link agents
- tick counter
- view update modes (tick-based, continuous)
- speed slider fast forwards
- input boxes in interface tab
- include multiple source files
- RGB colors
- slider bounds may be reporters
- HubNet client editor
- Mathematica-NetLogo link
- array, table, profiler, GIS extensions
- models run faster (partial compilation to JVM byte code)
- logging

Version 3.1 (April 2006)

- topologies (optional wrapping at world edges)
- randomized agentset ordering

Version 3.0 (September 2005)

- 3D view (for 2D models)
- formatted Info tabs
- System Dynamics Modeler
- `follow, ride, watch`
- drawing layer
- GoGo extension

Version 2.1 (December 2004)

- "headless" mode for command line operation
- "action keys" to trigger buttons by keypresses
- makes QuickTime movies of models
- `let, carefully`

Version 2.0.2 (August 2004)

- extensions and controlling APIs
- sound extension

Version 2.0 (December 2003)

- fast, flicker-free, non-grid-based graphics

Version 1.3 (June 2003)

- `run, runresult, map, foreach, filter, reduce`

Version 1.2 (March 2003)

- computers as HubNet clients

Version 1.1 (July 2002)

- Applets

Version 1.0 (April 2002)

- first full release (after a series of betas)

System Requirements

NetLogo runs on almost any current computer.

If you have any trouble with NetLogo not working, see Contacting Us.

Application Requirements

Windows

NetLogo runs on Windows 10, Windows 8, Windows 7 and Vista. NetLogo 5.2.1 was the last version to support Windows XP and Windows 2000.

The NetLogo installer for Windows includes Java 8 for NetLogo's private use only. Other programs on your computer are not affected.

Mac OS X

Mac OS X 10.8.3 or newer is required. (NetLogo 5.1 was the last version to support 10.5 and 10.4; NetLogo 5.2.1 was the last version to support 10.6 and 10.7)

The NetLogo application contains a distribution of the Java 8 runtime for NetLogo's private use only. Other programs on your computer will not be affected.

Linux

NetLogo should work on standard Debian-based and Red Hat-based Linux distributions. The NetLogo tarball includes a copy of the Java 8 runtime.

Start NetLogo by running the provided `NetLogo` executable.

3D Requirements

Occasionally an older, less powerful system is not able to use the 3D view or NetLogo 3D. Try it and see.

Some systems can use 3D but can't switch to full-screen mode. It depends on the graphics card or controller. (For example, the ATI Radeon IGP 345 and Intel 82845 probably will not work.)

32-bit or 64-bit?

For most users on Linux or Windows, the 32-bit version of NetLogo is the simplest way to a working NetLogo installation. Advanced users will want to understand the advantages of 64-bit NetLogo as well as how to determine whether their machine meets the requirements.

The primary advantage of the 64-bit version is the ability to add additional heap space by changing the "-Xmx" JVM option. For more information, see How big can my model be. You may also find it helpful to browse Oracle's documentation on the performance characteristics of the 64-bit JVM

To run 64-bit NetLogo, you must be running 64-bit Windows. To determine whether your version of Windows is 64-bit, see [Is my PC running the 32-bit or 64-bit version of Windows](#) if you're using Windows Vista, or Windows 7, or [Which Windows operating system am I running?](#) if you are running any other version of Windows.

For Linux users, the easiest way to determine whether your operating system is 64-bit is checking the output of

```
uname -m
```

If the output shows "x86_64" or "amd64", you should be able to run the 64-bit version.

Contacting Us

Feedback from users is essential to us in designing and improving NetLogo. We'd like to hear from you.

Web site

Our web site at ccl.northwestern.edu includes our mailing address and phone number. It also has information about our staff and our various research activities.

Feedback, questions, etc.

For help using NetLogo, try this group: http://groups.yahoo.com/group/netlogo-users/.

If you have feedback, suggestions, or questions, you may write us at feedback@ccl.northwestern.edu.

Reporting bugs

Our public bug tracker is on GitHub at https://github.com/NetLogo/NetLogo/issues. You can look here to report a new bug, check if a bug has already been reported, and so on.

When submitting a bug report, please try to include as much of the following information as possible:

- A complete description of the problem and how it occurred.
- The NetLogo model or code you are having trouble with. If possible, attach a complete model. (It's best if you can reduce the amount of code in the model to the minimum necessary to demonstrate the bug.)
- Your system information: NetLogo version, OS version, Java version, and so on. This information is available from NetLogo's "About NetLogo" menu item, then clicking the System tab. In applets, the same information is available by right-clicking the white background of the applet.
- Any error messages that were displayed. Please copy and paste the entire error message into your email, or make a screen capture if you are unable to copy and paste.

We also accept bug reports by email at bugs@ccl.northwestern.edu.

Open source

NetLogo is free, open source software. The source code is hosted at https://github.com/NetLogo/NetLogo. Contributions from interested users are welcome.

For discussion of NetLogo API's and the development of NetLogo itself, try http://groups.google.com/group/netlogo-devel.

Sample Model: Party

This activity gets you thinking about computer modeling and how you can use it. It also gives you insight into NetLogo itself. We encourage beginning users to start here.

At a Party

Have you ever been at a party and noticed how people cluster in groups? You may have also noticed that people don't just stay in a group. As they circulate, the groups change. If you watched these changes over time, you might notice patterns.

For example, in social settings, people may exhibit different behavior than at work or home. Individuals who are confident within their work environment may become shy and timid at a social gathering. And others who are reserved at work may be the "party starter" with friends.

These patterns can depend on the type of gathering. In some settings, people are trained to organize themselves into mixed groups; for example, party games or school-like activities. But in a non-structured atmosphere, people tend to group in a more random manner.

Is there any type of pattern to this kind of grouping?

Let's take a closer look at this question by using the computer to model human behavior at a party. NetLogo's "Party" model looks specifically at the question of grouping by gender at parties: why do groups tend to form that are mostly men, or mostly women?

Let's use NetLogo to explore this question.

What to do:

1. Start NetLogo.
2. Choose "Models Library" from the File menu.
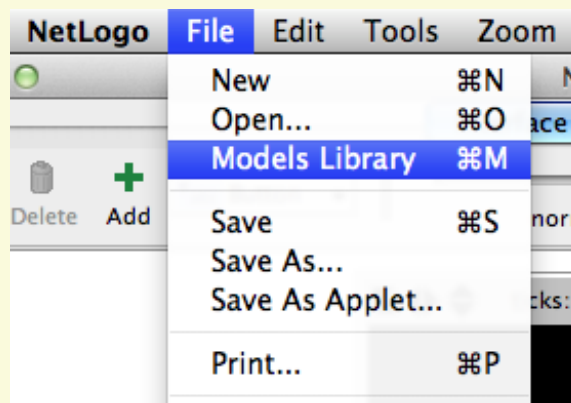
3. Open the "Social Science" folder.
4. Click on the model called "Party".
5. Press the "open" button.
6. Press the "setup" button.

In the view of the model, you will see pink and blue groups with numbers:
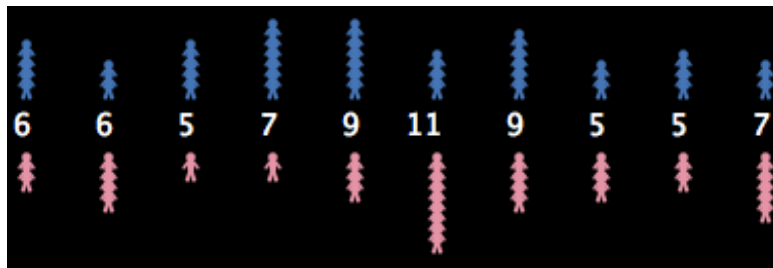
These lines represent mingling groups at a party. Men are shown as blue, women pink. The numbers are the sizes of the groups.

> ***Do all the groups have about the same number of people?***

> ***Do all the groups have about the same number of each sex?***

Let's say you are having a party and invited 150 people. You are wondering how people will gather together. Suppose 10 groups form at the party.

> ***How do you think they will group?***

Instead of asking 150 of your closest friends to gather and randomly group, let's have the computer simulate this situation for us.

> **What to do:**
>
> 1. Press the "go" button. (Pressing "go" again will stop the model manually.)
> 2. Observe the movement of people until the model stops.
> 3. Watch the plots to see what's happening in another way.
> 4. Use the speed slider if you need to slow the model down.

> ***Now how many people are in each group?***

Originally, you may have thought 150 people splitting into 10 groups, would result in about 15 people in each group. From the model, we see that people did not divide up evenly into the 10 groups. Instead, some groups became very small, whereas other groups became very large. Also, the party changed over time from all mixed groups of men and women to all single-sex groups.

> ***What could explain this?***

There are lots of possible answers to this question about what happens at real parties. The designer of this simulation thought that groups at parties don't just form randomly. The groups are determined by how the individuals at the party behave. The designer chose to focus on a particular variable, called "tolerance":

Tolerance is defined here as the percentage of people of the opposite sex an individual is "comfortable" with. If the individual is in a group that has a higher percentage of people of the opposite sex than their tolerance allows, then they become "uncomfortable" and leave the group to find another group.

For example, if the tolerance level is set at 25%, then males are only "comfortable" in groups that are less than 25% female, and females are only "comfortable" in groups that are less

than 25% male.

As individuals become "uncomfortable" and leave groups, they move into new groups, which may cause some people in that group to become "uncomfortable" in turn. This chain reaction continues until everyone at the party is "comfortable" in their group.

Note that in the model, "tolerance" is not fixed. You, the user, can use the tolerance "slider" to try different tolerance percentages and see what the outcome is when you start the model over again.

How to start over:

1. If the "go" button is pressed (black), then the model is still running. Press the button again to stop it.
2. Adjust the "tolerance" slider to a new value by dragging its red handle.
3. Press the "setup" button to reset the model.
4. Press the "go" button to start the model running again.

Challenge

As the host of the party, you would like to see both men and women mingling within the groups. Adjust the tolerance slider on the side of the view to get all groups to be mixed as an end result.

> *To make sure all groups of 10 have both sexes, at what level should we set the tolerance?*

Test your predictions on the model.

> *Can you see any other factors or variables that might affect the male to female ratio within each group?*

Make predictions and test your ideas within this model.

As you are testing your hypotheses, you will notice that patterns are emerging from the data. For example, if you keep the number of people at the party constant but gradually increase the tolerance level, more mixed groups appear.

> *How high does the tolerance value have to be before you get mixed groups?*

> *What percent tolerance tends to produce what percentage of mixing?*

Thinking with models

Using NetLogo to model a situation like a party allows you to experiment with a system in a rapid and flexible way that would be difficult to do in the real world. Modeling also gives you the opportunity to observe a situation or circumstance with less prejudice, as you can examine the underlying dynamics of a situation. You may find that as you model more and more, many of your preconceived ideas about various phenomena will be challenged. For example, a surprising result of the Party model is that even if tolerance is relatively high, a great deal of separation between the sexes occurs.

This is a classic example of an "emergent" phenomenon, where a group pattern results from

the interaction of many individuals. This idea of "emergent" phenomena can be applied to almost any subject.

What other emergent phenomena can you think of?

To see more examples and gain a deeper understanding of this concept and how NetLogo helps learners explore it, you may wish to explore NetLogo's Models Library. It contains models that demonstrate these ideas in systems of all kinds.

For a longer discussion of emergence and how NetLogo helps learners explore it, see "Modeling Nature's Emergent Patterns with Multi-agent Languages" (Wilensky, 2001).

What's next?

The section of the User Manual called Tutorial #1: Running Models goes into more detail about how to use the other models in the Models Library.

If you want to learn how to explore the models at a deeper level, Tutorial #2: Commands will introduce you to the NetLogo modeling language.

Eventually, you'll be ready for Tutorial #3: Procedures. There you can learn how to alter and extend existing models to give them new behaviors, and you can start to build your own models.

Tutorial #1: Models

If you read the [Sample Model: Party](#) section, you got a brief introduction to what it's like to explore a NetLogo model. This section will go into more depth about the features that are available while you're exploring the models in the Models Library.

Throughout all of the tutorials, we'll be asking you to make predictions about what the effects of making changes to the models will be. Keep in mind that the effects are often surprising. We think these surprises are exciting and provide excellent opportunities for learning.

You may want to print out the tutorials to make them easier to refer to while you're using NetLogo.

Sample Model: Wolf Sheep Predation

We'll open one of the Sample Models and explore it in detail. Let's try a biology model: Wolf Sheep Predation, a predator-prey population model.

- Open the Models Library from the File menu.
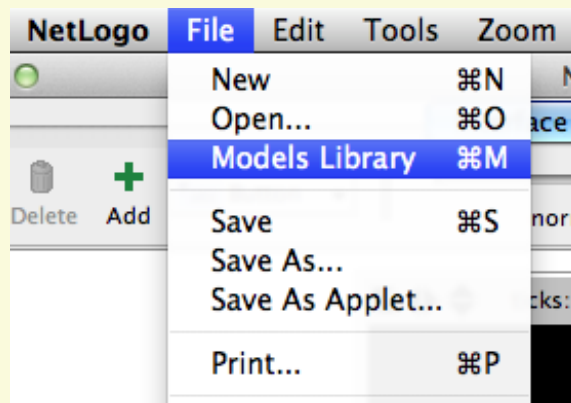
- Choose "Wolf Sheep Predation" from the Biology section and press "Open".

The Interface tab will fill up with lots of buttons, switches, sliders and monitors. These interface elements allow you to interact with the model. Buttons are blue; they set up, start, and stop the model. Sliders and switches are green; they alter model settings. Monitors and plots are beige; they display data.

If you'd like to make the window larger so that everything is easier to see, you can use the Zoom menu.

When you first open the model, you will notice that the "view" (the graphical display of the agents in the model) is empty (all black). To begin the model, you will first need to set it up.

- Press the "setup" button.

  What do you see appear in the view?

- Press the "go" button to start the simulation.

Controlling the Model: Buttons

When a button is pressed, the model responds with an action. A button can be a "once" button, or a "forever" button. You can tell the difference between these two types of buttons by a symbol on the face of the button. Forever buttons have two arrows in the bottom right corners, like this:
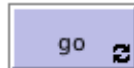
Once buttons don't have the arrows, like this:
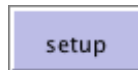
Once buttons do one action and then stop. When the action is finished, the button pops back up.

Forever buttons do an action over and over again. When you want the action to stop, press the button again. It will finish the current action, then pop back up.

Most models, including Wolf Sheep Predation, have a once button called "setup" and a forever button called "go". Many models also have a once button called "go once" or "step once" which is like "go" except that it advances the model by one tick (time step) instead of over and over. Using a once button like this lets you watch the progress of the model more closely.

Stopping a forever button is the normal way to pause or stop a model. After pausing you can make it resume by pressing the button again. (You can also stop a model with the "Halt" item on the Tools menu, but you should only do this if the model is stuck for some reason. Using "Halt" may interrupt the model in the middle of an action, and as the result the model could get confused.)

> - If you like, experiment with the "setup" and "go" buttons in the Wolf Sheep Predation model.
>
> ***Do you ever get different results if you run the model several times with the same settings?***

Controlling speed: Speed Slider

The speed slider allows you to control the speed of a model, that is, the speed at which turtles move, patches change color, and so on.

normal speed
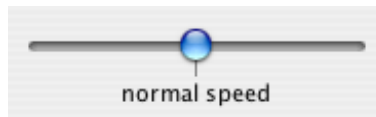
When you move the slider to the left the model slows down so there are longer pauses between each tick (time step). That makes it easier to see what is happening. You might even slow the model down so far as to see exactly what a single turtle is doing.

When you move the speed slider to the right the model speeds up. NetLogo will start skipping frames, that is, it won't update the view at the end of every tick, only some ticks. Updating takes time, so fewer view updates means the model progresses faster.

Note that if you push the speed slider well to the right, the view may update so infrequently that the model appears to have slowed down. It hasn't, as you can see by watching the tick counter race ahead. Only the frequency of view updates has lessened.

Adjusting Settings: Sliders and Switches

A model's settings let you explore different scenarios or hypotheses. Altering the settings and then running the model to see how it reacts can give you a deeper understanding of the phenomena being modeled.

Switches and sliders give you access to a model's settings. Here are the switches and sliders in Wolf Sheep Predation:
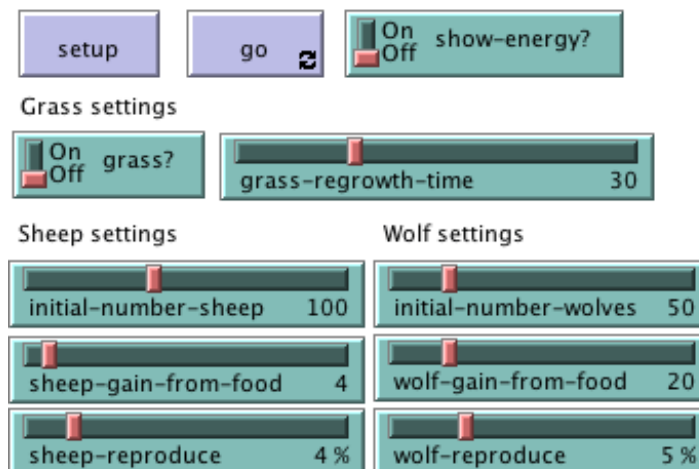
Let's experiment with their effect on the behavior of the model.

- Open Wolf Sheep Predation if it's not open already.
- Press "setup" and "go" and let the model run for about 100 ticks. (The tick count is shown above the view.)
- Stop the model by pressing the "go" button.

  What happened to the sheep over time?

  Let's take a look and see what would happen to the sheep if we change a setting.

- Turn the "grass?" switch on.
- Press "setup" and "go" and let the model run for a similar amount of time as before.

Turning the "grass?" switch on affected the outcome of the model. With the switch off, the amount of grass available always stayed the same. This is not a realistic look at the predator-prey relationship; so by setting and turning on a grass regrowth rate, we were able to model all three factors: sheep, wolf and grass populations.

Another type of setting is called a slider.

Besides switches, a model may also have sliders. While a switch has only two values, on and off, a slider has a whole range of numeric values. For example, the "initial-number-sheep" slider has a minimum value of 0 and a maximum value of 250. The model could run with 0 sheep or it could run with 250 sheep, or anywhere in between. Try this out and see what happens. As you move the marker from the minimum to the maximum value, the number on the right side of the slider changes; this is the number the slider is currently set to.

Let's investigate Wolf Sheep Predation's sliders.

- Change from the Interface to the Info tab to learn what each of this models' sliders represents.

The Info tab offers guidance and insight into the model. Within this tab you will find an explanation of the model, suggestions on things to try, and other information. You may want to read the Info tab before running a model, or you might want to just start experimenting, then look at the Info tab later.

What would happen to the sheep population if there were more sheep and less wolves initially?

- Turn the "grass?" switch off.
- Set the "initial-number-sheep" slider to 100.
- Set the "initial-number-wolves" slider to 20.
- Press "setup" and then "go".
- Let the model run for about 100 ticks.

Try running the model several times with these settings.

What happened to the sheep population?

Did this outcome surprise you? What other sliders or switches can be adjusted to help out the sheep population?

- Set "initial-number-sheep" to 80 and "initial-number-wolves" to 50. (This is close to how they were when you first opened the model.)
- Set "sheep-reproduce" to 10.0%.
- Press "setup" and then "go".
- Let the model run for about 100 time ticks.

What happened to the wolves in this run?

When you open a model, all the sliders and switches are on a default setting. If you open a new model or exit the program, your changed settings will not be saved, unless you choose to save them.

(Note: in addition to sliders and switches, some models have choosers and input boxes. The Wolf Sheep Predation doesn't have any of these, though.)

Gathering Information: Plots and Monitors

The view lets you see what's going on in a model. NetLogo also provides has other ways of giving you information about model run, such as plots and monitors.

Plots

The plot in Wolf Sheep Predation contains three lines: sheep, wolves, and grass / 4. (The grass count is divided by four so it doesn't make the plot too tall.) The lines show what's happening in the model over time. The plot legend shows what each line indicates. In this case, it's the population counts.

When a plot gets close to becoming filled up, the horizontal axis is compressed and all of the data from before gets squeezed into a smaller space. In this way, more room is made for the plot to grow.

If you want to save the data from a plot to view or analyze it in another application, use the "Export Plot" item on the File menu. It saves the plot data in a format that can by read back by spreadsheet and database programs such as Excel. You can also export a plot by right-clicking it and choosing "Export..." from the popup menu.

Monitors

Monitors are another means of displaying information from a model. Here are the monitors in Wolf Sheep Predation:

sheep	wolves	grass / 4
0	0	0
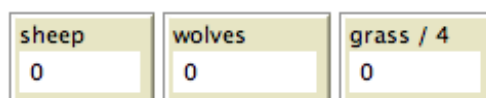
The monitors show us the population of sheep and wolves, and the amount of grass. (Remember, the amount of grass is divided by four to keep the plot from getting too tall.)

The numbers displayed in the monitors change as the model runs, whereas the plots show you data from the whole course of the model run.

Controlling the View

In the Interface tab, you'll see a toolbar of controls. Some of these control aspects of the view.

Let's experiment with the effect of these controls.

- Press "setup" and then "go" to start the model running.
- As the model runs, move the speed slider to the left.

What happens?

This slider is helpful if a model is running too fast for you to see what's going on in detail.

- Move the speed slider to the middle.
- Try moving the speed slider to the right.
- Now try checking and unchecking the "view updates" checkbox.

What happens?

Fast forwarding the model and turning off view updates are useful if you're impatient and want a model to run faster. Fast forwarding (moving the speed slider to the right) drops view updates so the model can run fast, since updating the view takes time that could be used for running the model itself.

When view updates are off completely, the model continues to run in the background, and plots and monitors still update. But if you want to see what's happening, you need to turn view updates back on by rechecking the box. Many models run much faster when view updates are off. For others, it makes little difference.

The size of the view is determined by five separate settings: min-pxcor, max-pxcor, min-pycor, max-pycor, and patch size. Let's take a look at what happens when we change the size of the view in the "Wolf Sheep Predation" model.

There are more model settings than there's room for in the toolbar. The "Settings..." button lets you get to the rest of the settings.

- Press the "Settings..." button in the toolbar.

A dialog will open containing all the settings for the view:
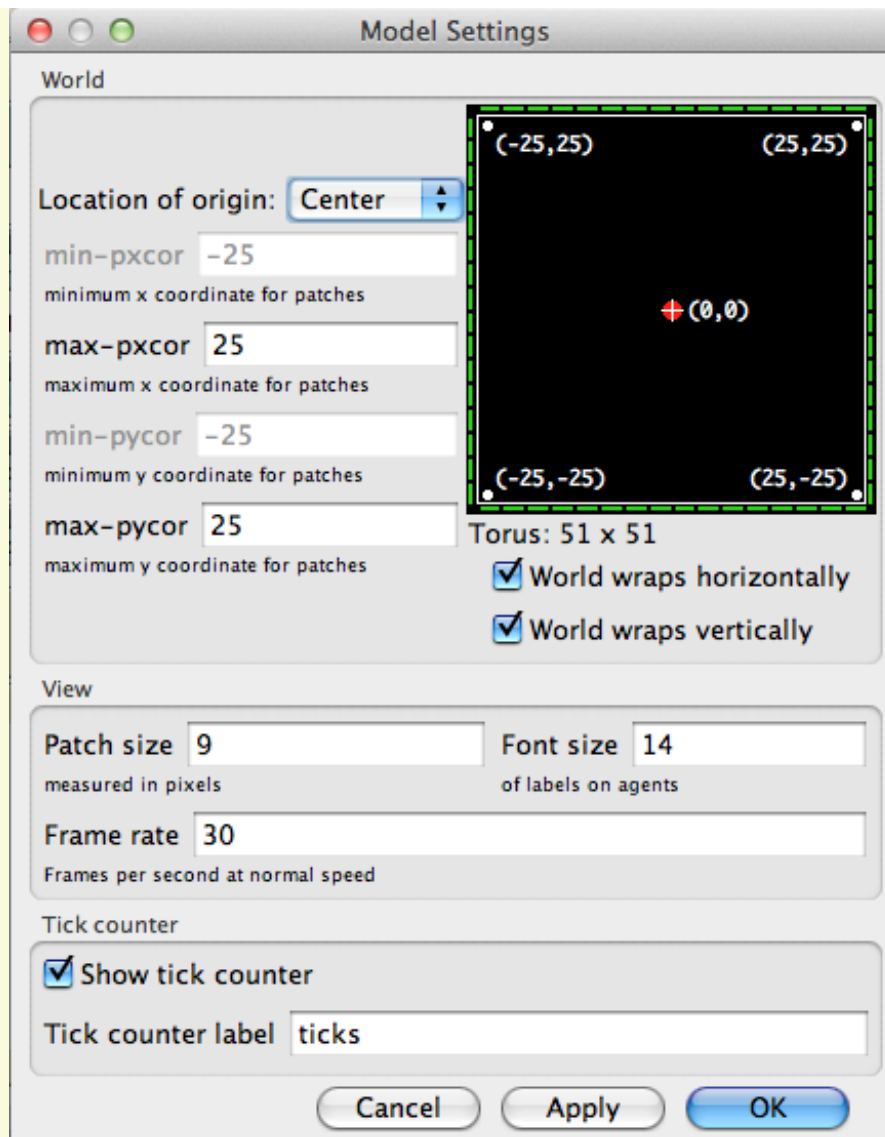
What are the current settings for min-pxcor, max-pxcor, min-pycor, max-pycor, and patch size?

- Press "cancel" to make this window go away without changing the settings.
- Place your mouse pointer next to, but still outside of, the view.

You will notice that the pointer turns into a crosshair.

- Hold down the mouse button and drag the crosshair over the view.

The view is now selected, which you know because it is now surrounded by a gray border.

- Drag one of the square black "handles". The handles are found on the edges and at the corners of the view.
- Unselect the view by clicking anywhere in the white background of the Interface tab.
- Press the "Settings..." button again and look at the settings.

What numbers changed?

What numbers didn't change?

The NetLogo world is a two dimensional grid of "patches". Patches are the individual squares

in the grid. In Wolf Sheep Predation, when the "grass?" switch is on the individual patches are easily seen, because some are green, others brown.

Think of the patches as being like square tiles in a room with a tile floor. By default, exactly in the middle of the room is a tile labeled (0,0); meaning that if the room was divided in half one way and then the other way, these two dividing lines would intersect on this tile. We now have a coordinate system that will help us locate objects within the room:
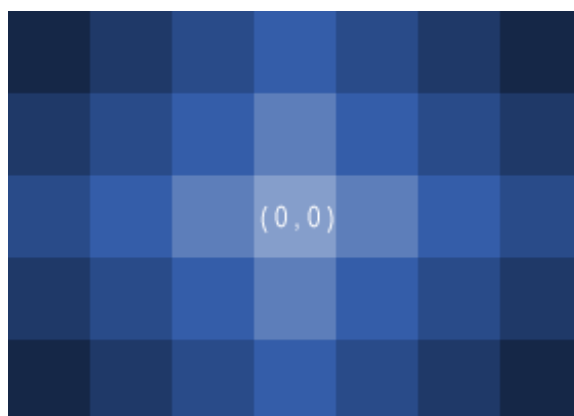
How many tiles away is the (0,0) tile from the right side of the room?

How many tiles away is the (0,0) tile from the left side of the room?

In NetLogo, the number of tiles from right to left is called world-width. And the number of tiles from top to bottom is world-height. These numbers are defined by top, bottom, left and right boundaries.
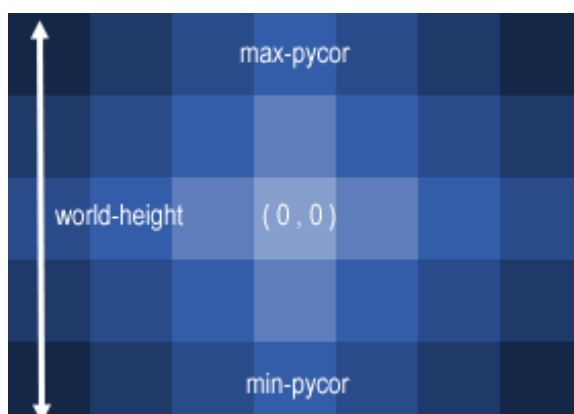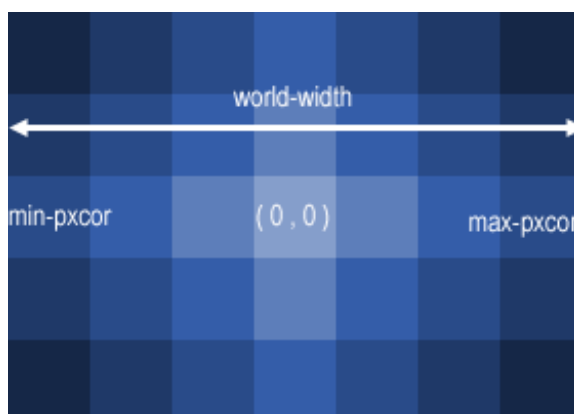
In these diagrams, max-pxcor is 3 , min-pxcor is -3, max-pycor is 2 and min-pycor is -2.

When you change the patch size, the number of patches (tiles) doesn't change, the patches only get larger or smaller in the view.

Let's look at the effect of changing the minimum and maximum coordinates in the world.

- Using the Settings dialog that is still open, change max-pxcor to 30 and max-pycor value to 10. Notice that min-pxcor and min-pycor change too. That's because by default the origin (0,0) is in the center of the world.

  What happened to the shape of the view?

- Press the "setup" button.

Now you can see the new patches you have created.

- Edit the view by pressing the "Settings..." button again.
- Change the patch size to 20 and press "OK".

  What happened to the size of the view? Did its shape change?

Editing the view also lets you change other settings. Feel free to experiment with these.

Once you are done exploring the Wolf Sheep Predation model, you may want to take some time just to explore some of the other models available in the Models Library.

Models Library

The library contains four sections: Sample Models, Curricular Models, Code Examples, and HubNet Activities.

Sample Models

The Sample Models section is organized by subject area and currently contains more than 200 models. We are continuously working on adding new models to it, so come visit this section at a later date to view the new additions to the library.

Some of the folders in Sample Models have folders inside them labeled "(unverified)". These models are complete and functional, but are still in the process of being reviewed for content, accuracy, and quality of code.

Curricular Models

These are models designed to be used in schools in the context of curricula developed by the CCL at Northwestern University. Some of these are models are also listed under Sample Models; others are unique to this section. See the Info tabs of the models for more information on the curricula they go with.

Code Examples

These are simple demonstrations of particular features of NetLogo. They'll be useful to you later when you're extending existing models or building new ones. For example, if you wanted to add a histogram to your model, you'd look at "Histogram Example" to find out how.

HubNet Activities

This section contains participatory simulations for use with groups. For more information about HubNet, see the HubNet Guide.

What's Next?

If you want to learn how to explore models at a deeper level, Tutorial #2: Commands will introduce you to the NetLogo modeling language.

In Tutorial #3: Procedures you can learn how to alter and extend existing models and build new ones.

Tutorial #2: Commands

So far you've successfully navigated your way through opening and running models, pressing buttons, changing sliders and switches, and gathering information from a model using plots and monitors.

In this section, the focus will start to shift from observing models to manipulating models. You will start to see the inner workings of the models and be able to change how they look.

Sample Model: Traffic Basic

- Open the Models Library (from the File menu).
- Open Traffic Basic, found in the "Social Science" section.
- Run the model for a while to get a feel for it.
- Consult the Info tab for any questions you may have.

In this model, you will notice one red car in a stream of blue cars. The stream of cars are all moving in the same direction. Every so often they "pile up" and stop moving. This is modeling how traffic jams can form without a specific cause like an accident.

You may alter the settings and observe a few runs to get a fuller understanding of the model.

> ***As you are using the Traffic Basic model, have you noticed any additions you would like to make to the model?***

Looking at the Traffic Basic model, you may notice the environment is fairly simple; a black background with a white street and number of blue cars and one red car. Changes that could be made to the model include: changing the color and shape of the cars, adding a house or street light, creating a stop light, or even creating another lane of traffic. Some of these suggested changes are visual, to enhance the look of the model, while others are more behavioral. We will be focusing more on the simpler or cosmetic changes throughout most of this tutorial. (Tutorial #3 will go into greater detail about behavioral changes, which require changing the Code tab.)

To make these simple changes we will be using the Command Center.

Command Center

The Command Center is found in the Interface tab. It allows you to enter commands or directions to a model. Commands are instructions you can give to NetLogo's agents: turtles, patches, links, and the observer.

In Traffic Basic:

- Press the "setup" button.
- Locate the Command Center.

- Click the mouse in the white box at the bottom of the Command Center.
- Type the text shown here:
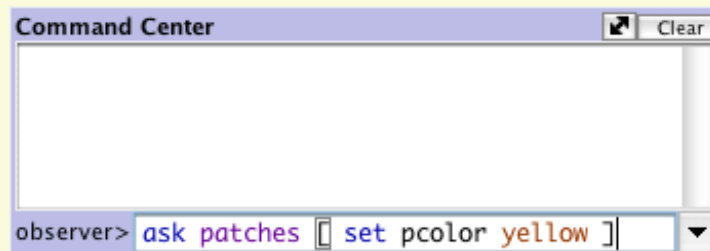
- Press the return key.

What happened to the View?

You may have noticed the background of the View has turned all yellow and the street has disappeared.

Why didn't the cars turn yellow too?

Looking back at the command that was written, we asked only the patches to change their color. In this model, the cars are represented by a different kind of agent, called "turtles". Therefore, the cars did not receive these instructions and thus did not change.

What happened in the Command Center?

You may have noticed that the command you just typed is now displayed in the Command Center as shown below:
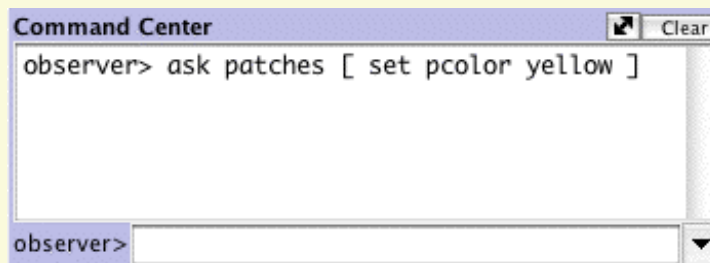
- Type in the bottom of the Command Center the text shown below:
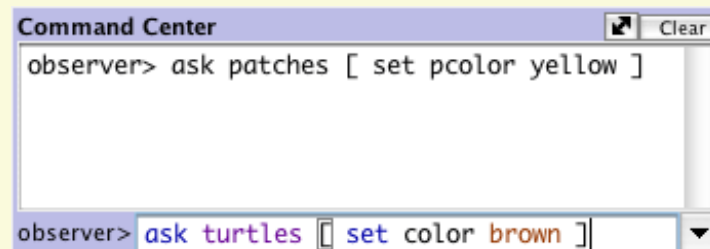
Was the result what you expected?

The view should have a yellow background with a line of brown cars in the middle:
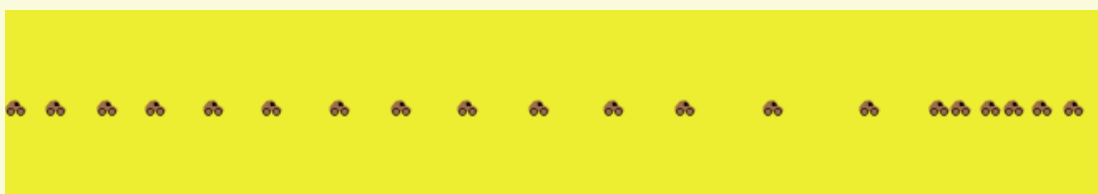
The NetLogo world is a two dimensional world that is made up of turtles, patches, links, and an observer. The patches are the ground over which the turtles move. Links are connections between turtles. And the observer is a being that oversees everything that is going on. (For more specifics, refer to the NetLogo Programming Guide.)

In the Command Center you can give commands to any of these types of agents. You choose which type to talk to by using the popup menu located in the bottom left corner. You can also use the tab key on your keyboard to cycle through the different types.

- In the Command Center, click on the "observer>" in the bottom left corner:
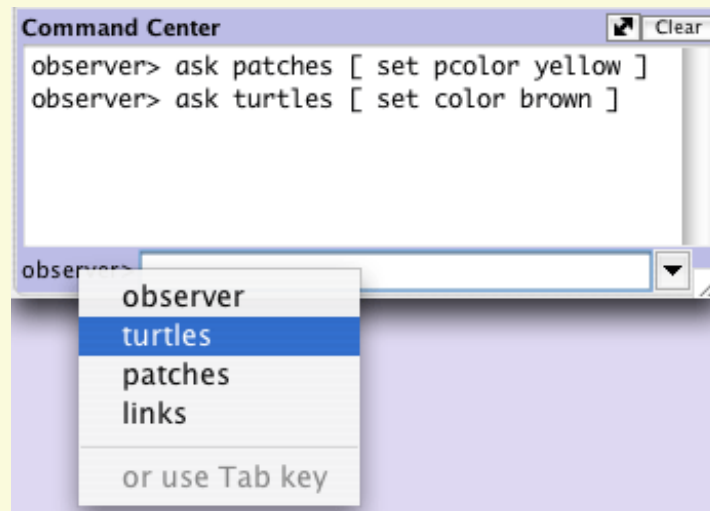
- Choose "turtles" from the popup menu.
- Type `set color pink` and press return.
- Press the tab key until you see "patches>" in the bottom left corner.
- Type `set pcolor white` and press return.

What does the View look like now?

Do you notice any differences between these two commands and the observer commands from earlier?

The observer oversees the world and therefore can give a command to the patches or turtles using `ask`. Like in the first example (observer> `ask patches [set pcolor yellow]`), the observer has to ask the patches to set their pcolor to yellow. But when a command is directly given to a group of agents like in the second example (patches> `set pcolor white`), you only have to give the command itself.

- Press "setup".

What happened?

Why did the View revert back to the old version, with the black background and white road? Upon pressing the "setup" button, the model will reconfigure itself back to the settings outlined

in the Code tab. The Command Center doesn't permanently change the model. It allows you to manipulate the NetLogo world directly to further answer those "What if" questions that pop up as you are investigating the models. (The Code tab is explained in the next tutorial, and in the Programming Guide.)

Now that we have familiarized ourselves with the Command Center, let's look at some more details about how colors work in NetLogo.

Working with colors

You may have noticed in the previous section that we used two different words for changing color: `color` and `pcolor`.
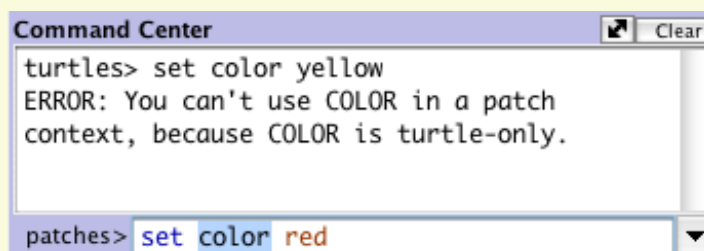
> ### *What is the difference between color and pcolor?*
>
> - Choose "turtles" from the popup menu in the Command Center (or use the tab key).
> - Type `set color blue` and press return.
>
> ### *What happened to the cars?*
>
> Think about what you did to make the cars turn blue, and try to make the patches turn red.
>
> If you try to ask the patches to `set color red`, an error message occurs:
>
> ```
> Command Center                              ⬈  Clear
> turtles> set color yellow
> ERROR: You can't use COLOR in a patch
> context, because COLOR is turtle-only.
>
> patches> set color red                      ▼
> ```
>
> - Type `set pcolor red` instead and press return.

We call `color` and `pcolor` "variables". Some commands and variables are specific to turtles and some are specific to patches. For example, the `color` variable is a turtle variable, while the `pcolor` variable is a patch variable.

Go ahead and practice altering the colors of the turtles and patches using the `set` command and these two variables.

To be able to make more changes to the colors of turtles and patches, or shall we say cars and backgrounds, we need to gain a little insight into how NetLogo deals with colors.

In NetLogo, colors have a numeric value. In all of the exercises we have been using the name of the color. This is because NetLogo recognizes 16 different color names. This does not mean that NetLogo only recognizes 16 colors. There are many shades in between these colors that can be used too. Here's a chart that shows the whole NetLogo color space:

black = 0   white = 9.9

	0	1	2	3	4	5	6	7	8	9	
gray = 5	0	1	2	3	4	5	6	7	8	9	9.9
red = 15	10	11	12	13	14	15	16	17	18	19	19.9
orange = 25	20	21	22	23	24	25	26	27	28	29	29.9
brown = 35	30	31	32	33	34	35	36	37	38	39	39.9
yellow = 45	40	41	42	43	44	45	46	47	48	49	49.9
green = 55	50	51	52	53	54	55	56	57	58	59	59.9
lime = 65	60	61	62	63	64	65	66	67	68	69	69.9
turquoise = 75	70	71	72	73	74	75	76	77	78	79	79.9
cyan = 85	80	81	82	83	84	85	86	87	88	89	89.9
sky = 95	90	91	92	93	94	95	96	97	98	99	99.9
blue = 105	100	101	102	103	104	105	106	107	108	109	109.9
violet = 115	110	111	112	113	114	115	116	117	118	119	119.9
magenta = 125	120	121	122	123	124	125	126	127	128	129	129.9
pink = 135	130	131	132	133	134	135	136	137	138	139	139.9

To get an intermediate shade, you refer to it by a number instead, or by adding or subtracting a number from a name. For example, when you type `set color red`, this does the same thing as if you had typed `set color 15`. And you can get a lighter or darker version of the same color by using a number that is a little larger or a little smaller, as follows.

- Choose "patches" from the popup menu in the Command Center (or use the tab key).
- Type `set pcolor red - 2` (The spacing around the "-" is important.)

By subtracting from red, you make it darker.

- Type `set pcolor red + 2`

By adding to red, you make it lighter.

You can use this technique on any of the colors listed in the chart.

Agent Monitors and Agent Commanders

In the previous activity, we used the `set` command to change the colors of all the cars. But if you recall, the original model contained one red car amongst a group of blue cars. Let's look at how to change only one car's color.

- Press "setup" to get the red car to reappear.
- Right-click on the red car.

- If there is another turtle close to the red turtle you'll see more than one turtle listed at the bottom of the menu. Move your mouse over the turtle selections, notice when your mouse highlights a turtle menu item that turtle is highlighted in the view. Select "inspect turtle" from the sub-menu for the red turtle.

A turtle monitor for that car will appear:
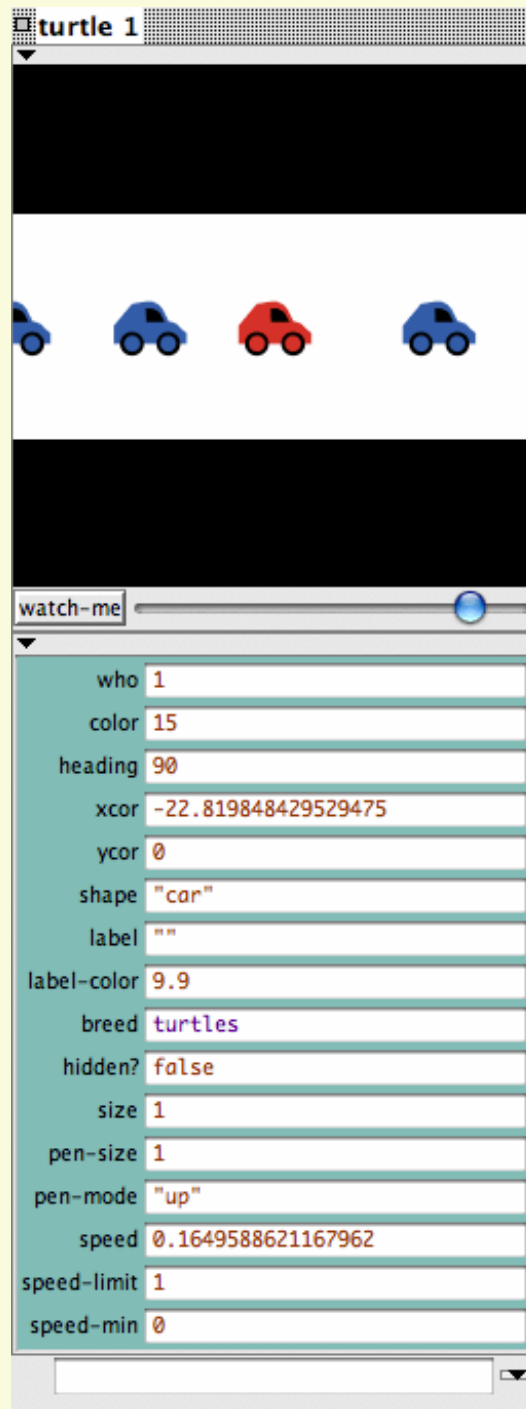
The mini-view at the top of the agent monitor will always stay centered on this agent. You can zoom the view in and out using the slider below the view and you can watch this turtle in the main view by pressing the "watch-me" button.

Taking a closer look at this turtle monitor, we can see all of the variables that belong to the red car. A variable is a location that holds a value.

Let's take a closer look at the turtle monitor:

This turtle monitor is showing a turtle who that has a who number of 0, a color of 15 (red -- see chart above), and the shape of a car.

There are two other ways to open a turtle monitor besides right-clicking. One way is to choose "Turtle Monitor" from the Tools menu, then type the who number of the turtle you want to inspect into the "who" field and press return. The other way is to type `inspect turtle 0` (or other who number) into the Command Center.

You close a turtle monitor by clicking the close box in the upper left hand corner (Mac) or upper right hand corner (other operating systems).

Now that we know more about Agent Monitors, we have three ways to change an individual turtle's color.

One way is to use the box called an Agent Commander found at the bottom of an Agent Monitor. You type commands here, just like in the Command Center, but the commands you type here are only done by this particular turtle.

- In the Agent Commander of the Turtle Monitor for turtle 0, type `set color pink.`

  What happens in the View?

  Did anything change in the Turtle Monitor?

A second way to change one turtle's color is to go directly to the color variable in the Turtle Monitor and change the value.

- Select the text to the right of "color" in the Turtle Monitor.
- Type in a new color such as `green + 2.`

  What happened?

The third way to change an individual turtle's or patch's color is to use the observer. Since, the observer oversees the NetLogo world, it can give commands that affect individual turtles, as well as groups of turtles.

- In the Command Center, select "observer" from the popup menu (or use the tab key).

- Type `ask turtle 0 [set color blue]` and press return.

  What happens?

Just as there are Turtle Monitors, there are also Patch Monitors. Patch monitors work very similarly to Turtle Monitors.

  Can you make a patch monitor and use it to change the color of a single patch?

If you try to have the observer `ask patch 0 [set pcolor blue]`, you'll get an error message:
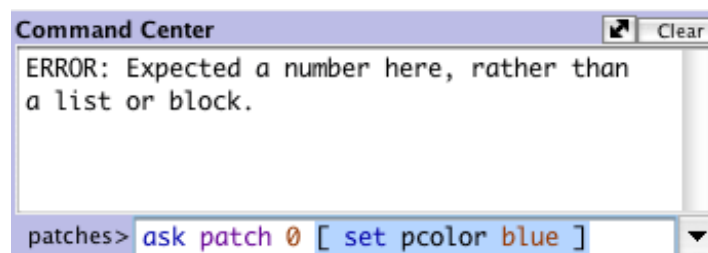
To ask an individual turtle to do something, we use its who number. But patches don't have who numbers, therefore we need to refer to them some other way.

Remember, patches are arranged on a coordinate system. Two numbers are needed to plot a point on a graph: an x-axis value and a y-axis value. Patch locations are designated in the same way as plotting a point.

- Open a patch monitor for any patch.
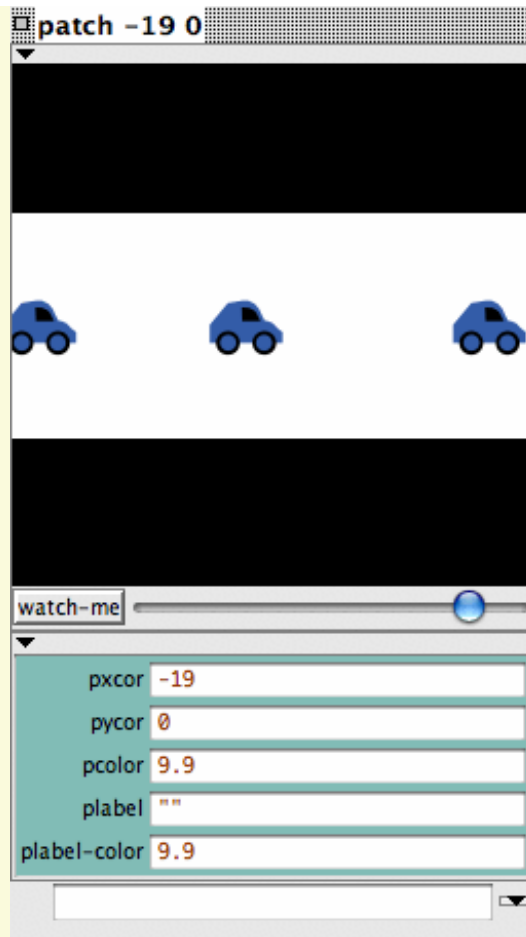
The monitor shows that for the patch in the picture, its `pxcor` variable is -19 and its `pycor` variable is 0. If we go back to the analogy of the coordinate plane and wanted to plot this point, the point would be found on the x axis left of the origin, where x=-19 and y=0.

To tell this particular patch to change color:

- In the bottom of the patch monitor, enter `set pcolor blue` and press return.

Typing a command in a turtle or patch monitor addresses only that turtle or patch.

You can also talk to a single patch from the Command Center:

- In the Command Center, enter `ask patch -19 0 [set pcolor green]` and press return.

What's Next?

At this point, you may want to take some time to try out the techniques you've learned on some of the other models in the Models Library.

In Tutorial #3: Procedures you can learn how to alter and extend existing models and build your own models.

Tutorial #3: Procedures

This tutorial leads you through the process of building a complete model, built up in stages, with every step explained along the way.

Agents and procedures

In Tutorial #2, you learned how to use the command center and agent monitors to inspect and modify agents and make them do things. Now you're ready to learn about the real heart of a NetLogo model: the Code tab.

You've seen that agents in NetLogo are divided into patches, turtles, links, and the observer. Patches are stationary and arranged in a grid. Turtles move over that grid. Links connect two turtles. The observer oversees everything that's going on and does whatever the turtles, patches and links can't do for themselves.

All four types of agents can run NetLogo commands. All four can also run "procedures". A procedure combines a series of NetLogo commands into a single new command that you define.

You will now learn to write procedures that make turtles move, eat, reproduce, and die. You will also learn how to make monitors, sliders, and plots. The model we'll build is a simple ecosystem model not unlike Wolf Sheep Predation from Tutorial #1.

Making the setup button

To start a new model, select "New" from the File menu. Then begin by creating a setup button:

- Click the "Add" icon in the toolbar at the top of the Interface tab.
- On the menu next to Add, select Button (if it isn't already selected).
- Click wherever you want the button to appear in the empty white area of the Interface tab.
- A dialog box for editing the button opens. Type `setup` in the box labeled "Commands".
- Press the OK button when you're done; the dialog box closes.

Now you have a setup button. Pressing the button runs a procedure called "setup". A procedure is a sequence of NetLogo commands that we assign a new name. We'll define that procedure soon, but we haven't yet. The button refers to a procedure that doesn't exist, so the button turns red:
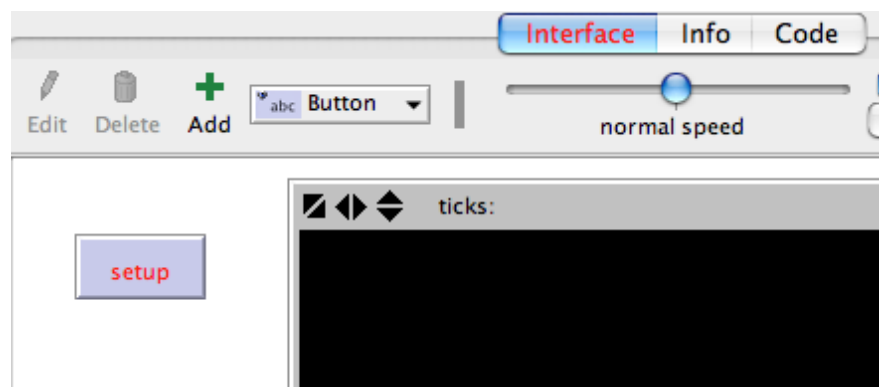
If you want to see the actual error message, click the button.

Now we'll create the "setup" procedure, so the error message will go away:

- Switch to the Code tab.
- Type the following:

```
to setup
  clear-all
  create-turtles 100 [ setxy random-xcor random-ycor ]
  reset-ticks
end
```

When you're done, the Code tab looks like this:
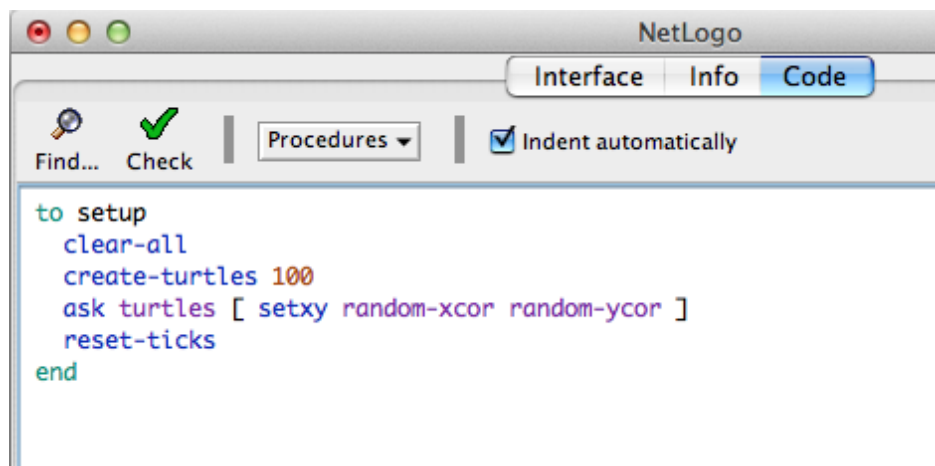
Note that some lines are indented. Most people find it helpful to indent their code. It isn't mandatory, but it makes the code easier to read and change.

Your procedure begins with `to` and ends with `end`. Every procedure begins and ends with these words.

Let's look at what you typed in and see what each line of your procedure does:

- `to setup` begins defining a procedure named "setup".
- `clear-all` resets the world to an initial, empty state. All the patches turn black and any turtles you might have created disappear. Basically, it wipes the slate clean for a new model run.
- `create-turtles 100` creates 100 turtles. They start out standing at the origin, that is, the center of patch 0,0.
- After `create-turtles` we can put commands for the new turtles to run, enclosed by square brackets.
- `setxy random-xcor random-ycor` is a command using "reporters". A reporter, as opposed to a command, reports a result. First each turtle runs the reporter `random-xcor` which will report a random number from the allowable range of turtle coordinates along the X axis. Then each turtle runs the reporter `random-ycor`, same for the Y axis. Finally each turtle runs the `setxy` command with those two numbers as inputs. That makes the turtle move to the point with those coordinates.
- `reset-ticks` starts the tick counter, now that setup is otherwise complete.
- `end` completes the definition of the "setup" procedure.

When you're done typing, switch to the Interface tab and press the setup button you made before. You will see the turtles scattered around the world:
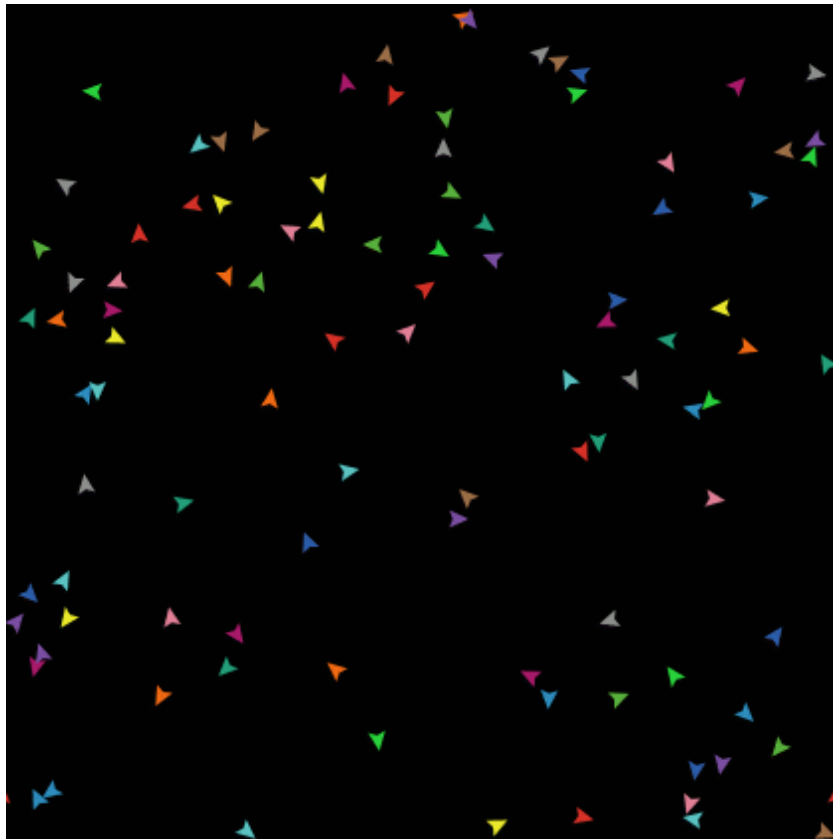
Press setup a couple more times, and see how the arrangement of turtles is different each time. Note that some turtles may be right on top of each other.

Think a bit about what you needed to do to make this happen. You needed to make a button in the interface and make a procedure that the button uses. The button only worked once you completed both of these separate steps. In the remainder of this tutorial, you will often have to complete two or more similar steps to add another feature to the model. If something doesn't appear to work after you completed what you thought is the final step for that new feature, continue to read ahead to see if there is still more to do. After reading ahead for a couple of paragraphs, you should then go back over the directions to see if there is any step you might have missed.

Switching to tick-based view updates

Now that we're using the tick counter (with `reset-ticks`), we should tell NetLogo that it only needs to update the view once per tick, instead of continuously updating it.

- Find the view updates menu. It's above the view and by default says "continuous".
- Choose "on ticks" instead.

This makes your model run faster and ensures a consistent appearance (since the updates will happen at consistent times). See the Programming Guide for a fuller discussion of view updates.

Making the go button

Now make a button called "go". Follow the same steps you used to make the setup button, except:

- For Commands enter `go` instead of `setup`.
- Check the "Forever" checkbox in the edit dialog.
- Check the "Disable until ticks start" checkbox too.
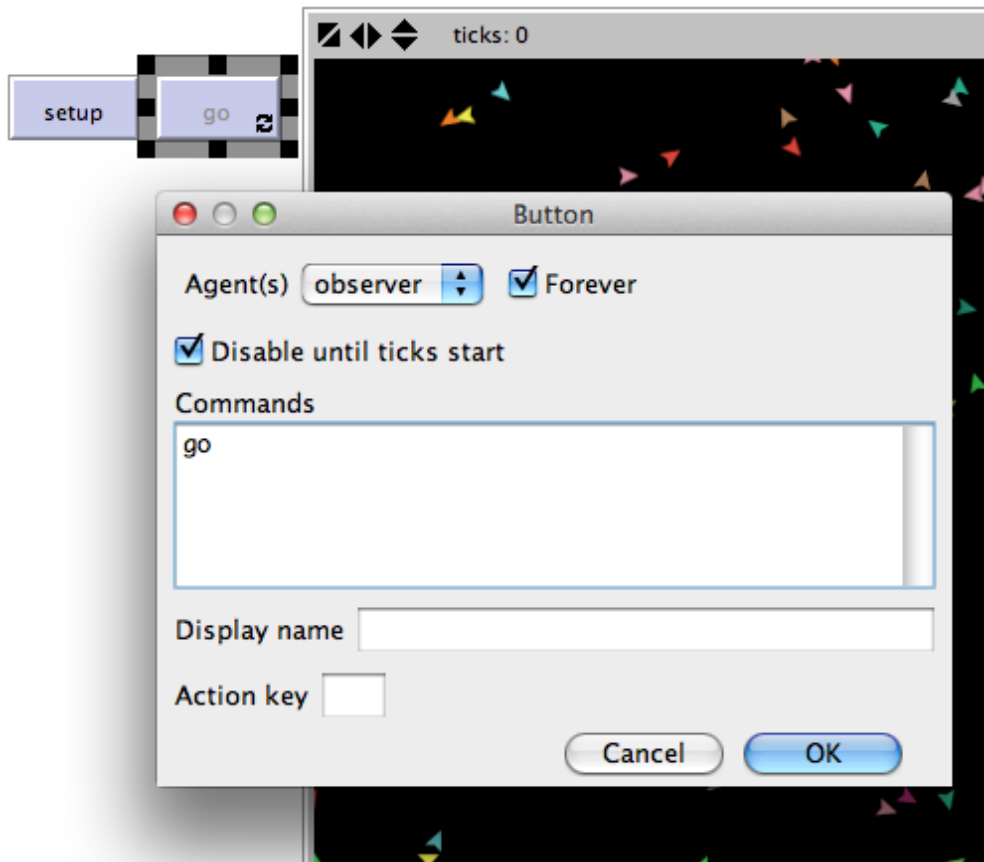
The "Forever" checkbox makes the button stay down once pressed, so its commands run over and over again, not just once.

The "Disable until ticks start" prevents you from pressing go before setup.

- Then add a go procedure to the Code tab:

```
to go
  move-turtles
  tick
end
```

`tick` is a primitive that advances the tick counter by one tick.

But what is `move-turtles`? Is it a primitive (in other words, built-in to NetLogo)? No, it's another procedure that you're about to add. So far, you have introduced two procedures that you added yourself: `setup` and `go`.

- Add the `move-turtles` procedure after the `go` procedure:

```
to go
```

```
      move-turtles
      tick
    end

    to move-turtles
      ask turtles [
        right random 360
        forward 1
      ]
    end
```

Note there are no spaces around the hyphen in `move-turtles`. In Tutorial #2 we used `red - 2`, with spaces, in order to subtract two numbers, but here we want `move-turtles`, without spaces. The "-" combines "move" and "turtles" into a single name.

Here is what each command in the `move-turtles` procedure does:

- `ask turtles [ ... ]` says that each turtle should run the commands in the brackets.
- `right random 360` is another command that uses a reporter. First, each turtle picks a random whole number between 0 and 359. (`random` doesn't include the number you give it as a possible result.) Then the turtle turns right this number of degrees.
- `forward 1` makes the turtle move forward one step.

Why couldn't we have just written all of these commands in `go` instead of in a separate procedure? We could have, but during the course of building your project, it's likely that you'll add many other parts. We'd like to keep `go` as simple as possible, so that it is easy to understand. Eventually, it will include many other things you want to have happen as the model runs, such as calculating something or plotting the results. Each of these things to do will have its own procedure and each procedure will have its own unique name.

The 'go' button you made in the Interface tab is a forever button, meaning that it will continually run its commands until you shut it off (by clicking on it again). After you have pressed 'setup' once, to create the turtles, press the 'go' button. Watch what happens. Turn it off, and you'll see that all the turtles stop in their tracks.

Note that if a turtle moves off the edge of the world, it "wraps", that is, it appears on the other side. (This is the default behavior. It can be changed; see the Topology section of the Programming Guide for more information.)

Experimenting with commands

We suggest you start experimenting with other turtle commands.

Type commands into the Command Center (like `turtles> set color red`), or add commands to `setup`, `go`, or `move-turtles`.

Note that when you enter commands in the Command Center, you must choose `turtles>`, `patches>`, `links>`, or `observer>` in the popup menu on the left, depending on which agents are going to run the commands. It's just like using `ask turtles` or `ask patches`, but saves typing. You can also use the tab key to switch agent types, which you might find more convenient than using the menu.

You might try typing `turtles> pen-down` into the Command Center and then pressing the go button.

Also, inside the `move-turtles` procedure you can try changing `right random 360` to `right random 45`.

Play around. It's easy and the results are immediate and visible -- one of NetLogo's many

strengths.

When you feel you've done enough experimenting for now, you're ready to continue improving the model you are building.

Patches and variables

Now we've got 100 turtles aimlessly moving around, completely unaware of anything else around them. Let's make things a little more interesting by giving these turtles a nice background against which to move.

- Go back to the `setup` procedure. We can rewrite it as follows:

```
to setup
  clear-all
  setup-patches
  setup-turtles
  reset-ticks
end
```

- The new definition of `setup` refers to two new procedures. To define `setup-patches`, add this:

```
to setup-patches
  ask patches [ set pcolor green ]
end
```

The `setup-patches` procedure sets the color of every patch to green to start with. (A turtle's color variable is `color`; a patch's is `pcolor`.)

The only part remaining in our new 'setup' that is still undefined is `setup-turtles`.

- Add this procedure too:

```
to setup-turtles
  create-turtles 100
  ask turtles [ setxy random-xcor random-ycor ]
end
```

Did you notice that the new `setup-turtles` procedure has most of the same commands as the old `setup` procedure?

- Switch back to the Interface tab.
- Press the setup button.

Voila! A lush NetLogo landscape complete with turtles and green patches appears:
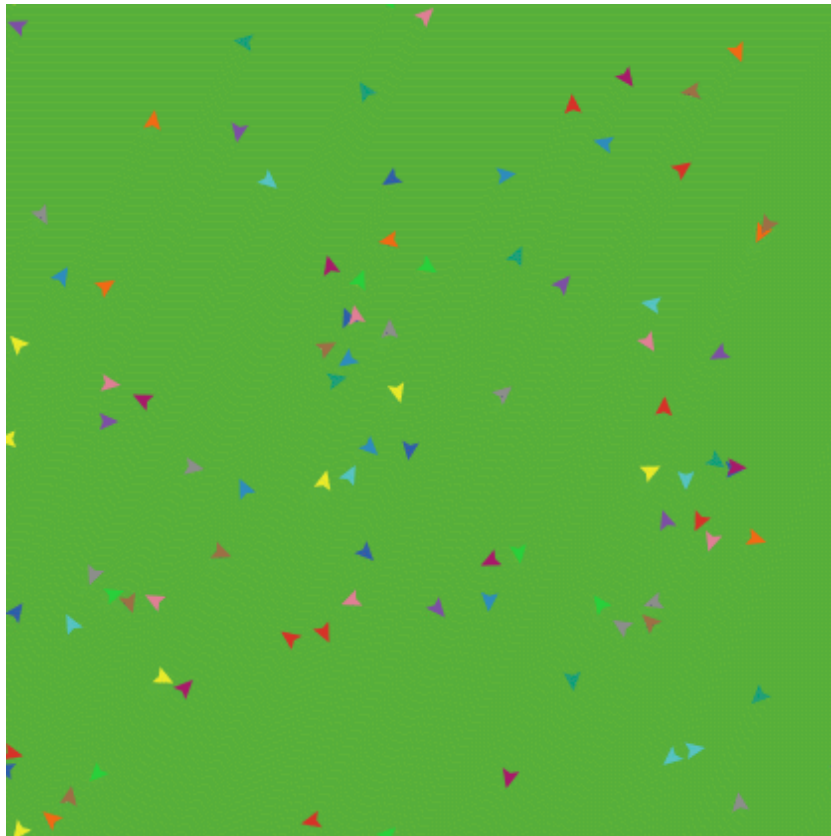
After seeing the new `setup` procedure work a few times, you may find it helpful to read through the procedure definitions again.

Turtle variables

So we have some turtles running around on a landscape, but they aren't doing anything with it. Let's add some interaction between the turtles and the patches.

We'll make the turtles eat "grass" (the green patches), reproduce, and die. The grass will gradually grow back after it is eaten.

We'll need a way of controlling when a turtle reproduces and dies. We'll determine that by keeping track of how much "energy" each turtle has. To do that we need to add a new turtle variable.

You've already seen built-in turtle variables like <u>color</u>. To make a new turtle variable, we add a <u>turtles-own</u> declaration at the top of the Code tab, before all the procedures. Call it `energy`:

```
turtles-own [energy]

to go
  move-turtles
  eat-grass
  tick
end
```

Let's use this newly defined variable (`energy`) to allow the turtles to eat.

- Switch to the Code tab.
- Rewrite the `go` procedure as follows:

```
to go
```

```
    move-turtles
    eat-grass
    tick
  end
```

- Add a new `eat-grass` procedure:

```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy energy + 10
    ]
  ]
end
```

We are using the `if` command for the first time. Look at the code carefully. Each turtle, when it runs these commands, compares the value of the patch color it is on (`pcolor`) to the value for `green`. (A turtle has direct access to the variables of the patch it is standing on.) If the patch color is green, the comparison reports `true`, and only then will the turtle run the commands inside the brackets (otherwise it skips them). The commands make the turtle change the patch color to black and increase its own energy by 10. The patch turns black to signify that the grass at that spot has been eaten. And the turtle is given more energy, from having just eaten.

Next, let's make the movement of turtles use up some of the turtle's energy.

- Rewrite `move-turtles` as follows:

```
to move-turtles
  ask turtles [
    right random 360
    forward 1
    set energy energy - 1
  ]
end
```

As each turtle wanders, it will lose one unit of energy at each step.

- Switch to the Interface tab now and press the setup button and the go button.

  You'll see the patches turn black as turtles travel over them.
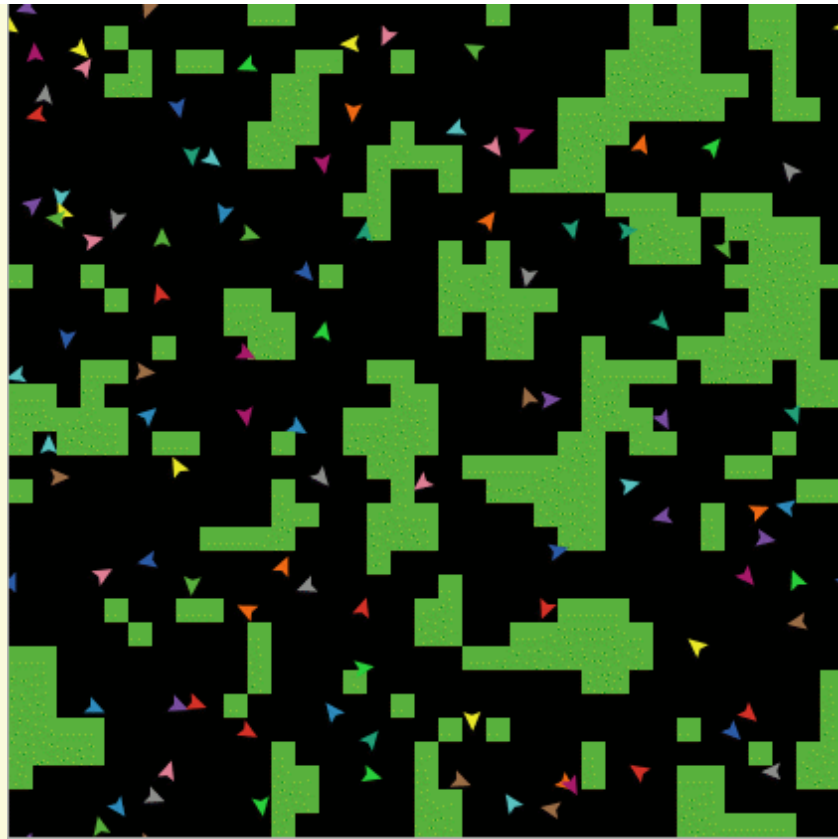
Monitors

Next you will create two monitors in the Interface tab with the toolbar. (You make them just like buttons and sliders, using the Add icon on the toolbar.) Let's make the first monitor now.

- Create a monitor by clicking the Add icon on the toolbar, selecting Monitor next to it, and clicking on an open spot in the Interface.

  A dialog box will appear.

- In the dialog type: `count turtles` (see image below).
- Press the OK button to close the dialog.
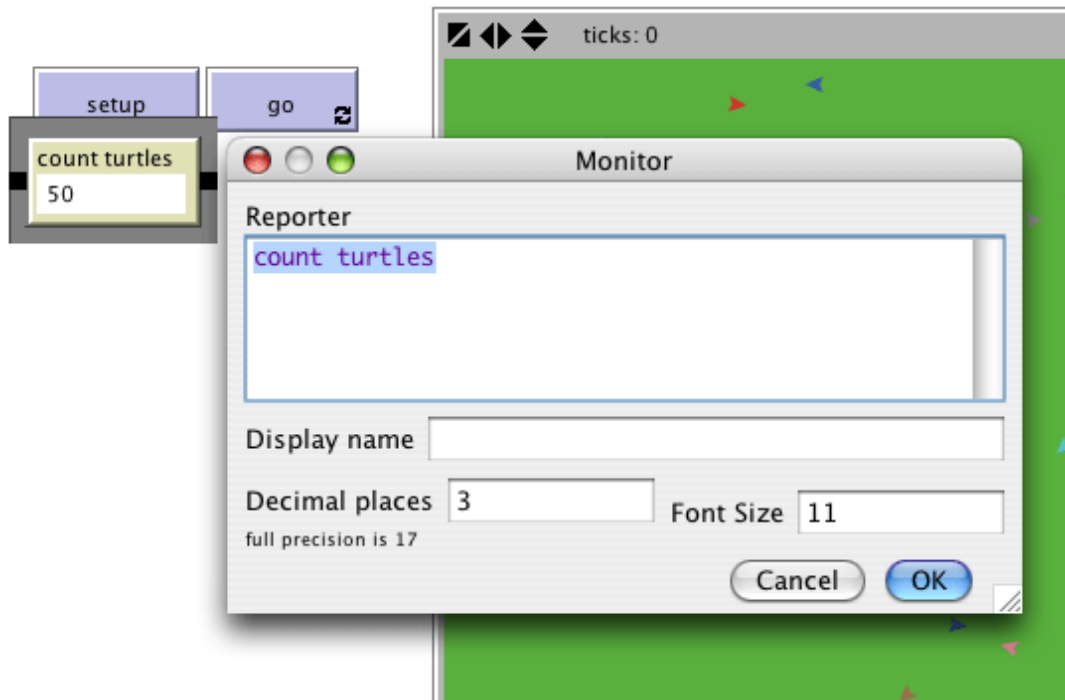
`turtles` is an "agentset", the set of all turtles. `count` tells us how many agents are in that set.

Let's make the second monitor now:

- Create a monitor by clicking the Add icon on the toolbar, selecting Monitor next to it, and clicking on an open spot in the Interface.

  A dialog box will appear.

- In the Reporter section of the dialog box type:`count patches with [pcolor = green]` (see image below).
- In the Display name section of the dialog box type:`green patches`
- Press the OK button to close the dialog box.
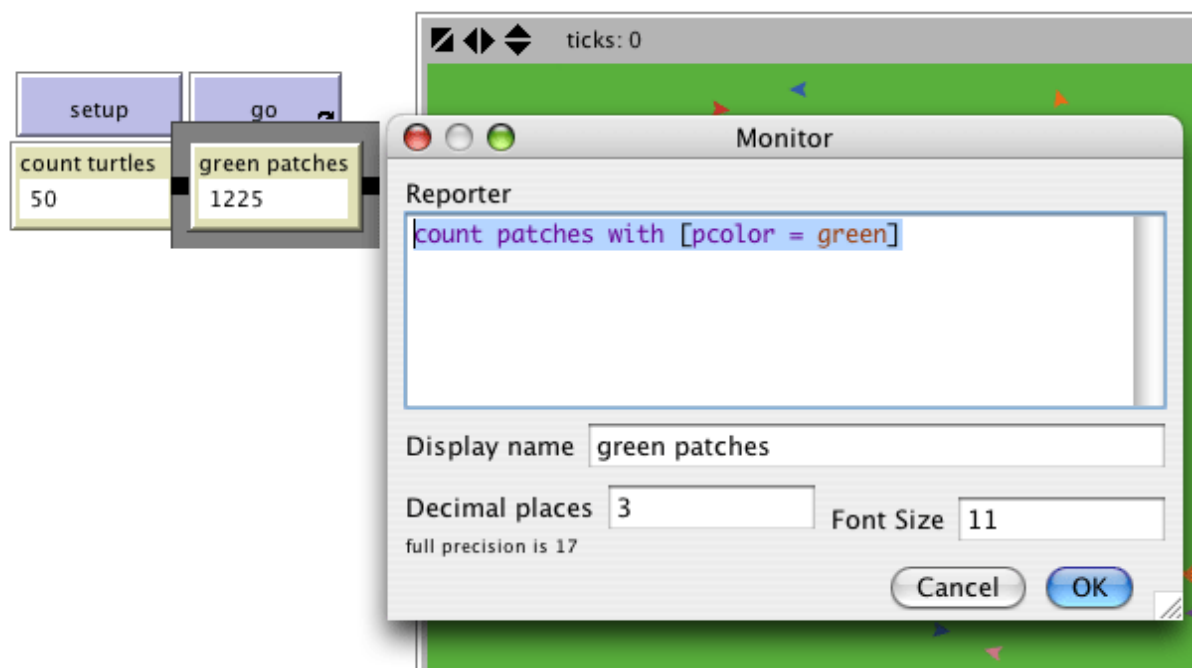
Here we're using `count` again to see how many agents are in an agentset. `patches` is the set of all the patches, but we don't just want to know how many patches there are total, we want to know how many of them are green. That's what `with` does; it makes a smaller agentset of just those agents for whom the condition in the brackets is true. The condition is `pcolor = green`, so that gives us just the green patches.

Now we have two monitors that will report how many turtles and green patches we have, to help us track what's going on in our model. As the model runs, the numbers in the monitors will automatically change.

- Use the setup and go buttons and watch the numbers in the monitors change.

Switches and labels

The turtles aren't just turning the patches black. They're also gaining and losing energy. As the model runs, try using a turtle monitor to watch one turtle's energy go up and down.

It would be nicer if we could see every turtle's energy all the time. We will now do exactly that, and add a switch so we can turn the extra visual information on and off.

- Click on the Add icon on the toolbar (in the Interface tab).
- Select Switch from the menu next to Add.
- Click on an open spot in the interface.

  A dialog will appear.

- Into the Global variable field, type `show-energy?` Don't forget to include the question mark in the name. (See image below.)
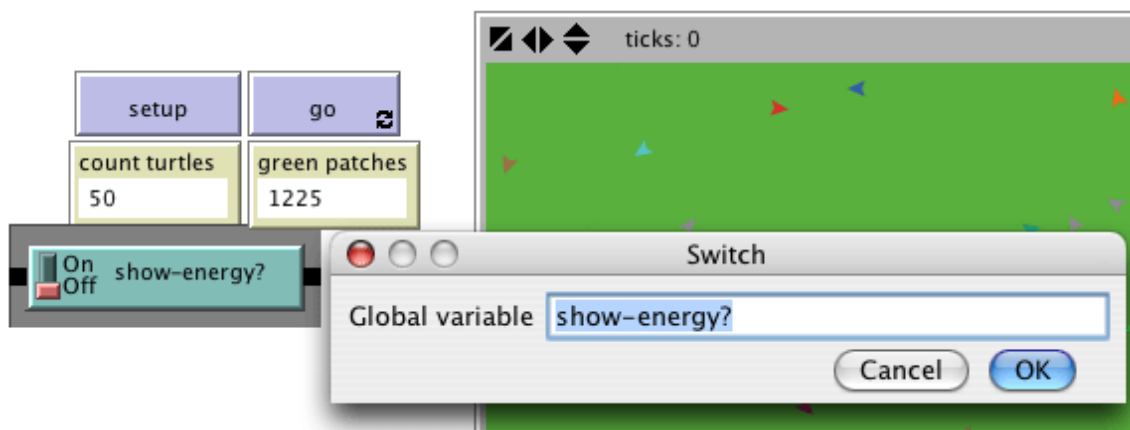
- Now go back to the 'go' procedure using the Code tab with the Toolbar.
- Rewrite the `eat-grass` procedure as follows:

```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy energy + 10
```

```
      ]
    ifelse show-energy?
      [ set label energy ]
      [ set label "" ]
    ]
  end
```

The `eat-grass` procedure introduces the <u>ifelse</u> command. Look at the code carefully. Each turtle, when it runs these new commands, checks the value of `show-energy?` (determined by the switch). If the switch is on, comparison is true and the turtle will run the commands inside the first set of brackets. In this case, it assigns the value for the energy to the label of the turtle. If the comparison is false (the switch is off) then the turtle runs the commands inside the second set of brackets. In this case, it removes the text labels (by setting the label of the turtle to be nothing).

(In NetLogo, a piece of text is called a "string", short for string of characters. A string is a sequence of letters or other characters, written between double quotes. Here we have two double quotes right next to each other, with nothing in between them. That's an empty string. If a turtle's label is an empty string, no text is attached to the turtle.)

- Test this in the Interface tab, by running the model (using the setup and go buttons) switching the `show-energy?` switch back and forth.

When the switch is on, you'll see the energy of each turtle go up each time it eats grass. You'll also see its energy going down whenever it moves.
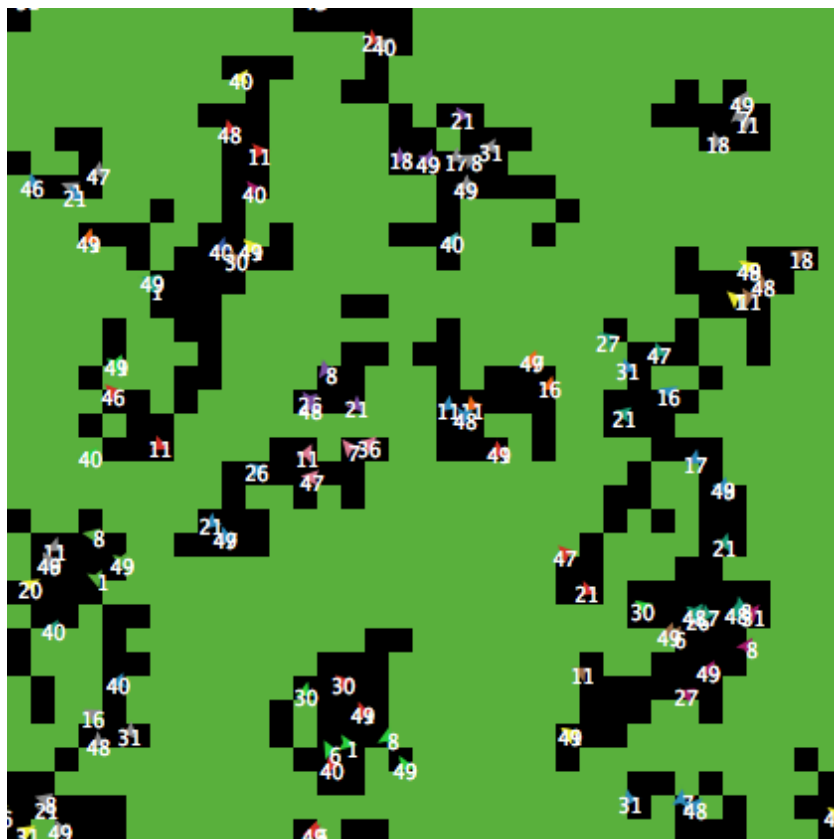
More procedures

Now our turtles are eating. Let's make them reproduce and die, too. And let's make the grass grow back. We'll add all three of these of these behaviors now, by making three separate

procedures, one for each behavior.

- Go to the Code tab.
- Rewrite the `go` procedure as follows:

```
to go
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
  tick
end
```

- Add the procedures for `reproduce`, `check-death`, and `regrow-grass` as shown below:

```
to reproduce
  ask turtles [
    if energy > 50 [
      set energy energy - 50
      hatch 1 [ set energy 50 ]
    ]
  ]
end

to check-death
  ask turtles [
    if energy <= 0 [ die ]
  ]
end

to regrow-grass
  ask patches [
    if random 100 < 3 [ set pcolor green ]
  ]
end
```

Each of these procedures uses the `if` command. Each turtle, when it runs `check-death` it will check to see if its energy is less or equal to 0. If this is true, then the turtle is told to `die` (die is a NetLogo primitive).

When each turtle runs `reproduce`, it checks the value of the turtle's `energy` variable. If it is greater than 50, then the turtle runs the commands inside the first set of brackets. In this case, it decreases the turtle's energy by 50, then 'hatches' a new turtle with an energy of 50. The `hatch` command is a NetLogo primitive which looks like this: `hatch` *number* [*commands*]. This turtle creates *number* new turtles, each identical to its parent, and asks the new turtle(s) that have been hatched to run *commands*. You can use the commands to give the new turtles different colors, headings, or whatever. In our case we run one command. We set the energy for the newly hatched turtle to be 50.

When each patch runs `regrow-grass` it will check to see if a random integer from 0 to 99 is less than 3. If so, the patch color is set to green. This will happen 3% of the time (on average) for each patch, since there are three numbers (0, 1, and 2) out of 100 possible that are less than 3.

- Switch to the Interface tab now and press the setup and go buttons.

You should see some interesting behavior in your model now. Some turtles die off, some new turtles are created (hatched), and some grass grows back. This is exactly what we set out to do.

If you continue to watch your monitors in your model, you will see that the **count turtles** and **green patches** monitors both fluctuate. Is this pattern of fluctuation predictable? Is there a relationship between the variables?

It'd be nice if we had a easier way to track the changes in the model behavior over time. NetLogo allows us to plot data as we go along. That will be our next step.

Plotting

To make plotting work, we'll need to create a plot in the Interface tab and put some commands inside it.

The commands we put in the plots will run automatically when our `setup` procedure calls `reset-ticks` and when our `go` procedure calls `tick`.

- Create a plot by clicking the Add icon on the toolbar, selecting Plot next to it, and clicking on an open spot in the Interface.
- Set its Name to "Totals" (see image below)
- Set the X axis label to "time"
- Set the Y axis label to "totals"
- Change the name of the "default" pen to "turtles".
- Enter `plot count turtles` under Pen Update Commands.
- Press the "Add Pen" button.
- Change the name of the new pen to "grass".
- Enter `plot count patches with [pcolor = green]` under Pen Update Commands.

When you're done, the dialog should look like this:

Totals

10

totals

0

	Plot

Name Totals

X axis label time   X min 0   X max 10

Y axis label totals   Y min 0   Y max 10

☑ Auto Scale?   ☐ Show legend?

▸ Plot setup commands

▸ Plot update commands

Plot pens

Color	Pen Name	Pen Update Commands	
	turtles	plot count turtles	✏ 🗑
	grass	plot count patches with [pcolor = green]	✏ 🗑

Add Pen

Cancel   Help   Apply   OK
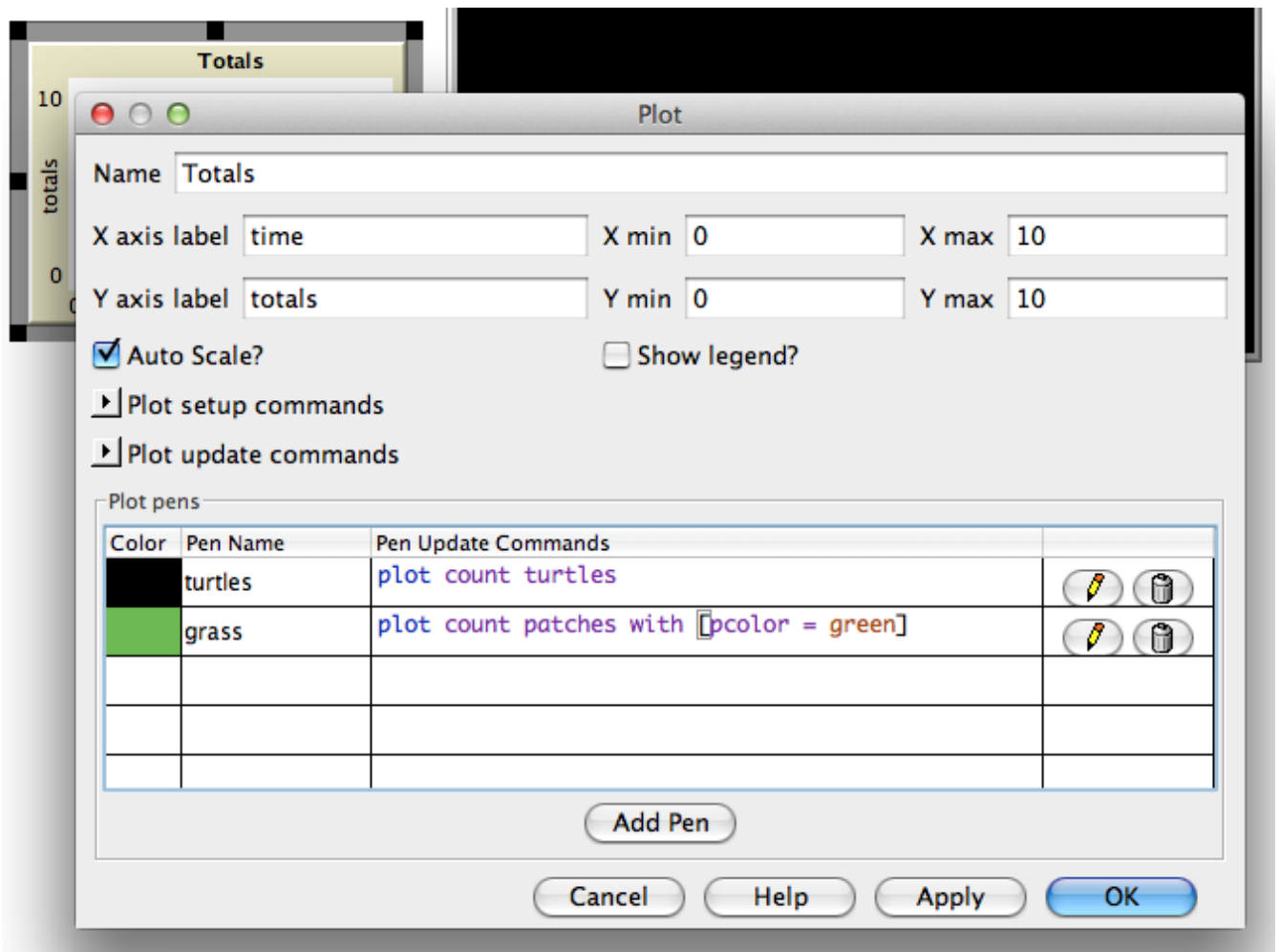
- Press OK in the Plot dialog to finish editing.

Note that when you create the plot you can also set the minimum and maximum values on the X and Y axes. You'll want to leave the "Auto Scale" checkbox checked, so that if anything you plot exceeds the minimum and maximum values for the axes, the axes will automatically grow so you can see all the data.

Note that we used the `plot` command to add the next point to a plot. This command moves the current plot pen to the point that has an X coordinate equal to 1 greater than the previously plotted X coordinate and a Y coordinate equal to the value given in the plot command (in the first case, the number of turtles, and in the second case, the number of green patches). As the pens move they each draw a line.

- Setup and run the model again.

You can now watch the plot being drawn as the model is running. Your plot should have the general shape of the one below, though your plot might not look exactly the same.

Remember that we left "Auto Scale?" on. This allows the plot to readjust itself when it runs out of room.
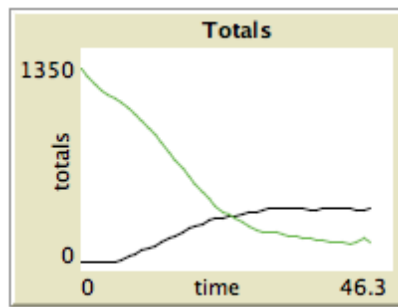
If you forget which pen is which, you can edit the plot and check the "Show legend?" checkbox.

You might try running the model several times to see what aspects of the plot are the same and which are different from run to run.

Tick counter

To make comparisons between plots from one model run and another, it is often useful to do the comparison for the same length of model run. Learning how to stop or start an action at a specific time can help make this happen by stopping the model at the same point each model run. Keeping track of how many times the `go` procedure is run is a useful way to cue these actions. That's what the tick counter does.

You're already using the tick counter in your model, with the `reset-ticks` and `tick` commands, which also trigger plotting.

You can also use the tick counter for other things, such as to set a limit on the total length of a run.

- Change the `go` procedure:

```
to go
  if ticks >= 500 [ stop ]
  move-turtles
  eat-grass
  check-death
  reproduce
  regrow-grass
  tick
end
```

- Now setup and run the model.

The graph and model won't keep running forever. They should stop automatically when the tick counter in the Interface tab's toolbar reaches 500.

The `tick` command advances the tick counter by 1. `ticks` is a reporter which reports the current value of the tick counter. `reset-ticks`, in your `setup` procedure, takes care of restarting the tick counter at 0 when a new run is set up and ready to begin.

Some more details

First, instead of always using 100 turtles, you can have a varying number of turtles.

- Create a slider named "number": click the Add icon on the toolbar, select Slider next to it, and click on an open spot in the interface.
- Try changing the minimum and maximum values in the slider.
- Then inside of `setup-turtles`, instead of `create-turtles 100` you can type:

```
to setup-turtles
  create-turtles number [ setxy random-xcor random-ycor ]
end
```

Test this change and compare how having more or fewer turtles initially affect the plots over time.

Second, wouldn't it be nice to adjust the energy the turtles gain and lose as they eat grass and reproduce?

- Make a slider called `energy-from-grass`.
- Make another slider called `birth-energy`.
- Then, inside of `eat-grass`, make this change:

```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy (energy + energy-from-grass)
    ]
    ifelse show-energy?
      [ set label energy ]
      [ set label "" ]
  ]
end
```

- And, inside of `reproduce`, make this change:

```
to reproduce
  ask turtles [
    if energy > birth-energy [
      set energy energy - birth-energy
      hatch 1 [ set energy birth-energy ]
    ]
  ]
end
```

Finally, what other slider could you add to vary how often grass grows back? Are there rules you can add to the movement of the turtles or to the newly hatched turtles that happen only at certain times? Try writing them.

What's next?

So now you have a simple model of an ecosystem. Patches grow grass. Turtles wander, eat the grass, reproduce, and die.

You have created an interface containing buttons, sliders, switches, monitors, and a plot. You've even written a series of procedures to give the turtles something to do.

That's where this tutorial leaves off.

If you'd like to look at some more documentation about NetLogo, the Interface Guide section of the manual walks you through every element of the NetLogo interface in order and explains its function. For a detailed description and specifics about writing procedures, refer to the Programming Guide. All of the primitives are listed and described in the NetLogo Dictionary.

Also, you can continue experimenting with and expanding this model if you'd like, experimenting with different variables and behaviors for the agents.

Alternatively, you may want to revisit the first model in the tutorial, Wolf Sheep Predation. This is the model you used in Tutorial #1. In the Wolf Sheep Predation model, you saw sheep move around, consume resources that are replenished occasionally (grass), reproduce under certain conditions, and die if they ran out of resources. But that model had another type of creature moving around -- wolves. The addition of wolves requires some additional procedures and some new primitives. Wolves and sheep are two different "breeds" of turtle. To see how to use breeds, study Wolf Sheep Predation.

Alternatively, you can look at other models (including the many models in the Code Examples section of the Models Library) or even go ahead and build your own model. You don't even have to model anything. It can be interesting just to watch patches and turtles forming patterns, to try to create a game to play, or whatever.

Hopefully you have learned some things, both in terms of the NetLogo language and about how to go about building a model. The entire set of procedures that was created above is shown below.

Appendix: Complete code

The complete model is also available in NetLogo's Models Library, in the Code Examples section. It's called "Tutorial 3".

Notice that this listing is full of "comments", which begin with semicolons. Comments let you mix an explanation the code right in with the code itself. You might use comments to help others understand your model, or you might use them as notes to yourself.

In the Code tab, comments are gray, so your eyes can pick them out easily.

```
turtles-own [energy] ;; for keeping track of when the turtle is ready
                     ;; to reproduce and when it will die

to setup
  clear-all
  setup-patches
  setup-turtles
  reset-ticks
end

to setup-patches
  ask patches [ set pcolor green ]
end

to setup-turtles
  create-turtles number     ;; uses the value of the number slider to create turtles
  ask turtles [ setxy random-xcor random-ycor ]
end

to go
  if ticks >= 500 [ stop ]  ;; stop after 500 ticks
  move-turtles
  eat-grass
  check-death
  reproduce
  regrow-grass
  tick                       ;; increase the tick counter by 1 each time through
```

```
    end

to move-turtles
  ask turtles [
    right random 360
    forward 1
    set energy energy - 1  ;; when the turtle moves it looses one unit of energy
  ]
end

to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
            ;; the value of energy-from-grass slider is added to energy
      set energy energy + energy-from-grass
    ]
  ifelse show-energy?
    [ set label energy ] ;; the label is set to be the value of the energy
    [ set label "" ]     ;; the label is set to an empty text value
  ]
end

to reproduce
  ask turtles [
    if energy > birth-energy [
      set energy energy - birth-energy  ;; take away birth-energy to give birth
      hatch 1 [ set energy birth-energy ] ;; give this birth-energy to the offspring
    ]
  ]
end

to check-death
  ask turtles [
    if energy <= 0 [ die ] ;; removes the turtle if it has no energy left
  ]
end

to regrow-grass
  ask patches [ ;; 3 out of 100 times, the patch color is set to green
    if random 100 < 3 [ set pcolor green ]
  ]
end
```

Interface Guide

This section of the manual explains the function of each element in NetLogo's user interface.

In NetLogo, you have the choice of viewing models found in the Models Library, adding to existing models, or creating your own models. The NetLogo interface was designed to meet all these needs.

The interface can be divided into two main parts: NetLogo menus, and the main NetLogo window. The main window is divided into tabs.

- Menus
- Tabs
- Interface Tab
  - Working with interface elements
  - The 2D and 3D views
  - Command Center
  - Plots
  - Sliders
  - Agent monitors
- Code tab
- Includes menu
- International usage

Menus

On a Mac, if you are running the NetLogo application, the menu bar is located at the top of the screen. On other platforms, the menu bar is found at the top of the NetLogo window.
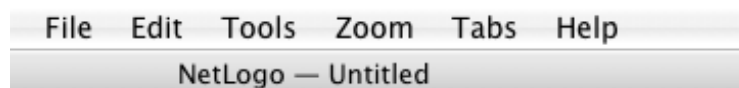
The functions available from the menus in the menubar are listed in the following chart.

Chart: NetLogo menus

File		
	New	Starts a new model.
	Open	Opens any NetLogo model on your computer.
	Recent Files	Re-opens any previously model opened with "File -> Open"
	Models Library	A collection of demonstration models.
	Save	Save the current model.

	Save As	Save the current model using a different name.
	Save As Applet	Saves a web page, in HTML format, that has your model embedded in it as a Java "applet". (Use of this feature is no longer recommended; Details here.)
	Upload to Modeling Commons	Uploads the model to Modeling Commons. See Modeling Commons
	Print	Sends the contents of the currently showing tab to your printer.
	Export World	Saves all variables, the current state of all turtles and patches, the drawing , the plots, the output area and the random state information to a file.
	Export Plot	Saves the data in a plot to a file.
	Export All Plots	Saves the data in all the plots to a file.
	Export View	Save a picture of the current view (2D or 3D) to a file (in PNG format).
	Export Interface	Save a picture of the current Interface tab. (in PNG format)
	Export Output	Save the contents of the output area or the output section of the command center to a file.
	Import World	Load a file that was saved by Export World.
	Import Patch Colors	Load an image into the patches; see the import-pcolors command.
	Import Patch Colors RGB	Load an image into the patches using RGB colors; see the import-pcolors-rgb command.
	Import Drawing	Load an image into the drawing, see the import-drawing command.
	Import HubNet Client Interface	Load the interface from another model into the HubNet Client Editor.
	Quit	Exits NetLogo. (On a Mac, this item is on the NetLogo menu instead.)

Edit

	Undo	Undo last text editing action you performed.
	Redo	Redo last undo action you performed.
	Cut	Cuts out or removes the selected text and temporarily saves it to the clipboard.
	Copy	Copies the selected text.
	Paste	Places the clipboard text where cursor is currently located.
	Delete	Deletes selected text.
	Select All	Select all the text in the active window.
	Find	Finds a word or sequence of characters within the Info or Code tabs.
	Find Next	Find the next occurrence of the word or sequence you last used Find with.
	Shift Left / Shift Right	Used in the Code tab to change the indentation level of code.
	Comment / Uncomment	Used in the Code tab to add or remove semicolons from code (semicolons are used in NetLogo code to indicate comments).
	Snap to Grid	When enabled new widgets stay on a 5 pixel grid so it is easier to line them up. (Note: this feature is disabled when zoomed in or out.)

Tools

	Halt	Stops all running code, including buttons and the command center. (Warning: since the code is interrupted in the middle of whatever it was doing, you may get unexpected results if you try to continue running the model without first pressing "setup" to start the model run over.)
	Globals Monitor	Displays the values of all global variables.

	Turtle Monitor	Displays the values of all of the variables in a particular turtle. You can can also edit the values of the turtle's variables and issue commands to the turtle. (You can also open a turtle monitor via the View; see the View section below.)
	Patch Monitor	Displays the values of all of the variables in a particular patch. You can can also edit the values of the patch's variables and issue commands to the patch. (You can also open a patch monitor via the View; see the View section below.)
	Link Monitor	Displays the values of all of the variables in a particular link. You can can also edit the values of the link's variables and issue commands to the link. (You can also open a link monitor via the View; see the View section below.)
	Close All Agent Monitors	Closes all open agent monitor windows.
	Close Dead Agent Monitors	Closes all open agent monitor windows targeting dead agents.
	Hide/Show Command Center	Makes the command center visible or invisible. (Note that the command center can also be shown or hidden, or resized, with the mouse.)
	3D View	Opens the 3D view. See the Views section for more information.
	Color Swatches	Opens the Color Swatches. See the Color Section of the Programming Guide for details.
	Turtle Shapes Editor	Draw turtle shapes. See the Shapes Editor Guide for more information.
	Link Shapes Editor	Draw link shapes. See the Shapes Editor Guide for more information.
	BehaviorSpace	Runs the model over and over with different settings. See the BehaviorSpace Guide for more information.
	System Dynamics Modeler	Opens the System Dynamics Modeler. See the System Dynamics Modeler Guide for more details.
	HubNet Client Editor	Opens the HubNet Client Editor. See the HubNet Authoring Guide for more details.

	HubNet Control Center	Disabled if no HubNet activity is open. See the HubNet Guide for more information.

Zoom

	Larger	Increase the overall screen size of the model. Useful on large monitors or when using a projector in front of a group.
	Normal Size	Reset the screen size of the model to the normal size.
	Smaller	Decrease the overall screen size of the model.

Tabs		This menu offers keyboard shortcuts for each of the tabs. (On Macs, it's Command 1 through Command 3. On Windows, it's Control 1 through Control 3.)

Help

	About NetLogo	Information on the current NetLogo version you are running. (On Macs, this menu item is on the NetLogo menu instead.)
	Look Up In Dictionary	Opens a browser with the dictionary entry for the selected command or reporter. (You may also use the F1 key for this.)
	NetLogo User Manual	Opens this manual in a web browser.
	NetLogo Dictionary	Opens the NetLogo Dictionary in a web browser.
	NetLogo Users Group	Opens the NetLogo Users Group site in a web browser.
	Donate	Opens the NetLogo donation page in a web browser.

Tabs

At the top of NetLogo's main window are three tabs labeled "Interface", "Info" and "Code" .
Only one tab at a time can be visible, but you can switch between them by clicking on the tabs
at the top of the window.
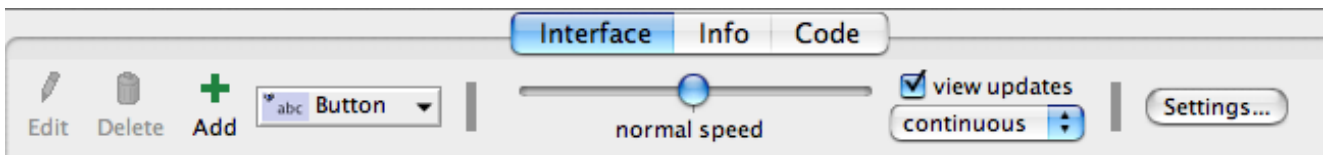
Right below the row of tabs is a toolbar containing a row of controls. The controls available vary from tab to tab.

Interface Tab

The Interface tab is where you watch your model run. It also has tools you can use to inspect and alter what's going on inside the model.

When you first open NetLogo, the Interface tab is empty except for the view, where the turtles and patches appear, and the Command Center, which allows you to issue NetLogo commands.

Working with interface elements

The toolbar on the Interface tab contains buttons that let you edit, delete, and create items in the Interface tab and a menu that lets you select different interface items (such as buttons and sliders).
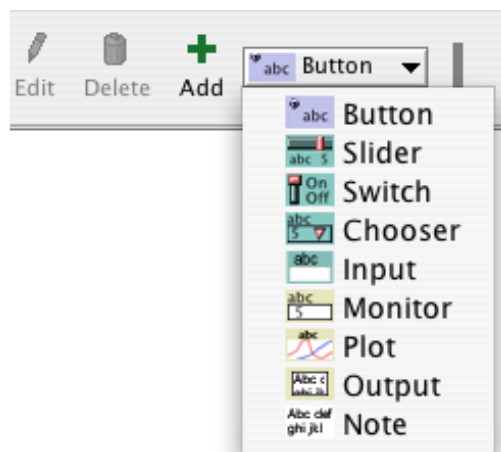
The buttons in the toolbar are described below.

Adding: To add an interface element, select the element from the drop down menu. Note that the Add button stays down. Then click on the white area below the toolbar. (If the menu is already showing the right type, you can just press the Add button instead of using the menu again.)

Selecting: To select an interface element, drag a rectangle around it with your mouse. A gray border with black handles will appear around the element to show it is selected.

Selecting multiple items: You can select multiple interface elements at the same time by including them in the rectangle you drag. If multiple elements are selected, one of them is the "key" item, which means that if you use the "Edit" or "Delete" buttons on the Interface Toolbar, only the key item is affected. The key item's border is darker gray.

Unselecting: To unselect all interface elements, click the mouse on the white background of the Interface tab. To unselect an individual element, right-click the element and choose "Unselect" from the popup menu.

Editing: To change the characteristics of an interface element, select the element, then press the "Edit" button on the Interface toolbar. You may also double click the element once it is selected. A third way to edit an element is to right-click it and choose "Edit" from the popup menu. If you use this last method, it is not necessary to select the element first.

Moving: Select the interface element, then drag it with your mouse to its new location. If you hold down the shift key while dragging, the element will move only straight up and down or straight left and right.

Resizing: Select the interface element, then drag the black "handles" in the selection border.

Deleting: Select the element or elements you want to delete, then press the "Delete" button on the Interface toolbar. You may also delete an element by right-clicking it and choosing "Delete" from the popup menu. If you use this latter method, it is not necessary to select the element first.

To learn more about the different kinds of interface elements, refer to the chart below.

Chart: Interface Toolbar

Icon & Name	Description
Button	A button is either *once* or *forever*. When you click on a once button, it executes its instructions once. The forever button executes the instructions over and over, until you click on the button again to stop the action. If you have assigned an action key to the button, pressing the corresponding keyboard key will act just like a button press when the button is in focus. Buttons with action keys have a letter in the upper right corner of the button to show what the action key is. If the keyboard focus is in another interface element such as the Command Center, pressing the action key won't trigger the button. The letter in the upper right hand corner of the button will be dimmed in this situation. To enable action keys, click in the white background of the Interface tab.
Slider	Sliders are global variables, which are accessible by all agents. They are used in models as a quick way to change a variable without having to recode the procedure every time. Instead, the user moves the slider to a value and observes what happens in the model.
Switch	Switches are a visual representation for a true/false global variable. You may set the variable to either on (true) or off (false) by flipping the switch.
Chooser	Choosers let you choose a value for a global variable from a list of choices, presented in a drop down menu. The choices may be strings, numbers, booleans, or lists.
Input	Input Boxes are global variables that contain strings or numbers. The model author chooses what types of values you can enter. Input boxes can be set to check the syntax of a string for commands or reporters. Number input boxes read any type of constant number reporter which allows a more open way to express numbers than a slider. Color input boxes offer a NetLogo color chooser.
Monitor	Monitors display the value of any reporter. The reporter could be a variable, a complex reporter, or a call to a reporter procedure. Monitors automatically update several times per second.
Plot	Plots show data the model is generating.

ᴬᵇᶜ Output	The output area is a scrolling area of text which can be used to create a log of activity in the model. A model may only have one output area.
ᴬᵇᶜ ᵈᵉᶠ Note	Notes lets you add informative text labels to the Interface tab. The contents of notes do not change as the model runs.

The other controls in the Interface toolbar allow you to control the view updates and various other model properties.
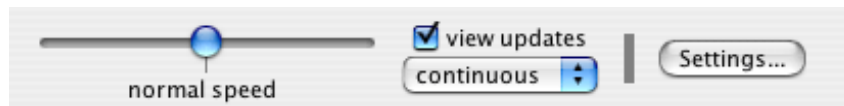
- The slider lets you control how fast the model runs. Slower can be valuable since some models run so fast they're hard to follow. You can also fast-forward the model by moving the slider to the right, reducing the frequency of view updates.
- The view updates checkbox controls whether view updates happen at all.
- The update mode menu allows you to switch between continuous and tick-based updates.
- The "Settings..." button allows you to change model settings.

"Continuous" updates means that NetLogo updates (that is, redraws) the view many times a second, regardless of what is going on in the model. "Tick-based" updates means that the view only updates when the tick counter advances. (For a fuller discussion of view updates, see the Programming Guide.)

The 2D and 3D views

The large black square in the Interface tab is the 2D view. It's a visual representation of the NetLogo world of turtles and patches. Initially it's all black because the patches are black and there are no turtles yet. You can open the 3D View, an alternate visual representation of the world, by clicking on the "3D" button in the View Control Strip.

The three sets of black arrows in the upper left let you change the size of the world. When the origin is centered the world will grow in increments of two, adding one to the maximum and subtracting one from the minimum. If one of the edges is set to 0 the world will grow by one in the other direction to keep the origin along the edge. If the origin is at a custom location the black arrows will be disabled.

There are a number of settings associated with a view, accessed through the "Settings..." button.

Notice that the control strip in the 3D View combines the ticks counter from the 2D view control strip and the controls from the right portion of the interface toolbar.
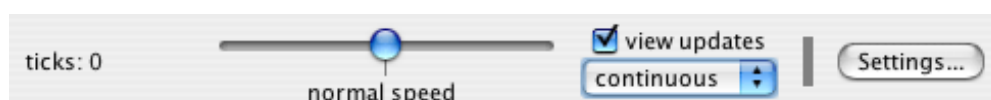
Here are the settings for the View (accessible by editing the View, or by pressing the "Settings..." button in the Interface Toolbar):
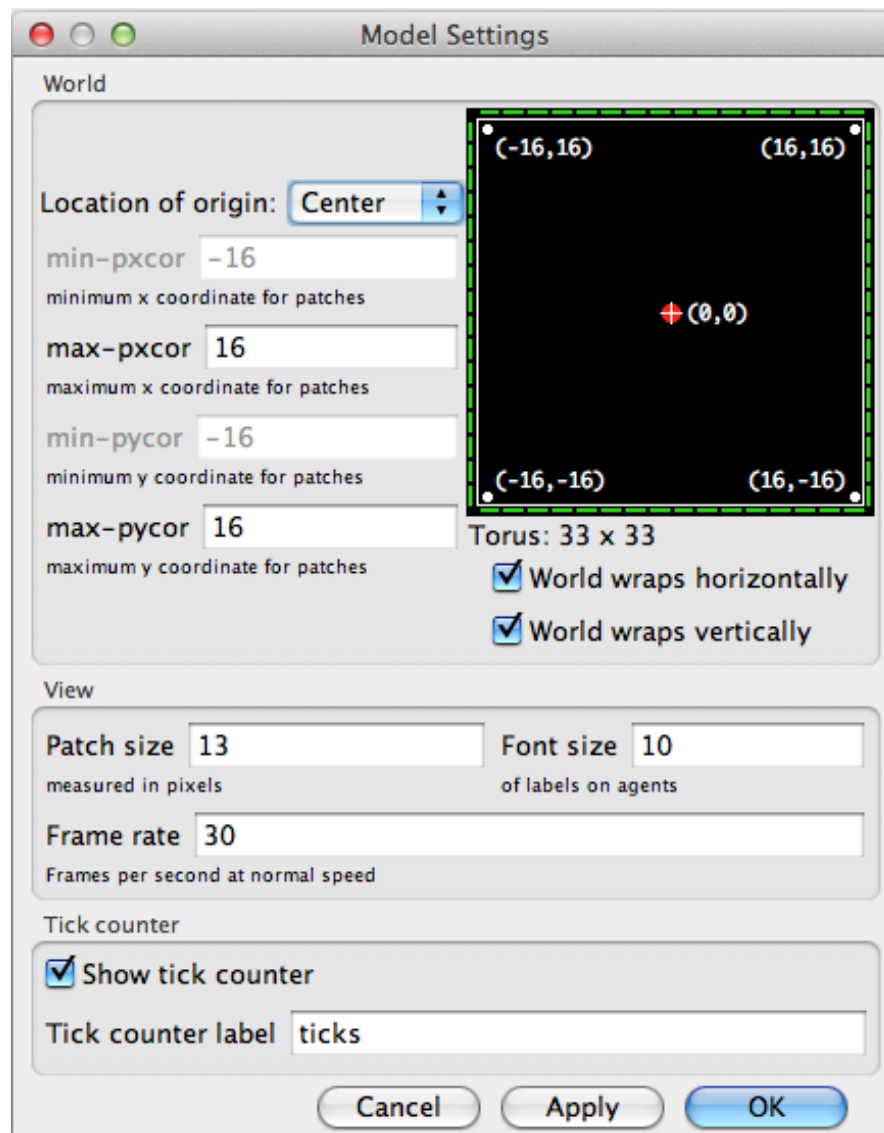
Notice that the settings are broken up into three groups. There are world, view, and ticks counter settings. World settings affect the properties of the world that the turtles live in (changing them may require resetting the world). View and tick counter settings only affect the appearance, changing them will not affect the outcome of the model.

The world settings allow you to define the boundaries and topology of the world. At the top of the left side of the world panel you can choose a location for the origin of the world either "Center", "Corner", "Edge", or "Custom". By default the world has a center configuration where (0,0) is at the center of the world and you define the number of patches from the center to the right and left boundaries and the number of patches from the center to the top and bottom boundaries. For example: if you set Max-Pxcor = 10, then Min-Pxcor will automatically be set to -10, thus there are 10 patches to the left of the origin and 10 patches to the right of patch 0 0, for a total of 21 patches in each row.

A Corner configuration allows you to define the location of the origin as one of the corners of the world, upper left, upper right, lower left, or lower right. Then you define the far boundary in the x and y directions. For example if you choose to put the origin in the lower left corner of the world you define the right and top (positive) boundaries.

Edge mode allows you to place the origin along one of the edges (x or y) then define the far boundary in that direction and both boundaries in the other. For example if you select edge mode along the bottom of the world, you must also define the top boundary, as well as the left and the right.

Finally, Custom mode allows you to place the origin at any location in the world, though patch

0 0 must still exist in the world.

As you change the settings you will notice that the changes you make are reflected in the preview on the right side of the panel which shows the origin and the boundaries. The width and height of the world are displayed below the preview.

Also below the preview there are two checkboxes, the world wrap settings. These allow you to control the topology of the world. Notice when you click the check boxes the preview indicates which directions allow wrapping, and the name of the topology is displayed next to the world dimensions. See the Topology section of the Programming Guide for more information.

The view settings allow you to customize the look of the view without changing the world. Changing view settings will never force a world reset. To change the size of the 2D View adjust the "Patch Size" setting, measured in pixels. This does not change the number of patches, only how large the patches appear in the 2D View. (Note that the patch size does not affect the 3D View, as you can simply make the 3D View larger by making the window larger.)

The font size setting lets you control the size of turtle, patch, and link labels.

The frame rate controls how often the view gets updated. This can have a dramatic effect on the default speed at which a model runs. For more details, see the view updates section of the Programming Guide.

The "Smooth edges" checkbox controls the use of anti-aliasing in the 3D view only and only appears when editing from the 3D view. Unchecking it makes lines appear more jagged but may speed up rendering.

Tick counter settings control the appearance of the tick counter which is visible (or not) in the view control strip.

Turtle, patch and link monitors are easily available through the View, just right-click on the turtle or patch you want to inspect, and choose "inspect turtle ..." or "inspect patch ..." from the popup menu. You can also watch, follow or ride a turtle by selecting the appropriate item in the turtle sub-menu. (Turtle, patch and link monitors can also be opened from the Tools menu or by using the `inspect` command.)

Some NetLogo models let you interact with the turtles and patches with your mouse by clicking and dragging in the View.

Manipulating the 3D View

At the bottom of the window there are buttons to move the observer, or change the perspective from which you are looking at the world.

A blue cross appears at the current focus point as you adjust these settings. The little blue triangle will always point along the y-axis in the positive direction, so you can orient yourself in case you get lost.
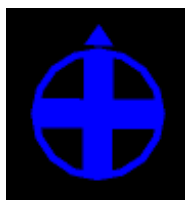
To look at the world from a different angle, press the "rotate" button, then click and drag the mouse. The observer will continue to face the same point as before (where the blue cross is) but its position in the relation to the xy-plane will change.

To move closer or farther away from the world or the agent you are watching, following or riding, press the "zoom" button and drag up and down. Note when you are in follow or ride mode zooming will switch you between ride and follow, since ride is just a special case of follow where the distance at which you are following is 0.

To change the position of the observer without changing the direction it is facing select the "move" button and drag the mouse inside the 3D View while holding down the mouse button.

To allow the mouse position and state to be passed to the model select the "interact" button and it will function just as the mouse does in the 2D view.

To return the observer and focus point to their default positions press the "Reset Perspective" button (or use the `reset-perspective` command).

Fullscreen Mode

To enter fullscreen mode, press the "Full Screen" button, to exit fullscreen mode, press the Esc key.

Note: Fullscreen mode doesn't work on every computer. It depends on your graphics card. See the System Requirements for details.

3D Shapes

Some shapes are automatically mapped to true 3D counterparts in the 3D view. For example, the 2D circle shape becomes a sphere in the 3D view.

Shape name	3D shape
default	3D turtle shape
circle	sphere
dot	small sphere
square	cube
triangle	cone
line	3D line
cylinder	3D cylinder
line-half	3D line-half
car	3D car

All other shapes are based on their 2D forms. If a shape is a rotatable shape, it is assumed to be a top view and it is extruded as if through a cookie cutter and oriented parallel to the xy-plane, as in Ants.
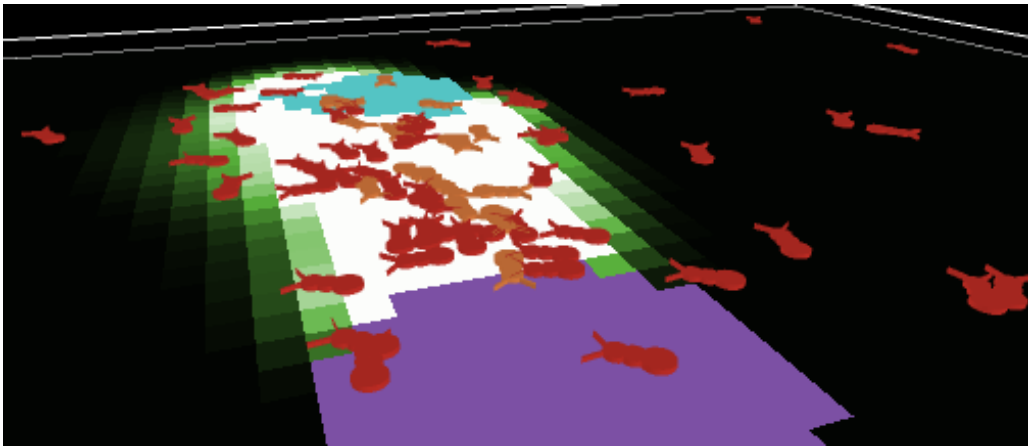
Or, if a shape is non-rotatable, it is assumed to be a side view, and it is drawn always facing the observer, as in Wolf Sheep Predation.
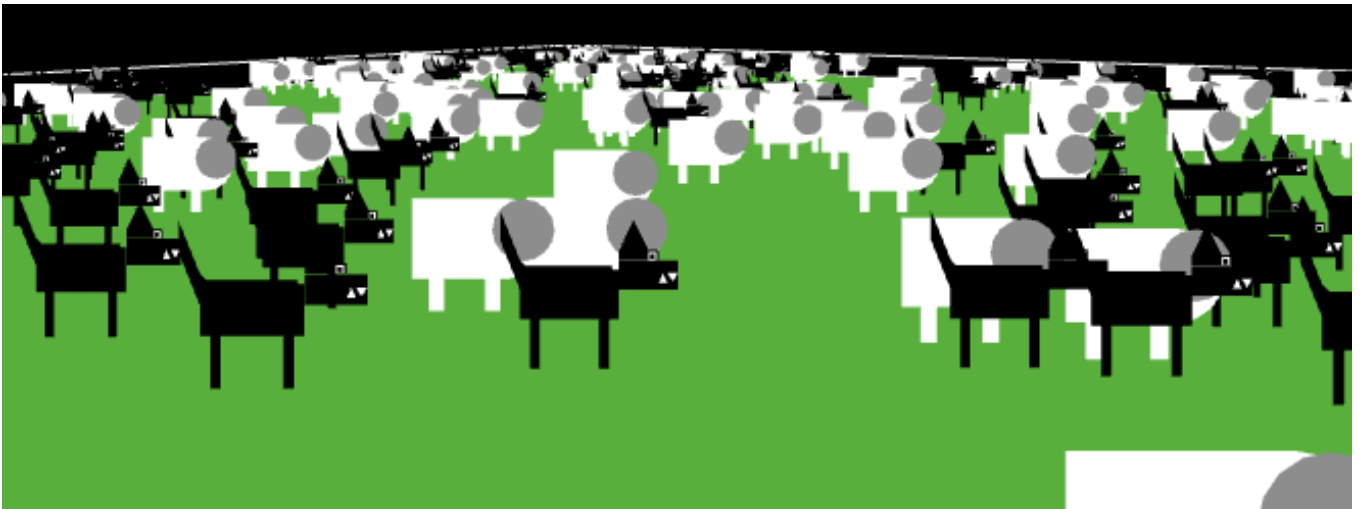
Command Center

The Command Center allows you to issue commands directly, without adding them to the model's procedures. This is useful for inspecting and manipulating agents on the fly.

(Tutorial #2: Commands is an introduction to using commands in the Command Center.)

Let's take a look at the design of the Command Center.

The smaller box, below the large box, is where you type a command. After typing it press the Return or Enter key to run it.

To the left of where you type is a popup menu that initially says "observer>". You can choose either observer, turtles, or patches, to specify which agents run the command you type.

Tip: a quicker way to change agent types is to use the Tab key.

Reporters

If you enter a reporter into the Command Center, the <u>show</u> command will be inserted before it automatically.

Accessing previous commands

After you type a command, it appears in the large scrolling box above the command line. You can use Copy on the Edit menu in this area to copy commands and then paste them elsewhere, such as the Code tab.

You can also access previous commands using the history popup, which is the small downward pointing triangle to the right of where you type commands. Click on the triangle and a menu of previously typed commands appears, so you can pick one to use again.

Tip: a quicker way to access previous commands is with the up and down arrow keys on your keyboard.

Clearing

To clear the large scrolling area containing previous commands and output, click "clear" in the top right corner.

To clear the history popup menu, choose "Clear History" on that menu.

Arranging

You can hide and show the command center using the Hide Command Center and Show Command Center items on the Tools menu.

To resize the command center, drag the bar that separates it from the model interface. Or, click one of the little arrows on the right end of the bar to make the command center either very big or hidden altogether.

To switch between a vertical command center and a horizontal one, click the button with the double-headed arrow, just to the left of "Clear".

Plots

When the mouse pointer is over the white area of a plot, the x and y coordinates of the mouse location appear. (Note that the mouse location might not correspond exactly to any actual data points in the plot. If you need to know the exact coordinates of plotted points, use the Export Plot menu item and inspect the resulting file in another program.)

When you create a plot, as with all widgets, the edit dialog automatically appears.
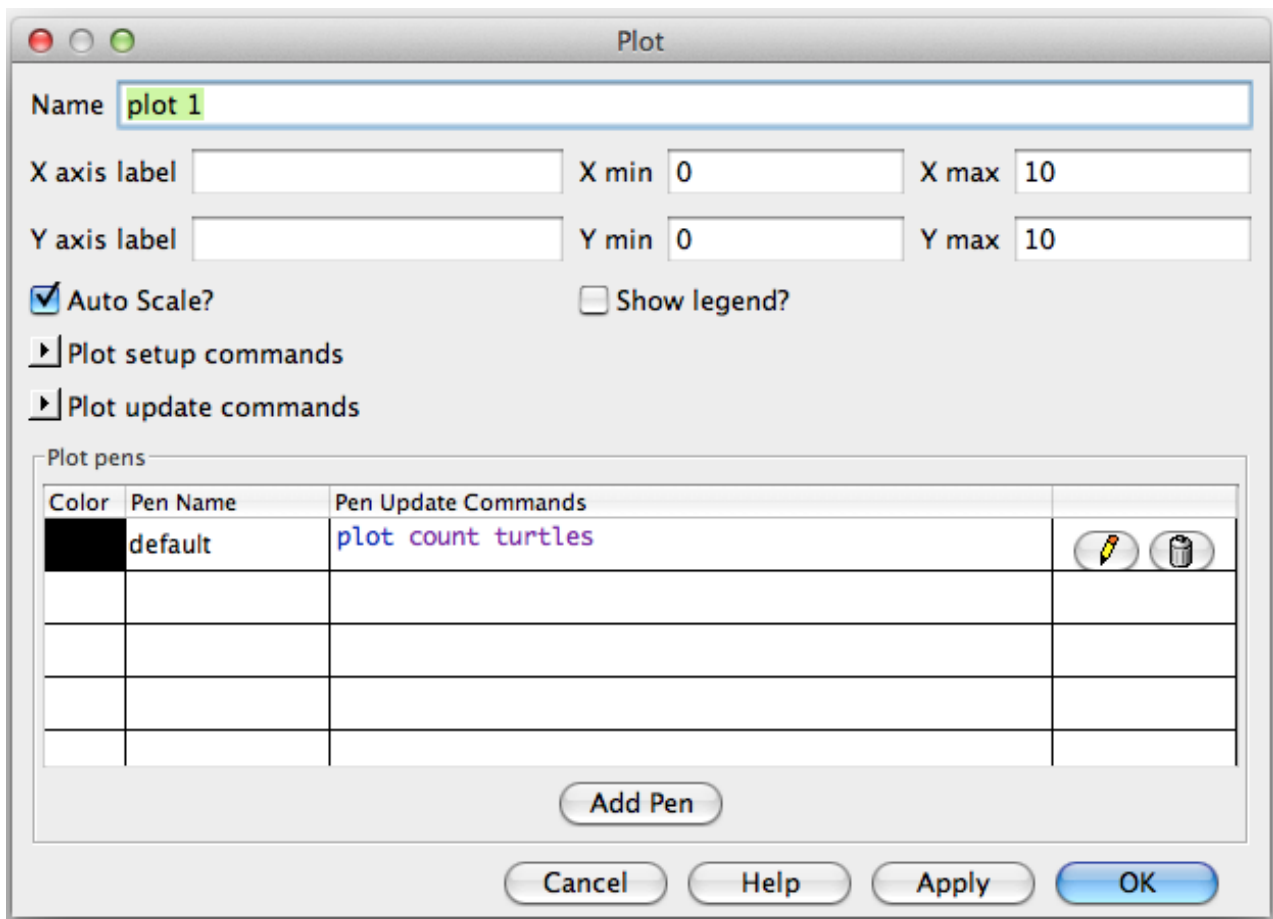
Many of the fields are fairly self-explanatory, such as the name of the plot, labels for the x and y axes, ranges for the axes, and the "Show legend?" checkbox.

If Auto Scale? is checked the x and y changes will automatically readjust as points are added to the plot if they are outside the current range.

Under "Plot setup commands" and "Plot update commands" you can enter commands that will automatically be run at appropriate times. Click the little triangle to open the text box for the commands. Plot commands are explained in more detail in the Plotting section of the Programming Guide.

Plot Pens

In the plot pens section of the dialog, you can create and customize your plot's pens. Each table row represents a pen. By default there is one pen named "default". (You may wish to change it to a name that has meaning in your model.)

To edit the color of a pen click the colored rectangle to the left of the pen's name. This will bring up a dialog that allows you to set the color to one of the NetLogo base hues or a custom color using the color swatches.

To edit the pen's name, double click the name.

In the "Pen Update Commands" column you can enter commands that will be run when `reset-ticks`, `tick`, or `update-plots` commands are run. This is explained in more detail in the Plotting section of the Programming Guide.

The last column has two buttons. Clicking the pencil icon will bring up an edit dialog with additional pen settings. The trash can button deletes the pen.

Plot Pen Advanced Settings

Clicking a pen's edit button will open this dialog:
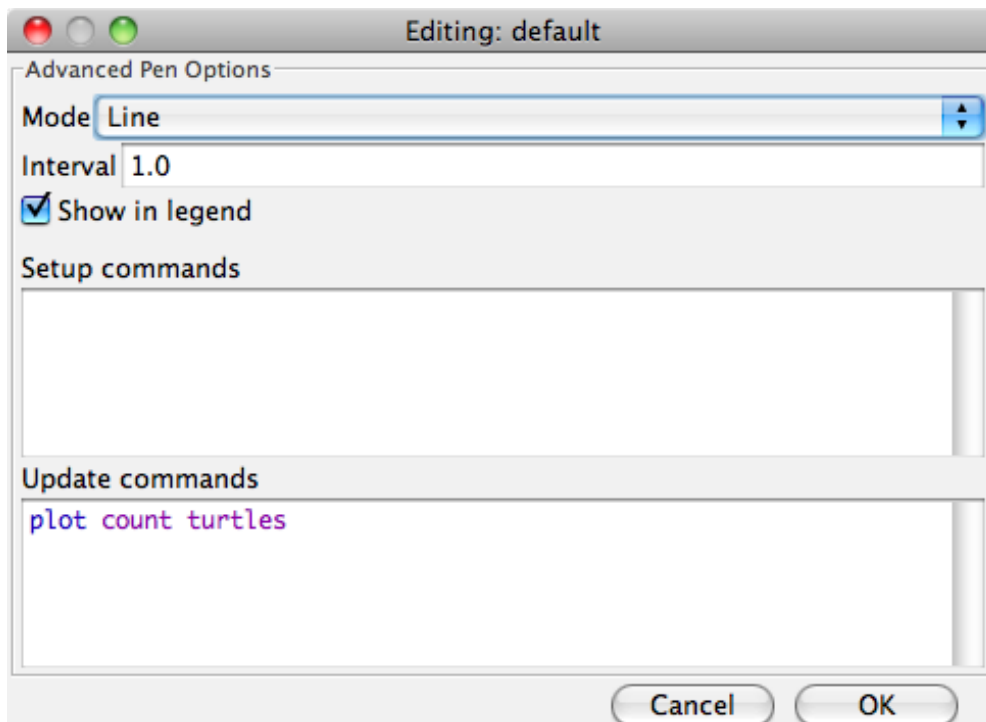
- Mode allows you to change the appearance of the plot pen: line, bar (for a bar chart), or point (a scatter plot of individual points).
- Interval is the amount by which x advances every time you use the `plot` command.
- If the "Show in legend" checkbox is checked the selected pen will be a part of the legend in the upper right hand corner of the plot.
- In the "Setup commands" field you can enter commands that will be run when `reset-ticks` or `setup-plots` runs.
- In the "Update commands" field you can enter commands that will be run when `tick` or `update-plots` runs. This field reappears in the advanced dialog to provide space for editing lengthier sets of commands.

For more detailed information on how each of these features works you can see the Plotting Section of the Programming Guide.

Sliders

A slider has an associated global variable. Moving the slider changes the variable's value.

When you place a slider in the Interface tab the edit dialog automatically opens, as with all widgets. Most of the fields will be familiar. However, it is important to notice the minimum, maximum and increment fields will take any reporter, not just constants. So, for example, you could make the minimum `min-pxcor` and the maximum `max-pxcor` and the slider bounds will automatically adjust when you change the size of the world.
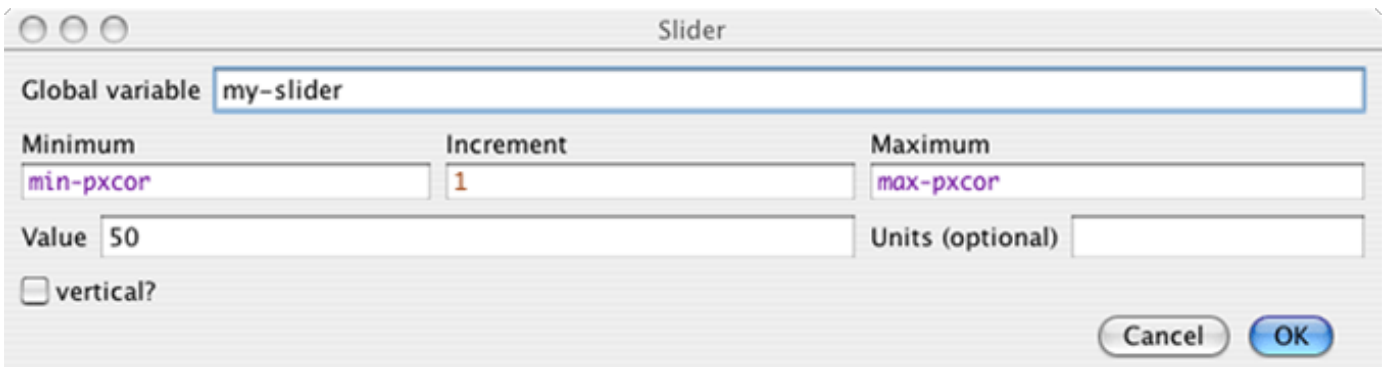
Agent Monitors

Agent monitors display both the values of all the variables for a particular agent and a mini-view that shows the agent and a small area around it.

You can open agent monitors through the Tools menu or the `inspect` command.

You can zoom in or out using the slider beneath the view and you can `watch` the agent in the main view using the `watch-me` button.

Below the slider the current value of each agent variable is displayed. You can enter a new value. It will be as if, for example, the code `set pcolor ...` had run.

pxcor	-13
pycor	-12
pcolor	105
plabel	""
plabel-color	9.9
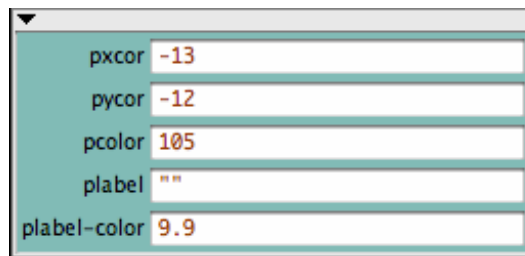
Below the agent variable area there is a mini-command center. Rather than running code as the observer, or talking to all of the turtles, patches, or links, the code entered in this command center is run only by this agent.

```
set pcolor blue
```
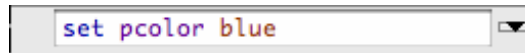
You can close an agent monitor by clicking the box in the upper left corner, or by pressing the Esc key. If you hold down shift while you click the box all open agent monitors will close or you can close all the agent monitors using the "Close All Monitors" option in the Tools Menu.

Code tab

The Code tab is where the code for the model is stored. Commands you only want to use immediately go in the Command Center; commands you want to save and use later, over and over again, are found in the Code tab.


```
globals [grass]   ;; keep track of how much grass there is
;; Sheep and wolves are both breeds of turtle.
breed [sheep a-sheep]   ;; sheep is its own plural, so we use "a-sheep" as the singular.
breed [wolves wolf]
turtles-own [energy]       ;; both wolves and sheep have energy
patches-own [countdown]

to setup
  clear-all
  ask patches [ set pcolor green ]
  ;; check GRASS? switch.
  ;; if it is true, then grass grows and the sheep eat it
  ;; if it false, then the sheep don't need to eat
  if grass? [
    ask patches [
```

To determine if the code has any errors, you may press the "Check" button. If there are any syntax errors, the Code tab will turn red, the code that contains the error will be highlighted, and an error message will appear.
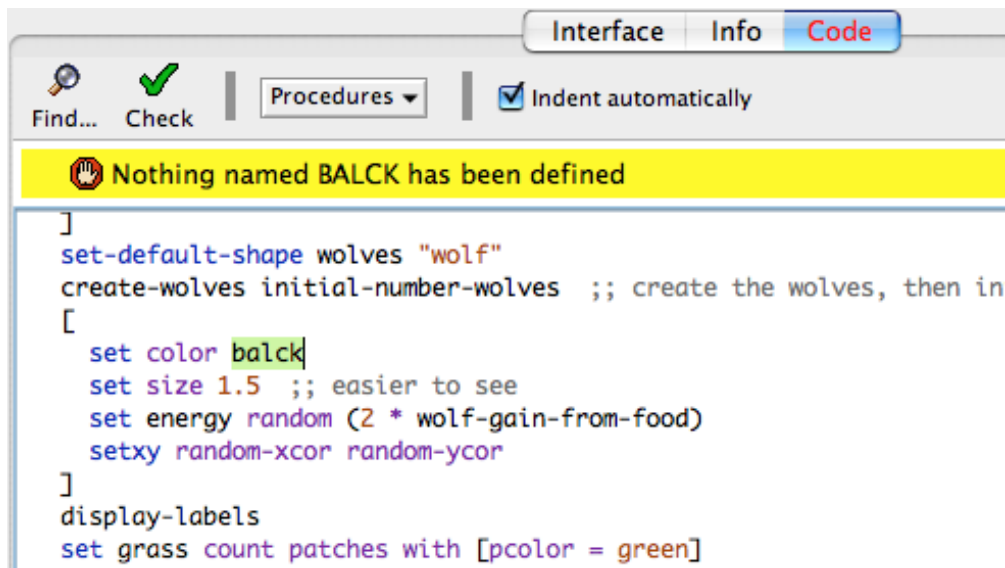
Switching tabs also causes the code to be checked, so if you just switch tabs, pressing the Check button first isn't necessary.

To find a fragment of code in the procedures, click on the "Find" button in the Code toolbar and the Find dialog will appear.
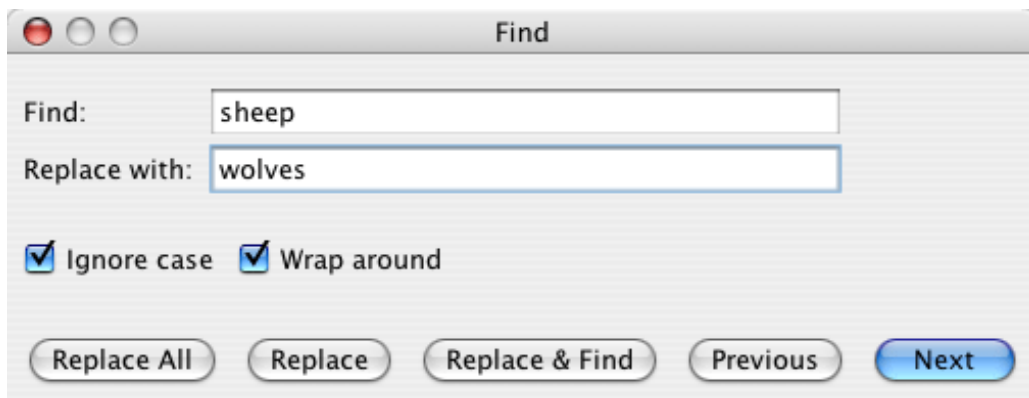
You may enter a word or phrase to find, and optionally also a new word or phrase to replace it with. The "Ignore case" checkbox controls whether the capitalization must be the same to indicate a match.

If the "Wrap around" checkbox is checked, the entire Code tab will be checked for the phrase, starting at the cursor position. When it reaches the end it will return to the top, otherwise only the area from the cursor position to the end of the Code tab will be searched. The "Next" and "Previous" buttons will move down and up to find another occurrence of the search phrase.

"Replace" changes the currently selected phrase with the replace phrase and "Replace & Find" changes the selected phrase and moves to the next occurrence. "Replace All" will change all instances of the find phrase in the search area with the replace phrase.

To find a particular procedure definition in your code, use the "Procedures" popup menu in the Code tab. The menu lists all procedures in alphabetical order.

The "Shift Left", "Shift Right", "Comment", and "Uncomment" items on the Edit menu are used in the Code tab to change the indentation level of your code or add and remove semicolons, which mark comments, from sections of code.

For more information about writing procedures, read Tutorial #3: Procedures and the Programming Guide.

Includes Menu

Caution: The includes facility is new and experimental.

When you add the __includes keyword to a model a menu to the right of the procedures menu appears. This is the includes menu which lists all the NetLogo source files (.nls) included in this file.

You can choose a file name from the menu to open a tab for that file, or you can open a new or existing file using New Source File and Open Source File, respectively.
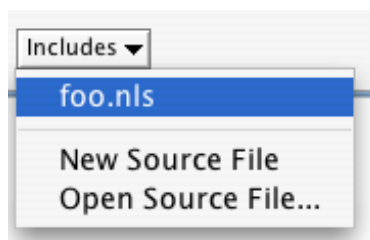
Once you've opened new tabs they become accessible from the Tabs menu, and you can use the keyboard to move from tab to tab (Command + number on Mac, Control + number on other operating systems).
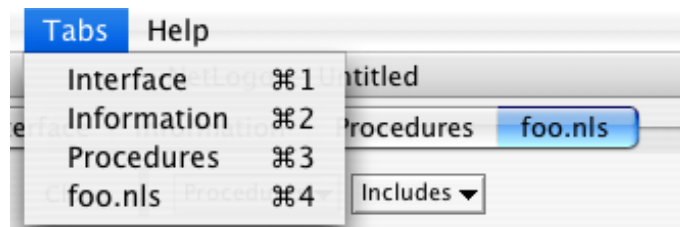
Automatic indentation

When the Indent Automatically checkbox is selected, NetLogo will automatically attempt to align your code in a logically structured format. For example, when you open a set of square brackets "[" (perhaps after an `if` statement), NetLogo will automatically add spaces so that the following lines of code are two spaces further indented than the bracket. When you close the square brackets the closing bracket will be lined up with the matching open bracket.

NetLogo will try to indent the code as you type, but you can also press the tab key anywhere

on any line to ask NetLogo to indent the line immediately. Or, you can select a whole region of code and press the tab key to re-indent all of it.

International Usage

Character sets

NetLogo always saves and loads models in the UTF-8 character encoding, which includes a wide range of international characters.

If you are in a locale other than U.S. English, let us know if you have any trouble using your local character set.

The [Transition Guide](#) has advice on converting models containing international characters from earlier NetLogo versions.

Languages

Most of NetLogo's GUI, and some of its error messages, are now "internationalized". This means that it is now possible to display NetLogo in different languages.

We say "possible" here because the current release of NetLogo only supports English, Spanish, Chinese, and Russian.

The work on internationalization is not complete. We are asking for help from the user community in helping us localize items such as the menus and error messages.

Default language

By default, NetLogo uses the same language your operating system is set to, if available. (If unavailable, you get English.)

You can record a preference for a different language by typing `__change-language` (that's two underscore characters at the start) into the Command Center. A dialog will open allowing you to choose from the supported languages. Once a new language is chosen you will have to restart NetLogo.

Support for translators

We hope NetLogo will become available in many different languages. If you would like to translate NetLogo to your language, see [this wiki page](#) for instructions.

Info Tab

The Info tab provides an introduction to a model. It explains what system is being modeled, how the model was created, and and how to use it. It may also suggest things to explore and ways to extend the model, or call your attention to particular NetLogo features the model uses.
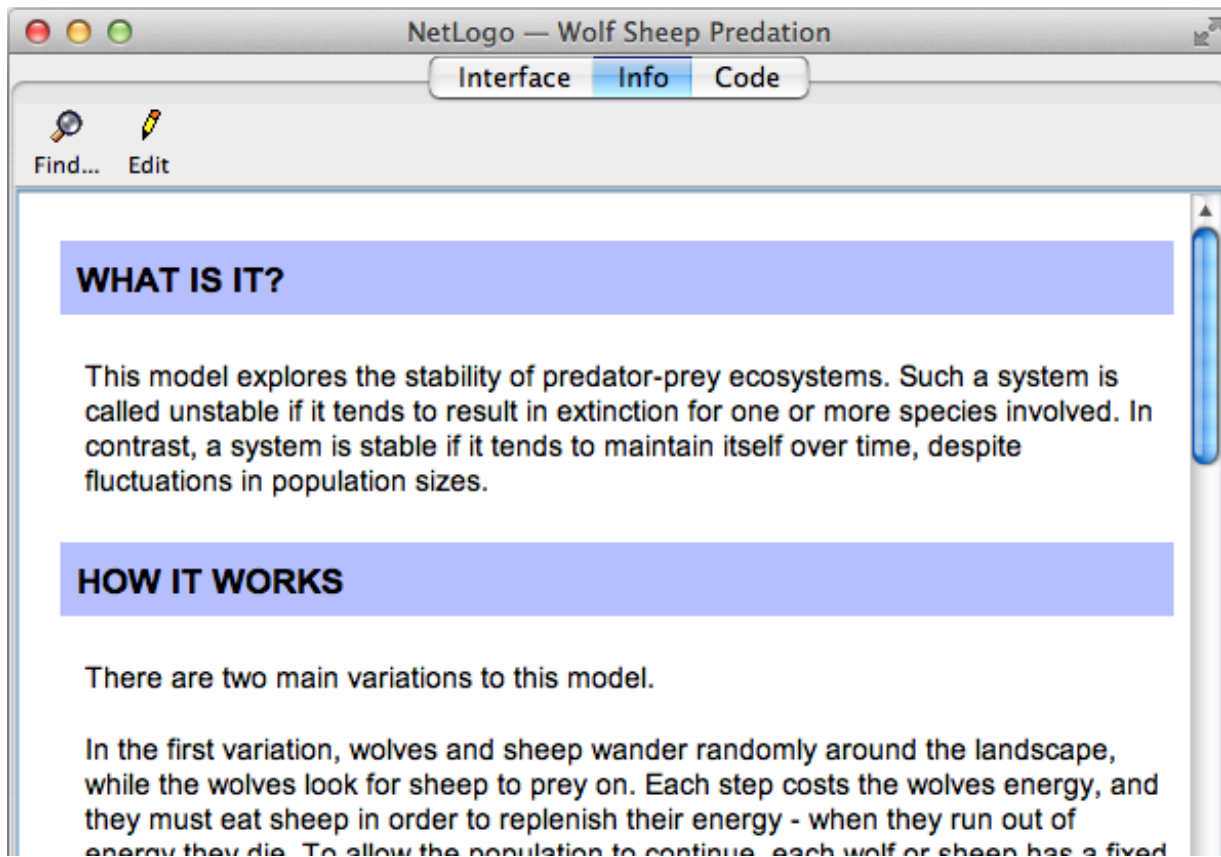
You may wish to read the Info tab before starting a model.

Editing

The normal, formatted view of the Info tab is not editable. To make edits, click the "Edit" button. When done editing, click the "Edit" button again.
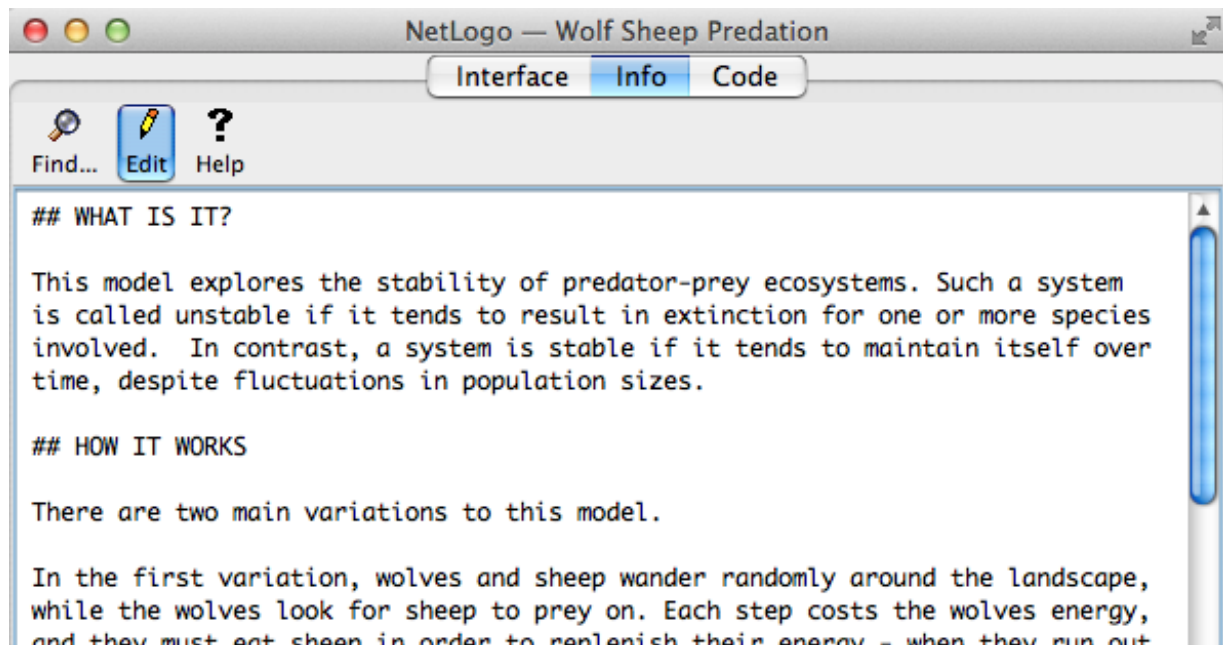
You edit the Info tab as unformatted plain text. When you're done editing, the plain text you entered is displayed in a more attractive format.

To control how the formatted display looks, you use a "markup language" called Markdown. You may have encountered Markdown elsewhere; it is used on a number of web sites. (There are other markup languages in use on the web; for example, Wikipedia used a markup language called MediaWiki. Markup languages differ in details.)

The remainder of this guide is a tour of Markdown.

- Headings
- Paragraphs
- Italicized and bold text
- Ordered lists
- Unordered lists
- Links
- Images
- Block quotations
- Code
- Code blocks
- Superscripts and subscripts
- Notes on usage
- Other features

Headings

A heading begins with one or more hash marks (#). First level headings get one hash, second level headings get two, and so on up to four levels.

Input

```
# First-level heading
## Second-level heading
### Third-level heading
#### Fourth-level heading
```

Paragraphs

```
This is a paragraph. There are no spaces before the word 'This'.

This is another paragraph. The first line has two sentences.
The entire paragraph has two lines and three sentences.

Line breaks in the input,
Make line breaks in the output,
Like this.
```

This is a paragraph. There are no spaces before the word 'This'.

This is another paragraph. The first line has two sentences. The entire paragraph has two lines and three sentences.

Line breaks in the input,
Make line breaks in the output,
Like this.

Italicized and bold text

```
For italics, surround  text with underscores:
_hello, world_.

For bold, surround text with two asterisks:
**hello, world**.

You can also combine them:
_**hello**_  and **_goodbye_**
```

For italics, surround text with underscores:
hello, world.

For bold, surround text with two asterisks:
hello, world.

You can also combine them:
hello and ***goodbye***

Ordered lists

```
We are about to start an ordered list.
```

```
   1. Ordered lists are indented 2 spaces.
     1. Subitems are indented 2 more spaces (4 in all).
   2. The next item in the list starts with the next number.
   3. And so on...
```

Formatted

We are about to start an ordered list.

1. Ordered lists are indented 2 spaces.
   1. Subitems are indented 2 more spaces (4 in all for a second level item).
2. The next item in the list starts with the next number.
3. And so on…

Unordered lists

Example

```
We are about to start an unordered list.

  * Like ordered lists, unordered lists are also indented 2 spaces.
  * Unlike ordered lists, unordered lists use stars instead of numbers.
    * Sub items are indented 2 more spaces.
    * Here's another sub item.
```

Formatted

We are about to start an unordered list.

- Like ordered lists, unordered lists are also indented 2 spaces.
- Unlike ordered lists, unordered lists use stars instead of numbers.
  - Sub items are indented 2 more spaces.
  - Here's another sub item.

Links

Automatic links

The simplest way to create a link is to just type it in:

Example

```
http://ccl.northwestern.edu/netlogo/
```

Formatted

http://ccl.northwestern.edu/netlogo/

Links with text

If you want to use your own text for the link, here's how:

```
[link text here](link.address.here)
```

Example

```
[NetLogo](http://ccl.northwestern.edu/netlogo/)
```

Formatted

[NetLogo](#)

Local links

It is also possible to link to a page on your computer, instead of a page somewhere on the Internet.

Local links have this form:

```
[alt text](file:path)
```

Any spaces in the path must be converted to %20. For example, this:

```
file:my page.html
```

must be written as:

```
file:my%20page.html
```

The path is relative to the directory that the model file is in.

Example

The easiest way to link to files on your computer is to put them into the same directory as your model. Assuming you have a file named `index.html` in the same directory as your model, the link would look like this:

```
[Home](file:index.html)
```

Example

Here is another example where the file lives in a directory called docs, and docs is in the same directory as your model:

```
[Home](file:docs/index.html)
```

Images

Images are very similar to links, but have an exclamation point in front:

```
![alt text](http://location/of/image)
```

(The alternate text is the text that gets displayed if the image is not found.)

Example

```
![NetLogo](http://ccl.northwestern.edu/netlogo/images/netlogo-title-new.jpg)
```

Formatted

Local images

Also very similar to links, it is possible to display an image on your computer instead of an image somewhere on the Internet. Assuming you have an image named `image.jpg`, local images look like this:

```
![alt text](file:path)
```

The path is relative to the directory that the model file is in.

As with local links, any spaces in the name of the file or the path must be converted to %20.

Example

Like local links, the easiest way to display images on your computer is to put them into the same directory as your model. This example displays the image "Perspective Example.png", which resides in the same directory as this model (Info Tab Example).

```
![Example](file:Perspective%20Example.png)
```

Formatted
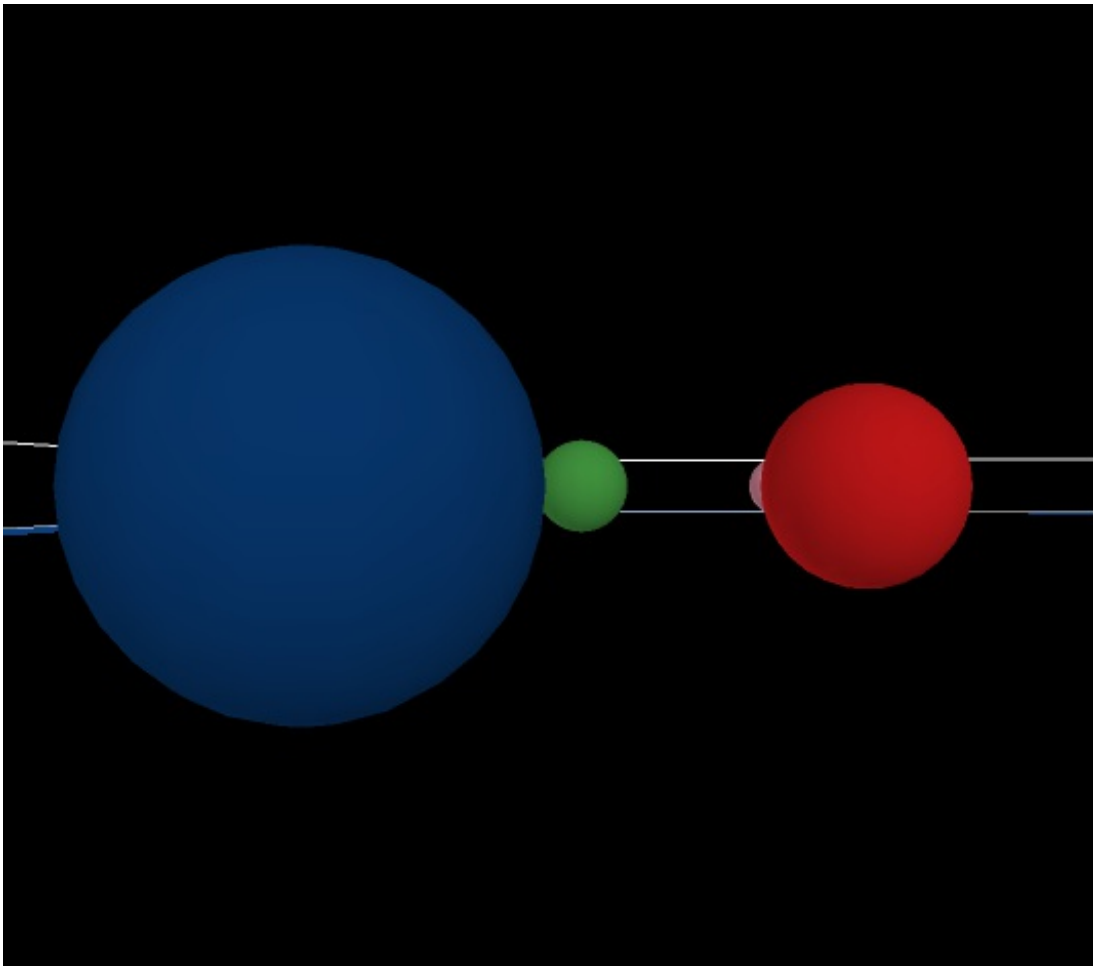
Block quotations

Consecutive lines starting with > will become block quotations.
You can put whatever text you like inside of it and you can also style it.

Example

```
> Let me see: four times five is twelve, and four times six is thirteen,
> and four times seven is --- _oh dear!_
> I shall never get to twenty at that rate!
```

Formatted

> Let me see: four times five is twelve, and four times six is thirteen,
> and four times seven is -- *oh dear!*
> I shall never get to twenty at that rate!

Code

To include a short piece of code in a sentence, surround it with backticks (`).

Example

```
You can create a single turtle with the `crt 1` command.
```

You can create a single turtle with the `crt 1` command.

Code blocks

It is also possible to have blocks of code. To create a code block, indent every line of the block by 4 spaces.

Example

```
About to start the code block.
Leave a blank line after this one, and then indent four spaces:

    ; a typical go procedure
    to go
      ask turtles
        [ fd 1 ]
      tick
    end
```

Formatted

About to start the code block.
Leave a blank line after this one, and then indent four spaces:

```
; a typical go procedure
to go
  ask turtles
    [ fd 1 ]
  tick
end
```

Superscripts and subscripts

Superscripts and subscripts are useful for writing formulas, equations, footnotes and more. Subscripts appear half a character below the baseline, and are written using the HTML tag `<sub>`. Superscripts appear half a character above the baseline, and are written using the HTML tag `<sup>`.

Example

```
H<sub>2</sub>O

2x<sup>4</sup> + x<sup>2</sup>

WWW<sup>[1]</sup>
```

Formatted

H_2O

$2x^4 + x^2 + 42$

WWW[1]

Notes on usage

- Paragraphs, lists, code blocks and other features should be separated from each other with a blank line. If you find that something isn't formatted the way you expected, it might be because you need to add a blank line before it.

- To prevent a special character from being treated as markup, put a backslash (\\) before it.

- We use GitHub flavored newlines (http://github.github.com/github-flavored-markdown/) instead of traditional Markdown handling of newlines. This means that newlines are treated as real line breaks, instead of being being combined with the previous line into a single paragraph.

Other features

Markdown has additional features that we have not shown here.

We have tested the features shown above on a variety of systems. If you use other Markdown features, you may find that they work on your computer, or not. Even a feature that works on your computer might work differently, or not work at all, for someone with a different operating system or Java virtual machine.

If you want all NetLogo users to be able to read your Info tab, use only the features shown above.

More information about Markdown is at http://daringfireball.net/projects/markdown/. For rendering Markdown, NetLogo uses the Pegdown library.

Programming Guide

This section describes the NetLogo programming language in detail.

The Code Example models mentioned throughout can be found in the Code Examples section of the Models Library.

Agents

The NetLogo world is made up of **agents**. Agents are beings that can follow instructions.

In NetLogo, there are four types of agents: **turtles**, **patches**, **links**, and the **observer**.

Turtles are agents that move around in the **world**. The world is two dimensional and is divided up into a grid of patches. Each patch is a square piece of "ground" over which turtles can move. Links are agents that connect two turtles. The observer doesn't have a location -- you can imagine it as looking out over the world of turtles and patches.

The observer doesn't observe passively -- it gives instructions to the other agents.

When NetLogo starts up, there are no turtles. The observer can make new turtles. Patches can make new turtles too. (Patches can't move, but otherwise they're just as "alive" as turtles.)

Patches have coordinates. The patch at coordinates (0, 0) is called the origin and the

coordinates of the other patches are the horizontal and vertical distances from this one. We call the patch's coordinates `pxcor` and `pycor`. Just like in the standard mathematical coordinate plane, `pxcor` increases as you move to the right and `pycor` increases as you move up.

The total number of patches is determined by the settings `min-pxcor`, `max-pxcor`, `min-pycor`, and `max-pycor` When NetLogo starts up, `min-pxcor`, `max-pxcor`, `min-pycor`, and `max-pycor` are -16, 16, -16, and 16 respectively. This means that `pxcor` and `pycor` both range from -16 to 16, so there are 33 times 33, or 1089 patches total. (You can change the number of patches with the Settings button.)

Turtles have coordinates too: `xcor` and `ycor`. A patch's coordinates are always integers, but a turtle's coordinates can have decimals. This means that a turtle can be positioned at any point within its patch; it doesn't have to be in the center of the patch.

Links do not have coordinates. Every link has two ends, and each end is a turtle. If either turtle dies, the link dies too. A link is represented visually as a line connecting the two turtles.

Procedures

In NetLogo, **commands** and **reporters** tell agents what to do. A command is an action for an agent to carry out, resulting in some effect. A reporter is instructions for computing a value, which the agent then "reports" to whoever asked it.

Typically, a command name begins with a verb, such as "create", "die", "jump", "inspect", or "clear". Most reporter names are nouns or noun phrases.

Commands and reporters built into NetLogo are called **primitives**. The NetLogo Dictionary has a complete list of built-in commands and reporters.

Commands and reporters you define yourself are called **procedures**. Each procedure has a name, preceded by the keyword `to` or `to-report`, depending on whether it is a command procedure or a reporter procedure. The keyword `end` marks the end of the commands in the procedure. Once you define a procedure, you can use it elsewhere in your program.

Many commands and reporters take **inputs** -- values that the command or reporter uses in carrying out its actions or computing its result.

Here are two command procedures:

```
to setup
  clear-all
  create-turtles 10
  reset-ticks
end

to go
  ask turtles [
    fd 1            ;; forward 1 step
    rt random 10    ;; turn right
    lt random 10    ;; turn left
  ]
  tick
end
```

Note the use of semicolons to add "comments" to the program. Comments can make your code easier to read and understand, but they don't affect its behavior.

In this program,

- `setup` and `go` are user-defined commands.
- `clear-all`, `create-turtles`, `reset-ticks`, `ask`, `lt` ("left turn"), `rt` ("right turn") and `tick`, are all primitive commands.
- `random` and `turtles` are primitive reporters. `random` takes a single number as an input and reports a random integer that is less than the input (in this case, between 0 and 9). `turtles` reports the agentset consisting of all the turtles. (We'll explain about agentsets later.)

`setup` and `go` can be called by other procedures, or by buttons, or from the Command Center.

Many NetLogo models have a once button that calls a procedure called `setup` and a forever button that calls a procedure called `go`.

In NetLogo, you may specify which agents -- turtles, patches, or links -- are to run each command. If you don't specify, the code is run by the observer. In the code above, the observer uses `ask` to make the set of all turtles run the commands between the square brackets.

`clear-all` and `create-turtles` can only be run by the observer. `fd`, on the other hand, can only be run by turtles. Some other commands and reporters, such as `set` and `ticks`, can be run by different agent types.

Here are some more advanced features you can take advantage of when defining your own procedures.

Procedures with inputs

Procedures can take inputs, just like many primitives do. To create a procedure that accepts inputs, put their names in square brackets after the procedure name. For example:

```
to draw-polygon [num-sides len]   ;; turtle procedure
  pen-down
  repeat num-sides [
    fd len
    rt 360 / num-sides
  ]
end
```

Elsewhere in the program, you might use the procedure by asking the turtles to each draw an octagon with a side length equal to its who number:

```
ask turtles [ draw-polygon 8 who ]
```

Reporter procedures

Just like you can define your own commands, you can define your own reporters. You must do two special things. First, use `to-report` instead of `to` to begin your procedure. Then, in the body of the procedure, use `report` to report the value you want to report.

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report (- number) ]
end
```

Variables

Agent variables

Agent variables are places to store values (such as numbers) in an agent. An agent variable can be a global variable, a turtle variable, a patch variable, or a link variable.

If a variable is a global variable, there is only one value for the variable, and every agent can access it. You can think of global variables as belonging to the observer.

Turtle, patch, and link variables are different. Each turtle has its *own* value for every turtle variable. The same goes for patches and links.

Some variables are built into NetLogo. For example, all turtles and links have a `color` variable, and all patches have a `pcolor` variable. (The patch variable begins with "p" so it doesn't get confused with the turtle variable, since turtles have direct access to patch variables.) If you set the variable, the turtle or patch changes color. (See next section for details.)

Other built-in turtle variables including `xcor`, `ycor`, and `heading`. Other built-in patch variables include `pxcor` and `pycor`. (There is a complete list here.)

You can also define your own variables. You can make a global variable by adding a switch, slider, chooser, or input box to your model, or by using the `globals` keyword at the beginning of your code, like this:

```
globals [score]
```

You can also define new turtle, patch and link variables using the `turtles-own` , `patches-own` and `links-own` keywords, like this:

```
turtles-own [energy speed]
patches-own [friction]
links-own [strength]
```

These variables can then be used freely in your model. Use the `set` command to set them. (Any variable you don't set has a starting value of zero.)

Global variables can be read and set at any time by any agent. As well, a turtle can read and set patch variables of the patch it is standing on. For example, this code:

```
ask turtles [ set pcolor red ]
```

causes every turtle to make the patch it is standing on red. (Because patch variables are shared by turtles in this way, you can't have a turtle variable and a patch variable with the same name.)

In other situations where you want an agent to read a different agent's variable, you can use `of`. Example:

```
show [color] of turtle 5
;; prints current color of turtle with who number 5
```

You can also use `of` with a more complicated expression than just a variable name, for example:

```
show [xcor + ycor] of turtle 5
```

```
;; prints the sum of the x and y coordinates of
;; turtle with who number 5
```

Local variables

A local variable is defined and used only in the context of a particular procedure or part of a procedure. To create a local variable, use the `let` command. If you use `let` at the top of a procedure, the variable will exist throughout the procedure. If you use it inside a set of square brackets, for example inside an "ask", then it will exist only inside those brackets.

```
to swap-colors [turtle1 turtle2]
  let temp [color] of turtle1
  ask turtle1 [ set color [color] of turtle2 ]
  ask turtle2 [ set color temp ]
end
```

Tick counter

In many NetLogo models, time passes in discrete steps, called "ticks". NetLogo includes a built-in tick counter so you can keep track of how many ticks have passed.

The current value of the tick counter is shown above the view. (You can use the Settings button to hide the tick counter, or change the word "ticks" to something else.)

In code, to retrieve the current value of the tick counter, use the `ticks` reporter. The `tick` command advances the tick counter by 1. The `clear-all` command clears the tick counter along with everything else.

When the tick counter is clear, it's an error to try to read or modify it. Use the `reset-ticks` command when your model is done setting up, to start the tick counter.

If your model is set to use tick-based updates, then the `tick` command will usually also update the view. See the later section, View Updates.

When to tick

Use `reset-ticks` at the end of your setup procedure.

Use `tick` at the end of your go procedure.

```
to setup
  clear-all
  create-turtles 10
  reset-ticks
end

to go
  ask turtles [ fd 1 ]
  tick
end
```

Fractional ticks

In most models, the tick counter starts at 0 and goes up 1 at a time, from integer to integer. But it's also possible for the tick counter to take on in-between floating point values.

To advance the tick counter by a fractional amount, use the `tick-advance` command. This command takes a numeric input specifying how far to advance the tick counter.

A typical use of fractional ticks is to approximate continuous or curved motion. See, for example, the GasLab models in the Models Library (under Chemistry & Physics). These models calculate the exact time at which a future event is to occur, then advance the tick counter to exactly that time.

Colors

NetLogo represents colors in different ways. A color can be number in the range 0 to 140, with the exception of 140 itself. Below is a chart showing the range of such colors you can use in NetLogo.
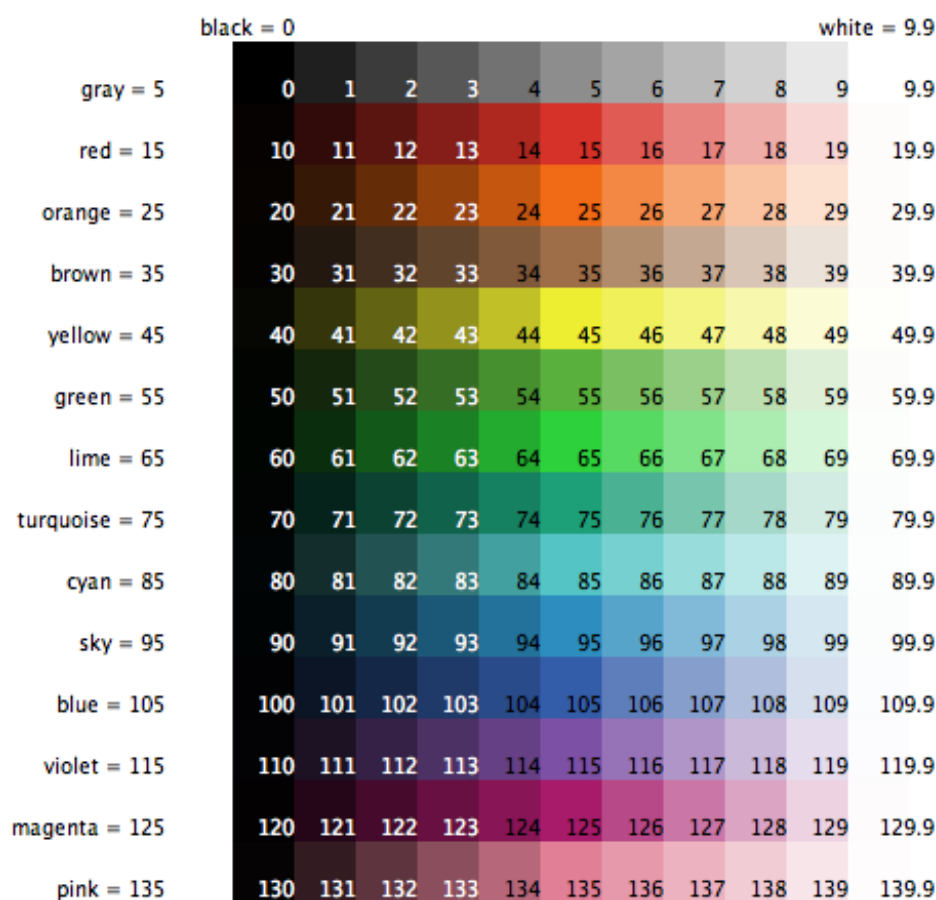
The chart shows that:

- Some of the colors have names. (You can use these names in your code.)
- Every named color except black and white has a number ending in 5.
- On either side of each named color are darker and lighter shades of the color.
- 0 is pure black. 9.9 is pure white.
- 10, 20, and so on are all so dark they are very nearly black.
- 19.9, 29.9 and so on are all so light they are very nearly white.

> **Code Example:** The color chart was made in NetLogo with the Color Chart Example model.

If you use a number outside the 0 to 140 range, NetLogo will repeatedly add or subtract 140 from the number until it is in the 0 to 140 range. For example, 25 is orange, so 165, 305, 445,

and so on are orange too, and so are -115, -255, -395, etc. This calculation is done automatically whenever you set the turtle variable `color` or the patch variable `pcolor`. Should you need to perform this calculation in some other context, use the `wrap-color` primitive.

If you want a color that's not on the chart, more exist between the integers. For example, 26.5 is a shade of orange halfway between 26 and 27. This doesn't mean you can make any color in NetLogo; the NetLogo color space is only a subset of all possible colors. It contains only a fixed set of discrete hues (one hue per row of the chart). Starting from one of those hues, you can either decrease its brightness (darken it) or decrease its saturation (lighten it), but you cannot decrease both brightness and saturation. Also, only the first digit after the decimal point is significant. Thus, color values are rounded down to the next 0.1, so for example, there's no visible difference between 26.5 and 26.52 or 26.58.

Color primitives

There are a few primitives that are helpful for working with colors.

We have already mentioned the `wrap-color` primitive.

The `scale-color` primitive is useful for converting numeric data into colors.

`shade-of?` will tell you if two colors are both "shades" of the same basic hue. For example, `shade-of? orange 27` is true, because 27 is a lighter shade of orange.

> **Code Example:** Scale-color Example demonstrates the scale-color reporter.

RGB and RGBA Colors

NetLogo also represents colors as RGB (red/green/blue) lists and RGBA (red/green/blue/alpha) lists. When using RGB colors the full range of colors is available to you. RGBA colors allow all the colors that RGB allows and you can also vary the transparency of a color. RGB and RGBA lists are made up of three or four integers, respectively, between 0 and 255 if a number is outside that range 255 is repeatedly subtracted until it is in the range. You can set any color variables in NetLogo (`color` for turtles and links and `pcolor` for patches) to an RGB list and that agent will be rendered appropriately. So you can set the color of patch 0 0 to pure red using the following code:

```
set pcolor [255 0 0]
```

Turtles, links, and labels can all contain RGBA lists as their color variables, however, patches cannot have RGBA `pcolors` You can set the color of a turtle to be approximately half transparent pure red with the following code:

```
set color [255 0 0 125]
```

You can convert from a NetLogo color to RGB or HSB (hue/saturation/brightness) using `extract-hsb` and `extract-rgb`. You can use `rgb` to generate rgb lists and `hsb` to convert from an HSB color to RGB.

Since many colors are missing from the NetLogo color space, `approximate-hsb` and `approximate-rgb` often can't give you the exact color you ask for, but they try to come as close as possible.

Example: you can change any turtle from it's existing NetLogo color to a half transparent

version of that color using:

```
set color lput 125 extract-rgb color
```

> **Code Examples:** HSB and RGB Example (lets you experiment with the HSB and RGB color systems), Transparency Example

Color Swatches dialog

The Color Swatches dialog helps you experiment with and choose colors. Open it by choosing Color Swatches on the Tools Menu.
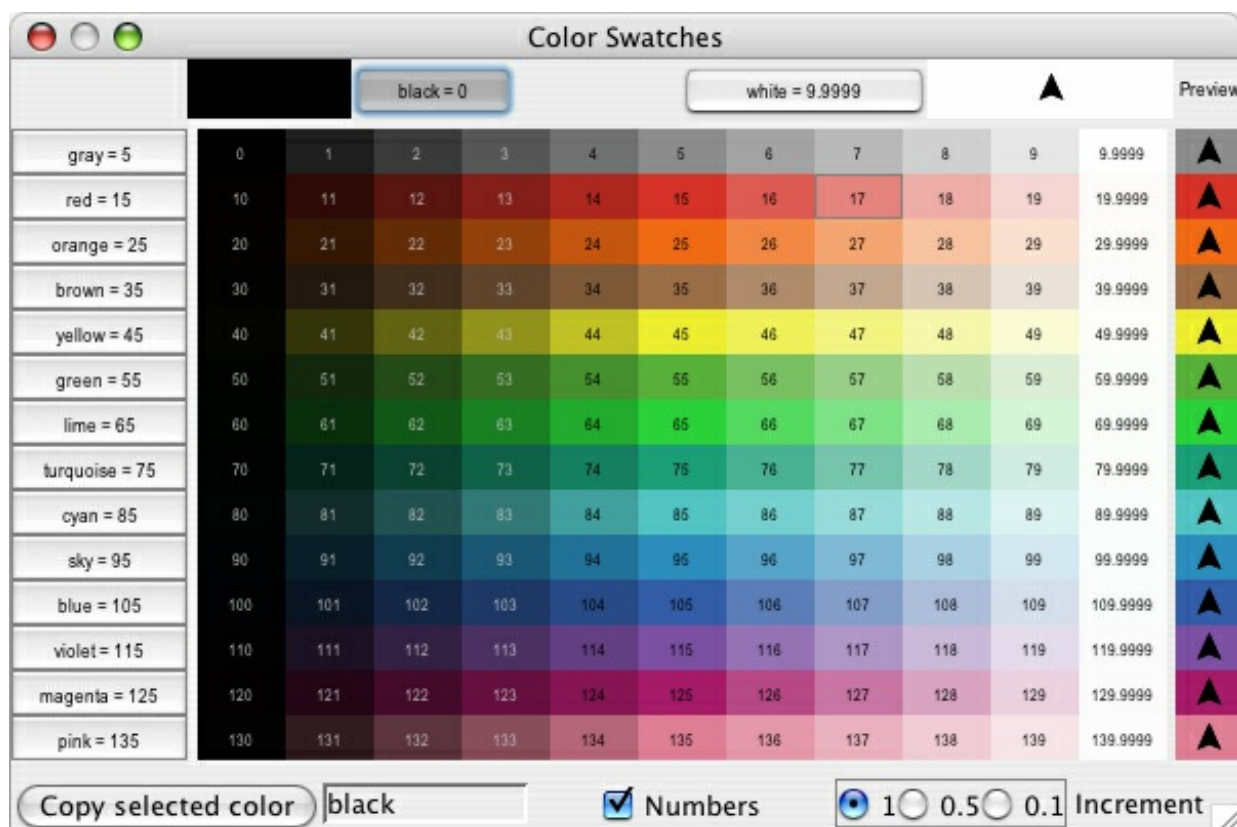
When you click on a color swatch (or a color button), that color will be shown against other colors. In the bottom left, the code for the currently selected color is displayed (for example, `red + 2`) so you can copy and paste it into your code. On the bottom right there are three increment options, 1, 0.5, and 0.1. These numbers indicate the difference between two adjacent swatches. When the increment is 1 there are 10 different shades in each row; when the increment is 0.1 there are 100 different shades in each row. 0.5 is an intermediate setting.

Ask

NetLogo uses the `ask` command to give commands to turtles, patches, and links. All code to be run by turtles *must* be located in a turtle "context". You can establish a turtle context in any of three ways:

- In a button, by choosing "Turtles" from the popup menu. Any code you put in the button will be run by all turtles.
- In the Command Center, by choosing "Turtles" from the popup menu. Any commands you enter will be run by all the turtles.
- By using `ask turtles`, `hatch`, or other commands which establish a turtle context.

The same goes for patches, links, and the observer, except that you cannot `ask` the observer. Any code that is not inside any `ask` is by default observer code.

Here's an example of the use of `ask` in a NetLogo procedure:

```
to setup
  clear-all
  create-turtles 100    ;; create 100 turtles with random headings
  ask turtles
    [ set color red     ;; turn them red
      fd 50 ]           ;; spread them around
  ask patches
    [ if pxcor > 0          ;; patches on the right side
        [ set pcolor green ] ]  ;; of the view turn green
  reset-ticks
end
```

The models in the Models Library are full of other examples. A good place to start looking is in the Code Examples section.

Usually, the observer uses `ask` to ask all turtles, all patches or all links to run commands. You can also use `ask` to have an individual turtle, patch or link run commands. The reporters `turtle`, `patch`, `link` and `patch-at` are useful for this technique. For example:

```
to setup
  clear-all
  crt 3                            ;; make 3 turtles
  ask turtle 0                     ;; tell the first one...
    [ fd 1 ]                       ;; ...to go forward
  ask turtle 1                     ;; tell the second one...
    [ set color green ]            ;; ...to become green
  ask turtle 2                     ;; tell the third one...
    [ rt 90 ]                      ;; ...to turn right
  ask patch 2 -2                   ;; ask the patch at (2,-2)
    [ set pcolor blue ]            ;; ...to become blue
  ask turtle 0                     ;; ask the first turtle
    [ ask patch-at 1 0             ;; ...to ask patch to the east
        [ set pcolor red ] ]       ;; ...to become red
  ask turtle 0                     ;; tell the first turtle...
    [ create-link-with turtle 1 ]  ;; ...make a link with the second
  ask link 0 1                     ;; tell the link between turtle 0 and 1
    [ set color blue ]             ;; ...to become blue
  reset-ticks
end
```

Every turtle created has a who number. The first turtle created is number 0, the second turtle number 1, and so forth.

The `turtle` primitive reporter takes a who number as an input, and reports the turtle with that who number. The `patch` primitive reporter takes values for pxcor and pycor and reports the patch with those coordinates. The `link` primitive takes two inputs, the who numbers of the two turtles it connects. And the `patch-at` primitive reporter takes *offsets*: distances, in the x and y directions, *from* the first agent. In the example above, the turtle with who number 0 is asked to get the patch east (and no patches north) of itself.

You can also select a subset of turtles, or a subset of patches, or a subset of links and ask them to do something. This involves using **agentsets**. The next section explains them in detail.

When you ask a set of agents to run more than one command, each agent must finish before the next agent starts. One agent runs all of the commands, then the next agent runs all of them, and so on. For example, if you write:

```
ask turtles
  [ fd 1
    set color red ]
```

first one turtle moves and turns red, then another turtle moves and turns red, and so on.

But if you write it this way:

```
ask turtles [ fd 1 ]
ask turtles [ set color red ]
```

first all the turtles move, then they all turn red.

Agentsets

An agentset is exactly what its name implies, a set of agents. An agentset can contain either turtles, patches or links, but not more than one type at once.

An agentset is not in any particular order. In fact, it's always in a random order. And every time you use it, the agentset is in a *different* random order. This helps you keep your model from treating any particular turtles, patches or links differently from any others (unless you want them to be). Since the order is random every time, no one agent always gets to go first.

You've seen the turtles primitive, which reports the agentset of all turtles, the patches primitive, which reports the agentset of all patches and the links primitive which reports the agentset of all links.

But what's powerful about the agentset concept is that you can construct agentsets that contain only *some* turtles, *some* patches or *some* links. For example, all the red turtles, or the patches with pxcor evenly divisible by five, or the turtles in the first quadrant that are on a green patch or the links connected to turtle 0. These agentsets can then be used by ask or by various reporters that take agentsets as inputs.

One way is to use turtles-here or turtles-at, to make an agentset containing only the turtles on my patch, or only the turtles on some other patch at some x and y offsets. There's also turtles-on so you can get the set of turtles standing on a given patch or set of patches, or the set of turtles standing on the same patch as a given turtle or set of turtles.

Here are some more examples of how to make agentsets:

```
;; all other turtles:
other turtles
;; all other turtles on this patch:
other turtles-here
;; all red turtles:
turtles with [color = red]
;; all red turtles on my patch
turtles-here with [color = red]
;; patches on right side of view
patches with [pxcor > 0]
;; all turtles less than 3 patches away
turtles in-radius 3
;; the four patches to the east, north, west, and south
patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
;; shorthand for those four patches
neighbors4
;; turtles in the first quadrant that are on a green patch
turtles with [(xcor > 0) and (ycor > 0)
              and (pcolor = green)]
;; turtles standing on my neighboring four patches
turtles-on neighbors4
```

```
;; all the links connected to turtle 0
[my-links] of turtle 0
```

Note the use of `other` to exclude this agent. This is common.

Once you have created an agentset, here are some simple things you can do:

- Use `ask` to make the agents in the agentset do something
- Use `any?` to see if the agentset is empty
- Use `all?` to see if every agent in an agentset satisfies a condition.
- Use `count` to find out exactly how many agents are in the set

And here are some more complex things you can do:

- Pick a random agent from the set using `one-of`. For example, we can make a randomly chosen turtle turn green:

  ```
  ask one-of turtles [ set color green ]
  ```

  Or tell a randomly chosen patch to `sprout` a new turtle:

  ```
  ask one-of patches [ sprout 1 ]
  ```

- Use the `max-one-of` or `min-one-of` reporters to find out which agent is the most or least along some scale. For example, to remove the richest turtle, you could say

  ```
  ask max-one-of turtles [sum assets] [ die ]
  ```

- Make a histogram of the agentset using the `histogram` command (in combination with `of`).
- Use `of` to make a list of values, one for each agent in the agentset. Then use one of NetLogo's list primitives to do something with the list. (See the "Lists" section below.) For example, to find out how rich turtles are on the average, you could say

  ```
  show mean [sum assets] of turtles
  ```

- Use `turtle-set`, `patch-set` and `link-set` reporters to make new agentsets by gathering together agents from a variety of possible sources.
- Use `no-turtles`, `no-patches` and `no-links` reporters to make empty agentsets.
- Check whether two agentsets are equal using `=` or `!=`.
- Use `member?` to see whether a particular agent is a member of an agentset.

This only scratches the surface. See the Models Library for many more examples, and consult the NetLogo Dictionary for more information about all of the agentset primitives.

More examples of using agentsets are provided in the individual entries for these primitives in the NetLogo Dictionary.

Special agentsets

The agentsets `turtles` and `links` have special behavior because they always hold the sets of *all* turtles and *all* links. Therefore, these agentsets can grow.

The following interaction shows the special behavior. Assume the Code tab has `globals [g]`.

Then:

```
observer> clear-all
observer> create-turtles 5
observer> set g turtles
observer> print count g
5
observer> create-turtles 5
observer> print count g
10
observer> set g turtle-set turtles
observer> print count g
10
observer> create-turtles 5
observer> print count g
10
observer> print count turtles
15
```

The `turtles` agentset grows when new turtles are born, but other agentsets don't grow. If I write `turtle-set turtles`, I get a new, normal agentset containing just the turtles that *currently* exist. New turtles don't join when they're born.

Breed agentsets are special in the same way as `turtles` and `links`. Breeds are introduced and explained below.

Agentsets and lists

Earlier, we said that agentsets are always in random order, a different random order every time. If you need your agents to do something in a fixed order, you need to make a list of the agents instead. See the Lists section below.

> **Code Example:** Ask Ordering Example

Breeds

NetLogo allows you to define different "breeds" of turtles and breeds of links. Once you have defined breeds, you can go on and make the different breeds behave differently. For example, you could have breeds called `sheep` and `wolves`, and have the wolves try to eat the sheep or you could have link breeds called `streets` and `sidewalks` where foot traffic is routed on sidewalks and car traffic is routed on streets.

You define turtle breeds using the <u>breed</u> keyword, at the top of the Code tab, before any procedures:

```
breed [wolves wolf]
breed [sheep a-sheep]
```

You can refer to a member of the breed using the singular form, just like the <u>turtle</u> reporter. When printed, members of the breed will be labeled with the singular name.

Some commands and reporters have the plural name of the breed in them, such as <u>create-<breeds></u>. Others have the singular name of the breed in them, such as <u><breed></u>

The order in which breeds are declared is also the order in which they are layered in the view. So breeds defined later will appear on top of breeds defined earlier; in this example, sheep

will be drawn over wolves.

When you define a breed such as `sheep`, an agentset for that breed is automatically created, so that all of the agentset capabilities described above are immediately available with the `sheep` agentset.

The following new primitives are also automatically available once you define a breed: `create-sheep`, `hatch-sheep`, `sprout-sheep`, `sheep-here`, `sheep-at`, `sheep-on`, and `is-a-sheep?`.

Also, you can use `sheep-own` to define new turtle variables that only turtles of the given breed have. (It's allowed for more than one breed to own the same variable.)

A turtle's breed agentset is stored in the `breed` turtle variable. So you can test a turtle's breed, like this:

```
if breed = wolves [ ... ]
```

Note also that turtles can change breeds. A wolf doesn't have to remain a wolf its whole life. Let's change a random wolf into a sheep:

```
ask one-of wolves [ set breed sheep ]
```

The `set-default-shape` primitive is useful for associating certain turtle shapes with certain breeds. See the section on shapes below.

Who numbers are assigned irrespective of breeds. If you already have a `frog 0`, then the first mouse will be `mouse 1`, not `mouse 0`, since the who number 0 is already taken.

Here is a quick example of using breeds:

```
breed [mice mouse]
breed [frogs frog]
mice-own [cheese]
to setup
  clear-all
  create-mice 50
    [ set color white
      set cheese random 10 ]
  create-frogs 50
    [ set color green ]
  reset-ticks
end
```

> **Code Example:** Breeds and Shapes Example

Link breeds

Link breeds are very similar to turtle breeds, however, there are a few differences.

When you declare a link breed you must declare whether it is a breed of directed or undirected links by using the `directed-link-breed` and `undirected-link-breed` keywords.

```
directed-link-breed [streets street]
undirected-link-breed [friendships friendship]
```

Once you have created a breeded link you cannot create unbreeded links and vice versa. (You can, however, have directed and undirected links in the same world, just not in the same breed)

Unlike with turtle breeds the singular breed name is required for link breeds, as many of the link commands and reports use the singular name, such as `<link-breed>-neighbor?`.

The following primitives are also automatically available once you define a directed link breed: `create-street-from` `create-streets-from` `create-street-to` `create-streets-to` `in-street-neighbor?` `in-street-neighbors` `in-street-from` `my-in-streets` `my-out-streets` `out-street-neighbor?` `out-street-neighbors` `out-street-to`

And the following are automatically available when you define an undirected link breed: `create-friendship-with` `create-friendships-with` `friendship-neighbor?` `friendship-neighbors` `friendship-with` `my-friendships`

Multiple link breeds may declare the same `-own` variable, but a variable may not be shared between a turtle breed and a link breed.

Just as with turtle breeds the order in which link breeds are declared defines the order in which the links are drawn, so the friendships will always be on top of streets (if for some reason these breeds were in the same model). You can also use `<link-breeds>-own` to declare variables of each link breed separately.

You can change the breed of a link with `set breed`. (However, you cannot change a breeded link to an unbreeded one, to prevent having breeded and unbreeded links in the same world.)

```
ask one-of friendships [ set breed streets ]
ask one-of friendships [ set breed links ] ;; produces a runtime error
```

`set-default-shape` may also be used with link breeds to associate it with a particular link shape.

> **Code Example:** Link Breeds Example

Buttons

Buttons in the interface tab provide an easy way to control the model. Typically a model will have at least a "setup" button, to set up the initial state of the world, and a "go" button to make the model run continuously. Some models will have additional buttons that perform other actions.

A button contains some NetLogo code. That code is run when you press the button.

A button may be either a "once button", or a "forever button". You can control this by editing the button and checking or unchecking the "Forever" checkbox. Once buttons run their code once, then stop and pop back up. Forever buttons keep running their code over and over again.

A forever button stops if the user presses the button again to stop it. The button waits until the current iteration has finished, then pops up.

A forever button can also be stopped from code. If the forever button directly calls a procedure, then when that procedure stops, the button stops. (In a turtle or patch forever

button, the button won't stop until every turtle or patch stops -- a single turtle or patch doesn't have the power to stop the whole button.)

Normally, a button is labeled with the code that it runs. For example, a button that says "go" on it usually contains the code "go", which means "run the go procedure". (Procedures are defined in the Code tab; see below.) But you can also edit a button and enter a "display name" for the button, which is a text that appears on the button instead of the code. You might use this feature if you think the actual code would be confusing to your users.

When you put code in a button, you must also specify which agents you want to run that code. You can choose to have the observer run the code, or all turtles, or all patches, or all links. (If you want the code to be run by only some turtles or some patches, you could make an observer button, and then have the observer use the `ask` command to ask only some of the turtles or patches to do something.)

When you edit a button, you have the option to assign an "action key". This makes that key on the keyboard behave just like a button press. If the button is a forever button, it will stay down until the key is pressed again (or the button is clicked). Action keys are particularly useful for games or any model where rapid triggering of buttons is needed.

Buttons take turns

More than one button can be pressed at a time. If this happens, the buttons "take turns", which means that only one button runs at a time. Each button runs its code all the way through once while the other buttons wait, then the next button gets its turn.

In the following examples, "setup" is a once button and "go" is a forever button.

Example #1: The user presses "setup", then presses "go" immediately, before the "setup" has popped back up. Result: "setup" finishes before "go" starts.

Example #2: While the "go" button is down, the user presses "setup". Result: the "go" button finishes its current iteration. Then the "setup" button runs. Then "go" starts running again.

Example #3: The user has two forever buttons down at the same time. Result: first one button runs its code all the way through, then the other runs its code all the way through, and so on, alternating.

Note that if one button gets stuck in an infinite loop, then no other buttons will run.

Turtle, patch, and link forever buttons

There is a subtle difference between putting commands in a turtle, patch or link forever button, and putting the same commands in an observer button that does `ask turtles`, `ask patches` or `ask links`. An "ask" doesn't complete until all of the agents have finished running all of the commands in the "ask". So the agents, as they all run the commands concurrently, can be out of sync with each other, but they all sync up again at the end of the ask. The same isn't true of turtle, patch and link forever buttons. Since `ask` was not used, each turtle or patch runs the given code over and over again, so they can become (and remain) out of sync with each other.

At present, this capability is very rarely used in the models in our Models Library. A model that does use the capability is the Termites model, in the Biology section of Sample Models. The "go" button is a turtle forever button, so each termite proceeds independently of every other termite, and the observer is not involved at all. This means that if, for example, you wanted to add ticks and/or a plot to the model, you would need to add a second forever button (an observer forever button), and run both forever buttons at the same time. Note also that a model like this cannot be used with BehaviorSpace.

> **Code Example:** State Machine Example shows how Termites can be recoded in a tick-based way, without using a turtle forever button.

At present, NetLogo has no way for one forever button to start another. Buttons are only started when you press them.

Lists

In the simplest models, each variable holds only one piece of information, usually a number or a string. Lists let you store multiple pieces of information in a single value by collecting that information in a list. Each value in the list can be any type of value: a number, or a string, an agent or agentset, or even another list.

Lists allow for the convenient packaging of information in NetLogo. If your agents carry out a repetitive calculation on multiple variables, it might be easier to have a list variable, instead of multiple number variables. Several primitives simplify the process of performing the same computation on each value in a list.

The [NetLogo Dictionary](#) has a section that lists all of the list-related primitives.

Constant lists

You can make a list by simply putting the values you want in the list between brackets, like this: `set mylist [2 4 6 8]`. Note that the individual values are separated by spaces. You can make lists that contain numbers and strings this way, as well as lists within lists, for example `[[2 4] [3 5]]`.

The empty list is written by putting nothing between the brackets, like this: `[]`.

Building lists on the fly

If you want to make a list in which the values are determined by reporters, as opposed to being a series of constants, use the `list` reporter. The `list` reporter accepts two other reporters, runs them, and reports the results as a list.

If I wanted a list to contain two random values, I might use the following code:

```
set random-list list (random 10) (random 20)
```

This will set `random-list` to a new list of two random integers each time it runs.

To make longer or shorter lists, you can use the `list` reporter with fewer or more than two inputs, but in order to do so, you must enclose the entire call in parentheses, e.g.:

```
(list random 10)
(list random 10 random 20 random 30)
```

For more information, see [Varying number of inputs](#).

Some kinds of lists are most easily built using the `n-values` reporter, which allows you to construct a list of a specific length by repeatedly running a given reporter. You can make a list of the same value repeated, or all the numbers in a range, or a lot of random numbers, or many other possibilities. See dictionary entry for details and examples.

The `of` primitive lets you construct a list from an agentset. It reports a list containing each agent's value for the given reporter. (The reporter could be a simple variable name, or a more complex expression -- even a call to a procedure defined using `to-report`.) A common idiom is

```
max [...] of turtles
sum [...] of turtles
```

and so on.

You can combine two or more lists using the `sentence` reporter, which concatenates lists by combining their contents into a single, larger list. Like `list`, `sentence` normally takes two inputs, but can accept any number of inputs if the call is surrounded by parentheses.

Changing list items

Technically, lists can't be modified, but you can construct new lists based on old lists. If you want the new list to replace the old list, use `set`. For example:

```
set mylist [2 7 5 Bob [3 0 -2]]
; mylist is now [2 7 5 Bob [3 0 -2]]
set mylist replace-item 2 mylist 10
; mylist is now [2 7 10 Bob [3 0 -2]]
```

The `replace-item` reporter takes three inputs. The first input specifies which item in the list is to be changed. 0 means the first item, 1 means the second item, and so forth.

To add an item, say 42, to the end of a list, use the `lput` reporter. (`fput` adds an item to the beginning of a list.)

```
set mylist lput 42 mylist
; mylist is now [2 7 10 Bob [3 0 -2] 42]
```

But what if you changed your mind? The `but-last` (`bl` for short) reporter reports all the list items but the last.

```
set mylist but-last mylist
; mylist is now [2 7 10 Bob [3 0 -2]]
```

Suppose you want to get rid of item 0, the 2 at the beginning of the list.

```
set mylist but-first mylist
; mylist is now [7 10 Bob [3 0 -2]]
```

Suppose you wanted to change the third item that's nested inside item 3 from -2 to 9? The key is to realize that the name that can be used to call the nested list [3 0 -2] is `item 3 mylist`. Then the `replace-item` reporter can be nested to change the list-within-a-list. The parentheses are added for clarity.

```
set mylist (replace-item 3 mylist
                  (replace-item 2 (item 3 mylist) 9))
; mylist is now [7 10 Bob [3 0 9]]
```

Iterating over lists

If you want to do some operation on each item in a list in turn, the `foreach` command and the `map` reporter may be helpful.

`foreach` is used to run a command or commands on each item in a list. It takes an input list and a command name or block of commands, like this:

```
foreach [1 2 3] show
=> 1
=> 2
=> 3
foreach [2 4 6]
  [ crt ?
    show (word "created " ? " turtles") ]
=> created 2 turtles
=> created 4 turtles
=> created 6 turtles
```

In the block, the variable `?` holds the current value from the input list.

Here are some more examples of `foreach`:

```
foreach [1 2 3] [ ask turtles [ fd ? ] ]
;; turtles move forward 6 patches
foreach [true false true true] [ ask turtles [ if ? [ fd 1 ] ] ]
;; turtles move forward 3 patches
```

`map` is similar to `foreach`, but it is a reporter. It takes an input list and a reporter name or reporter block. Note that unlike `foreach`, the reporter comes first, like this:

```
show map round [1.2 2.2 2.7]
;; prints [1 2 3]
```

`map` reports a list containing the results of applying the reporter to each item in the input list. Again, use `?` to refer to the current item in the list.

Here are a couple more examples of `map`:

```
show map [? < 0] [1 -1 3 4 -2 -10]
;; prints [false true false false true true]
show map [? * ?] [1 2 3]
;; prints [1 4 9]
```

Besides `map` and `foreach`, other primitives for processing whole lists in a configurable way include `filter`, `reduce`, and `sort-by`.

These primitives aren't always the solution for every situation in which you want to operate on an entire list. In some situations, you may need to use some other technique such as a loop using `repeat` or `while`, or a recursive procedure.

The blocks of code we're giving to `map` and `foreach` in these examples are actually **tasks**. Tasks are explained in more detail in Tasks, below.

Varying number of inputs

Some commands and reporters involving lists and strings may take a varying number of inputs. In these cases, in order to pass them a number of inputs other than their default, the primitive and its inputs must be surrounded by parentheses. Here are some examples:

```
show list 1 2
=> [1 2]
show (list 1 2 3 4)
=> [1 2 3 4]
show (list)
=> []
```

Note that each of these special primitives has a default number of inputs for which no parentheses are required. The primitives which have this capability are `list`, `word`, `sentence`, `map`, `foreach`, `run`, and `runresult`.

Lists of agents

Earlier, we said that agentsets are always in random order, a different random order every time. If you need your agents to do something in a fixed order, you need to make a list of the agents instead.

There are two primitives that help you do this, `sort` and `sort-by`.

Both `sort` and `sort-by` can take an agentset as input. The result is always a new list, containing the same agents as the agentset did, but in a particular order.

If you use `sort` on an agentset of turtles, the result is a list of turtles sorted in ascending order by `who` number.

If you use `sort` on an agentset of patches, the result is a list of patches sorted left-to-right, top-to-bottom.

If you use `sort` on an agentset of links, the result is a list of links, sorted in ascending order first by `end1` then by `end2` any remaining ties are resolved by breed in the order they are declared in the Code tab.

If you need descending order instead, you can combine `reverse` with `sort`, for example `reverse sort turtles`.

If you want your agents to be ordered by some other criterion than the standard ones `sort` uses, you'll need to use `sort-by` instead.

Here's an example:

```
sort-by [[size] of ?1 < [size] of ?2] turtles
```

This returns a list of turtles sorted in ascending order by their turtle variable `size`.

There's a common pattern to get a list of agents in a random order, using a combination of `of` and `self`, in the rare case that you cannot just use `ask`:

```
[self] of my-agentset
```

Asking a list of agents

Once you have a list of agents, you might want to ask them each to do something. To do this, use the `foreach` and `ask` commands in combination, like this:

```
foreach sort turtles [
  ask ? [
    ...
```

```
        ]
    ]
```

This will ask each turtle in ascending order by who number. Substitute "patches" for "turtles" to ask patches in left-to-right, top-to-bottom order.

Note that you can't use `ask` directly on a list of turtles. `ask` only works with agentsets and single agents.

Performance of lists

The data structure underlying NetLogo's lists is a sophisticated tree-based data structure on which most operations run in near-constant time. That includes `fput`, `lput`, `butfirst`, `butlast`, `length`, `item`, and `replace-item`.

One exception to the fast-performance rule is that concatenating two lists with `sentence` requires traversing and copying the whole second list. (This may be fixed in a future version.)

Technically, "near-constant time" is actually logarithmic time, proportional to the depth of the underlying tree, but these trees have large nodes and a high branching factor, so they are never more than a few levels deep. This means that changes can be made in at most a few steps. The trees are immutable, but they share structure with each other, so the whole tree doesn't need to be copied to make a changed version.

The actual data structure used is the immutable Vector class from the Scala collections library. These are 32-wide hash array mapped tries, as implemented by Tiark Rompf, based in part on work by Phil Bagwell and Rich Hickey.

Math

All numbers in NetLogo are stored internally as double precision floating point numbers, as defined in the IEEE 754 standard. They are 64 bit numbers consisting of one sign bit, an 11-bit exponent, and a 52-bit mantissa. See the IEEE 754 standard for details.

An "integer" in NetLogo is simply a number that happens to have no fractional part. No distinction is made between 3 and 3.0; they are the same number. (This is the same as how most people use numbers in everyday contexts, but different from some programming languages. Some languages treat integers and floating point numbers as distinct types.)

Integers are always printed by NetLogo without the trailing ".0":

```
show 1.5 + 1.5
observer: 3
```

If a number with a fractional part is supplied in a context where an integer is expected, the fractional part is simply discarded. So for example, `crt 3.5` creates three turtles; the extra 0.5 is ignored.

The range of integers is +/-9007199254740992 (2^{53}, about 9 quadrillion). Calculations that exceed this range will not cause runtime errors, but precision will be lost when the least significant (binary) digits are rounded off in order fit the number into 64 bits. With very large numbers, this rounding can result in imprecise answers which may be surprising:

```
show 2 ^ 60 + 1 = 2 ^ 60
=> true
```

Calculations with smaller numbers can also produce surprising results if they involve fractional quantities, since not all fractions can be precisely represented and roundoff may occur. For example:

```
show 1 / 6 + 1 / 6 + 1 / 6 + 1 / 6 + 1 / 6 + 1 / 6
=> 0.9999999999999999
show 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9 + 1 / 9
=> 1.0000000000000002
```

Any operation which produces the special quantities "infinity" or "not a number" will cause a runtime error.

Scientific notation

Very large or very small floating point numbers are displayed by NetLogo using "scientific notation". Examples:

```
show 0.000000000001
=> 1.0E-12
show 50000000000000000000
=> 5.0E19
```

Numbers in scientific notation are distinguished by the presence of the letter E (for "exponent"). It means "times ten to the power of", so for example, 1.0E-12 means 1.0 times 10 to the -12 power:

```
show 1.0 * 10 ^ -12
=> 1.0E-12
```

You can also use scientific notation yourself in NetLogo code:

```
show 3.0E6
=> 3000000
show 8.123456789E6
=> 8123456.789
show 8.123456789E7
=> 8.123456789E7
show 3.0E16
=> 3.0E16
show 8.0E-3
=> 0.0080
show 8.0E-4
=> 8.0E-4
```

These examples show that numbers with fractional parts are displayed using scientific notation if the exponent is less than -3 or greater than 6. Numbers outside of NetLogo's integer range of -9007199254740992 to 9007199254740992 (+/-2^53) are also always shown in scientific notation:

```
show 2 ^ 60
=> 1.15292150460684698E18
```

When entering a number, the letter E may be either upper or lowercase. When printing a number, NetLogo always uses an uppercase E:

```
show 4.5e20
=> 4.5E20
```

Floating point accuracy

Because numbers in NetLogo are subject to the limitations of how floating point numbers are represented in binary, you may get answers that are slightly inaccurate. For example:

```
show 0.1 + 0.1 + 0.1
=> 0.30000000000000004
show cos 90
=> 6.123233995736766E-17
```

This is an inherent issue with floating point arithmetic; it occurs in all programming languages that use floating point numbers.

If you are dealing with fixed precision quantities, for example dollars and cents, a common technique is to use only integers (cents) internally, then divide by 100 to get a result in dollars for display.

If you must use floating point numbers, then in some situations you may need to replace a straightforward equality test such as `if x = 1 [ ... ]` with a test that tolerates slight imprecision, for example `if abs (x - 1) < 0.0001 [ ... ]`.

Also, the `precision` primitive is handy for rounding off numbers for display purposes. NetLogo monitors round the numbers they display to a configurable number of decimal places, too.

Random numbers

The random numbers used by NetLogo are what is called "pseudo-random". (This is typical in computer programming.) That means they appear random, but are in fact generated by a deterministic process. "Deterministic" means that you get the same results every time, if you start with the same random "seed". We'll explain in a minute what we mean by "seed".

In the context of scientific modeling, pseudo-random numbers are actually desirable. That's because it's important that a scientific experiment be reproducible -- so anyone can try it themselves and get the same result that you got. Since NetLogo uses pseudo-random numbers, the "experiments" that you do with it can be reproduced by others.

Here's how it works. NetLogo's random number generator can be started with a certain seed value, which must be an integer in the range -2147483648 to 2147483647. Once the generator has been "seeded" with the `random-seed` command, it always generates the same sequence of random numbers from then on. For example, if you run these commands:

```
random-seed 137
show random 100
show random 100
show random 100
```

You will always get the numbers 79, 89, and 61 in that order.

Note, however, that you're only guaranteed to get those same numbers if you're using the same version of NetLogo. Sometimes when we make a new version of NetLogo the random number generator changes. (Presently, we use a generator known as the Mersenne Twister.)

To create a number suitable for seeding the random number generator, use the `new-seed` reporter. `new-seed` creates a seed, evenly distributed over the space of possible seeds, based on the current date and time. It never reports the same seed twice in a row.

If you don't set the random seed yourself, NetLogo sets it to a value based on the current date and time. There is no way to find out what random seed it chose, so if you want your model run to be reproducible, you must set the random seed yourself ahead of time.

The NetLogo primitives with "random" in their names (random, random-float, and so on) aren't the only ones that use pseudo-random numbers. Many other operations also make random choices. For example, agentsets are always in random order, `one-of` and `n-of` choose agents randomly, the `sprout` command creates turtles with random colors and headings, and the `downhill` reporter chooses a random patch when there's a tie. All of these random choices are governed by the random seed as well, so model runs can be reproducible.

In addition to the uniformly distributed random integers and floating point numbers generated by `random` and `random-float`, NetLogo also offers several other random distributions. See the dictionary entries for `random-normal`, `random-poisson`, `random-exponential`, and `random-gamma`.

Auxiliary generator

Code run by buttons or from the command center uses the main random number generator.

Code in monitors uses an auxiliary random generator, so even if a monitor does a calculation that uses random numbers, the outcome of the model is not affected. The same is true of code in sliders.

Local randomness

You may want to explicitly specify that a section of code does not affect the state of the main random generator, so the outcome of the model is not affected. The `with-local-randomness` command is provided for this purpose. See its entry in the NetLogo Dictionary for more information.

Turtle shapes

In NetLogo, turtle shapes are vector shapes. They are built up from basic geometric shapes; squares, circles, and lines, rather than a grid of pixels. Vector shapes are fully scalable and rotatable. NetLogo caches bitmap images of vector shapes size 1, 1.5, and 2 in order to speed up execution.

A turtle's shape is stored in its `shape` variable and can be set using the `set` command.

New turtles have a shape of "default". The `set-default-shape` primitive is useful for changing the default turtle shape to a different shape, or having a different default turtle shape for each breed of turtle.

The `shapes` primitive reports a list of currently available turtle shapes in the model. This is useful if, for example, you want to assign a random shape to a turtle:

```
ask turtles [ set shape one-of shapes ]
```

Use the Turtle Shapes Editor to create your own turtle shapes, or to add shapes to your model from our shapes library, or to transfer shapes between models. For more information,

see the Shapes Editor section of this manual.

The thickness of the lines used to draw the vector shapes can be controlled by the `__set-line-thickness` primitive.

Code Examples: Breeds and Shapes Example, Shape Animation Example

---

Link shapes

Link Shapes are similar to turtle shapes, only you use the Link Shape Editor to create and edit them. Link shapes consist of between 0 and 3 lines which can have different patterns and a direction indicator that is composed of the same elements as turtle shapes. Links also have a `shape` variable that can be set to any link shape that is in the model. By default links have the "default" shape, though you can change that using `set-default-shape`. The `link-shapes` reporter reports all the link shapes included in the current model.

The thickness of the lines in the link shape is controlled by the `thickness` link variable.

View updates

The "view" in NetLogo lets you see the agents in your model on your computer's screen. As your agents move and change, you see them moving and changing in the view.

Of course, you can't really see your agents directly. The view is a picture that NetLogo paints, showing you how your agents look at a particular instant. Once that instant passes and your agents move and change some more, that picture needs to be repainted to reflect the new state of the world. Repainting the picture is called "updating" the view.

When does the view get updated? This section discusses how NetLogo decides when to update the view, and how you can influence when it gets updated.

NetLogo offers two updates modes, "continuous" updates and "tick-based" updates. You can switch between NetLogo's two view update modes using a popup menu at the top of the Interface tab.

Continuous updates are the default when you start up NetLogo or start a new model. Nearly every model in our Models Library, however, uses tick-based updates.

Continuous updates are simplest, but tick-based updates give you more control over when and how often updates happen.

It's important exactly when an update happens, because when updates happen determines what you see on the screen. If an update comes at an unexpected time, you may see something unexpected -- perhaps something confusing or misleading.

It's also important how often updates happen, because updates take time. The more time NetLogo spends updating the view, the slower your model will run. With fewer updates, your model runs faster.

Continuous updates

Continuous updates are very simple. With continuous updates, NetLogo updates the view a certain number of times per second -- by default, 30 times a second when the speed slider is

in the default, middle setting.

If you move the speed slider to a slower setting, NetLogo will update more than 30 times a second, effectively slowing down the model. On a faster setting, NetLogo will update less than 30 times a second. On the fastest setting, updates will be separated by several seconds.

At extremely slow settings, NetLogo will be updating so often that you will see your agents moving (or changing color, etc.) one at a time.

If you need to temporarily shut off continuous updates, use the `no-display` command. The `display` command turns updates back on, and also forces an immediate update (unless the user is fast-forwarding the model using the speed slider).

Tick-based updates

As discussed above in the [Tick Counter](#) section, in many NetLogo models, time passes in discrete steps, called "ticks". Typically, you want the view to update once per tick, between ticks. That's the default behavior with tick-based updates.

If you want additional view updates, you can force an update using the `display` command. (The update may be skipped if the user is fast-forwarding the model using the speed slider.)

You don't have to use the tick counter to use tick-based updates. If the tick counter never advances, the view will update only when you use the `display` command.

If you move the speed slider to a fast enough setting, eventually NetLogo will skip some of the updates that would ordinarily have happened. Moving the speed slider to a slower setting doesn't cause additional updates; rather, it makes NetLogo pause after each update. The slower the setting, the longer the pause.

Even under tick-based updates, the view also updates whenever a button in the interface pops up (both once and forever buttons) and when a command entered in the Command Center finishes. So it's not necessary to add the `display` command to once buttons that don't advance the tick counter. Many forever buttons that don't advance the tick counter do need to use the `display` command. An example in the Models Library is the Life model (under Computer Science -> Cellular Automata). The forever buttons that let the user draw in the view use the `display` command so the user can see what they are drawing, even though the tick counter is not advancing.

Choosing a mode

Advantages of tick-based updates over continuous updates include:

1. Consistent, predictable view update behavior which does not vary from computer to computer or from run to run.
2. Continuous updates can confuse the user of your model by letting them see model states they aren't supposed to see, which may be misleading.
3. Increased speed. Updating the view takes time, so if one update per tick is enough, then enforcing there is only one update per tick will make your model faster.
4. Since `setup` buttons don't advance the tick counter, they are unaffected by the speed slider; this is normally the desired behavior.

Nearly every model in our Models Library uses tick-based updates.

Continuous updates are occasionally useful for those rare models in which execution is not divided into short, discrete phases. An example in the Models Library is Termites. (See also,

however, the State Machine Example model, which shows how to re-code Termites using ticks.)

Even for models that would normally be set to tick-based updates, it may be useful to switch to continuous updates temporarily for debugging purposes. Seeing what's going on within a tick, instead of only seeing the end result of a tick, could help with troubleshooting. After switching to continuous updates, you may want to use the speed slider to slow the model down until you see your agents moving one at a time. Don't forget to change back to tick-based updates when you are done, as the choice of update mode is saved with the model.

Frame rate

One of the model settings in NetLogo's "Settings..." dialog is "Frame rate" which defaults to 30 frames per second.

The frame rate setting affects both continuous updates and tick-based updates.

With continuous updates, the setting directly determines the frequency of updates.

With tick-based updates, the setting is a *ceiling* on how many updates per second you get. If the frame rate is 30, then NetLogo will ensure that the model never runs faster than that when the speed slider is in the default position. If any frame takes less than 1/30 of a second to compute and display, NetLogo will pause and wait until the full 1/30 of a second has passed before continuing.

The frame rate settings lets you set what you consider to be a normal speed for your model. Then you, or the user of your model, can use the speed slider to temporarily get a faster or slower speed.

Plotting

NetLogo's plotting features let you create plots to help you understand what's going on in your model.

Before you can plot, you need to create one or more plots in the Interface tab. For more information on using and editing plots in the Interface tab, see the Interface Guide.

Plotting points

The two basic commands for actually plotting things are `plot` and `plotxy`.

With `plot` you need only specify the y value you want plotted. The x value will automatically be 0 for the first point you plot, 1 for the second, and so on. (That's if the plot pen's "interval" is the default value of 1; you can change the interval.)

The `plot` command is especially handy when you want your model to plot a new point at every time step. Example:

```
plot count turtles
```

If you need to specify both the x and y values of the point you want plotted, then use `plotxy` instead. This example assumes that a global variable called `time` exists:

```
plotxy time count-turtles
```

Plot commands

Each plot and its pens have setup and update code fields that may contain commands (usually containing `plot` or `plotxy`). These commands are run automatically triggered by other commands in NetLogo.

Plot setup commands and pen setup commands are run when the either `reset-ticks` or `setup-plots` commands are run. If the `stop` command is run in the body of the plot setup commands then the pen setup commands will not run.

Plot update commands and pen update commands are run when the either `reset-ticks`, `tick` or `update-plots` commands are run. If the `stop` command is run in the body of the plot update commands then the pen update commands will not run.

Here are the four commands that trigger plotting explained in more detail.

- `setup-plots` executes commands for one plot at a time. For each plot, the plot's setup commands are executed. If the stop command is not encountered while running those commands, then each of the plot's pens will have their setup code executed.
- `update-plots` is very similar to `setup-plots`. For each plot, the plot's update commands are executed. If the stop command is not encountered while running those commands, then each of the plot's pens will have their update code executed.
- `tick` is exactly the same as `update-plots` except that the tick counter is incremented before the plot commands are executed.
- `reset-ticks` first resets the tick counter to 0, and then does the equivalent of `setup-plots` followed by `update-plots`.

A typical model will use `reset-ticks` and `tick` like so:

```
to setup
  clear-all
  ...
  reset-ticks
end

to go
  ...
  tick
end
```

Note that in this example we plot from both the `setup` and `go` procedures (because `reset-ticks` runs plot setup and plot update commands). We do this because we want our plot to include the initial state of the system at the end of `setup`. We plot at the end of the `go` procedure, not the beginning, because we want the plot always to be up to date after the go button stops.

Models that don't use ticks but still want to do plotting will instead use `setup-plots` and `update-plots`. In the previous code, replace `reset-ticks` with `setup-plots update-plots` and replace tick with `update-plots`.

> Code Example: Plotting Example

Other kinds of plots

By default, NetLogo plot pens plot in line mode, so that the points you plot are connected by a line.

If you want to move the pen without plotting, you can use the `plot-pen-up` command. After this command is issued, the `plot` and `plotxy` commands move the pen but do not actually draw anything. Once the pen is where you want it, use `plot-pen-down` to put the pen back down.

If you want to plot individual points instead of lines, or you want to draw bars instead of lines or points, you need to change the plot pen's "mode". Three modes are available: line, bar, and point. Line is the default mode.

Normally, you change a pen's mode by editing the plot. This changes the pen's default mode. It's also possible to change the pen's mode temporarily using the `set-plot-pen-mode command`. That command takes a number as input: 0 for line, 1 for bar, 2 for point.

Histograms

A histogram is a special kind of plot that measures how frequently certain values, or values in certain ranges, occur in a collection of numbers that arise in your model.

For example, suppose the turtles in your model have an age variable. You could create a histogram of the distribution of ages among your turtles with the histogram command, like this:

```
histogram [age] of turtles
```

The numbers you want to histogram don't have to come from an agentset; they could be any list of numbers.

Note that using the histogram command doesn't automatically switch the current plot pen to bar mode. If you want bars, you have to set the plot pen to bar mode yourself. (As we said before, you can change a pen's default mode by editing the plot in the Interface tab.)

Like other types of plots, histograms can be set to auto scale. However, auto scaled histograms do not automatically resize themselves horizontally like other plot types do. To set the range programmatically, you can use the `set-plot-x-range` primitive.

The width of the bars in a histogram is controlled by the plot pen's interval. You can set a plot pen's default interval by editing the plot in the Interface tab. You can also change the interval temporarily with the `set-plot-pen-interval` command or the `set-histogram-num-bars`. If you use the latter command, NetLogo will set the interval appropriately so as to fit the specified number of bars within the plot's current x range.

Code Example: Histogram Example

Clearing and resetting

You can clear the current plot with the `clear-plot` command, or clear every plot in your model with `clear-all-plots`. The `clear-all command` also clears all plots, in addition to clearing

everything else in your model.

If you want to remove only the points that a particular pen has drawn, use `plot-pen-reset`.

When a whole plot is cleared, or when a pen is reset, that doesn't just remove the data that has been plotted. It also restores the plot or pen to its default settings, as they were specified in the Interface tab when the plot was created or last edited. Therefore, the effects of such commands as `set-plot-x-range` and `set-plot-pen-color` are only temporary.

Ranges and auto scaling

The default x and y ranges for a plot are fixed numbers, but they can be changed at setup time or as the model runs.

To change the ranges at any time, use `set-plot-x-range` and `set-plot-y-range`. Or, you can let the ranges grow automatically. Either way, when the plot is cleared the ranges will return to their default values.

By default, all NetLogo plots have the auto scaling feature enabled. This means that if the model tries to plot a point which is outside the current displayed range, the range of the plot will grow along one or both axes so that the new point is visible. Histogram plots, however, do not auto scale horizontally.

In the hope that the ranges won't have to change every time a new point is added, when the ranges grow they leave some extra room: 25% if growing horizontally, 10% if growing vertically.

If you want to turn off this feature, edit the plot and uncheck the Auto Scale? checkbox. At present, it is not possible to enable or disable this feature only on one axis; it always applies to both axes.

Using a Legend

You can show the legend of a plot by checking the "Show legend" checkbox in the edit dialog. If you don't want a particular pen to show up in the legend you can uncheck the "Show in Legend" checkbox for that pen also in the advanced plot pen settings (the advanced plot pen settings can be opened by clicking the pencil button for that pen in the plot pens table in the plot edit dialog).

Temporary plot pens

Most plots can get along with a fixed number of pens. But some plots have more complex needs; they may need to have the number of pens vary depending on conditions. In such cases, you can make "temporary" plot pens from code and then plot with them. These pens are called "temporary" because they vanish when the plot is cleared (by the `clear-plot`, `clear-all-plots`, or `clear-all` commands).

To create a temporary plot pen, use the `create-temporary-plot-pen` command. Typically, this would be done in the Code tab, but it is also possible to use this command from plot setup or plot update code (in the edit dialog). By default, the new pen is down, is black in color, has an interval of 1, and plots in line mode. Commands are available to change all of these settings; see the Plotting section of the NetLogo Dictionary.

Before you can use the pen, you'll have to use the use the `set-current-plot` and `set-`

`current-plot-pen` commands. These are explained in the next section.

set-current-plot and set-current-plot-pen

Before NetLogo 5, it was not possible to put plot commands in the plot itself. All of the plot code was written in the Code tab with the rest of the code. For backwards compatibility, and for temporary plot pens, this is still supported. Models in previous versions of NetLogo (and those using temporary plot pens) have to explicitly state which plot is the current plot with the `set-current-plot` command and which pen is the current pen with the `set-current-plot-pen` command.

To set the current plot use the `set-current-plot` command with the name of the plot enclosed in double quotes, like this:

```
set-current-plot "Distance vs. Time"
```

The name of the plot must be exactly as you typed it when you created the plot. Note that later if you change the name of the plot, you'll also have to update the `set-current-plot` calls in your model to use the new name. (Copy and paste can be helpful here.)

For a plot with multiple pens, you can manually specify which pen you want to plot with. If you don't specify a pen, plotting will take place with the first pen in the plot. To plot with a different pen, the `set-current-plot-pen` command was used with the name of the pen enclosed in double quotes, like this:

```
set-current-plot-pen "distance"
```

Once the current pen is set, then commands like `plot count turtles` can be executed for that pen.

Older models with plots usually had their own `do-plotting` procedure that looked something like this:

```
to do-plotting
  set-current-plot "populations"
  set-current-plot-pen "sheep"
  plot count sheep
  set-current-plot-pen "wolves"
  plot count wolves

  set-current-plot "next plot"
  ...
end
```

Once again, this is no longer necessary in NetLogo 5, unless you are using temporary plot pens.

Conclusion

Not every aspect of NetLogo's plotting system has been explained here. See the Plotting section of the NetLogo Dictionary for information on additional commands and reporters related to plotting.

Many of the Sample Models in the Models Library illustrate various advanced plotting techniques. Also check out the following code examples:

Strings

Strings may contain any Unicode characters.

To input a constant string in NetLogo, surround it with double quotes.

The empty string is written by putting nothing between the quotes, like this: `""`.

Most of the list primitives work on strings as well:

```
but-first "string" => "tring"
but-last "string" => "strin"
empty? "" => true
empty? "string" => false
first "string" => "s"
item 2 "string" => "r"
last "string" => "g"
length "string" => 6
member? "s" "string" => true
member? "rin" "string" => true
member? "ron" "string" => false
position "s" "string" => 0
position "rin" "string" => 2
position "ron" "string" => false
remove "r" "string" => "sting"
remove "s" "strings" => "tring"
replace-item 3 "string" "o" => "strong"
reverse "string" => "gnirts"
```

A few primitives are specific to strings, such as `is-string?`, `substring`, and `word`:

```
is-string? "string" => true
is-string? 37 => false
substring "string" 2 5 => "rin"
word "tur" "tle" => "turtle"
```

Strings can be compared using the =, !=, <, >, <=, and >= operators.

If you need to embed a special character in a string, use the following escape sequences:

- `\n` = newline
- `\t` = tab
- `\"` = double quote
- `\\` = backslash

Output

This section is about output to the screen. Output to the screen can also be later saved to a file using the `export-output` command. If you need a more flexible method of writing data to external files, see the next section, File I/O.

The basic commands for generating output to the screen in NetLogo are `print`, `show`, `type`, and `write`. These commands send their output to the Command Center.

For full details on these four commands, see their entries in the NetLogo Dictionary. Here is

how they are typically used:

- `print` is useful in most situations.
- `show` lets you see which agent is printing what.
- `type` lets you print several things on the same line.
- `write` lets you print values in a format which can be read back in using `file-read`.

A NetLogo model may optionally have an "output area" in its Interface tab, separate from the Command Center. To send output there instead of the Command Center, use the `output-print`, `output-show`, `output-type`, and `output-write` commands.

The output area can be cleared with the `clear-output` command and saved to a file with `export-output`. The contents of the output area will be saved by the `export-world` command. The `import-world` command will clear the output area and set its contents to the value in imported world file. It should be noted that large amounts of data being sent to the output area can increase the size of your exported worlds.

If you use `output-print`, `output-show`, `output-type`, `output-write`, `clear-output`, or `export-output` in a model which does not have a separate output area, then the commands apply to the output portion of the Command Center.

File I/O

In NetLogo, there is a set of primitives that give you the power to interact with outside files. They all begin with the prefix **file-**.

There are two main modes when dealing with files: reading and writing. The difference is the direction of the flow of data. When you are reading in information from a file, data that is stored in the file flows into your model. On the other hand, writing allows data to flow out of your model and into a file.

When working with files, always begin by using the primitive `file-open`. This specifies which file you will be interacting with. None of the other primitives work unless you open a file first.

The next **file-** primitive you use dictates which mode the file will be in until the file is closed, reading or writing. To switch modes, close and then reopen the file.

The reading primitives include `file-read`, `file-read-line`, `file-read-characters`, and `file-at-end?` Note that the file must exist already before you can open it for reading.

> **Code Examples:** File Input Example

The primitives for writing are similar to the primitives that print things in the Command Center, except that the output gets saved to a file. They include `file-print`, `file-show`, `file-type`, and `file-write`. Note that you can never "overwrite" data. In other words, if you attempt to write to a file with existing data, all new data will be appended to the end of the file. (If you want to overwrite a file, use `file-delete` to delete it, then open it for writing.)

> **Code Examples:** File Output Example

When you are finished using a file, you can use the command `file-close` to end your session with the file. If you wish to remove the file afterwards, use the primitive `file-delete` to delete it. To close multiple opened files, one needs to first select the file by using `file-open` before

closing it.

```
;; Open 3 files
file-open "myfile1.txt"
file-open "myfile2.txt"
file-open "myfile3.txt"

;; Now close the 3 files
file-close
file-open "myfile2.txt"
file-close
file-open "myfile1.txt"
file-close
```

Or, if you know you just want to close every file, you can use file-close-all.

Two primitives worth noting are file-write and file-read . These primitives are designed to easily save and retrieve NetLogo constants such as numbers, lists, booleans, and strings. file-write will always output the variable in such a manner that file-read will be able to interpret it correctly.

```
file-open "myfile.txt"  ;; Opening file for writing
ask turtles
  [ file-write xcor file-write ycor ]
file-close

file-open "myfile.txt"  ;; Opening file for reading
ask turtles
  [ setxy file-read file-read ]
file-close
```

> **Code Examples:** File Input Example and File Output Example

Letting the user choose

The user-directory, user-file, and user-new-file primitives are useful when you want the user to choose a file or directory for your code to operate on.

Movies

This section describes how to capture a QuickTime movie of a NetLogo model.

First, use the movie-start command to start a new movie. The filename you provide should end with .mov, the extension for QuickTime movies.

To add a frame to your movie, use either movie-grab-view or movie-grab-interface, depending on whether you want the movie to show just the current view, or the entire Interface tab. In a single movie, you must use only one **movie-grab-** primitive or the other; you can't mix them.

When you're done adding frames, use movie-close.

```
;; export a 30 frame movie of the view
setup
movie-start "out.mov"
movie-grab-view ;; show the initial state
repeat 30
[ go
  movie-grab-view ]
```

```
movie-close
```

By default, a movie will play back at 15 frames per second. To make a movie with a different frame rate, call `movie-set-frame-rate` with a different number of frames per second. You must set the frame rate after `movie-start` but before grabbing any frames.

To check the frame rate of your movie, or to see how many frames you've grabbed, call `movie-status`, which reports a string that describes the state of the current movie.

To throw away a movie and delete the movie file, call `movie-cancel`.

> **Code Example:** Movie Example

Movies can only be generated from the NetLogo GUI, not when running headless, or by background runs in a parallel BehaviorSpace experiment.

NetLogo movies are exported as uncompressed QuickTime files. To play a QuickTime movie, you can use QuickTime Player, a free download from Apple.

Since the movies are not compressed, they can take up a lot of disk space. You will probably want to compress your movies with third-party software. The software may give you a choice of different kinds of compression. Some kinds of compression are lossless, while others are lossy. "Lossy" means that in order to make the files smaller, some of the detail in the movie is lost. Depending on the nature of your model, you may want to avoid using lossy compression, for example if the view contains fine pixel-level detail.

One software package that can compress QuickTime movies on both the Mac and Windows platforms is QuickTime Pro. On Macs, iMovie works as well. PNG compression is a good choice for lossless compression.

Perspective

The 2D and the 3D view show the world from the perspective of the observer. By default the observer is looking down on the world from the positive z-axis at the origin. You can change the perspective of the observer by using the `follow`, `ride` and `watch` observer commands and `follow-me`, `ride-me` and `watch-me` turtle commands. When in follow or ride mode the observer moves with the subject agent around the world. The difference between follow and ride is only visible in the 3D view. In the 3D view the user can change the distance behind the agent using the mouse. When the observer is following at zero distance from the agent it is actually riding the agent. When the observer is in watch mode it tracks the movements of one turtle without moving. In both views you will see a spotlight appear on the subject and in the 3D view the observer will turn to face the subject. To determine which agent is the focus you can use the `subject` reporter.

> **Code Example:** Perspective Example

Drawing

The drawing is a layer where turtles can make visible marks.

In the view, the drawing appears on top of the patches but underneath the turtles. Initially, the drawing is empty and transparent.

You can see the drawing, but the turtles (and patches) can't. They can't sense the drawing or react to it. The drawing is just for people to look at.

Turtles can draw and erase lines in the drawing using the `pen-down` and `pen-erase` commands. When a turtle's pen is down (or erasing), the turtle draws (or erases) a line behind it whenever it moves. The lines are the same color as the turtle. To stop drawing (or erasing), use `pen-up`.

Lines drawn by turtles are normally one pixel thick. If you want a different thickness, set the `pen-size` turtle variable to a different number before drawing (or erasing). In new turtles, the variable is set to 1.

Lines made when a turtle moves in a way that doesn't fix a direction, such as with `setxy` or `move-to`, the shortest path line that obeys the topology will be drawn.

Here's some turtles which have made a drawing over a grid of randomly shaded patches. Notice how the turtles cover the lines and the lines cover the patch colors. The `pen-size` used here was 2:
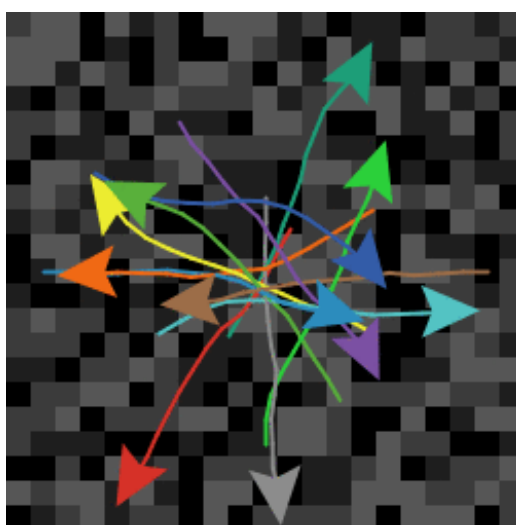
The `stamp` command lets a turtle leave an image of itself behind in the drawing and `stamp-erase` lets it remove the pixels below it in the drawing.

To erase the whole drawing, use the observer commmand `clear-drawing`. (You can also use `clear-all`, which clears everything else too.)

Importing an image

The observer command `import-drawing` command allows you to import an image file from disk into the drawing.

`import-drawing` is useful only for providing a backdrop for people to look at. If you want turtles and patches to react to the image, you should use `import-pcolors` or `import-pcolors-rgb` instead.

Comparison to other Logos

Drawing works somewhat differently in NetLogo than some other Logos.

Notable differences include:

- New turtles' pens are up, not down.
- Instead of using a `fence` command to confine the turtle inside boundaries, in NetLogo you edit the world and turn wrapping off.
- There is no `screen-color`, `bgcolor`, or `setbg`. You can make a solid background by

coloring the patches, e.g. `ask patches [ set pcolor blue ]`.

Drawing features not supported by NetLogo:

- There is no `window` command. This is used in some other Logos to let the turtle roam over an infinite plane.
- There is no `flood` or `fill` command to fill an enclosed area with color.

Topology

The way the world of patches is connected can change. By default the world is a torus which means it isn't bounded, but "wraps" -- so when a turtle moves past the edge of the world, it disappears and reappears on the opposite edge and every patch has the same number of "neighbor" patches. If you're a patch on the edge of the world, some of your "neighbors" are on the opposite edge.

However, you can change the wrap settings with the Settings button. If wrapping is not allowed in a given direction then in that direction (x or y) the world is bounded. Patches along that boundary will have fewer than 8 neighbors and turtles will not move beyond the edge of the world.

The topology of the NetLogo world has four potential values, torus, box, vertical cylinder, or horizontal cylinder. The topology is controlled by enabling or disabling wrapping in the x or y directions. The default world is a torus.

A torus wraps in both directions, meaning that the top and bottom edges of the world are connected and the left and right edges are connected. So if a turtle moves beyond the right edge of the world it appears again on the left and the same for the top and bottom.

A box does not wrap in either direction. The world is bounded so turtles that try to move off the edge of the world cannot. Note that the patches around edge of the world have fewer than eight neighbors; the corners have three and the rest have five.

Horizontal and vertical cylinders wrap in one direction but not the other. A horizontal cylinder wraps vertically, so the top of the world is connected to the bottom. but the left and right edges are bounded. A vertical cylinder is the opposite; it wraps horizontally so the left and right edges are connected, but the top and bottom edges are bounded.

> **Code Example:** Neighbors Example

When coordinates wrap, turtles and links wrap visually in the view, too. If a turtle shape or link extends past an edge, part of it will appear at the other edge. (Turtles themselves are points that take up no space, so they cannot be on both sides of the world at once, but in the view, they appear to take up space because they have a shape.)

Wrapping also affects how the view looks when you are following a turtle. On a torus, wherever the turtle goes, you will always see the whole world around it:
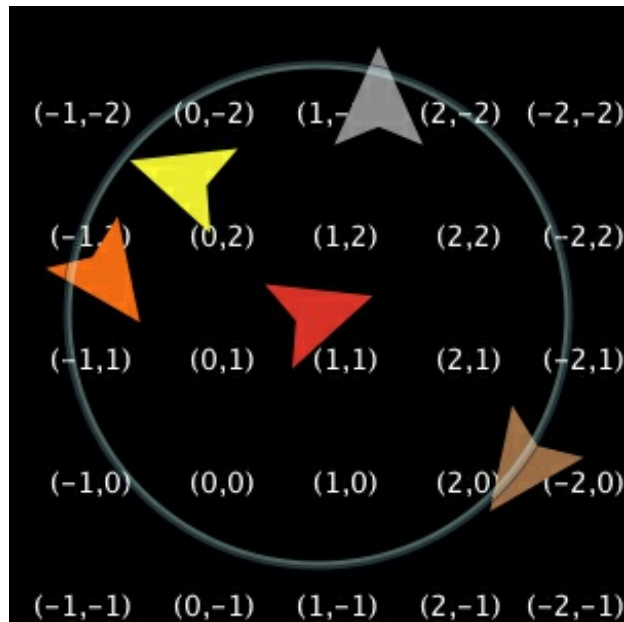
Whereas in a box or cylinder the world has edges, so the areas past those edges show up in the view as gray:
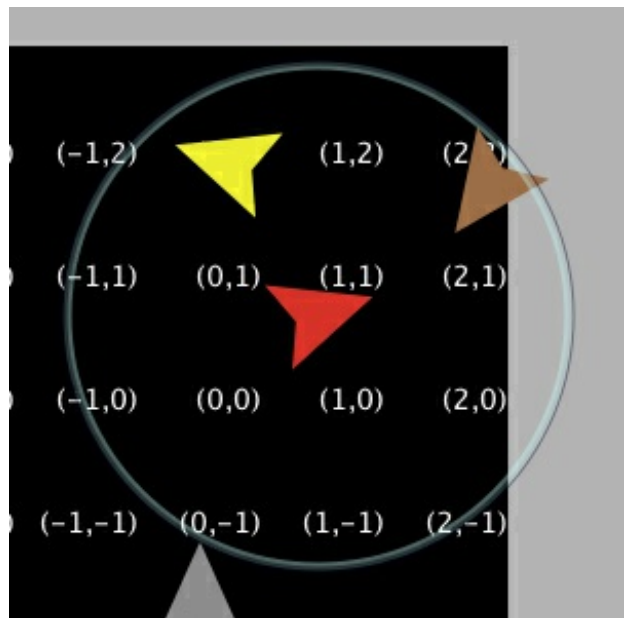
The topology settings also control the behavior of the distance(xy), in-radius, in-cone, face(xy), and towards(xy) primitives. The topology controls whether the primitives wrap or not. They always use the shortest path allowed by the topology. For example, the distance from the center of the patches in the bottom right corner (min-pxcor, min-pycor) and the upper left corner (max-pxcor, max-pycor) will be as follows for each topology given that the min and max pxcor and pycor are +/-2:

- Torus - sqrt(2) ~ 1.414 (this will be the same for all world sizes since the patches are directly diagonal to each other in a torus.)
- Box - sqrt(world-width^2 + world-height^2) ~ 7.07
- Vertical Cylinder - sqrt(world-height^2 + 1) ~ 5.099
- Horizontal Cylinder - sqrt(world-width^2 + 1) ~ 5.099

All the other primitives will act similarly to distance. If you formerly used -nowrap primitives in

your model we recommend removing them and changing the topology of the world instead.

If your model has turtles that move around you'll need to think about what happens to them when they reach the edge of the world, if the topology you're using has some non-wrapping edges. There are a few common options: the turtle is reflected back into the world (either systematically or randomly), the turtle exits the system (dies), or the turtle is hidden. It is no longer necessary to check the bounds using turtle coordinates, instead we can just ask NetLogo if a turtle is at the edge of the world. There are a couple ways of doing this, the simplest is to use the <u>can-move?</u> primitive.

```
if not can-move? distance [ rt 180 ]
```

can-move? merely returns true if the position *distance* in front of the turtle is inside the NetLogo world, false otherwise. In this case, if the turtle is at the edge of the world it simple goes back the way it came. You can also use `patch-ahead 1 != nobody` in place of <u>can-move?</u>. If you need to do something smarter that simply turning around it may be useful to use <u>patch-at</u> with <u>dx</u> and <u>dy</u>.

```
if patch-at dx 0 = nobody [
  set heading (- heading)
]
if patch-at 0 dy = nobody [
  set heading (180 - heading)
]
```

This tests whether the turtle is hitting a horizontal or vertical wall and bounces off that wall.

In some models if a turtle can't move forward it simply dies (exits the system, like in Conductor or Mousetraps).

```
if not can-move? distance[ die ]
```

If you are moving turtles using `setxy` rather than `forward` you should test to make sure the patch you are about to move to exists since `setxy` throws a runtime error if it is given coordinates outside the world. This is a common situation when the model is simulating an infinite plane and turtles outside the view should simply be hidden.

```
let new-x new-value-of-xcor
let new-y new-value-of-ycor

ifelse patch-at (new-x - xcor) (new-y - ycor) = nobody
  [ hide-turtle ]
  [ setxy new-x new-y
    show-turtle ]
```

Several models in the Models Library use this technique, Gravitation, N-Bodies, and Electrostatics are good examples.

The `diffuse` and `diffuse4` commands behave correctly in all topologies. Each patch diffuses and equal amount of the diffuse variable to each of its neighbors, if it has fewer than 8 neighbors (or 4 if you are using `diffuse4`), the remainder stays on the diffusing patch. This means that the overall sum of patch-variable across the world remains constant. However, if you want the diffuse matter to still fall off the edges of the world as it would on an infinite plane you still need to clear the edges each step as in the Diffuse Off Edges Example.

Links

A link is an agent that connects two turtles. These turtles are sometimes also called nodes.

The link is always drawn as a line between the two turtles. Links do not have a location as turtles do, they are not considered to be on any patch and you cannot find the distance from a link to another point.

There are two flavors of links, undirected and directed. A directed link is *out* of, or *from*, one node and *in*to, or *to*, another node. The relationship of a parent to a child could be modeled as a directed link. An undirected link appears the same to both nodes, each node has a link *with* another node. The relationship between spouses, or siblings, could be modeled as an undirected link.

There is a global agentset of all links, just as with turtles and patches. You can create undirected links using the `create-link-with` and `create-links-with` commands; and directed links using the `create-link-to`, `create-links-to`, `create-link-from`, and `create-links-from` commands. Once the first link has been created directed or undirected, all unbreeded links must match (links also support breeds, much like turtles, which will be discussed shortly); it's impossible to have two unbreeded links where one is directed and the other is undirected. A runtime error occurs if you try to do it. (If all unbreeded links die, then you can create links of that breed that are different in flavor from the previous links.)

In general, primitives that work with directed links have "in", "out", "to", and "from" in their names. Undirected ones either omit these or use "with".

A link's `end1` and `end2` variables contain the two turtles the link connects. If the link is directed, it goes from end1 to end2. If the link is undirected, end1 is always the older of the two turtles, that is, the turtle with the smaller who number.

Link breeds, like turtle breeds, allow you to define different types of links in your model. Link breeds must either be directed or undirected, unlike unbreeded links this is defined at compile time rather than run time. You declare link breeds using the keywords `undirected-link-breed` and `directed-link-breed`. Breeded links can be created using the commands `create-<breed>-with` and `create-<breeds>-with` for undirected breeds and the commands `create-<breed>-to`, `create-<breeds>-to`, `create-<breed>-from`, and `create-<breeds>-from` for directed links.

There cannot be more than one undirected link of the same breed (or more than one unbreeded undirected link) between a pair of agents, nor more than one directed link of the same breed in the same direction between a pair of agents. You can have two directed links of the same breed (or two unbreeded directed links) between a pair if they are in opposite directions.

Layouts

As part of our network support we have also added several different primitives that will help you to visualize the networks. The simplest is `layout-circle` which evenly spaces the agents around the center of the world given a radius.
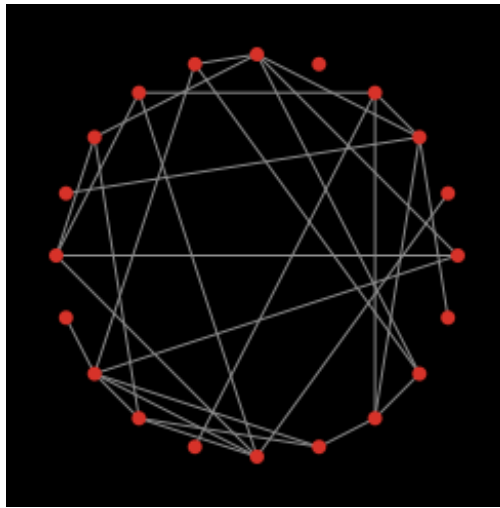
`layout-radial` is a good layout if you have something like a tree structure, though even if there are some cycles in the tree it will still work, though as there are more and more cycles it will probably not look as good. `layout-radial` takes a root agent to be the central node places it at (0,0) and arranges the nodes connected to it in a concentric pattern. Nodes one degree away from the root will be arranged in a circular pattern around the central node and the next level around those nodes and so on. `layout-radial` will attempt to account for asymmetrical graphs and give more space to branches that are wider. `layout-radial` also takes a breed as an input so you use one breed of links to layout the network and not another.
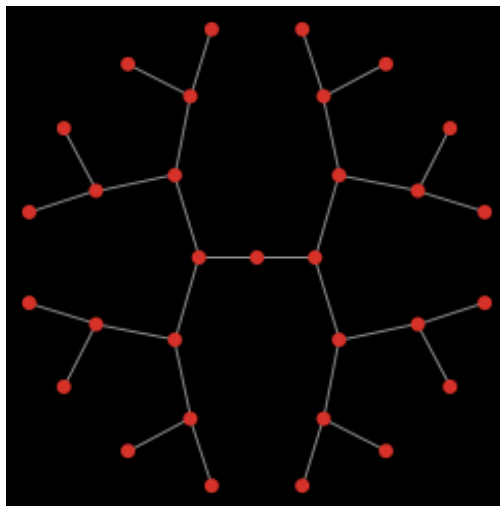
Given a set of anchor nodes `layout-tutte` places all the other nodes at the center of mass of the nodes it is linked to. The anchor set is automatically arranged in a circle layout with a user defined radius and the other nodes will converge into place (this of course means that you may have to run it several times before the layout is stable.)
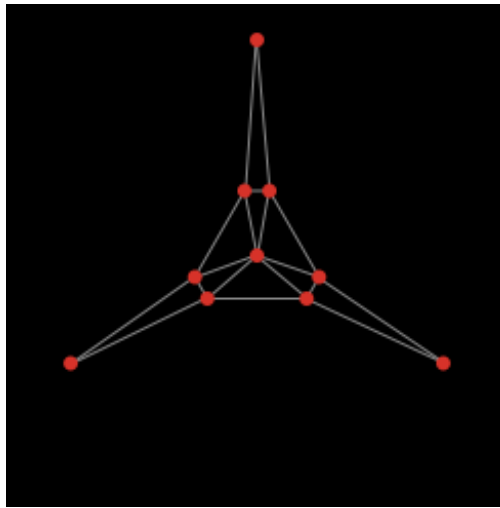
[layout-spring](#) is useful for many kinds of networks. The drawback is that is relatively slow since it takes many iterations to converge. In this layout the links act as springs that pull the nodes they connect toward each other and the nodes repel each other. The strength of the forces is controlled by inputs to the primitives. These inputs will always have a value between 0 and 1; keep in mind that very small changes can still affect the appearance of the network. The springs also have a length (in patch units), however, because of all the forces involved the nodes will not end up exactly that distance from each other.

> **Code Examples:** Network Example, Network Import Example, Giant Component, Small Worlds, Preferential Attachment

Tasks

Tasks let you store code to be run later. There are two kinds, command tasks and reporter tasks.

Tasks are values, which means that a task may be passed as input, reported as a result, or stored in a variable.

A given task might be run once, multiple times, or not at all.

In other programming languages tasks are known as first-class functions, closures, or lambda.

Task primitives

Primitives specific to tasks are `task`, `is-command-task?`, and `is-reporter-task?`.

The `task` primitive creates a task. The task it reports might be a command task or a reporter task, depending on what kind of block you pass it. For example `task [ fd 1 ]` reports a command task, because `fd` is a command, while `task [ count turtles ]` reports a reporter task, because `count` is a reporter.

These primitives require tasks as input: `foreach`, `map`, `reduce`, `filter`, `n-values`, `sort-by`. When calling these primitives, the `task` primitive is optional, so for example one may write simply `foreach mylist [ print ? ]` instead of `foreach mylist task [ print ? ]`, though the latter is also accepted. (Because `task` is optional, the syntax for these primitives is backward-compatible with existing code from previous NetLogo versions.)

The `run` command accepts command tasks as well as strings.

The `runresult` reporter accepts reporter tasks as well as strings.

`run` and `runresult` allow passing inputs to a task. As with all primitives accepting varying number of inputs, the whole call must be surrounded with parentheses, so for example `(run my-command-task 5)` or `(runresult my-reporter-task "foo" 2)`. When not passing input, no parentheses are required.

Task inputs

A task may take zero or more inputs. The inputs are referenced using the special variables `?` (aka `?1`), `?2`, `?3`, etc. Any extra inputs are ignored.

Tasks and strings

Creating and running tasks is fast. To use `run` or `runresult` on a new string for the first time is about 100x slower than running a task. Modelers should normally use tasks instead of running strings, except when running strings entered by the user.

Concise syntax

Simple uses of `foreach`, `map`, `reduce`, `filter`, `n-values`, and `sort-by` can be written with an especially concise syntax. You can write:

```
map abs [1 -2 3 -4] => [1 2 3 4]
reduce + [1 2 3 4] => 10
filter is-number? [1 "x" 3] => [1 3]
foreach [1 2 3 4] print
```

In older NetLogo versions, these had to be written:

```
map [abs ?] [1 -2 3 -4] => [1 2 3 4]
reduce [?1 + ?2] [1 2 3 4] => 10
filter [is-number? ?] [1 "x" 3] => [1 3]
foreach [1 2 3 4] [ print ? ]
```

The old syntax remains valid.

The concise syntax can also be used with the `task` primitive. So for example `task die` is short for `task [ die ]`, `task fd` is short for `task [fd ?]`, and `task +` is short for `task [?1 + ?2]`.

Tasks as closures

Tasks are "closures"; that means they capture or "close over" the bindings (not just the current values) of local variables and procedure inputs. They do not capture agent variables and do not capture the identity (or even the agent type) of the current agent.

Nonlocal exits

The `stop` and `report` commands exit from the dynamically enclosing procedure, not the enclosing task. (This is backward-compatible with older NetLogo versions.)

Tasks and extensions

The extensions API supports writing primitives that accept tasks as input. Write us for sample code.

Limitations

We hope to address at least some of the following limitations in future NetLogo versions:

- `import-world` does not support tasks.
- Task inputs can't be given names. This can make nesting reporter tasks difficult. (Command tasks can use `let` to give their inputs names, if needed.)
- Tasks can't be variadic (accept a varying number of inputs).
- Reporter tasks can't contain commands, only a single reporter expression. So for example you must use `ifelse-value` not `if`, and you don't use `report` at all. If your code is too complex to be written as one reporter, you'll need to move the code to a separate reporter procedure, and then call that procedure from your task, passing it any needed inputs.
- Tasks are not interchangeable with command blocks and reporter blocks. Only the primitives listed above accept tasks as input. Control primitives such as `ifelse` and `while` and agent primitives such as `of` and `with` don't accept tasks. So for example if I have a reporter task `r` and two command tasks `c1` and `c2`, I can't write `ifelse r c1 c2`, I must write `ifelse runresult r [ run c1 ] [ run c2 ]`.
- The concise syntax where `task` may be omitted is only available to primitives and extension primitives, not ordinary procedures. So for example if I have a procedure `p` that accepts a task as input, it must be called as e.g. `p task [ ... ]` not `p [ ... ]`.

Code example

> **Code Example:** State Machine Example

Ask-Concurrent

NOTE: The following information is included only for backwards compatibility. We don't recommend using the `ask-concurrent` primitive at all in new models.

In very old versions of NetLogo, `ask` had simulated concurrent behavior by default. Since NetLogo 4.0 (2007), `ask` is serial, that is, the agents run the commands inside the ask one at a time.

The following information describes the behavior of the `ask-concurrent` command, which behaves the way the old `ask` behaved.

`ask-concurrent` produces simulated concurrency via a mechanism of turn-taking. The first agent takes a turn, then the second agent takes a turn, and so on until every agent in the asked agentset has had a turn. Then we go back to the first agent. This continues until all of the agents have finished running all of the commands.

An agent's "turn" ends when it performs an action that affects the state of the world, such as moving, or creating a turtle, or changing the value of a global, turtle, patch, or link variable. (Setting a local variable doesn't count.)

The forward (fd) and back (bk) commands are treated specially. When used inside ask-concurrent, these commands can take multiple turns to execute. During its turn, the turtle can only move by one step. Thus, for example, `fd 20` is equivalent to `repeat 20 [ fd 1 ]`, where the turtle's turn ends after each run of `fd`. If the distance specified isn't an integer, the last fraction of step takes a full turn. So for example `fd 20.3` is equivalent to `repeat 20 [ fd 1 ] fd 0.3`.

The jump command always takes exactly one turn, regardless of distance.

To understand the difference between ask and ask-concurrent, consider the following two commands:

```
ask turtles [ fd 5 ]
ask-concurrent turtles [ fd 5 ]
```

With ask, the first turtle takes five steps forward, then the second turtle takes five steps forward, and so on.

With ask-concurrent, all of the turtles take one step forward. Then they all take a second step, and so on. Thus, the latter command is equivalent to:

```
repeat 5 [ ask turtles [ fd 1 ] ]
```

> **Code Example:** Ask-Concurrent Example shows the difference between ask and ask-concurrent.

The behavior of ask-concurrent cannot always be so simply reproduced using ask, as in this example. Consider this command:

```
ask-concurrent turtles [ fd random 10 ]
```

In order to get the same behavior using ask, we would have to write:

```
turtles-own [steps]
ask turtles [ set steps random 10 ]
while [any? turtles with [steps > 0]] [
  ask turtles with [steps > 0] [
    fd 1
    set steps steps - 1
  ]
]
```

To prolong an agent's "turn", use the without-interruption command. (The command blocks inside some commands, such as create-turtles and hatch, have an implied without-interruption around them.)

Note that the behavior of ask-concurrent is completely deterministic. Given the same code and the same initial conditions, the same thing will always happen (if you are using the same version of NetLogo and begin your model run with the same random seed).

In general, we suggest you not use ask-concurrent at all. If you do, we suggest you write your model so that it does not depend on the exact details of how ask-concurrent works. We make no guarantees that its semantics will remain the same in future versions of NetLogo, or that it will continue to be supported at all.

Tie

Tie connects two turtles so that the movement of one turtles affects the location and heading of another. Tie is a property of links so there must be a link between two turtles to create a tie relationship.

When a link's `tie-mode` is set to "fixed" or "free" `end1` and `end2` are tied together. If the link is directed `end1` is the "root agent" and `end2` is the "leaf agent". That is when `end1` moves (using `fd`, `jump`, `setxy`, etc.) `end2` also moves the same distance and direction. However when `end2` moves it does not affect `end1`.

If the link is undirected it is a reciprocal tie relationship, meaning, if either turtle moves the other turtle will also move. So depending on which turtle is moving either turtle can be considered the root or the leaf. The root turtle is always the turtle that initiates the movement.

When the root turtle turns right or left, the leaf turtle rotates around the root turtle the same amount as if a stiff were attaching the turtles. When `tie-mode` is set to "fixed" the heading of the leaf turtle changes by the same amount. If the `tie-mode` is set to "free" the heading of the leaf turtle is unchanged.

The `tie-mode` of a link can be set to "fixed" using the `tie` command and set to "none" (meaning the turtles are no longer tied) using `untie` to set the mode to "free" you need to: `set tie-mode "free"`.

> **Code Example:** Tie System Example

Multiple source files

The `__includes` keyword allows you to use multiple source files in a single NetLogo model.

The keyword begins with two underscores to indicate that the feature is experimental and may change in future NetLogo releases.

When you open a model that uses the `__includes` keyword, or if you add it to the top of a model and hit the Check button, the includes menu will appear in the toolbar. From the includes menu you can select from the files included in this model.

When you open included files they appear in additional tabs. See the [Interface Guide](#) for more details.

You can have anything in external source files (`.nls`) that you would normally put in the Code tab: `globals`, `breed`, `turtles-own`, `patches-own`, `breeds-own`, procedure definitions, etc. Note though that these declarations all share the same namespace. That is, if you declare a global `my-global` in the Code tab you cannot declare a global (or anything else) with the name `my-global` in any file that is included in the model. `my-global` will be accessible from all the included files. The same would be true if `my-global` were declared in one of the included files.

Syntax

Colors

In the Code tab and elsewhere in the NetLogo user interface, program code is color-coded by

the following scheme:

- Keywords are green
- Constants are orange
- Comments are gray
- Primitive commands are blue
- Primitive reporters are purple
- Everything else is black

Notice

The remainder of this section contains technical terminology which will be unfamiliar to some readers.

Keywords

The only keywords in the language are `globals`, `breed`, `turtles-own`, `patches-own`, `to`, `to-report`, and `end`, plus `extensions` and the experimental `__includes` keyword. (Built-in primitive names may not be shadowed or redefined, so they are effectively a kind of keyword as well.)

Identifiers

All primitives, global and agent variable names, and procedure names share a single global case-insensitive namespace; local names (`let` variables and the names of procedure inputs) may not shadow global names or each other. Identifiers may contain any Unicode letter or digit and the following ASCII characters:

```
.?=*!<>:#+/%$_^'&-
```

Some primitive names begin with two underscores to indicate that they are experimental and are especially likely to change or be removed in future NetLogo releases.

Identifiers beginning with a question mark are reserved.

Scope

NetLogo is lexically scoped. Local variables (including inputs to procedures) are accessible within the block of commands in which they are declared, but not accessible by procedures called by those commands.

Comments

The semicolon character introduces a comment, which lasts until the end of the line. There is no multi-line comment syntax.

Structure

A program consists of optional declarations (`globals`, `breed`, `turtles-own`, `patches-own`, `<BREED>-own`, `extensions`) in any order, followed by zero or more procedure definitions. Multiple breeds may be declared with separate `breed` declarations; the other declarations may

appear once only.

Every procedure definition begins with <u>to</u> or <u>to-report</u>, the procedure name, and an optional bracketed list of input names. Every procedure definition ends with <u>end</u>. In between are zero or more commands.

Commands and reporters

Commands take zero or more inputs; the inputs are reporters, which may also take zero or more inputs. No punctuation separates or terminates commands; no punctuation separates inputs. Identifiers must be separated by whitespace or by parentheses or square brackets. (So for example, `a+b` is a single identifier, but `a(b[c]d)e` contains five identifiers.)

All commands are prefix. All user-defined reporters are prefix. Most primitive reporters are prefix, but some (arithmetic operators, boolean operators, and some agentset operators like with and in-points) are infix.

All commands and reporters, both primitive and user-defined, take a fixed number of inputs by default. (That's why the language can be parsed though there is no punctuation to separate or terminate commands and/or inputs.) Some primitives are variadic, that is, may optionally take a different number of inputs than the default; parentheses are used to indicate this, e.g. `(list 1 2 3)` (since the <u>list</u> primitive only takes two inputs by default). Parentheses are also used to override the default operator precedence, e.g. `(1 + 2) * 3`, as in other programming languages.

Sometimes an input to a primitive is a command block (zero or more commands inside square brackets) or a reporter block (a single reporter expression inside square brackets). User-defined procedures may not take a command or reporter block as input.

Operator precedences are as follows, high to low:

- <u>with</u>, <u>at-points</u>, <u>in-radius</u>, <u>in-cone</u>
- (all other primitives and user-defined procedures)
- `^`
- `*`, `/`, `mod`
- `+`, `-`
- `<`, `>`, `<=`, `>=`
- `=`, `!=`
- <u>and</u>, <u>or</u>, <u>xor</u>

Compared to other Logos

There is no agreed-upon standard definition of Logo; it is a loose family of languages. We believe that NetLogo has enough in common with other Logos to earn the Logo name. Still, NetLogo differs in some respects from most other Logos. The most important differences are as follows.

Surface differences

- The precedence of mathematical operators is different. Infix math operators (like `+`, `*`, etc.) have lower precedence than reporters with names. For example, in many Logos, if you write `sin x + 1`, it will be interpreted as `sin (x + 1)`. NetLogo, on the other hand, interprets it the way most other programming languages would, and the way the same expression would be interpreted in standard mathematical notation, namely as `(sin x)`

```
+ 1.
```

- The `and` and `or` reporters are special forms, not ordinary functions, and they "short circuit", that is, they only evaluate their second input if necessary.
- Procedures can only be defined in the Code tab, not interactively in the Command Center.
- Reporter procedures, that is, procedures that "report" (return) a value, must be defined with `to-report` instead of `to`. The command to report a value from a reporter procedure is `report`, not `output`.
- When defining a procedure, the inputs to the procedure must be enclosed in square brackets, e.g. `to square [x]`.
- Variable names are always used without any punctuation: always `foo`, never `:foo` or `"foo`. (To make this work, instead of a `make` command taking a quoted argument we supply a `set` special form which does not evaluate its first input.) As a result, procedures and variables occupy a single shared namespace.

The last three differences are illustrated in the following procedure definitions:

most Logos

```
to square :x
output :x * :x
end
```

NetLogo

```
to-report square [x]
report x * x
end
```

Deeper differences

- NetLogo's local variables and inputs to procedures are lexically scoped, not dynamically scoped.
- NetLogo has no "word" data type (what Lisp calls "symbols"). Eventually, we may add one, but since it is seldom requested, it may be that the need doesn't arise much in agent-based modeling. We do have strings. In most situations where traditional Logo would use words, we simply use strings instead. For example in Logo you could write `[see spot run]` (a list of words), but in NetLogo you must write `"see spot run"` (a string) or `["see" "spot" "run"]` (a list of strings) instead.
- NetLogo's `run` command works on tasks and strings, not lists (since we have no "word" data type), and does not permit the definition or redefinition of procedures.
- Control structures such as `if` and `while` are special forms, not ordinary functions. You can't define your own special forms, so you can't define your own control structures. (You can do something similar using tasks, but you must use the `task`, `run`, and `runresult` primitives for that, you cannot make them implicit.)
- Tasks (aka function values or lambda) are true lexically-scoped closures. This feature is available in NetLogo and in modern Lisps, but not in standard Logo.

Of course, the NetLogo language also contains other features not found in most Logos, most importantly agents and agentsets.

Transition Guide

Many models created in earlier versions of NetLogo also work in NetLogo 5.0. However, some models will need changes. If an old model isn't working, this section of the User Manual may be able to help you.

What issues may arise depends on what version of NetLogo the model was created with.

This guide only covers changes most likely to cause issues for users. See the Release notes for more complete details on differences between versions.

- Changes for NetLogo 5.2
- Changes for NetLogo 5.0
- Changes for NetLogo 4.1
- Changes for NetLogo 4.0
- Changes for NetLogo 3.1

Changes for NetLogo 5.2

hsb primitives

In 5.2, the hsb primitives have been changed to work with the standard scale values of 360 for hue, and 100 for saturation and brightness. This affects the primitives hsb, extract-hsb and approximate-hsb.

The old primitives, scaled to 255, are automatically transitioned to and have been renamed __hsb-old, __extract-hsb-old and __approximate-hsb-old.

GoGo extension

The GoGo extension has been upgraded to use newer GoGo boards with the HID interface. Many of the older primitives no longer work, and will alert you to upgrading your GoGo board's firmware.

If you need to continue to use a serial interface, you can use the bundled gogo-serial extension. Change your model to use gogo-serial as opposed to gogo. More details [here] (https://github.com/NetLogo/NetLogo/wiki/GoGo-Upgrade).

If you cannot upgrade to use the new HID extension nor the new serial extension, the original extension can be found at https://github.com/NetLogo/GoGo-RXTX-Extension

Changes for NetLogo 5.0

Plotting

In 5.0, you don't have to put your plotting code in the Code tab anymore. Instead, you can put it inside the plots themselves, in the Interface tab.

Nonetheless, the old style and all of the existing plotting primitives are still supported. We recommend changing your model to use the new style, but if you don't, it should still work.

The following example shows how to change a model to use the new style. Suppose you

have a typical NetLogo 4.1 model with one plot called "populations" and two pens called "robots" and "humans". The old code might look like:

```
to setup
  clear-all
  ...
  do-plotting
end

to go
  ...
  tick
  do-plotting
end

to do-plotting
  set-current-plot "populations"
  set-current-plot-pen "robots"
  plot count robots
  set-current-plot-pen "humans"
  plot count humans
end
```

Here are the steps to make the transition:

- Copy the `plot count robots` command and paste it into the Update Commands field for the robots pen in the plot edit dialog. Remove it from the `do-plotting` procedure.
- The `plot count humans` command can be moved in the same way for the humans pen.
- After those lines are removed from the `do-plotting` procedure, it doesn't actually do anything anymore! Remove it.
- The final step is to replace the `do-plotting` procedure calls in `setup` and `go`. In `setup`, the `do-plotting` call should be changed to `reset-ticks`. In `go`, the do-plotting call should be changed to `tick`. `reset-ticks` and `tick` will both cause plotting to happen automatically.

The resulting (much simpler) code looks like this:

```
to setup
  clear-all
  ...
  reset-ticks
end

to go
  ...
  tick
end
```

For more details on how plotting works in NetLogo 5.0, see the Plotting Sections of the Programming Guide and the Interface Guide. For details on how plotting interacts with the tick counter, read on.

Tick counter

The way the tick counter works has changed in 5.0. Instead of being initially set to 0, the tick counter is initially blank.

`reset-ticks`

You must use `reset-ticks` to start the tick counter at 0 before using `ticks`, `tick` or `tick-advance` for the first time.

`reset-ticks` should go at the end of your setup procedure. Putting it there will allow your model to work with 5.0's new plotting features.

`reset-ticks` and plotting

In 5.0, you don't have to put your plotting code in the Code tab anymore. Instead, you can put it inside the plots themselves, in the Interface tab. Code inside plots is triggered by `reset-ticks` and `tick`. Resetting the tick counter runs plot setup code, and then it also runs plot update code to plot the initial state of the model. The initial state of the model won't be in place until the end of `setup`, so that's why `reset-ticks` should go at the end.

`__clear-all-and-reset-ticks`

In order for models from previous NetLogo versions to work in 5.0 without changes, when an old model is opened in 5.0, any occurrences of `clear-all` (or `ca`) are automatically changed to `__clear-all-and-reset-ticks`, which combines the effects of `clear-all` and `reset-ticks`. The two underscores on the name indicate that this is not a normal primitive, but exists only for backwards compatibility.

You should remove `__clear-all-and-reset-ticks` from your code, replace it with `clear-all`, and put `reset-ticks` at the end of your setup procedure. (This doesn't happen automatically because the structure of NetLogo models is too free-form for an automatic converter to reliably make the change for you.)

Unicode characters

NetLogo 5.0 fully supports international characters cross-platform, using the Unicode character set. NetLogo 5.0 model files always represent Unicode characters using the UTF-8 encoding.

Previous versions of NetLogo allowed Unicode characters to be used in some contexts. However, model files were saved in the platform's default encoding, which on most systems was something other than UTF-8. Characters were handled correctly on the same platform (e.g. two Windows machines), but could be altered if the model was moved between platforms (e.g. from Windows to Mac or vice versa).

When opening an existing model in NetLogo 5.0, if the model contains international or other non-ASCII characters, the characters may be interpreted incorrectly, because they were originally written in a platform-specific encoding, but then read back in in UTF-8.

If only a few characters are affected, you might find it easiest just to fix them manually.

But if you expect a large number of characters to be affected, and you want them translated automatically, you can use a third party utility to re-encode your .nlogo file from its original encoding into UTF-8. After conversion, open the model in NetLogo 5.0 and all characters should be correct.

Info tabs

NetLogo 5.0 uses the Markdown markup language to allow you to format your Info tab, including headers, bold and italics, images, and so forth.

Earlier versions of NetLogo used a custom markup language with much more limited

capabilities.

When opening a model from an older version, NetLogo 5.0 translates your old markup into Markdown. Most of the time this produces good results, but you may want to check the results yourself and make sure that your Info tab still looks good.

Model speed

In NetLogo 5.0 every model has a "target frame rate" which affects the default speed at which the model runs, when the speed slider is in the middle, on the "normal speed" setting.

The default target frame rate for new models, and for models that were created in earlier versions of NetLogo, is 30 frames per second. If you are using tick-based updates, as we recommend for most models, then that translates to 30 ticks per second.

If your model runs slower in 5.0 than it ran in 4.1, it's probably just because its speed is being limited by this rate. If you want, you can press the Settings button in the Interface tab and change the frame rate to a higher number.

Some old models used the `every` command to set a default speed. In most case this can be now removed from the code, and the target frame rate setting used instead.

List performance

The underlying data structure for NetLogo lists has changed.

In NetLogo 4.1, a NetLogo list was represented internally as a singly linked list. Some operations on singly linked lists are fast (such as `first` and `butfirst`) but others are slow because they could require traversing the whole list (such as `item` and `last`).

In NetLogo 5.0, lists are now actually trees internally. As a result, some operations are a little slower, but other operations are drastically faster on long lists. See the Lists section of the Programming Guide for details.

Some models may run a little slower with the new data structure, especially if you make heavy use of short lists. But other models will run faster -- perhaps dramatically faster.

Some special ways of writing list-processing code that were useful in NetLogo 4.1 are no longer needed in 5.0. For example, since in 4.1 `fput` was fast and `lput` was slow, modelers sometimes built up lists in reverse order using `fput`, perhaps calling `reverse` later to restore the intended order. In NetLogo 5.0, you don't need to code that way anymore. `fput` and `lput` are the same speed.

Extensions API

If you are the author of an extension, you will need to recompile it against the 5.0 NetLogo.jar and lib directory for it to work with 5.0.

You may also need to be aware of the following changes:

Syntax constants

The code for specifying the syntax of a primitive has changed slightly, for example `Syntax.TYPE_STRING` is now `Syntax.StringType()`. (From Java, the pair of parentheses at the

end is required. In Scala, you can omit them.)

LogoList construction

One significant change is that `org.nlogo.api.LogoList` no longer has a public constructor. Instead, there are two new ways to construct a `LogoList`.

If you have a `java.lang.Iterable`, you can copy the contents into a fresh `LogoList` by passing it to the static method `LogoList.fromJava()`. See the array extension source code for a sample usage.

Or, to build up a new list one item a time, use `org.nlogo.api.LogoListBuilder`. The [Extensions Guide](#) has sample code showing the use of `LogoListBuilder`.

Primitive classes

In prior NetLogo versions, the extensions API required that each extension primitive have its own separate top-level class with a no-argument constructor. These limitations have now been lifted. Also, api.Primitive objects are now made only once, when the extension is loaded, instead of every time the Code tab was recompiled.

Changes for NetLogo 4.1

Combining `set` and `of`

The following syntax is no longer supported:

```
set [<variable>] of <agent> <value>
```

Commands of this form must be rewritten using `ask`:

```
ask <agent> [ set <variable> <value> ]
```

Or, if the new value must be computed by the asking agent and not by the agent whose variable is being set:

```
;; OPTION #1 (using let):
let new-value <value>
ask <agent> [ set <variable> new-value ]

;; OPTION #2 (using myself):
ask <agent> [ set <variable> [value] of myself ]
```

So for example, this:

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

Can be rewritten as:

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

It is not necessary to use `let` or `myself` since red is red from the point of view of both agents.

However, this:

```
set [color] of turtle 0 color
```

Must be rewritten as:

```
let new-color color
ask turtle 0 [ set color new-color ]
```

or

```
ask turtle 0 [ set color [color] of myself ]
```

in order not to change the meaning, since the two agents may have different starting values for `color`. The form using `myself` is briefer, but the former using `let` may be considered clearer, depending on context and individual preference.

Changes for NetLogo 4.0

Who numbering

Prior to NetLogo 4.0, a dead turtle's who number (stored in the `who` turtle variable) could be reassigned to a later newborn turtle. In NetLogo 4.0, who numbers are never reused until who numbering is reset to 0 by the `clear-all` or `clear-turtles` command. This change in behavior may break a few old models.

Turtle creation: randomized vs. "ordered"

NetLogo 4.0 provides two different observer commands for creating turtles, `create-turtles` (`crt`) and `create-ordered-turtles` (`cro`).

`crt` gives the new turtles random colors and random integer headings. `cro` assigns colors sequentially and gives the turtles sequential equally spaced headings, with the first turtle facing north (heading of 0).

Prior to NetLogo 4.0, the `crt` command behaved the way `cro` does now. If your old model depends on the "ordered" behavior, you will need to change your code to use `cro` instead of `crt`.

It is common for old models that used `crt` to contain extra commands to randomize the new turtles' headings, for example `rt random 360` or `set heading random 360`. These commands are no longer necessary when used inside `crt`.

Adding strings and lists

Prior to NetLogo 4.0, the `+` (addition) operator could be used to concatenate strings and join lists. In current NetLogo, + only works on numbers. To concatenate strings, use the `word` primitive; to join lists together, use the `sentence` primitive. This language change was made to

increase the speed of code that uses `+`.

Old code:

```
print "There are " + count turtles + " turtles."
```

New code:

```
print (word "There are " count turtles " turtles.")
```

Likewise, if you need to concatenate lists, use SENTENCE.

This change is not handled automatically when converting old models; users will need to change their code by hand.

We know this change will be awkward for users who are used to the old syntax. We have made this change for efficiency and consistency. We can implement an addition operator that only adds numbers much more efficiently than one that handles several different data types. Because addition is such a common operation, NetLogo's overall speed is affected.

The `-at` primitives

The observer may no longer use `patch-at`, `turtles-at`, and `BREEDS-at`. Use `patch`, `turtles-on patch`, and `BREEDS-on patch` instead. Note that `patch` now rounds its inputs (before it only accepted integer inputs).

Links

NetLogo 3.1 had supports for using links to connect turtles to make networks, graphs, and geometric figures. The links were themselves turtles.

In NetLogo 4.0, instead of links being turtles, links are now an independent fourth agent type, right alongside observer, turtles, patches. The primitives involving links are no longer considered experimental; they are now fully part of the language.

Models that use the old, experimental turtle-based link primitives will need to be updated to use link agents. The differences are not huge, but hand updating is required.

Links are documented in the Links section of the Programming Guide, and in the NetLogo Dictionary entries for the link primitives. See the Networks section of the Models Library for example models that use links. There are also some link-based Code Examples.

First you will need to remove any breeds called "links" if you are only using one type of links then you will not have to use breeds at all. If you are using multiple types of links see `undirected-link-breed` and `directed-link-breed`. Commands and reporters that contain the word "links" (like `__create-links-with`, etc.) will automatically be converted to the new form without underscores (`create-links-with`). However, primitives that use a different breed name (such as "edges") will not be converted. You will need to remove the underscores by hand and unless you are declaring a link breed with that name you will need to change the breed designation to "links".

The commands `remove-link(s)-with/from/to` no longer exist. Instead you should ask the links in question to `die`.

For example:

```
ask turtle 0 [ __remove-links-with link-neighbors ]
```

becomes

```
ask turtle 0 [ ask my-links [ die ] ]
```

Several of the layout commands have slightly different inputs, the first two inputs are generally a turtle agentset and a link agentset to perform the layout on. See the dictionary entries for details. layout-spring, layout-radial layout-tutte

You may also need to rearrange the declaration of turtles-own variables, since links were once actually turtles. Any variables that apply to links should be moved into a links-own block.

Since links are no longer turtles they no longer have the built-in turtle variables (though some of the link variables are the same such as color and label. If you formerly used the location of link turtles you will now need to calculate the midpoint of the link. This is fairly simple in a non-wrapping world.

```
to-report link-xcor
  report mean [xcor] of both-ends
end

to-report link-ycor
  report mean [ycor] of both-ends
end
```

it is a little bit trickier in a wrapping world but still fairly straightforward.

```
to-report link-xcor
  let other-guy end2
  let x 0
  ask end1
  [
    hatch 1
    [
      face other-guy
      fd [distance other-guy] of myself / 2
      set x xcor
      die
    ]
  ]
  report x
end
```

and similarly for ycor.

If you used either the size or heading of the link turtles you can use the reporters link-length and link-heading instead.

New "of" syntax

We have replaced three different language constructs, -of (with hyphen), value-from, and values-from with a single of construct (no hyphen).

old	new

color-of turtle 0	[color] of turtle 0
value-from turtle 0 [size * size]	[size * size] of turtle 0
mean values-from turtles [size]	mean [size] of turtles

When `of` is used with a single agent, it reports a single value. When used with an agentset, it reports a list of values (in random order, since agentsets are always in random order).

Note that when opening old models in the new version, `-of`, `value-from`, and `values-from` will automatically be converted to use "of" instead, but some nested uses of these constructs are too complex for the converter and must be converted by hand.

Serial `ask`

The `ask` command is now serial rather than concurrent. In other words, the asked agents will run one at a time. Not until one agent completely finishes the entire body of the `ask` does the next agent start.

Note that even the old `ask` was never truly concurrent; we simulated concurrent execution by interleaving execution among the agents using a turn-taking mechanism described in the NetLogo FAQ.

We have made this change because in our experience, users often wrote models that behaved in unexpected ways due to the simulated concurrency, but rarely wrote models that benefited from the simulated concurrency. Models exhibiting unexpected behavior could usually be fixed by adding the `without-interruption` command in the right places, but it was difficult for users to know whether that command was needed and if so, where.

In NetLogo 4.0, `without-interruption` is no longer necessary unless your model uses `ask-concurrent` (or a turtle or patch forever button containing code that depends on simulated concurrency). In most models, all uses of `without-interruption` can be removed.

The simulated concurrency formerly employed by "ask" is still accessible in three ways:

- You may use the `ask-concurrent` primitive instead of `ask` to get the old simulated concurrency. (We don't recommend this, though.)
- Commands issued in the Command Center directly to turtles, patches, or links have an implied `ask-concurrent`.
- Turtle, patch, and link forever buttons have an implied `ask-concurrent` as well.

Note that `ask` itself is always serial regardless of the context in which it is used, however.

In our own Models Library, models that make use of this concurrency are rare. A prominent example, though, is Termites, which uses a concurrent turtle forever button.

Tick counter

NetLogo now has a built-in tick counter for representing the passage of simulated time.

You advance the counter by one using the `tick` command. If you need to read its value, there's a reporter called `ticks`. The `clear-all` command resets the tick counter; so does `reset-ticks`.

In most models the tick counter will be integer-valued, but if you want to use smaller increments of time, you can use the `tick-advance` command to advance the tick counter by any positive amount, including fractional amounts. Some Models Library models that use `tick-advance` are Vector Fields and the GasLab models.

The value of the tick counter is displayed in the toolbar at the top of the Interface tab. (You can use the Settings... button in the toolbar to hide the tick counter, or change the word "ticks" to something else.)

View update modes

In the past, NetLogo always tried to update the view about 20 times a second. We're now calling that "continuous" view updates. The biggest problem with it was that you usually want updates to happen between model ticks, not in the middle of a tick, so we had a checkbox on buttons that (by default) forced a display update after every button iteration. That made sure updates happened between ticks, but it didn't get rid of the intermediate updates. You had to use `no-display` and `display` to lock them out.

We still support continuous updates. They are the default when you start up NetLogo. But most Models Library models now use tick-based updates. With tick-based updates, updates happen only when the tick counter advances. (The `display` command can be used to force additional updates; see below.)

The advantages of tick-based updates as we see them are as follows:

1. Consistent, predictable view update behavior which does not vary from computer to computer or from run to run.
2. Intermediate updates can confuse the user of your model by letting them see things they aren't supposed to see, which may be misleading.
3. Increased speed. Updating the view takes time, so if one update per tick is enough, then enforcing than there is only one update per tick will make your model faster.
4. Instead of having a "force view update" checkbox in every button like in NetLogo 3.1, we only need one choice which applies to the entire model.
5. Using the speed slider to slow down a model now just inserts pauses between ticks. So with tick-based updates, `setup` buttons are no longer affected by the speed slider. This was a real annoyance with the old speed slider. (The annoyance persists for models that use continuous updates, though.)

As mentioned above, most models in our Models Library now use tick-based updates.

Even for models that would normally be set to tick-based updates, it may be useful to switch to continuous updates temporarily for debugging purposes. Seeing what's going on within a tick, instead of only seeing the end result of a tick, could help with troubleshooting.

If you switch your model to use tick-based updates, you'll also need to add the `tick` command to your code, otherwise the view won't update. (Note that the view still always updates when a button pops up or a command entered in the command center finishes, though. So it's not like the view will just stay frozen indefinitely.)

How to make a model use ticks and tick-based updates

Here are the steps to follow to convert your model to use ticks and tick-based updates in NetLogo 4.0:

1. In the Interface tab toolbar, on the right hand side where it says "update view:", change the setting from "continuously" to "on ticks".
2. Add the `tick` command to your go procedure, at or near the end. In Models Library models we always put `tick` after the agents move but before any plotting commands. That's because the plotting commands might contain something like `plotxy ticks ...` and we want the new value of the tick counter used, not the old one. Most models don't

refer to the tick counter in their plotting commands, but nonetheless, for consistency and to avoid mistakes we suggest always putting `tick` before the plotting commands.

Some models will require some additional changes:

1. If your model already has a global "ticks" or "clock" or "time" variable, get rid of it. Use the `tick` command and `ticks` reporter instead. (If your model uses fractional increments of time, use `tick-advance` instead of `tick`.) If you had a monitor for that variable, you can get rid of it; there's now a tick counter in the toolbar.
2. `clear-all` resets the tick counter to zero. If you don't use `clear-all` in your setup procedure, then you may need to add `reset-ticks` to reset the counter to zero.
3. If you used `no-display` and `display` to prevent view updates from happening in the middle of go, you can get rid of them.
4. If your model needs to update the view without advancing the tick counter (examples: Party, Dice Stalagmite, network models with animated layout, models with mouse interaction buttons), use the `display` command to force additional view updates so the user can see what is going on.

Speed slider

Previous versions of NetLogo had a speed slider that could be used to make models run slower, so you can see what's going on.

In NetLogo 4.0, the slider can be used to speed up models as well. It does this by updating the view less frequently. Updating the view takes time, so the fewer updates, the faster the model runs.

The default position of the slider is in the center. When you're at the center, the slider says "normal speed".

As you move the slider away from the center position, the model will gradually run faster or slower.

At very high speeds, view updates become very infrequent and may be separated by several seconds. It may feel like the model is actually running slower, since the updates are so infrequent. But watch the tick counter, or other indicators such as plots, and you'll see that yes, the model really is running faster. If the infrequent updates are disconcerting, don't push the slider so far over.

When using tick-based updates, slowing the model down does not cause additional view updates. Rather, NetLogo simply pauses after each tick.

When using continuous updates, slowing the model down means view updates become more closely spaced. If you push the speed slider more than halfway to the left, the model will be running so slowly that you can watch turtles moving one at a time! This is new in NetLogo 4.0; in previous NetLogo versions, no matter how slowly you ran a model, you would never see the agents in an `ask` moving one at a time; all the agents in an `ask` always appeared to move together.

Numbers

NetLogo no longer maintains an internal distinction between integers and floating point numbers. So for example:

Old:

```
observer> print 3
3
observer> print 3.0
3.0
observer> print 1 + 2
3
observer> print 1.5 + 1.5
3.0
observer> print 3 = 3.0
true
```

(The last line shows that although the distinction between integer 3 and floating point 3.0 was maintained, the two numbers were still considered equal.)

New:

```
observer> print 3
3
observer> print 3.0
3
observer> print 1 + 2
3
observer> print 1.5 + 1.5
3
observer> print 3 = 3.0
true
```

We expect that only rare models will be negatively impacted by this change.

A benefit of this change is that NetLogo now supports a much larger range of integers. The old range was -2,147,483,648 to 2,147,483,647 (around +/- 2 billion); the new range is +/- 9,007,199,254,740,992 (around +/- 9 quadrillion).

Agentset building

NetLogo 3.1 (and some earlier versions) included primitives called `turtles-from` and `patches-from` that were occasionally useful for building agentsets. In NetLogo 4.0, these primitives have been replaced with new primitives called turtle-set and patch-set that are much more flexible and powerful. (link-set exists as well.) See the entries for these primitives in the NetLogo Dictionary. Models that use the old `turtles-from` and `patches-from` will need to be altered by hand to use the new primitives.

RGB Colors

In NetLogo 3.1 RGB and HSB colors could be approximated as NetLogo colors using the `rgb` and `hsb` primitives. These have been renamed to approximate-rgb and approximate-hsb and now expect inputs in the range 0-255, not 0-1.

The full RGB spectrum is now available in NetLogo so it may no longer be necessary to use these primitives at all. You can set any color variable to a three-item RGB list, with values in the 0-255 range, and get that exact color. See the Color section of the Programming Guide for details.

Tie

In previous versions `__tie` was provided as an experimental feature. As of NetLogo 4.0 links have a tie-mode variable which can be set to `"none"`, `"free"`, or `"fixed"`. In 4.0 tie is now a

link-only primitive. This means that to tie turtle 1 to turtle 0 you write:

```
ask turtle 0 [ create-link-to turtle 1 [ tie ] ]
```

See the Tie section of the programming guide for details.

Changes for NetLogo 3.1

Agentsets

If your model is behaving strangely or incorrectly, it may be because since NetLogo 3.1, agentsets are now always in random order. In prior versions of NetLogo, agentsets were always in a fixed order. If your code depended on that fixed order, then it won't work anymore. How to fix your model to work with randomized agentsets depends on the details of what your code is doing. In some situations, it is helpful to use the `sort` or `sort-by` primitives to convert an agentset (random order) into a list of agents (fixed order). See "Lists of agents" in the Lists section of the Programming Guide.

Wrapping

If you are seeing pieces of turtle shapes wrapping around the view edges, it's because NetLogo 3.0 allowed you to turn off such wrapping in the view without affecting the behavior of the model. Since NetLogo 3.1, if you don't want the view to wrap you must make it so the world doesn't wrap, using the new topology feature. Making this change may require other changes to your model, though. See the Topology section of the Programming Guide for a thorough discussion of how to convert your model to take advantage of this new feature.

Random turtle coordinates

Many models made in NetLogo 3.0 or earlier use `setxy random world-width random world-height` to scatter turtles randomly, using either `random` or `random-float`. It only works if world wrapping is on.

(Why? Because when wrapping is on, you can set coordinates of turtles to numbers beyond the edge of the world and NetLogo will wrap the turtle to the other side. But in worlds that don't wrap setting the x or y coordinates of a turtle to a point outside the bounds of the world causes a runtime error. The world wrap settings were added in NetLogo 3.1. See the Topology section of the Programming Guide for more information.)

To fix your model so that it works regardless of the wrapping settings, use one of these two commands instead:

```
setxy random-xcor random-ycor
setxy random-pxcor random-pycor
```

The two commands are a bit different. The first command puts the turtle on a random point in the world. The second command puts the turtle on the center of a random patch. An even more concise way to put a turtle on the center of a random patch is:

```
move-to one-of patches
```

Applets

For now, deprecated legacy support still exists in NetLogo for running models as Java applets inside a web browser.

WARNING AND DISCLAIMER: However, applets are dead technology. We no longer recommend them as a way of delivering models to a general audience. Details here, including discussion of alternatives.

Making an applet

You can make a model into an applet by choosing Save As Applet on NetLogo's File menu. If your model has unsaved changes you will first be prompted to save it. Then you will also be prompted to save an HTML file containing the applet.

For applets to work, the HTML file, your model file (ending in `.nlogo`), and the files NetLogoLite.jar and NetLogoLite.jar.pack.gz must all be in the same folder. (You can copy NetLogoLite.jar and NetLogoLite.jar.pack.gz from the folder where you installed NetLogo.)

On some systems, you can test the applet locally on your computer before uploading it to a web server. It doesn't work on all systems, though, so if it doesn't work from your hard drive, please try uploading it to a web server.

You don't need to include everything in the html file in your page. If you want, you can just take the HTML code beginning with <applet> and ending with </applet>, and paste it into any HTML file you want. It's even OK to put multiple <applet> tags on a single page.

Additional files

Applets can read files on the web server. If your applet requires additional files, such as text files it reads, images it imports, and so on, you will also need to put those files in the same folder. These files should appear in the same location relative to the model file as they appear on your computer. Applets cannot read or write files on the user's computer, only the web server. Applets cannot browse web server or the user's computer, meaning, that `user-file` and `user-new-file` do nothing in an applet. All files required to run your model including the model file itself, NetLogoLite.jar, and NetLogoLite.jar.pack.gz must be readable by the web server user.

Extensions

Many extensions can be used in applets. Simply place the folder containing the extension jar in the same folder as the model.

Extensions that require native libraries don't work from applets. This includes the QTJ and GoGo extensions.

Using an alternate jar location

If the NetLogoLite files and your model are in different directories, you must modify the archive= and value= lines in the HTML code to point to their actual locations. (For example, if you have multiple applets in different directories on the same web server, you may want to put a single copy of the NetLogoLite files in one central place and change the archive= lines

of all the HTML files to point to that one central copy. This will save disk space for you and download time for your users.)

Java requirements

Getting the right version

Current versions of NetLogo require that your web browser support Java 6 or higher. Here's how to get the right Java:

- If you're on Windows, you need to download the Java browser plugin from http://www.java.com/en/download/windows_manual.jsp.
- If you're on Mac OS X, you need Mac OS X 10.4 or higher. (NetLogo 4.0 was the last version to support Mac OS X 10.2 and 10.3.)
- If you're on Linux or another Unix, you will need version 5 (or higher) of the Java Runtime Environment. It is available for download at http://www.java.com/. Check your browser's home page for information about installing the Java plugin.

If you think you have the right browser and plugin, but it still doesn't work, check your browser's preferences to make sure that Java is enabled.

The following web site may be helpful for figuring out what Java you have and getting the right version running: http://www.javatester.org/.

Increasing the available memory

Some NetLogo applets may require more memory than the browser normally makes available. This may happen if you have large numbers of agents. On Windows, you can increase the available memory ("heap") space in the Java Control Panel's applet runtime settings.

Mac OS X 10.4 users, note that Mac OS X 10.4 initially had a rather low memory limit for Java applets, namely 64 megabytes. Eventually a Java update from Apple raised it to 96 megabytes. You can get the update through Software Update.

If your browser is using the browser plug-in that comes with the Oracle JDK or JRE then instructions for starting the **Java Plug-In Control Panel** are available here. In the *Advanced* tab of the Control Panel add the following to the *Java Runtime Parameters* field: "-Xmx1024m".

Features not supported in applets

- Only some extensions work. (Extension authors should document whether their extensions work in applets.)
- Extensions that require native libraries won't work.
- The 3D view is not supported.
- The bytecode generator is not used in applets (which means that some models run somewhat slower as applets).
- Web servers that return custom error messages may cause Java exceptions. See the FAQ for the workaround.

Note to extension authors

You may wish to document whether your extension works in applets.

One reason an extension might fail to work is that the NetLogo build process uses [ProGuard](#) aggressively to reduce the size of NetLogoLite.jar. This can break some extensions, but often the problem is trivially fixable by adding an entry to the `project/proguard/lite.txt` file in the NetLogo build. If you discover that something your extension does works if you substitute NetLogo.jar, but doesn't work with NetLogoLite.jar, the NetLogo development team may be able to help you resolve the issue, if you ask about it on [netlogo-devel](#).

Shapes Editor Guide

The Turtle and Link Shape Editors allows you to create and save turtle and link designs.

NetLogo uses fully scalable and rotatable vector shapes, which means you can create designs by combining basic geometric elements, which can appear on-screen in any size or orientation.

Getting started

To begin making shapes, choose **Turtle Shapes Editor** or **Link Shapes Editor** in the Tools menu. A new window will open listing all the shapes currently in the model, beginning with *default*, the default shape. The Shapes Editor allows you to edit shapes, create new shapes, and borrow from another model. You can also import turtle shapes from a library of pre-existing shapes.

Importing shapes

Every new model in NetLogo starts off containing a small core set of frequently used shapes. Many more turtle shapes are available by using the **Import from library...** button. This brings up a dialog where you can select one or more shapes and bring them into your model. Select the shapes, then press the **Import** button.

Similarly, you can use the **Import from model...** button to borrow shapes from another model.

Default shapes

Here are the turtle shapes that are included by default in every new NetLogo model:
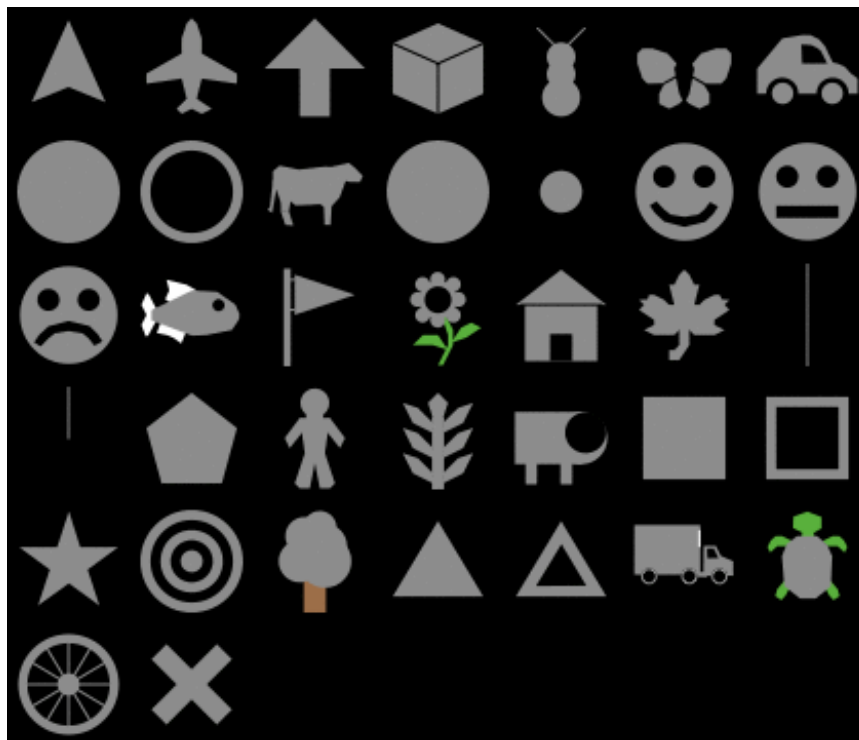
First row: default, airplane, arrow, box, bug, butterfly, car
Second row: circle, circle 2, cow, cylinder, dot, face happy, face neutral
Third row: face sad, fish, flag, flower, house, leaf, line
Fourth row: line half, pentagon, person, plant, sheep, square, square 2

Fifth row: star, target, tree, triangle, triangle 2, truck, turtle
Sixth row: wheel, x

Shapes library

And here are the shapes in the shapes library (including all of the default shapes, too):

By default there is only one Link shape in a model, that is "default". This shape is simply a single straight line with a simple arrowhead (if the link happens to be directed).
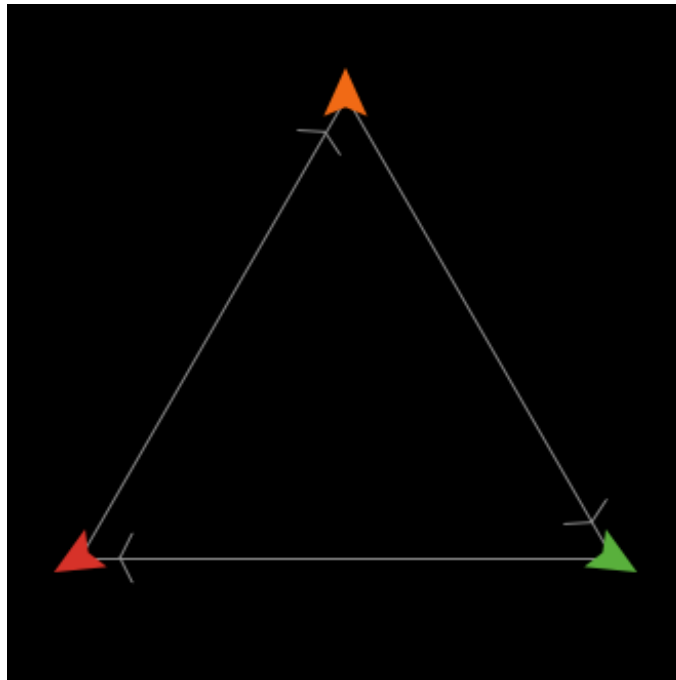
Creating and editing turtle shapes

Pressing the **New** button will make a new shape. Or, you may select an existing shape and press **Edit**.

Tools

In the upper left corner of the editing window is a group of drawing tools. The arrow is the selection tool, which selects an already drawn element.

To draw a new element, use one of the other seven tools:

- The **line** tool draws line segments.
- The **circle**, **square**, and **polygon** tools come in two versions, solid and outline.

When using the polygon tool, click the mouse to add a new segment to the polygon. When you're done adding segments, double click.

After you draw a new element, it is selected, so you can move, delete, or reshape it if you want:

- To move it, drag it with the mouse
- To delete it, press the Delete button.
- To reshape it, drag the small "handles" that appear on the element only when it is selected.
- To change its color, click on the new color.

Previews

As you draw your shape, you will also see it in five smaller sizes in the five preview areas found near the bottom of the editing window. The previews show your shape as it might appear in your model, including how it looks as it rotates. The number below each preview is the size of the preview in pixels. When you edit the view, patch size is also measured in pixels. So for example, the preview with "20" below it shows you how your shape would look on a turtle (of size 1) on patches of size 20 pixels.

The rotatable feature can be turned off if you want a shape that always faces the same way, regardless of the turtle's heading.

Overlapping shapes

New elements go on top of previous elements. You can change the layering order by selecting an element and then using the **Bring to front** and **Send to back** buttons.

Undo

At any point you can use the **Undo** button to undo the edit you just performed.

Colors

Elements whose color matches the *Color that changes* (selected from a drop-down menu -- the default is gray) will change color according to the value of each turtle's *color* variable in your model. Elements of other colors don't change. For example, you could create cars that always have yellow headlights and black wheels, but different body colors.

Other buttons

The "Rotate Left" and "Rotate Right" buttons rotate elements by 90 degrees. The "Flip Horizontal" and "Flip Vertical" buttons reflect elements across the axes.

These four buttons will rotate or flip the entire shape, unless an element is selected, in which case only that element is affected.

These buttons are especially handy in conjunction with the "Duplicate" button if you want to make shapes that are symmetrical. For example, if you were making a butterfly, you could draw the butterfly's left wing with the polygon tool, then duplicate the wing with the "Duplicate" button, then turn the copy into a right wing with the "Flip Horizontal" button.

Shape design

It's tempting to draw complicated, interesting shapes, but remember that in most models, the patch size is so small that you won't be able to see very much detail. Simple, bold, iconic shapes are usually best.

Keeping a shape

When the shape is done, give it a name and press the **Done** button at the bottom of the editing window. The shape and its name will now be included in the list of shapes along with the "default" shape.

Creating and editing link shapes

Managing link shapes is very similar to managing turtle shapes. So, you can create a new shape by pressing the **New** button or you can edit existing shapes. When you are done editing a shape press **Done** if you want to keep it.

Changing link shape properties

There are several different properties for each link shape that you are allowed to change:

- Name - link shapes can have the same name as turtle shapes but must be unique among link shapes.
- Direction Indicator - the direction indicator (the little arrow on directed links) is just like the turtle vector shapes, you can edit it using the same editor by pressing the **Edit** button.
- Curviness - this is the amount of bend in a link expressed in patches (this is particularly useful if you have directed links going in both directions so you can discern both links)
- Number of lines: You can have 1, 2, or 3 lines in each link shape, you control this by selecting line patterns in the "left line", "middle line", and "right line" selection boxes.
- Dash pattern of lines: There are several dashed line patterns available in the selection boxes so not all lines need be solid.

Here are some link shapes with various properties:
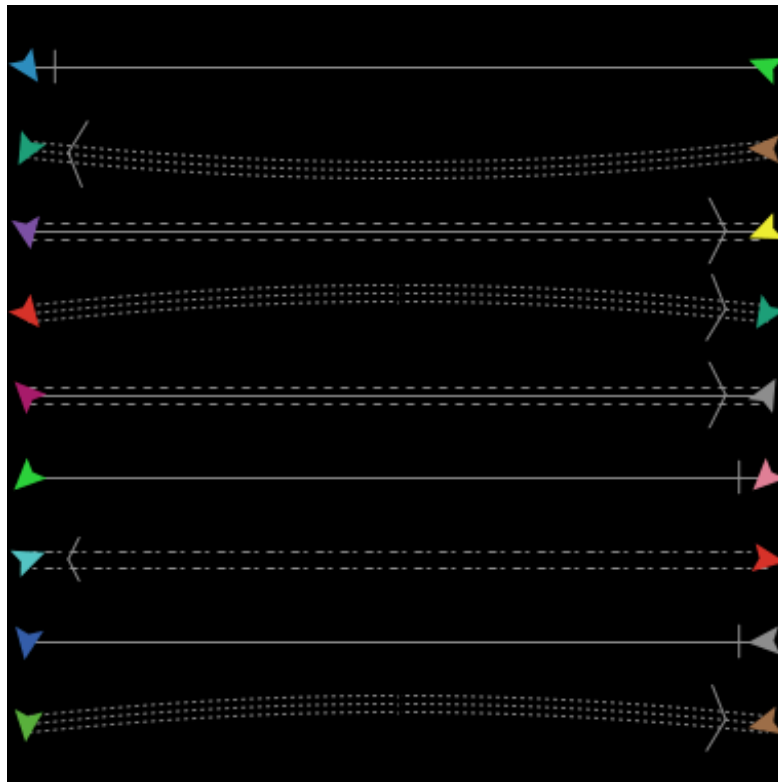
Using shapes in a model

In the model's code or in the command center, you can use any of the shapes that are in the model (though only turtles can have turtle shapes and only links can have link shapes). For example, suppose you want to create 50 turtles with the shape "rabbit". Provided there is some turtle shape called *rabbit* in this model, give this command to the observer in the command center:

```
observer> crt 50
```

And then give these commands to the turtles to spread them out, then change their shape:

```
turtles> fd random 15
turtles> set shape "rabbit"
```

Voila! Rabbits! Note the use of double quotes around the shape name. Shape names are strings.

Similarly, you can set the shape variable of links. Assuming there is a link shape called "road" in this model:

```
observer> crt 5 [ create-links-with other turtles ]
turtles> fd 5
links> set shape "road"
```

The `set-default-shape` command is also useful for assigning shapes to turtles and links.

BehaviorSpace Guide

This guide has three parts:

- **What is BehaviorSpace?**: A general description of the tool, including the ideas and principles behind it.
- **How It Works**: Walks you through how to use the tool and highlights its most commonly used features.
- **Advanced Usage**: How to use BehaviorSpace from the command line, or from your own Java code.

What is BehaviorSpace?

BehaviorSpace is a software tool integrated with NetLogo that allows you to perform experiments with models.

BehaviorSpace runs a model many times, systematically varying the model's settings and recording the results of each model run. This process is sometimes called "parameter sweeping". It lets you explore the model's "space" of possible behaviors and determine which combinations of settings cause the behaviors of interest.

If your computer has multiple processor cores, then by default, model runs will happen in parallel, one per core.

Why BehaviorSpace?

The need for this type of experiment is revealed by the following observations. Models often have many settings, each of which can take a range of values. Together they form what in mathematics is called a parameter space for the model, whose dimensions are the number of settings, and in which every point is a particular combination of values. Running a model with different settings (and sometimes even the same ones) can lead to drastically different behavior in the system being modeled. So, how are you to know which particular configuration of values, or types of configurations, will yield the kind of behavior you are interested in? This amounts to the question of where in its huge, multi-dimension parameter space does your model perform best?

For example, suppose you want speedy synchronization from the agents in the Fireflies model. The model has four sliders -- number, cycle-length, flash-length and number-flashes -- that have approximately 2000, 100, 10 and 3 possible values, respectively. That means there are 2000 * 100 * 10 * 3 = 600,000 possible combinations of slider values! Trying combinations one at a time is hardly an efficient way to learn which one will evoke the speediest synchronization.

BehaviorSpace offers you a much better way to solve this problem. If you specify a subset of values from the ranges of each slider, it will run the model with each possible combination of those values and, during each model run, record the results. In doing so, it samples the model's parameter space -- not exhaustively, but enough so that you will be able to see relationships form between different sliders and the behavior of the system. After all the runs are over, a dataset is generated which you can open in a different tool, such as a spreadsheet, database, or scientific visualization application, and explore.

By enabling you to explore the entire "space" of behaviors a model can exhibit, BehaviorSpace can be a powerful assistant to the modeler.

How It Works

To begin using BehaviorSpace, open your model, then choose the BehaviorSpace item on NetLogo's Tools menu.

Managing experiment setups

The dialog that opens lets you create, edit, duplicate, delete, and run experiment setups. Experiments are listed by name and how by model runs the experiment will consist of.

Experiment setups are considered part of a NetLogo model and are saved as part of the model.

To create a new experiment setup, press the "New" button.

Creating an experiment setup

In the new dialog that appears, you can specify the following information. Note that you don't always need to specify everything; some parts can be left blank, or left with their default values, depending on your needs.

Experiment name: If you have multiple experiments, giving them different names will help you keep them straight.

Vary variables as follows: This is where you specify which settings you want varied, and what values you want them to take. Variables can include sliders, switches, choosers, and any global variables in your model.

Variables can also include `max-pxcor`, `min-pxcor`, `max-pycor` and `min-pycor`, `world-width`, `world-height` and `random-seed`. These are not, strictly speaking, variables, but BehaviorSpace lets you vary them as if they were. Varying the world dimensions lets you explore the effect of world size upon your model. Since setting `world-width` and `world-height` does not necessarily define the bounds of the world how they are varied depends on the location of the origin. If the origin is centered, BehaviorSpace will keep it centered so the values `world-width` or `world-height` must be odd. If one of the bounds is at zero that bound will be kept at zero and the other bound will move, for example if you start with a world with `min-pxcor = 0 max-pxcor = 10` and you vary `world-width` like this:

```
["world-width" [11 1 14]]
```

`min-pxcor` will stay at zero and `max-pxcor` will set to 11, 12, and 13 for each of the runs. If neither of these conditions are true, the origin is not centered, nor at the edge of the world you cannot vary `world-height` or `world-width` directly but you should vary `max-pxcor`, `max-pycor`, `min-pxcor` and `min-pycor` instead.

Varying `random-seed` lets you repeat runs by using a known seed for the NetLogo random number generator. Note that you're also free to use the `random-seed` command in your experiment's setup commands. For more information on random seeds, see the Random Numbers section of the Programmer's Guide.

You may specify values either by listing the values you want used, or by specifying that you want to try every value within a given range. For example, to give a slider named `number` every value from 100 to 1000 in increments of 50, you would enter:

```
["number" [100 50 1000]]
```

Or, to give it only the values of 100, 200, 400, and 800, you would enter:

```
["number" 100 200 400 800]
```

Be careful with the brackets here. Note that there are fewer square brackets in the second example. Including or not including this extra set of brackets is how you tell BehaviorSpace whether you are listing individual values, or specifying a range.

Also note that the double quotes around the variable names are required.

You can vary as many settings as you want, including just one, or none at all. Any settings that you do not vary will retain their current values. Not varying any settings is useful if you just want to do many runs with the current settings.

What order you list the variables in determines what order the runs will be done in. All values for a later variable will be tried before moving to the next value for an earlier variable. So for example if you vary both x and y from 1 to 3, and x is listed first, then the order of model runs will be: x=1 y=1, x=1 y=2, x=1 y=3, x=2 y=1, and so on.

Repetitions: Sometimes the behavior of a model can vary a lot from run to run even if the settings don't change, if the model uses random numbers. If you want to run the model more than once at each combination of settings, enter a higher number.

Measure runs using these reporters: This is where you specify what data you want to collect from each run. For example, if you wanted to record how the population of turtles rose and fell during each run, you would enter:

```
count turtles
```

You can enter one reporter, or several, or none at all. If you enter several, each reporter must be on a line by itself, for example:

```
count frogs
count mice
count birds
```

If you don't enter any reporters, the runs will still take place. This is useful if you want to record the results yourself your own way, such as with the <u>export-world</u> command.

Measure runs at every step: Normally NetLogo will measure model runs at every step, using the reporters you entered in the previous box. If you're doing very long model runs, you might not want all that data. Uncheck this box if you only want to measure each run after it ends.

Setup commands: These commands will be used to begin each model run. Typically, you will enter the name of a procedure that sets up the model, typically `setup`. But it is also possible to include other commands as well.

Go commands: These commands will be run over and over again to advance to the model to the next "step". Typically, this will be the name of a procedure, such as `go`, but you may include any commands you like.

Stop condition: This lets you do model runs of varying length, ending each run when a certain condition becomes true. For example, suppose you wanted each run to last until there were no more turtles. Then you would enter:

```
not any? turtles
```

If you want the length of runs to all be of a fixed length, just leave this blank.

The run may also stop because the go commands use the `stop` command, in the same way that `stop` can be used to stop a forever button. The `stop` command may be used directly in the go commands, or in a procedure called directly by the go commands. (The intent is that the same `go` procedure should work both in a button and in a BehaviorSpace experiment.) Note that the step in which `stop` is used is considered to have been aborted, so no results will be recorded for that step. Therefore, the stopping test should be at the beginning of the go commands or procedure, not at the end.

Final commands: These are any extra commands that you want run once, when the run ends. Usually this is left blank, but you might use it to call the `export-world` command or record the results of the run in some other way.

Time limit: This lets you set a fixed maximum length for each run. If you don't want to set any maximum, but want the length of the runs to be controlled by the stop condition instead, enter 0.

Special primitives for BehaviorSpace experiments

Currently there are only two, `behaviorspace-run-number` and `behaviorspace-experiment-name`. The run number reported by the former primitive matches the run number used in the results files generated by BehaviorSpace. The experiment name reported by the latter matches the name with which the experiment was set up.

Running an experiment

When you're done setting up your experiment, press the "OK" button, followed by the "Run" button. A dialog titled "Run options" will appear.

Run options: formats

The run options dialog lets you select the formats you would like the data from your experiment saved in. Data is collected for each run or step, according to the setting of **Measure runs at every step** option. In either case, the initial state of the system is recorded, after the setup commands run but before the go commands run for the first time.

Table format lists each interval in a row, with each metric in a separate column. Table data is written to the output file as each run completes. Table format is suitable for automated processing of the data, such as importing into a database or a statistics package.

Spreadsheet format calculates the min, mean, max, and final values for each metric, and then lists each interval in a row, with each metric in a separate column. Spreadsheet data is more

human-readable than Table data, especially if imported into a spreadsheet application.

(Note however that spreadsheet data is not written to the results file until the experiment finishes. Since spreadsheet data is stored in memory until the experiment is done, very large experiments could run out of memory. So you should disable spreadsheet output unless you really want it. If you do want spreadsheet output, note that if anything interrupts the experiment, such as a runtime error, running out of memory, or a crash or power outage, no spreadsheet results will be written. For long experiments, you may want to also enable table format as a precaution so that if something happens and you get no spreadsheet output you'll at least get partial table output.)

After selecting your output formats, BehaviorSpace will prompt you for the name of a file to save the results to. The default name ends in ".csv". You can change it to any name you want, but don't leave off the ".csv" part; that indicates the file is a Comma Separated Values (CSV) file. This is a plain-text data format that is readable by any text editor as well as by most popular spreadsheet and database programs.

Run options: parallel runs

The run options dialog also lets you select whether you want multiple model runs to happen in parallel, and if so, how many are allowed to be simultaneously active. This number will default to the number of processor cores in your computer.

There are a few cautions associated with parallel runs.

First, if multiple runs are active, only one of them will be in the "foreground" and cause the view and plots to update. The other runs will happen invisibly in the background.

Second, invisible background runs can't use primitives that only work in the GUI. For example, a background run can't make a movie.

Third, since parallel runs progress independently of each other, table format output may contain interleaved, out-of-order results. When you analyze your table data, you may wish to sort it by run number first. (Spreadsheet format output is not affected by this issue, since it is not written until the experiment completes or is aborted.)

Fourth, using all available processor cores may make your computer slow to use for other tasks while the experiment is running.

Fifth, doing runs in parallel will multiply the experiment's memory requirements accordingly. You may need to increase NetLogo's memory ceiling (see [this FAQ entry](#)).

Observing runs

After you complete the run options dialog, another dialog will appear, titled "Running Experiment". In this dialog, you'll see a progress report of how many runs have been completed so far and how much time has passed. If you entered any reporters for measuring the runs, and if you left the "Measure runs at every step" box checked, then you'll see a plot of how they vary over the course of each run.

You can also watch the runs in the main NetLogo window. (If the "Running Experiment" dialog is in the way, just move it to a different place on the screen.) The view and plots will update as the model runs. If you don't need to see them update, then use the checkboxes in the "Running Experiment" dialog to turn the updating off. This will make the experiment go faster.

If you want to stop your experiment before it's finished, press the "Abort" button. Any results

generated so far will still be saved.

When all the runs have finished, the experiment is complete.

Advanced usage

Running from the command line

It is possible to run BehaviorSpace experiments "headless", that is, from the command line, without any graphical user interface (GUI). This is useful for automating runs on a single machine or a cluster of machines.

No Java programming is required. Experiment setups can be created in the GUI and then run later from the command line, or, if you prefer, you can create or edit experiment setups directly using XML.

How to use it

Run Java with the org.nlogo.headless.Main class. The Main.main() method supports these arguments:

- `--model <path>`: pathname of model to open (required)
- `--setup-file <path>`: read experiment setups from this file instead of the model file
- `--experiment <name>`: name of experiment to run
- `--table <path>`: pathname to send table output to (or – for standard output)
- `--spreadsheet <path>`: pathname to send table output to (or – for standard output)
- `--threads <number>`: use this many threads to do model runs in parallel, or 1 to disable parallel runs. defaults to one thread per processor.
- `--min-pxcor <number>`: override world size setting in model file
- `--max-pxcor <number>`: override world size setting in model file
- `--min-pycor <number>`: override world size setting in model file
- `--max-pycor <number>`: override world size setting in model file

`--model` is required. If you don't specify `--experiment`, you must specify `--setup-file`. By default no results are generated, so you'll usually want to specify either `--table` or `--spreadsheet`, or both. If you specify any of the world dimensions, you must specify all four.

Examples

It is easiest if you create your experiment setup ahead of time in the GUI, so it is saved as part of the model. To run an experiment setup saved in a model, here is an example command line:

```
java -Xmx1024m -Dfile.encoding=UTF-8 -cp NetLogo.jar \
  org.nlogo.headless.Main \
  --model Fire.nlogo \
  --experiment experiment1 \
  --table -
```

(For this to work, `NetLogo.jar` must be present along with the `lib` subdirectory containing necessary libraries. Both `NetLogo.jar` and `lib` are included with NetLogo.)

After the named experiment has run, the results are sent to standard output in table format,

as CSV. ("-" is how you specify standard output instead of output to a file.)

When running the headless.Main class as an application, it forces the system property `java.awt.headless` to be true. This tells Java to run in headless mode, allowing NetLogo to run on machines when a graphical display is not available.

Note the use of `-Xmx` to specify a maximum heap size of one gigabyte. If you don't specify a maximum heap size, you will get your VM's default size, which may be unusably small. (One gigabyte is an arbitrary size which should be more than large enough for most models; you can specify a different limit if you want.)

Note the use of `-Dfile.encoding=UTF-8`. This forces all file I/O to use UTF-8 encoding. Doing so ensures that NetLogo can load all models consistently, and that file-* primitives work consistently on all platforms, including models containing Unicode characters.

The required `--model` argument is used to specify the model file you want to open.

The `--experiment` argument is used to specify the name of the experiment you want to run. (At the time you create an experiment setup in the GUI, you assign it a name.)

Here's another example that shows some additional, optional arguments:

```
java -Xmx1024m -Dfile.encoding=UTF-8 -cp NetLogo.jar \
  org.nlogo.headless.Main \
  --model Fire.nlogo \
  --experiment experiment2 \
  --max-pxcor 100 \
  --min-pxcor -100 \
  --max-pycor 100 \
  --min-pycor -100
```

Note the use of the optional `--max-pxcor`, `--max-pycor`, etc. arguments to specify a different world size than that saved in the model. (It's also possible for the experiment setup to specify values for the world dimensions; if they are specified by the experiment setup, then there is no need to specify them on the command line.)

Since neither --table nor --spreadsheet is specified, no results will be generated. This is useful if the experiment setup generates all the output you need by some other means, such as exporting world files or writing to a text file.

Yet another example:

```
java -Xmx1024m -Dfile.encoding=UTF-8 -cp NetLogo.jar \
  org.nlogo.headless.Main \
  --model Fire.nlogo \
  --experiment experiment2 \
  --table table-output.csv \
  --spreadsheet spreadsheet-output.csv
```

The optional `--table <filename>` argument specifies that output should be generated in a table format and written to the given file as CSV data. If `-` is specified as the filename, than the output is sent to the standard system output stream. Table data is written as it is generated, with each complete run.

The optional `--spreadsheet <filename>` argument specified that spreadsheet output should be generated and written to the given file as CSV data. If `-` is specified as the filename, than the output is sent to the standard system output stream. Spreadsheet data is not written out until all runs in the experiment are finished.

Note that it is legal to specify both `--table` and `--spreadsheet`, and if you do, both kinds of output file will be generated.

Here is one final example that shows how to run an experiment setup which is stored in a separate XML file, instead of in the model file:

```
java -Xmx1024m -Dfile.encoding=UTF-8 -cp NetLogo.jar \
  org.nlogo.headless.Main \
  --model Fire.nlogo \
  --setup-file fire-setups.xml \
  --experiment experiment3
```

If the XML file contains more than one experiment setup, it is necessary to use the `--experiment` argument to specify the name of the setup to use.

In order to run any of these experiments in 3D add `-Dorg.nlogo.is3d=true` to any of these startup commands, for example:

```
java -Dorg.nlogo.is3d=true -Xmx1024m -Dfile.encoding=UTF-8 -cp NetLogo.jar \
  org.nlogo.headless.Main \
  --model Fire3D.nlogo \
  --experiment experiment1 \
  --table -
```

Note that you should supply a 3D model and there are also 3D arguments `--max-pzcor <number>` and `--min-pzcor <number>`.

The next section has information on how to create standalone experiment setup files using XML.

Setting up experiments in XML

We don't yet have detailed documentation on authoring experiment setups in XML, but if you already have some familiarity with XML, then the following pointers may be enough to get you started.

The structure of BehaviorSpace experiment setups in XML is determined by a Document Type Definition (DTD) file. The DTD is stored in NetLogo.jar, as `system/behaviorspace.dtd`. (JAR files are also zip files, so you can extract the DTD from the JAR using Java's "jar" utility or with any program that understands zip format.)

The easiest way to learn what setups look like in XML, though, is to author a few of them in BehaviorSpace's GUI, save the model, and then examine the resulting .nlogo file in a text editor. The experiment setups are stored towards the end of the .nlogo file, in a section that begins and ends with a `experiments` tag. Example:

```
<experiments>
  <experiment name="experiment" repetitions="10" runMetricsEveryStep="true">
    <setup>setup</setup>
    <go>go</go>
    <exitCondition>not any? fires</exitCondition>
    <metric>burned-trees</metric>
    <enumeratedValueSet variable="density">
      <value value="40"/>
      <value value="0.1"/>
      <value value="70"/>
    </enumeratedValueSet>
  </experiment>
</experiments>
```

In this example, only one experiment setup is given, but you can put as many as you want between the beginning and ending `experiments` tags.

Between looking at the DTD, and looking at examples you create in the GUI, it will hopefully be apparent how to use the tags to specify different kind of experiments. The DTD specifies which tags are required and which are optional, which may be repeated and which may not, and so forth.

When XML for experiment setups is included in a model file, it does not begin with any XML headers, because not the whole file is XML, only part of it. If you keep experiment setups in their own file, separate from the model file, then the extension on the file should be .xml not .nlogo, and you'll need to begin the file with proper XML headers, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE experiments SYSTEM "behaviorspace.dtd">
```

The second line must be included exactly as shown. In the first line, you may specify a different encoding than `UTF-8`, such as `UTF-8`.

Controlling API

If BehaviorSpace is not sufficient for your needs, a possible alternative is to use our Controlling API, which lets you write Java code that controls NetLogo. The API lets you run BehaviorSpace experiments from Java code, or, you can write custom code that controls NetLogo more directly to do BehaviorSpace-like things. See the Controlling section of the User Manual for further details on both possibilities.

System Dynamics Guide

This guide has three parts:

- **What is the System Dynamics Modeler?**: A general description of the tool, including the ideas and principles behind it.
- **How It Works**: Describes the interface and how you use it.
- **Tutorial: Wolf-Sheep Predation (aggregate)**: Walks you through creating a model with the System Dynamics Modeler.

What is the NetLogo System Dynamics Modeler?

System Dynamics is a type of modeling where you try to understand how things relate to one another. It is a little different from the agent-based approach we normally use in NetLogo models.

With the agent-based approach we usually use in NetLogo, you program the behavior of individual agents and watch what emerges from their interaction. In a model of Wolf-Sheep Predation, for example, you provide rules for how wolves, sheep and grass interact with each other. When you run the simulation, you watch the emergent aggregate-level behavior: for example, how the populations of wolves and sheep change over time.

With the System Dynamics Modeler, you don't program the behavior of individual agents. Instead, you program how populations of agents behave as a whole. For example, using System Dynamics to model Wolf-Sheep Predation, you specify how the total number of sheep would change as the total number of wolves goes up or down, and vice versa. You then run the simulation to see how both populations change over time.

The System Dynamics Modeler allows you to draw a diagram that defines these populations, or "stocks", and how they affect each other. The Modeler reads your diagram and generates the appropriate NetLogo code -- global variables, procedures and reporters -- to run your System Dynamics model inside of NetLogo.

Basic Concepts

A System Dynamics diagram is made of four kinds of elements: Stocks, Variables, Flows and Links.

A **Stock** is a collection of stuff, an aggregate. For example, a Stock can represent a population of sheep, the water in a lake, or the number of widgets in a factory.

A **Flow** brings things into, or out of a Stock. Flows look like pipes with a faucet because the faucet controls how much stuff passes through the pipe.

A **Variable** is a value used in the diagram. It can be an equation that depends on other Variables, or it can be a constant.

A **Link** makes a value from one part of the diagram available to another. A link transmits a number from a Variable or a Stock into a Stock or a Flow.

The System Dynamics Modeler figures out how the value of your Stocks change over time by estimating them over and over. The estimation isn't always perfect, but you can affect its accuracy by changing the value of **dt**. As dt decreases, you estimate the model more frequently, so it gets more accurate. However, decreasing dt also makes the model slower.

Sample Models

There are four basic models in the Sample Models section of the NetLogo Models Library that

demonstrate the use of the System Dynamics Modeler. All four basic models explore population growth (and, in models with predation, population decline).

Exponential Growth and **Logistic Growth** are simple examples of growth in one stock.

Wolf Sheep Predation (System Dynamics) is an example of a system with multiple stocks influencing one another. It models a predator-prey ecosystem using the System Dynamics Modeler.

Wolf Sheep Predation (Docked Hybrid) is an example of a model that runs both the a System Dynamics model and an agent-based model side-by-side. It runs the System Dynamics implementation of Wolf-Sheep Predation next to the agent-based Wolf Sheep Predation model from the Biology section of Sample Models.

How it Works

To open the System Dynamics Modeler, choose the System Dynamics Modeler item in the Tools menu. The System Dynamics Modeler window will appear.
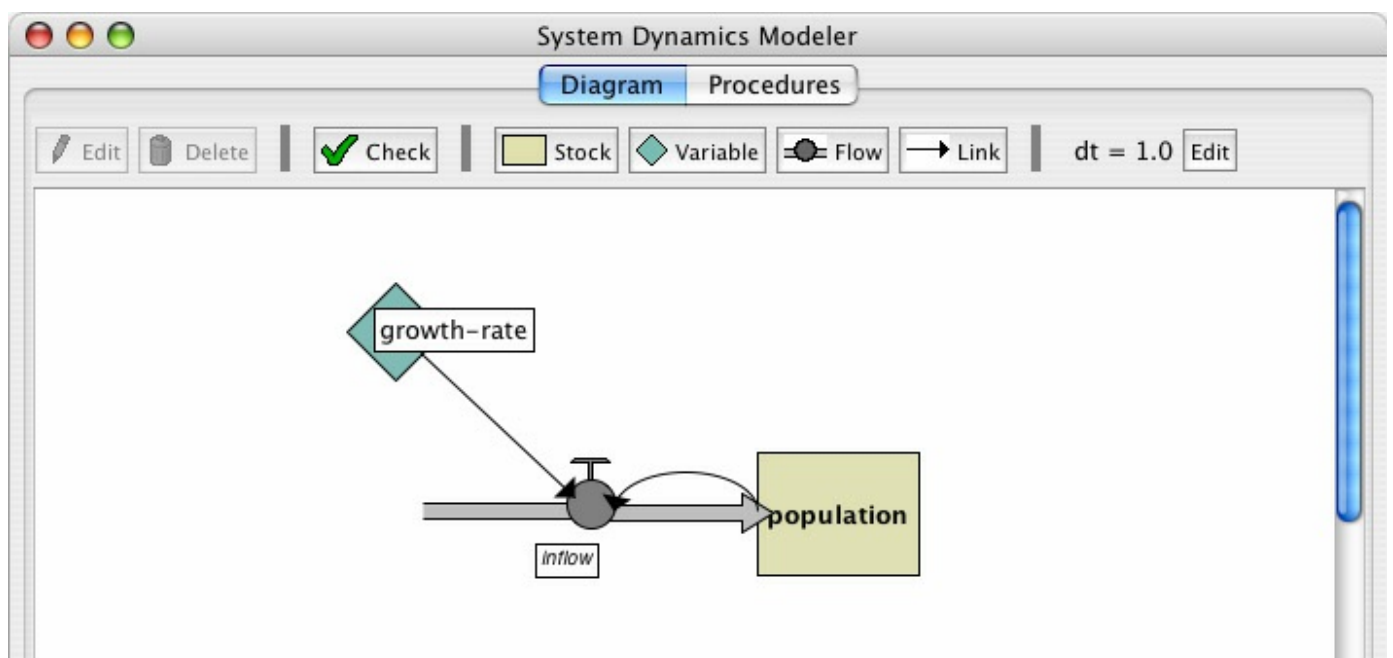
Diagram Tab

The Diagram tab is where you draw your System Dynamics diagram.

The toolbar contains buttons to edit, delete and create items in your diagram.

Creating Diagram Elements
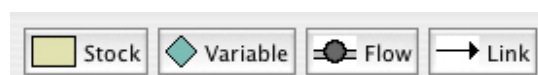
A System Dynamics diagram is made up of four kinds of components: Stocks, Variables, Flows and Links.

Stock

> To create a Stock, press the Stock button in the toolbar and click in the diagram area below. A new Stock appears. Each Stock requires a unique name, which becomes a global variable. Stocks also require an **Initial value**. It can be a number, a variable, a complex NetLogo

expression, or a call to a NetLogo reporter.

Variable

> To create a Variable, press the Variable button and click on the diagram. Each Variable in the System Dynamics Model requires a unique name, which becomes the name of a procedure, or a global variable. Variables also require an **Expression**. This expression can be a number, a variable, a complex NetLogo expression, or a call to a NetLogo reporter.

Flow

> To create a Flow, press the Flow button. Click and hold where you want the Flow to begin -- either on a Stock or in an empty area -- and drag the mouse to where you want the Flow to end -- on a Stock or in an empty area. Each Flow requires a unique name, which becomes a NetLogo reporter. Flows require an **Expression**, which is the rate of flow from the input to the output. This expression can be a number, a variable, a complex NetLogo expression, or a call to a NetLogo reporter. If the value is negative, the flow is in the opposite direction.

> When more than one Flow is connected to a Stock, it is important to consider how they should interact with one another. NetLogo will not enforce that the Flows out of a stock occur in any particular order. Also, NetLogo will not ensure that the sum of Flows out of a Stock are less than or equal to the value of the Stock. These behaviors can be implemented explicitly when creating the **Expression** for a Flow.

> For example, if the Flow is defined as a constant value, 10, you can ensure it never draws more than the value of the Stock by using the `min` primitive: `min (list stock 10)`. If I want Flow A to deplete a Stock before Flow B is calculated, I can link Flow A to Flow B and modify Flow B to subtract Flow A's value from the stock: `min (list (max (list 0 (stock - flow-a))) 10)`.

Link

> To create a Link, click and hold on the starting point for the link -- a Variable, Stock or Flow -- and drag the mouse to the destination Variable or Flow.

Working with Diagram Elements

When you create a Stock, Variable, or Flow, you see a red question-mark on the element. The question-mark indicates that the element doesn't have a name yet. The red color indicates that the Stock is incomplete: it's missing one or more values required to generate a System Dynamics model. When a diagram element is complete, the name turns black.

Selecting: To select a diagram element, click on it. To select multiple elements, hold the shift key. You can also select one or more elements by dragging a selection box.

Editing: To edit a diagram element, select the element and press the "Edit" button on the toolbar. Or just double-click the element. (You can edit Stocks, Flows and Variables, but you can't edit Links).

Moving: To move a diagram element, select it and drag the mouse to a new location.
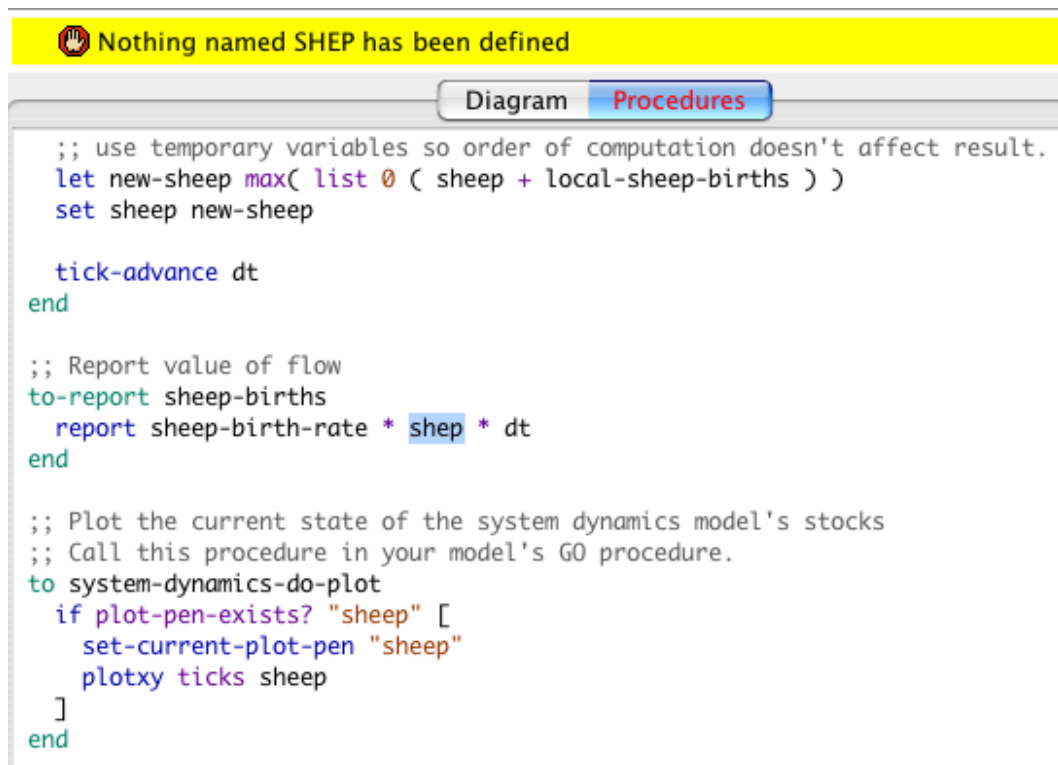
Editing dt

dt = 1.0 Edit
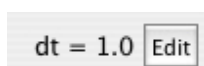
On the right side of the toolbar is the default dt, the interval used to approximate the results of your System Dynamics model. To change the value of the default dt for your aggregate model, press the Edit button next to the dt display and enter a new value.

Errors

When you click the "check" button or when you edit a stock, flow, or variable the modeler will automatically generate the NetLogo code the corresponds to your diagram and try to compile that code. If there is an error the Code tab will turn red and a message will appear, and the portion of the generated code that is causing the trouble will be highlighted.

This should give you a better idea which element in the diagram is causing the problem.
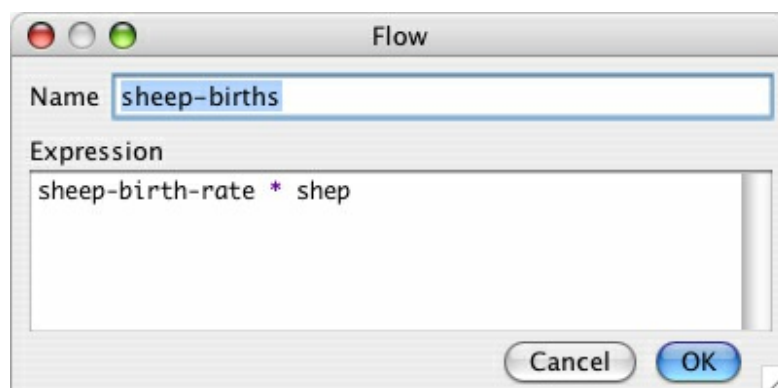
Code Tab

The System Dynamics Modeler generates NetLogo variables and procedures based on the contents of your diagram. These procedures are what make the diagram actually perform calculations. The Code tab in the System Dynamics Modeler window displays the NetLogo procedures generated from your diagram.

You can't edit the contents of the Code tab. To modify your System Dynamics mode, edit the diagram.

Let's take a closer look at how the generated code relates to the diagram.:

- Stocks correspond to a global variable that is initialized to the value or expression you provided in the **Initial value** field. Each Stock will be updated every step based on the Flows

in and out.

- Flows correspond to a procedure that contains the expression you provided in the **Expression** field.

- Variables can either be global variables or procedures. If the **Expression** you provided is a constant it will be a global variable and initialized to that value. If you used a more complicated **Expression** to define the Variable it will create a procedure like a Flow.

The variables and procedures defined in this tab are accessible in the main NetLogo window, just like the variables and procedures you define yourself in the main NetLogo Code tab. You can call the procedures from the main Code tab, from the Command Center, or from buttons in the Interface tab. You can refer to the global variables anywhere, including in the main Code tab and in monitors.

There are three important procedures to notice: `system-dynamics-setup`, `system-dynamics-go`, and `system-dynamics-do-plot`.

`system-dynamics-setup` initializes the aggregate model. It sets the value of `dt`, calls `reset-ticks`, and initializes your stocks and your converters. Converters with a constant value are initialized first, followed by the stocks with constant values. The remaining stocks are initialized in alphabetical order.

`system-dynamics-go` runs the aggregate model for `dt` time units. It computes the values of Flows and Variables and updates the value of Stocks. It also calls `tick-advance` with the value of `dt`. Converters and Flows with non-constant **Expressions** will be calculated only once when this procedure is called, however, their order of evaluation is undefined

`system-dynamics-do-plot` plots the values of Stocks in the aggregate model. To use this, first create a plot in the main NetLogo window. You then need to define a plot pen for each Stock you want to be plotted. This procedure will use the current plot, which you can change using the `set-current-plot` command.

The System Dynamics Modeler and NetLogo

The diagram you create with the System Dynamics Modeler, and the procedures generated from your diagram, are part of your NetLogo model. When you a save the NetLogo model, your diagram is saved with it, in the same file.

Tutorial: Wolf-Sheep Predation

Let's create a model of Wolf-Sheep Predation with the System Dynamics Modeler.
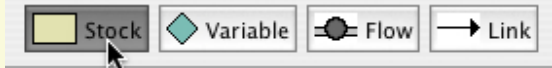
Step 1: Sheep Reproduction

- Open a new model in NetLogo.
- Launch the System Dynamics Modeler in the Tools menu.
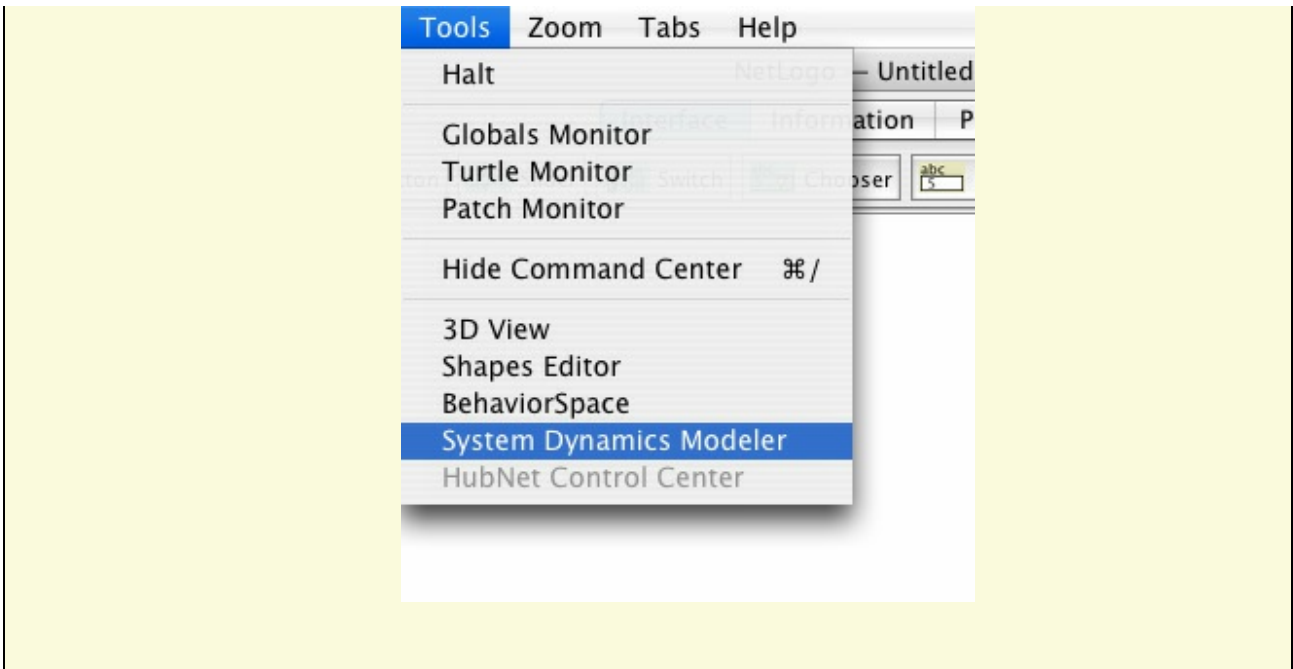
Our model will have a population of wolves and a population of sheep. Let's start with the sheep. First, create a Stock that holds a population of Sheep.

- Press the Stock button in the toolbar.

- Click in the diagram area.

You see a Stock with a red question-mark in the middle.
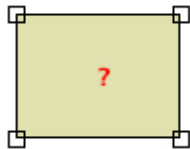
- Double-click the Stock to edit.
- Name the stock `sheep`
- Set the initial value to `100`.
- Deselect the Allow Negative Values checkbox. It doesn't make sense to have negative sheep!
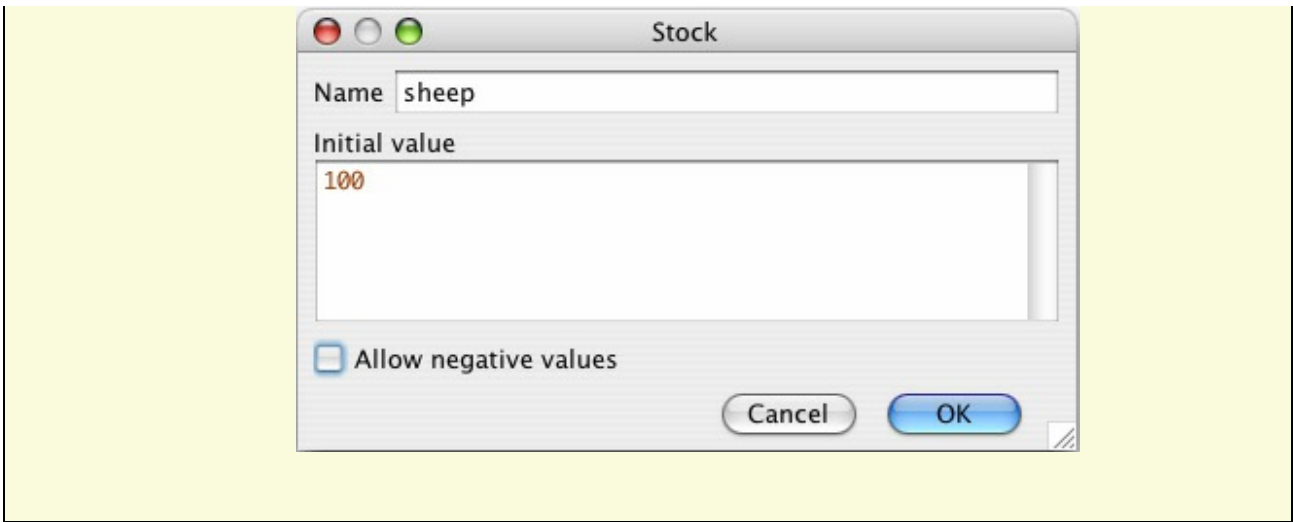
Our sheep population can increase if new sheep are born. To add this to our diagram, we create a Flow into the stock of sheep.

- Click on the Flow button in the toolbar and press the mouse button in an empty area to the left of the sheep Stock. Drag the Flow to the right until it connects to the sheep Stock and let go.
- Edit the Flow and name it `sheep-births`.
- For now, enter a constant, such as `1`, into the Expression field.

The number of sheep born during a period of time depends on the number of sheep that are alive: more sheep means more reproduction.

- Draw a Link from the `sheep` Stock to the `sheep-births` Flow.

The rate of sheep births also depends on some constant factors that are beyond the scope of this model: the rate of reproduction, etc.

- Create a Variable and name it `sheep-birth-rate`. Set its value to `0.04`
- Draw a Link from the `sheep-birth-rate` Variable to the `sheep-births`.

Your diagram should look something like this:
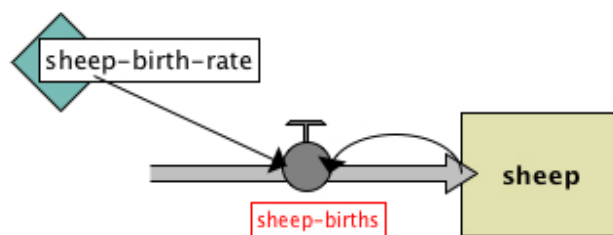
The `sheep-births` Flow has a red label because we haven't given it an expression. Red indicates that there's something missing from that part of the diagram.

The amount of sheep flowing into our stock will depend positively with the number of sheep and the sheep birth rate.

> - Edit the `sheep-births` Flow and set the expression to `sheep-birth-rate * sheep`.

We now have a complete diagram. To see the NetLogo code generated by our diagram, you can click on the Code tab of the System Dynamics Modeler window. It looks like this:

```
                                          Diagram   Procedures

;; System dynamics model globals
globals [
  ;; constants
  sheep-birth-rate
  ;; stock values
  sheep
  ;; size of each step, see SYSTEM-DYNAMICS-GO
  dt
]

;; Initializes the system dynamics model.
;; Call this in your model's SETUP procedure.
to system-dynamics-setup
  reset-ticks
  set dt 0.1
  ;; initialize constant values
  set sheep-birth-rate .04
  ;; initialize stock values
  set sheep 100
end

;; Step through the system dynamics model by performing next iteration of Euler's method.
;; Call this in your model's GO procedure.
to system-dynamics-go

  ;; compute variable and flow values once per step
  let local-sheep-births sheep-births
```

Step 2: NetLogo Integration

Once you create an aggregate model with the System Dynamics Modeler, you can interact with the model through the main NetLogo interface window. Let's build our NetLogo model to run the code generated by our diagram. We'll need a `setup` and `go` buttons which call the `system-dynamics-setup` and `system-dynamics-go procedures` created by the System Dynamics Modeler. And we'll want a monitor and a plot to watch the changes in sheep population.

> - Select the main NetLogo window
> - In the Code tab, write:
>
> ```
> to setup
>   ca
>   system-dynamics-setup
> end
>
> to go
>   system-dynamics-go
>   system-dynamics-do-plot
> end
> ```

- Move to the Interface tab
- Create a `setup` button
- Create a `go` button (don't forget to make it forever)
- Create a `sheep` monitor.
- Create a plot called "populations" with a pen named "sheep".

Now we're ready to run our model.

- Press the setup button.
- Don't press the "go" button yet. Instead, type `go` four or five times into the Command Center

Notice what happens. The sheep population increases exponentially. After four or five iterations, we have an enormous number of sheep. That's because we have sheep reproduction, but our sheep never die.

To fix that, let's finish our diagram by introducing a population of wolves which eat sheep.

Step 3: Wolf Predation

- Move back to the System Dynamics window
- Add a stock of wolves
- Add Flows, Variables and Links to make your diagram look like this:
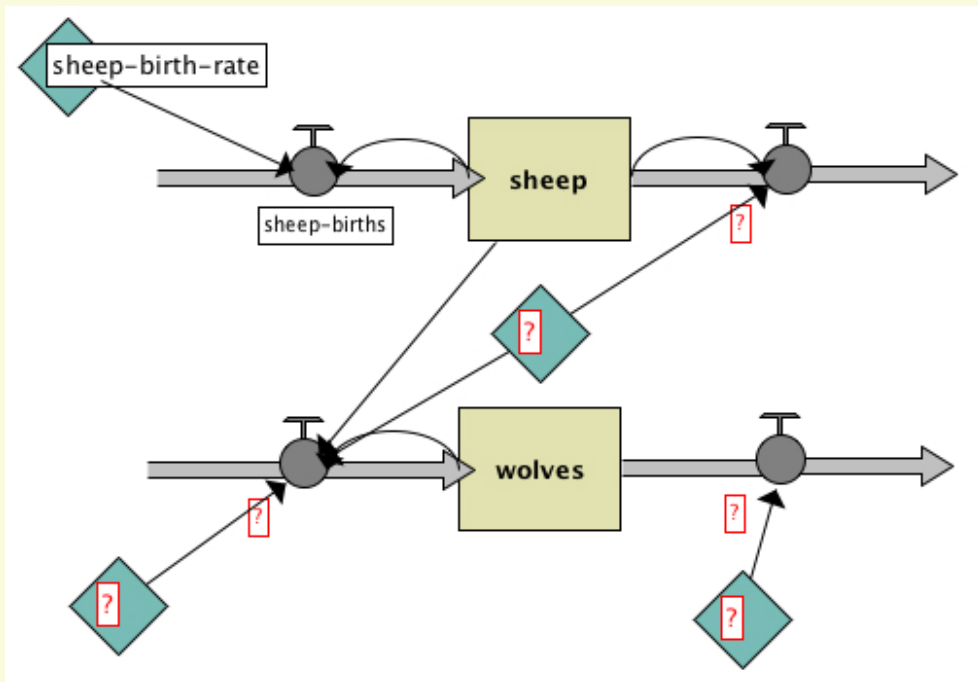
- Add one more Flow from the `wolves` Stock to the Flow that goes out of the Sheep stock.
- Fill in the names of the diagram elements so it looks like this:
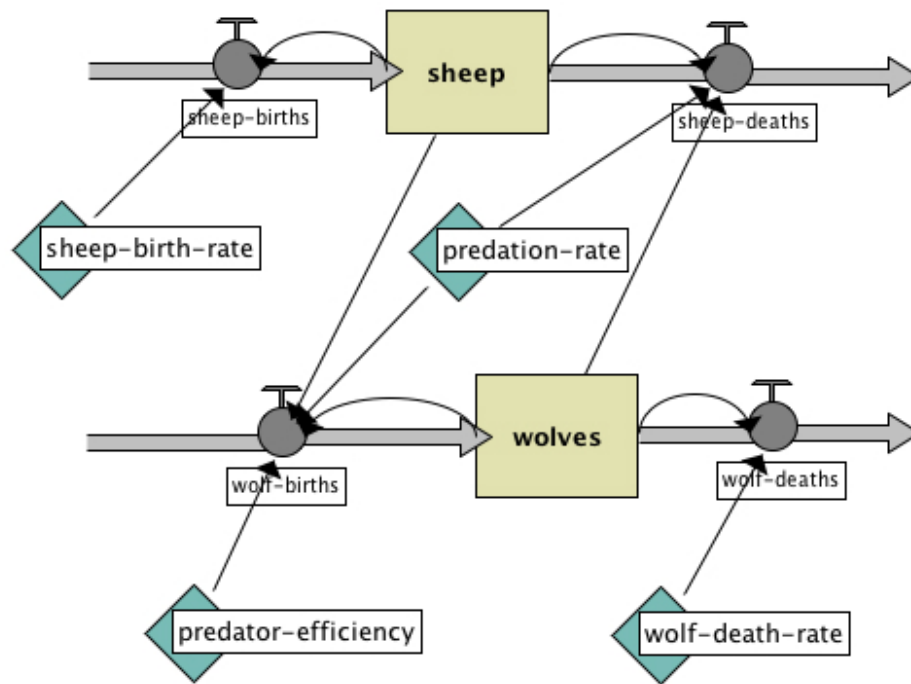
where
initial-value of `wolves` is `30`,
`wolf-deaths` is `wolves * wolf-death-rate` ,
`wolf-death-rate` is `0.15`,
`predator-efficiency` is `.8`,
`wolf-births` is `wolves * predator-efficiency * predation-rate * sheep`,
`predation-rate` is `3.0E-4`,
and `sheep-deaths` is `sheep * predation-rate * wolves`.

Now we're really done.

- Go back to the main NetLogo window
- Add a plot pen named "wolves" to the population plot
- Press `setup` and `go` to see your System Dynamics Modeler diagram in action.

You see a plot of the populations that looks like this:

setup

go

sheep
7379.098

wolves
467.279

ON
OFF More...

populations

Pens
■ wolves
■ sheep

8070

0

0   582
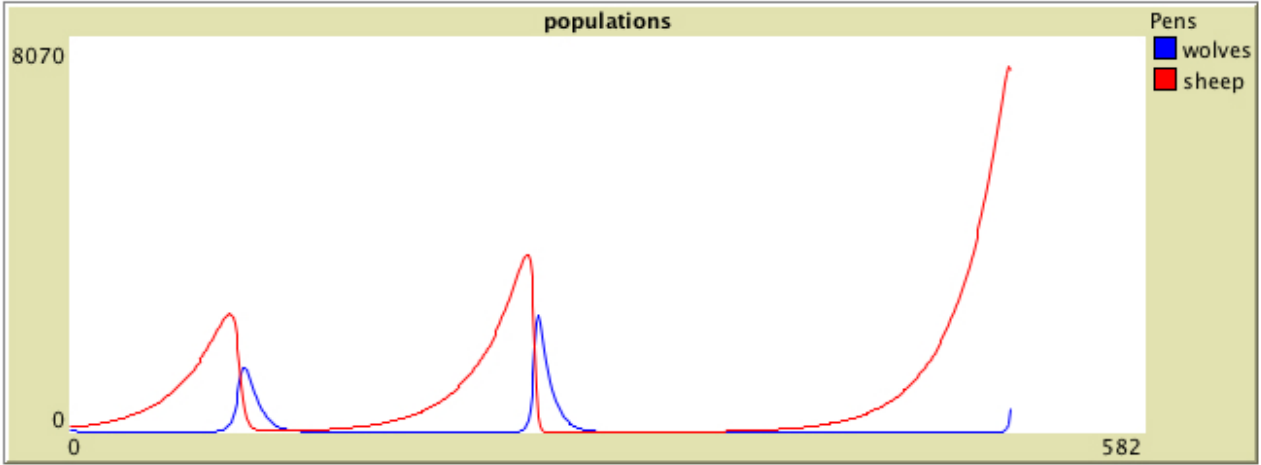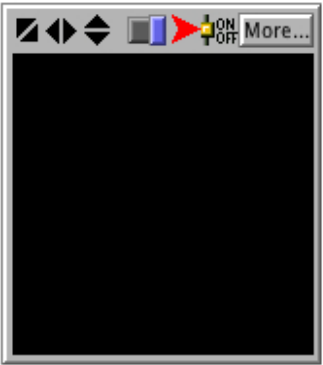
HubNet Guide

This section of the User Manual introduces the HubNet system and includes instructions to set up and run a HubNet activity.

HubNet is a technology that lets you use NetLogo to run *participatory simulations* in the classroom. In a participatory simulation, a whole class takes part in enacting the behavior of a system as each student controls a part of the system by using an individual device, such as a networked computer or Texas Instruments graphing calculator.

For example, in the Gridlock simulation, each student controls a traffic light in a simulated city. The class as a whole tries to make traffic flow efficiently through the city. As the simulation runs, data is collected which can afterwards be analyzed on a computer or calculator.

For more information on participatory simulations and their learning potential, please visit the [Participatory Simulations Project web site](#).

Understanding HubNet

NetLogo

NetLogo is a programmable modeling environment. It comes with a large library of existing simulations, both participatory and traditional, that you can use and modify. Content areas include social science and economics, biology and medicine, physics and chemistry, and mathematics and computer science. You and your students can also use it to build your own simulations.

In traditional NetLogo simulations, the simulation runs according to rules that the simulation author specifies. HubNet adds a new dimension to NetLogo by letting simulations run not just according to rules, but by direct human participation.

Since HubNet builds upon NetLogo, we recommend that before trying HubNet for the first time, you become familiar with the basics of NetLogo. To get started using NetLogo models, see [Tutorial #1: Running Models](#) in the NetLogo Users Manual.

HubNet Architecture

HubNet simulations are based on a client/server architecture. The activity leader uses the NetLogo application to run a HubNet activity. When NetLogo is running a HubNet activity, we refer to it as a HubNet server. Participants use a client application to log in and interact with the HubNet server.

There are two types of HubNet available. With [Computer HubNet](#), participants run the HubNet Client application on computers connected by a regular computer network. In [Calculator HubNet](#), created in conjunction with Texas Instruments, participants use Texas Instruments graphing calculators as clients which communicate via the TI-Navigator system.

We hope to add support for other types of clients such as tablets and phones.

Computer HubNet

Activities

The following activities are available in the Models Library, in the HubNet Activities folder. Information on how to run the models and activities can be found in the Info tab of each model. Additional discussion of educational goals and ways to incorporate many of the activities into your classroom in the Participatory Simulations Guide on the Participatory Simulations Project web site.

- Bug Hunters Camouflage - students hunt bugs and camouflaging emerges.
- Dice Stalagmite HubNet - students roll dice and explore the space of dependent and independent events.
- Disease - A disease spreads through the simulated population of students.
- Disease Doctors - A slight modification to the Disease activity where some students can recover from the disease.
- Gridlock - Students use traffic lights to control the flow of traffic through a city.
- Polling - Ask students questions and plot their answers.
- Root Beer Game - An adaptation of a popular game created at MIT in the early 1960s that shows how small delays in a distribution system can create big problems.
- Sampler - Students engage in statistical analysis as individuals and as a classroom. Through these activities, students discover the meaning and use of basic concepts in statistics.
- Tragedy of the Commons - Students work as farmers sharing a common resource.

Clients

To use the client application you simply need to launch the HubNet client application that is bundled with NetLogo.

Requirements

To use Computer HubNet, you need a networked computer with NetLogo installed for the server. When using the client application you will also need a networked computer with NetLogo installed for each participant. When using in classroom settings we also suggest an attached projector for the leader to project the entire simulation to the participants.

Starting an activity

You'll find the HubNet activities in NetLogo's Models Library, in the HubNet Activities folder. We suggest doing a few practice runs of an activity before trying it in front of a class.
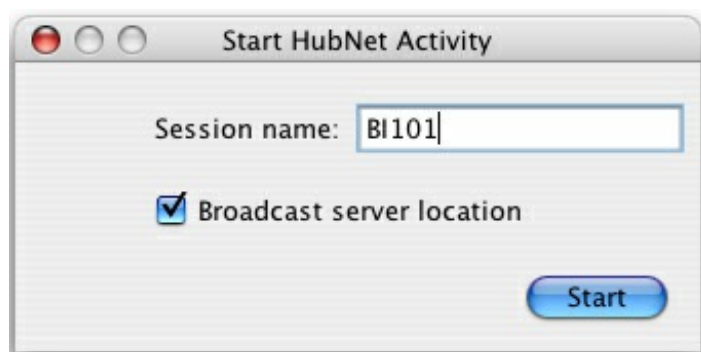
Open a Computer HubNet model. NetLogo will prompt you to enter the name of your new HubNet session. This is the name that participants will use to identify this activity. Enter a name and press Start.

NetLogo will open the HubNet Control Center, which lets you interact with the HubNet server.

You, as the leader, should then notify everyone that they may join. To join the activity, participants launch the HubNet Client application and enter their name. They should see your activity listed and can join your activity by selecting it and pressing Enter. If the activity you started is not listed the student can enter the server address manually which can be found in the HubNet Control Center.

HubNet Control Center
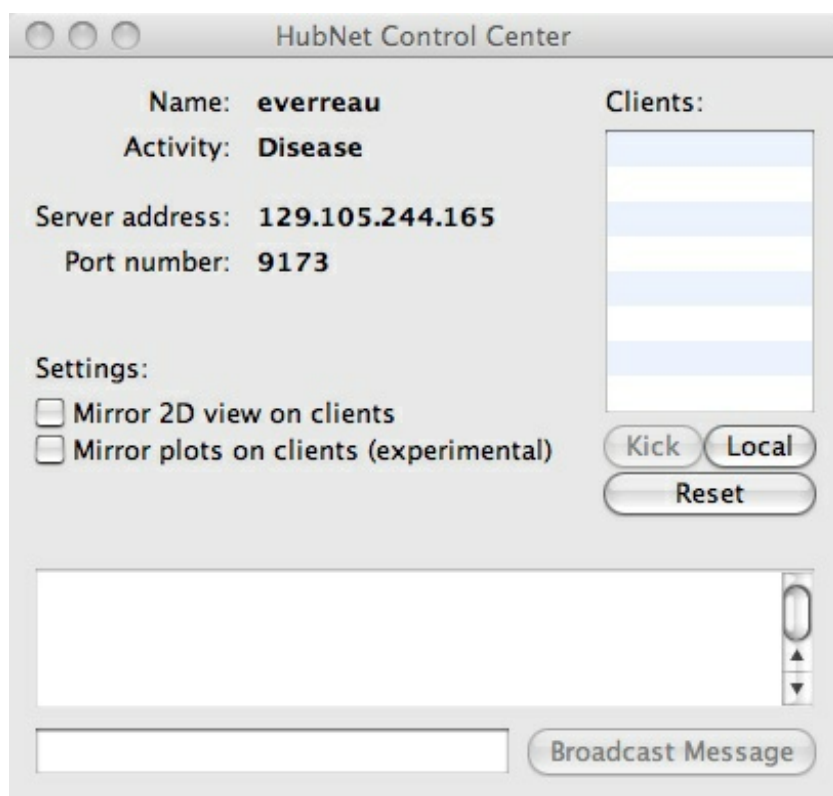
The HubNet Control Center lets you interact with the HubNet server. It displays the name, activity, address and port number of your server. The "Mirror 2D View on clients" checkbox controls whether the HubNet participants can see the view on their clients, assuming there is a view in the client setup. The "Mirror plots on clients" checkbox controls whether participants will receive plot information.

The client list on the right displays the names of clients that are currently connected to you activity. To remove a participant from the activity, select their name in the list and press the Kick button. To launch your own HubNet client press the Local button, this is particularly useful when you are debugging an activity. The "Reset" button kicks out all currently logged in clients and reloads the client interface.

The lower part of the Control Center displays messages when a participant joins or leaves the activity. To broadcast a message to all the participants, click on the field at the bottom, type your message and press Broadcast Message.

Troubleshooting

I started a HubNet activity, but when participants open a HubNet Client, my activity isn't listed.

On some networks, the HubNet Client cannot automatically detect a HubNet server. Tell your participants to manually enter the server address and port of your HubNet server, which appear in the HubNet Control Center.

Note: The technical details on this are as follows. In order for the client to detect the server, multicast routing must be available between them. Not all networks support multicast routing. In particular, networks that use the IPsec protocol typically do not support multicast. The IPsec protocol is used on many virtual private networks (VPNs).

When a participant tries to connect to an activity, nothing happens (the client appears to hang or gives an error saying that no server was found).

If your computer or network has a firewall, it may be impeding the HubNet server from communicating. Make sure that your computer and network are not blocking ports used by the HubNet server (ports 9173-9180).

The view on the HubNet client is gray.

- Verify that the "Mirror 2D view on clients" checkbox in the HubNet Control Center is selected.
- Make sure that the display switch in the model is on.
- If you have made changes to the size of the view on the server you may need to press the "Reset" button in the Control Center to ensure the clients get the new size.

There is no view on the HubNet client.

Some activities don't have a view on the client. If you want to add a view simply select "HubNet Client Editor" from the Tools Menu and add a view like any other widget. Make sure to press the "Reset" button before having clients log in.

I can't quit a HubNet client.

You will have to force the client to quit. On OS X, force quit the application by selecting Force Quit... in the Apple menu. On Windows, press Ctrl-Alt-Delete to open the Task Manager, select HubNet Client and press End Task.

My computer went to sleep while running a HubNet activity. When I woke the computer up, I got an error and HubNet wouldn't work anymore.

The HubNet server may stop working if the computer goes to sleep. If this happens, quit the NetLogo application and start over. Change the settings on your computer so it won't sleep again.

My problem is not addressed on this page.

See Contacting Us.

Known Limitations

If HubNet malfunctions, see the bug reporting information at Contacting Us.

Please note that:

- HubNet has not yet been extensively tested with large numbers of clients (i.e. more than about 25). Unexpected results may occur with more clients.
- Out-of-memory conditions are not handled gracefully
- Sending large amounts of plotting messages to the clients can take a long time.
- NetLogo does not handle malicious clients in a robust manner (in other words, it is likely vulnerable to denial-of-service type attacks).
- Performance does not degrade gracefully over slow or unreliable network connections.
- If you are on a wireless network or sub-LAN, the IP address in the HubNet Control Center is not always the entire IP address of the server.
- Computer HubNet has only been tested on LANs, and not on dial-up connections or WANs.

HubNet applet

WARNING: The HubNet client applet option is experimental and unsupported. It will probably be removed in a future NetLogo version, and it may not work even in this version. We don't recommend using applets at all; details here.

Client applets use the same client interface as the clients run in the HubNet application. Client applets and clients run through the HubNet application can be used at the same time in the same activity. In order to use a client applet you first have to save the client interface as an applet. You can do so by pressing the "Save Client As Applet..." button in the HubNet Client Editor toolbar.
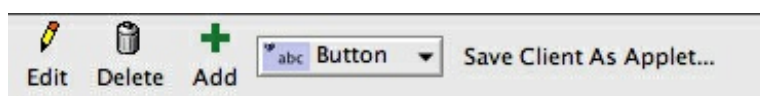
To access the client over the Internet you need to put the generated html file and HubNet.jar (you do not need the model file) somewhere that is web accessible on the machine that you intend to run the server on. You must run the server on the same computer as you host the client applet or the client applet will not be able to connect to the server due to security restrictions.

Calculator HubNet

Calculator HubNet for TI-Navigator

The TI-Navigator Classroom Learning System is a wireless classroom network for TI graphing calculators. TI-Navigator users can install a free NetLogo extension, which integrates with TI-Navigator and allows the calculators to act as clients for participatory simulations like the ones that are available for Computer HubNet. The Calculator HubNet extension is available from Inquire Learning, LLC, in collaboration with Texas Instruments. Inquire Learning also offers support, curricular materials, and professional development for the Calculator HubNet system. For more information on the TI-Navigator system itself, please visit the Texas Instruments web site, at http://education.ti.com/navigator. For more information on the Calculator HubNet extension for TI-Navigator, please contact Inquire Learning, at calc-hubnet@inquirelearning.com, or visit http://www.inquirelearning.com/calc-hubnet.html.

Teacher workshops

For information on upcoming workshops and NetLogo and HubNet use in the classroom, please contact us.

HubNet Authoring Guide

To learn about authoring or modifying HubNet activities, see the HubNet Authoring Guide.

Running HubNet in headless mode

To learn about running HubNet activities from the command line, with no GUI on the server, see the HubNet section in the Controlling Guide.

Getting help

If you have any questions about HubNet or need help getting started, contact us.

HubNet Authoring Guide

This guide shows how to understand and modify the code of existing HubNet activities and write your own new ones. It assumes you are familiar with running HubNet activities, basic NetLogo code and NetLogo interface elements. For more general information about HubNet see the HubNet Guide.

- Coding HubNet activities
  - Setup
  - Receiving information from clients
  - Sending information to clients
  - Examples
- How to make a client interface
- View updates on the clients
- Clicking in the view on clients
- Customizing the client's view
- Plot updates on the clients

Coding HubNet activities

Many HubNet activities will share bits of the same code. That is the code that it used to setup the network and the code that is used to receive information from and send information to the clients. If you understand this code you should be able to easily make modifications to existing activities and you should have a good start on writing your own activities. To get you started we have provided a Template model (in HubNet Activities -> Code Examples) that contains the most basic components that will be in the majority of HubNet activities. You should be able to use this activity as a starting point for most projects.

> **Code Example:** Template

Setup

To make a NetLogo model into a HubNet activity you must first initialize the network. In most HubNet activities you will use the `startup` procedure to initialize the network. `startup` is a special procedure that NetLogo runs automatically when you open any model. That makes it a good place to put code that you want to run once and only once (no matter how many times the user runs the model). For HubNet we put the command that initializes the network in `startup` because once the network is setup we don't need to do so again. We initialize the system using `hubnet-reset`, which will ask the user for a session name and open up the HubNet Control Center. Here is the startup procedure in the template model:

```
to startup
  hubnet-reset
end
```

Now that the network is all setup you don't need to worry about calling `hubnet-reset` again. Take a look at the setup procedure in the template model:

```
to setup
  cp
  cd
  clear-output
```

```
  ask turtles
  [
    set step-size 1
    hubnet-send user-id "step-size" step-size
  ]
end
```

For the most part it looks like most other setup procedures, however, you should notice that it does not call `clear-all`. In this model, and in the great majority of HubNet activities in the Models Library, we have a breed of turtles that represent the currently logged in clients. In this case we've called this breed `students`. Whenever a client logs in we create a student and record any information we might need later about that client in a turtle variable. Since we don't want to require users to log out and log back in every time we setup the activity we don't want to kill all the turtles, instead, we want to set all the variables back to initial values and notify the clients of any changes we make (more on that later).

Receiving messages from clients

During the activity you will be transferring data between the HubNet clients and the server. Most HubNet activities will call a procedure in the `go` loop that checks for new messages from clients in this case it's called listen clients:

```
to listen-clients
  while [ hubnet-message-waiting? ]
  [
    hubnet-fetch-message
    ifelse hubnet-enter-message?
    [ create-new-student ]
    [
      ifelse hubnet-exit-message?
      [ remove-student ]
      [ execute-command hubnet-message-tag ]
    ]
  ]
end
```

As long as there are messages in the queue this loop fetches each message one at a time. hubnet-fetch-message makes the next message in the queue the current message and sets the reporters hubnet-message-source, hubnet-message-tag, and hubnet-message to the appropriate values. The clients send messages when the users login and logout any time the user manipulates one of the interface elements, that is, pushes a button, moves a slider, clicks in the view, etc. We step through each message and decide what action to take depending on the type of message (enter, exit, or other), the hubnet-message-tag (the name of the interface element), and the hubnet-message-source of the message (the name of the client the message came from).

On an enter message we create a turtle with a `user-id` that matches the hubnet-message-source which is the name that each user enters upon entering the activity, it is guaranteed to be unique.

```
to create-new-student
  create-students 1
  [
    set user-id hubnet-message-source
    set label user-id
    set step-size 1
    send-info-to-clients
  ]
end
```

At this point we set any other client variables to default values and send them to the clients if appropriate. We declared a `students-own` variable for every interface element on the client that holds state, that is, anything that would be a global variable on the server, sliders, choosers, switches and input boxes. It is important to make sure that these variables stay synchronized with the values visible on the client.

When the clients logout they send an exit message to the server which gives you a chance to clean up any information you have been storing about the client, in this case we merely have to ask the appropriate turtle to die.

```
to remove-student
  ask students with [user-id = hubnet-message-source]
  [ die ]
end
```

All other messages are interface elements identified by the `hubnet-message-tag` which is the name that appears in the client interface. Every time an interface element changes a message is sent to the server. Unless you store the state of the values currently displayed in the client interface will not be accessible in other parts of the model. That's why we've declared a `students-own` variable for every interface element that has a state (sliders, switches, etc). When we receive the message from the client we set the turtle variable to the content of the message:

```
if hubnet-message-tag = "step-size"
[
  ask students with [user-id = hubnet-message-source]
    [ set step-size hubnet-message ]
]
```

Since buttons don't have any associated data there is generally no associated turtle variable, instead they indicate an action taken by the client, just as with a regular button there is often procedure associated with each button that you call whenever you receive a message indicating the button has been pressed. Though it is certainly not required, the procedure is often a turtle procedure, that is, something that the student turtle associated with the message source can execute:

```
if command = "move left"
[ set heading 270
  fd 1 ]
```

Sending messages to clients

As mentioned earlier you can also send values to any interface elements that display information: monitors, sliders, switches, choosers, and input boxes (note that plots and the view are special cases that have their own sections).

There are two primitives that allow you to send information `hubnet-send` and `hubnet-broadcast`. Broadcast sends the information to all the clients; send sends to one client, or a selected group.

As suggested earlier, nothing on the client updates automatically. If a value changes on the server, it is your responsibility as the activity author to update monitors on the client.

For example, say you have a slider on the client called step-size and a monitor called Step Size (note that the names must be different) you might write updating code like this:

```
if hubnet-message-tag = "step-size"
[
  ask student with [ user-id = hubnet-message-source ]
  [
    set step-size hubnet-message
    hubnet-send user-id "Step Size" step-size
  ]
]
```

You can send any type of data you want, numbers, strings, lists, lists of lists, lists of strings, however, if the data is not appropriate for the receiving interface element (say, if you were to send a string to a slider) the message will be ignored. Here are a few code examples for different types of data:

data type	`hubnet-broadcast` example	`hubnet-send` example
number	`hubnet-broadcast "A" 3.14`	`hubnet-send "jimmy" "A" 3.14`
string	`hubnet-broadcast "STR1" "HI THERE"`	`hubnet-send ["12" "15"] "STR1" "HI THERE"`
list of numbers	`hubnet-broadcast "L2" [1 2 3]`	`hubnet-send hubnet-message-source "L2" [1 2 3]`
matrix of numbers	`hubnet-broadcast "[A]" [[1 2] [3 4]]`	`hubnet-send "susie" "[A]" [[1 2] [3 4]]`
list of strings	`hubnet-broadcast "user-names" [["jimmy" "susie"] ["bob" "george"]]`	`hubnet-send "teacher" "user-names" [["jimmy" "susie"] ["bob" "george"]]`

Examples

Study the models in the "HubNet Activities" section of the Models Library to see how these primitives are used in practice in the Code tab. Disease is a good one to start with.

How to make a client interface

Open the HubNet Client Editor, found in the Tools Menu. Add any buttons, sliders, switches, monitors, plots, choosers, or notes that you want just as you would in the interface tab. You'll notice that the information you enter for each of the widgets is slightly different than in the Interface panel. Widgets on the client don't interact with the model in the same way. Instead of a direct link to commands and reporters the widgets send messages back to the server and the model then determines how those messages affect the model. All widgets on the client have a tag which is a name that uniquely identifies the widget. When the server receives a message from that widget the tag is found in `hubnet-message-tag`

For example, if you have a button called "move left", a slider called "step-size", a switch called "all-in-one-step?", and a monitor called "Location:", the tags for these interface elements will be as follows:

interface element	tag
move left	move left
step-size	step-size
all-in-one-step?	all-in-one-step?

Location:	Location:

Note that you can only have **one** interface element with a specific name. Having more than one interface element with the same tag in the client interface will result in unpredictable behavior since it is not clear which element you intended to send the information to.

View updates on the clients

View mirroring lets views of the world be displayed in clients as well on the server. View mirroring is enabled using a checkbox in the HubNet Control Center.

When mirroring is enabled, client views update whenever the view on the server does. To avoid excessive network traffic, the view should not update more often than necessary. Therefore we strongly recommend using tick-based updates, rather than continuous updates. See the View Updates section of the Programming Guide for an explanation of the two types of updates.

With tick-based updates, updates happen when a `tick` or `display` command runs. We recommend using these commands only inside an `every` block, to limit the frequency of view updates and thus also limit network traffic. For example:

```
every 0.1
[
  display
]
```

If there is no View in the clients or if the Mirror 2D View on Clients checkbox in the HubNet Control Center is not checked, then no view updates are sent to the clients.

Clicking in the view on clients

If the View is included in the client, two messages are sent to the server every time the user clicks in the view. The first message, when the user presses the mouse button, has the tag "View". The second message, sent when the user releases the mouse button, has the tag "Mouse Up". Both messages consist of a two item list of the x and y coordinates. For example, to turn any patch that was clicked on by the client red, you would use the following NetLogo code:

```
if hubnet-message-tag = "View"
[
  ask patches with [ pxcor = (round item 0 hubnet-message) and
                     pycor = (round item 1 hubnet-message) ]
  [ set pcolor red ]
]
```

Customizing the client's view

When view mirroring is enabled, by default clients see the same view the activity leader sees on the server. But you can change this so that each client sees something different, not just a literal "mirror".

You can change what a client sees in two distinct ways. We call them "client perspectives" and "client overrides".

Changing a client's perspective means making it "watch" or "follow" a particular agent, much

like the `watch` and `follow` commands that work with ordinary NetLogo models. See the dictionary entries for `hubnet-send-watch`, `hubnet-send-follow`, and `hubnet-reset-perspective`.

> **Code Example:** Client Perspective Example

Client overrides let you change the appearance of patches, turtles, and links in the client views. You can override any of the variables affecting an agent's appearance, including the `hidden?` variable causing a turtle or link to be visible or invisible. See the dictionary entries for `hubnet-send-override`, `hubnet-clear-override`, and `hubnet-clear-overrides`.

> **Code Example:** Client Overrides Example

Plot updates on the clients

If plot mirroring is enabled (in the HubNet Control Center) and a plot in the NetLogo model changes and a plot with the exact same name exists on the clients, a message with that change is sent to the clients causing the client's plot to make the same change. For example, let's pretend there is a HubNet model that has a plot called Milk Supply in NetLogo and the clients. Milk Supply is the current plot in NetLogo and in the Command Center you type:

```
plot 5
```

This will cause a message to be sent to all the clients telling them that they need to plot a point with a y value of 5 in the next position of the plot. Notice, if you are doing a lot of plotting all at once, this can generate a lot of plotting messages to be sent to the clients.

Modeling Commons Guide

Introduction

The Modeling Commons (http://modelingcommons.org/) is a Web-based collaboration system for NetLogo modelers. Users of the Modeling Commons can share, download, modify, create variations of, comment on, and run NetLogo models -- both those that are a part of the NetLogo models library, and also those that have been uploaded by other NetLogo users.

By uploading your NetLogo models to the Modeling Commons, you make it easy for others to see, review, and comment on your work. You can optionally keep the model private, either to yourself or to a group of your choice, if you aren't comfortable with letting everyone see the model. You can always change the permissions associated with a model, if you change your mind later on.

NetLogo now makes it possible to save models to the Modeling Commons, just as you can save them to `.nlogo` files on your own computer. You can access this functionality by selecting "Upload to Modeling Commons" from the "File" menu.

Use of the Modeling Commons is free of charge. You may use it for your own personal work, for your research group or company, or for a class in which you are a student or teacher. The Modeling Commons is sponsored by the CCL, the same group that develops and distributes NetLogo.

Modeling Commons Accounts

In order to upload models to the Modeling Commons, you must first be a registered user. Unregistered users can view and download models, but cannot upload, edit, or comment on them.

The first time that you invoke "Save to Modeling Commons" in NetLogo, you will be prompted to enter your e-mail address and password. If you already have an account, then you can enter this information and click on the "Login" button.

If you don't yet have an account with the Modeling Commons, then you will need to create one. Click on the "Create Account" button, and enter the requested information. Once you have done so, click on the "Create Account" button. If there are no errors, then you will be prompted to upload a NetLogo model. Alternatively, you may go to the Modeling Commons itself and register with your Web browser.

Uploading Models

There are three ways to upload a model to the Modeling Commons: Uploading, updating, and creating a child ("forking"). The following sections describe these in detail.

Upload A New Model

A new model will be created in the Modeling Commons, with its own page, description, and forum. You should use this function the first time that you save a model to the Modeling Commons.

You must give your model a name. Model names are not required to be unique; you could have 2 or more models with the same name, though we recommend that you not do this.

By default, anyone can view, fork, and update your model. You can restrict the ability to view and fork your model by changing the visibility permission. You can restrict the ability to update your model by changing the changeability permission. In order to set permissions for multiple people, assign your model to a group, and then restrict visibility or changeability to members of that group. Groups can be created from the Modeling Commons. Once you have uploaded your model, you can edit the permissions from the model's Modeling Commons page.

You can optionally upload a preview image to your model. The preview image will be displayed alongside your model whenever it is shown on the Modeling Commons. While uploading a preview image is optional, we highly recommend that you do so, in one of the following three ways:

- The "Use current image" option tells NetLogo to use the current view as your preview. We recommend that you first run the model, such that it shows off the key visual features.
- The "Auto-generate image" feature auto-generates a preview image by running `random-seed 0 setup repeat 75 [ go ]`. This option will only be enabled if you have defined `setup` and `go` procedures for NetLogo to run.
- The "Image from file" feature allows you to upload any PNG image. Preview images work

best when they are square.

Upload A Child Of An Existing Model ("forking")

Saving a model in this way, sometimes known as "forking," does not change or overwrite the original model. Rather, it creates a new model on the Modeling Commons, much as a plain "save" would do, simultaneously creating a parent-child relationship between the old model and the new one. This relationship can be seen on the "family" tab for a given model. You may fork any model for which you have "view" permissions, including one that you cannot change. You may wish, for example, to create a variation on a model in the NetLogo models library.

To fork a model, you must give your new child a name, as well as select an existing model to fork. To indicate the existing model, start typing the name of the model that you would like to fork. Select its name from among the search results.

Finally, you must enter a description about what you are changing in your child model, and how it relates to its parent.

Updating An Existing Model

Use this option if you have improved a model that already exists in the Modeling Commons. Existing attachments, discussions, and social tags will be preserved, but the model that users can display, run, and download will be updated. You may only update a model for which you have "write" permissions.

All versions of a model are saved in the Modeling Commons, so you should feel free to experiment with new ideas. If something goes wrong, you can always refer to an old version from

the "history" tab on a model's page.

To indicate which model should be updated, start typing the name of the model. Select the name that pops up with the search results. Finally, enter a description about what you are changing in your new version.

Logging

NetLogo's logging facility allows researchers to record student actions for later analysis.

Logging in NetLogo, once initiated, is invisible to the student. The researcher can choose the type of events logged through a configuration file.

NetLogo uses the log4j package for logging. If you have previous experience with this package you'll find logging in NetLogo familiar.

Logging is supported only by the special NetLogo Logging application.

Starting logging

This depends on what operating system you are using.

Mac OS X or Windows

There is a special logging launcher in the NetLogo directory called NetLogo Logging. Double click on the icon.

On Windows, the NetLogo directory can be found at `C:\Program Files`, unless you chose a different location when you installed NetLogo.

Linux and others

To enable logging, invoke the `netlogo.sh` script as follows:

```
netlogo.sh --logging netlogo_logging.xml
```

You could also modify the script to include these flags, or copy the script and modify the copy.

You can replace netlogo_logging.xml with any valid log4j XML configuration file, which will be discussed in more detail later.

Using logging

When NetLogo starts up it will ask for a user name. This name will appear in all the logs generated during this session.

Where logs are stored

Logs are stored in the OS-specific temp directory. On most Unix-like systems that is `/tmp`. On Windows Vista the logs can be found in `c:\Users\<user>\AppData\Local\Temp`, where `<user>` is the logged in user. On Mac OS X, the temp directory varies for each user. You can determine your temp directory by opening the Terminal application and typing `echo $TMPDIR` at the prompt.

There are two convenience commands that will help you manage the logs. `__zip-log-files` *filename* will gather all the logs in the temp directory and put them in one zip file, at the location specified. After doing `__zip-log-files` the existing logs are **not** deleted, you can do so

explicitly by using `__delete-log-files`.

The following is a chart describing the name of the loggers available, the type of events each logs, at what level, and provides a sample output using the XMLLayout. All the loggers are found in `org.nlogo.log.Logger`. When referring to the loggers in the configuration file you should use the fully qualified name. So, for example, the logger `GLOBALS` would actually be `org.nlogo.log.Logger.GLOBALS`

Logger	Events	Level	Example
GLOBALS	a global variable changes	info, debug	```<event logger="org.nlogo.log.Logger.GLOBALS" timestamp="1177341065988" level="INFO" type="globals"> <name>FOO</name> <value>51.0</value> </event>```
GREENS	sliders, switches, choosers, input boxes are changed through the interface	info	```<event logger="org.nlogo.log.Logger.GREENS" timestamp="1177341065988" level="INFO" type="slider"> <action>changed</action> <name>foo</name> <value>51.0</value> <parameters> <min>0.0</min> <max>100.0</max> <inc>1.0</inc> </parameters> </event>```
CODE	code is compiled, including: command center, Code tab, slider bounds, and buttons	info	```<event logger="org.nlogo.log.Logger.CODE" timestamp="1177341072208" level="INFO" type="command center"> <action>compiled</action> <code>crt 1</code> <agentType>O</agentType> <errorMessage>success</errorMessage> </event>```
WIDGETS	a widget is added or removed from the interface	info	```<event logger="org.nlogo.log.Logger.WIDGETS" timestamp="1177341058351" level="INFO" type="slider"> <name></name> <action>added</action> </event>```
BUTTONS	a button is pressed or released	info	```<event logger="org.nlogo.log.Logger.BUTTONS" timestamp="1177341053679" level="INFO" type="button"> <name>show 1</name> <action>released</action> <releaseType>once</releaseType>```

| | | | ```
</event>
``` |
| SPEED | the speed slider changes | info | ```
<event logger="org.nlogo.log.Logger.SPEED"
       timestamp="1177341042202"
       level="INFO"
       type="speed">
  <value>0.0</value>
</event>
``` |
| TURTLES | turtles die or are born | info | ```
<event logger="org.nlogo.log.Logger.TURTLES"
       timestamp="1177341094342"
       level="INFO"
       type="turtle">
  <name>turtle 1</name>
  <action>born</action>
  <breed>TURTLES</breed>
</event>
``` |
| LINKS | links die or are born | info | ```
<event logger="org.nlogo.log.Logger.LINKS"
       timestamp="1177341094342"
       level="INFO"
       type="link">
  <name>link 2 7</name>
  <action>born</action>
  <breed>LINKS</breed>
</event>
``` |

How to configure the logging output

The default logging configuration (netlogo_logging.xml) looks something like this:

NetLogo defines 8 loggers, all descend directly from the root logger, which means unless you explicitly set the properties (appender, layout, and output level) in the configuration they will inherit them from the root. In the default configuration the root is set to level INFO, the appender is org.nlogo.log.XMLFileAppender and layout is org.nlogo.log.XMLLayout. Together these generate a nicely formatted XML file as defined in the netlogo_logging.dtd which is based on the log4j dtd. If the appender is a FileAppender (including the XMLFileAppender) a new file is start each time the user opens a model.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration debug="false" xmlns:log4j='http://jakarta.apache.org/log4j/'>

        <appender name="A1" class="org.nlogo.log.XMLFileAppender">
          <layout class="org.nlogo.log.XMLLayout"/>
        </appender>

        <category name="org.nlogo.log.Logger.WIDGETS">
          <priority value="off" />
        </category>

        <category name="org.nlogo.log.Logger.TURTLES">
          <priority value="off" />
        </category>

        <category name="org.nlogo.log.Logger.LINKS">
```

```
            <priority value="off" />
        </category>

        <root>
            <priority value ="info" />
            <appender-ref ref="A1" />
        </root>

    </log4j:configuration>
```

This configuration, first defines an appender named "A1" of type XMLFileAppender with an XMLLayout. The appender defines where the logging data goes, in this case the data goes into a file. In fact, if NetLogo is given a FileAppender it will automatically start a new file every time the user opens a new model. The XMLFileAppender also does some formatting and writes the appropriate headers to the file. The layout defines how to write each individual message. Unless you are an advanced user there is no need change (or worry about) the appender or the layout.

At the end of the configuration notice the definition of the root logger. All of the other loggers descend from the root logger and, thus, inherit the properties of the root unless explicitly set. This case is fairly simple, having set up the appender A1 we make that the default appender for the root (and all other loggers) and make the default priority "INFO". Messages that are logged at the INFO level or higher will be written, messages logged at lower levels will not. Note that with only one exception NetLogo always logs at level INFO. Sets to globals that don't change the value of the global are logged at level DEBUG. Which means that these messages are disabled by default, since debug is lower level than info. The rest of the body of the configuration file overrides properties of the root logger in a few specific loggers (or categories as they are known in the configuration file, the terms can be assumed to be synonymous for the proposes of this document). That is it turns off the WIDGET, TURTLES, and LINKS loggers, by default. To re-enable them you can changes the priority from off to info, like this:

```
        <category name="org.nlogo.log.Logger.TURTLES">
          <priority value="info" />
        </category>
```

or you can simply remove the entire reference to the category from the configuration file, as it is not serving any other purpose.

Advanced Configuration

This is only a basic introduction to configuration files for logging in NetLogo. There are many more configuration options available through the log4j framework. See the log4j documentation.

Controlling Guide

NetLogo can be invoked and controlled by another program running on the Java Virtual Machine. For example, you might want to call NetLogo from a small program that does something simple like automate a series of model runs. Or, you might want to embed NetLogo models in a larger application.

For more information, go here.

Mathematica Link

What is it?

The NetLogo-Mathematica link provides modelers with an easy to use, real-time link between NetLogo and Mathematica. Together, these tools can provide users with a highly interactive, self-documenting work flow that neither can provide alone.

Mathematica includes many of the tools that agent-based modelers rely on throughout the research process: advanced import capabilities, statistical functions, data visualization, and document creation. With the NetLogo-Mathematica link, you can run all of these tools side-by-side with NetLogo.

Because all Mathematica documents, or notebooks, contain comments, code, images, annotations, and interactive objects, the integration of NetLogo and Mathematica provides a more complete solution for complex model exploration for students and researchers alike.

The basic functionality of the link is much like the NetLogo Controlling API: you can load models, execute commands, and report back data from NetLogo. Unlike the Controlling API, which is based on Java, all interactions with the link are interpreted, making it ideal not only for rapidly designing custom BehaviorSpace-like experiments, but also as a companion to NetLogo in debugging your model.

For more information about Mathematica, please visit the [Wolfram Research web site](#).

What can I do with it?

Here are a few examples of what you can do with the Mathematica-NetLogo link.

- Analyze your model in real-time with seamless two-way data conversion
- Develop high quality, custom visualizations of model data
- Collect detailed simulation data across large multi-dimensional parameter spaces
- Rapidly develop interactive interfaces for exploring model behavior
- Have direct access to patches and network data with built-in functions

Installation

The NetLogo-Mathematica link supports Mathematica 10 or greater. To install the NetLogo-Mathematica link:

- Go to the menu bar in Mathematica
- Click on File and select Install...
- In the Install Mathematica Item dialog
- Select Package for Type of item to install
- Click Source, and select From file...
- In the file browser, go to the location of your NetLogo installation,
- click on the Mathematica Link subfolder, and select NetLogo.m.
- For Install Name, enter NetLogo.

You can either install the NetLogo link in your user base directory or in the system-wide directory. If the NetLogo link is installed in the user base directory, other users on the system must also go through the NetLogo-Mathematica link installation process to use it. This option might be preferable if you do not have permission to modify files outside of your home directory. Otherwise, you can install NetLogo-Mathematica link in the system-wide

Mathematica base directory.

Usage

This section will very briefly introduce how to use the NetLogo-Mathematica Link. It will show you how to load the NetLogo-Mathematica link package, start NetLogo, execute commands, and retrieve data from NetLogo.

Loading the package: Once the NetLogo-Mathematica link is installed, you can load the package by entering the following into your Mathematica notebook:

```
<<NetLogo`
```

Launching NetLogo from Mathematica: To begin your NetLogo session in Mathematica, type the following into your notebook:

```
NLStart["your netlogo path"];
```

where *"your netlogo path"* is the directory that netlogo is located in. Typically on a Mac this will be "/Applications/NetLogo 5.3.1/"

Loading a model: To load a model, you must specify the full path of the model. In this example we will load the Forest Fire model, and the path will be given using the typical Mac install location.

```
NLLoadModel["/Applications/NetLogo 5.3.1/models/Sample Models/Earth
Science/Fire.nlogo"];
```

Executing a NetLogo command: Commands can be executed by passing a string of commands to `NLCommand[]`. The `NLCommand[]` function automatically splices common Mathematica data types into strings suitable for NetLogo. The following commands set the density using a single string, or set the density using a Mathematica defined variable, `myDensity`.

```
NLCommand["set density 50"];
myDensity = 60;
NLCommand["set density", myDensity];
```

Reporting information from NetLogo: NetLogo data can be reported back to Mathematica using `NLReport[]`. This includes numbers, strings, boolean values, and lists.

```
NLReport["count turtles"];
NLReport["[(list pxcor pycor)] of n-of 10 patches"]
```

For more information, see the NetLogo-Mathematica Tutorial notebook included with NetLogo. The notebook walks you through the process of using the link, with many examples along the way. If you do not have Mathematica, but are considering using the link, you can find a PDF of the notebook included with NetLogo in the "Mathematica Link" directory.

Known Issues

- A NetLogo session cannot be quit without exiting J/Link (the Java-Mathematica link) entirely. This may disrupt other packages that make use of J/Link.
- If a model loaded with the NetLogo-Mathematica link uses a NetLogo extension, the extension must be located in the same directory as the model itself. If the extension is located in NetLogo's application-wide extensions directory, it will not be found.
- Calls to NetLogo, such as `NLCommand[]` and `NLReport[]`, cannot be aborted.

Source code

The source code for the NetLogo-Mathematica link is in the public domain. It is hosted on line at https://github.com/NetLogo/Mathematica-Link.

Credits

The primary developer of the NetLogo-Mathematica link was Eytan Bakshy.

To refer to this package in academic publications, please use: Bakshy, E., Wilensky, U. (2007). NetLogo-Mathematica Link. http://ccl.northwestern.edu/netlogo/mathematica.html. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

NetLogo 3D

NetLogo includes the NetLogo 3D application that allows you to create 3D worlds.

Notice: NetLogo's support for 3D is less developed than NetLogo 2D. Models created with this release may not be compatible with future versions. While we've made efforts to ensure a quality product, NetLogo 3D has not been subject to the same level of quality control as the main application.

- [Introduction](#)
- [Tutorial](#)
- [Dictionary](#)

Introduction

To get started using NetLogo 3D, launch the NetLogo 3D application and check out the Sample Models in the 3D section of the Models Library.

When you're ready to write your own 3D model, look at the Code Examples in the 3D section of the Models Library.

> **Code Example:** Turtle Perspective Example 3D helps you learn about the different perspectives.
>
> **Code Example:** Turtle and Observer Motion Example 3D helps you understand how turtles and the observer move in 3D. You can also step through this model with the tutorial below.

3D Worlds

> *An unspeakable horror seized me. There was a darkness; then a dizzy, sickening sensation of sight that was not like seeing; I saw a Line that was no Line; Space that was not Space: I was myself, and not myself. When I could find voice, I shrieked loud in agony, "Either this is madness or it is Hell."*
>
> *"It is neither," calmly replied the voice of the Sphere, "it is Knowledge; it is Three Dimensions: open your eye once again and try to look steadily."*
> -- Edwin A. Abbott, *Flatland: A romance in many dimensions*

NetLogo 3D's world has width, height and depth. Patches are cubes. In addition to `pxcor` and `pycor`, patches have `pzcor`.

Turtles have three Cartesian coordinates, instead of two, to describe position. In addition to `xcor` and `ycor`, turtles have `zcor`.

A turtle's orientation is defined by three turtle variables, `heading`, `pitch` and `roll`. You can imagine the turtle as having two vectors to define its orientation in 3D space. One vector comes straight out of the nose of the turtle, this is the direction the turtle will travel when it moves forward. The second vector is perpendicular to the forward vector and comes out of the right side of the turtle (as if the turtle were to stick its right arm straight out from its body). Heading is the angle between the forward vector of the turtle projected onto the xy-plane and the vector [0 1 0]. Pitch is the angle between the forward vector of the turtle and the xy-plane and finally roll is the angle between the right vector of the turtle and the xy-

plane. When turtle turns right or left in 3D space it rotates around the down vector, that is the vector that is perpendicular to both the forward and right vectors. Depending on the orientation of the turtle more than one of the internal turtle variables may change as the result of a turn.

The observer and the 3D view

The point of view that you see the world from is considered the location and orientation of the observer. This is similar to the 3D view in NetLogo 2D. However, there are a few more ways to control the observer. You can set the point that the observer is facing by using `face` and `facexyz` which work the same way as the turtle commands, the observer turns so the center of the view is on the given point or the location of the given agent at the time it is called. You can change the location of the observer using `setxyz`. The observer will move to view the world as if standing on the given location, the point the observer faces will stay the same. For example create a new model and observer will be located at (0, 0, 49.5), that is, on the z-axis 49.5 patch units away from the origin and the observer is facing the origin, (0, 0, 0). If you `setxyz 0 49.5 0` the observer will move so it is on the positive y-axis but it will keep the origin at the center of the view. You can also move the observer using the rotation primitives that will allow you to move the observer around the world as if on the surface of a sphere where the center is the location the observer is facing. You may notice from the above examples that the observer is not constrained to be within the bounds of the world.

Custom Shapes

NetLogo automatically interprets 2D shapes so they are extruded, like a cookie cutter shape in the 3D view. You can also use the primitive load-shapes-3d to load shapes described in an external file in a custom format described here. Currently we do not import shapes in any standard formats.

For each shape in a custom 3D shape file, a 2D shape of the same name must exist as well. You can create the 2D shape in the Turtle Shapes Editor.

The input file may contain any number of shapes with any number of rectangular or triangular surfaces. The format of the input file should be as follows:

```
number of shapes in file
name of first shape
type of surface ( quads or tris )
surface1
surface2
.
.
.
stop
type of surface
surfaceA
.
.
.
stop
end-shape
```

Each surface is defined by a unit normal vector and the vertices listed in clockwise order, tris should have three vertices and quads should have four.

```
normal: xn yn zn
x1 y1 z1
```

```
x2 y2 z2
x3 y3 z3
x4 y4 z4
```

A file declaring just a two dimensional, patch-sized, square in the xy-plane centered at the origin would look like this:

```
1
square
quads
normal: 0 0 1
0.15 0.15 0
-0.15 0.15 0
-0.15 -0.15 0
0.15 -0.15 0
normal: 0 0 -1
0.15 0.15 0
0.15 -0.15 0
-0.15 -0.15 0
-0.15 0.15 0
stop
end-shape
```

Tutorial

Step 1: Depth

One of the first things you will notice when you open NetLogo 3D is that the world is a cube instead of a square.
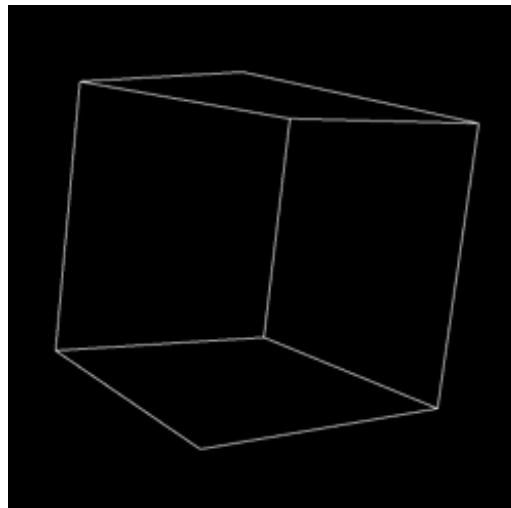
You can open up the Model Settings, by clicking on the "Settings..." button at the top of the 3D View. You'll notice in addition to `max-pxcor`, `min-pxcor`, `max-pycor`, and `min-pycor`, there is also `max-pzcor` and `min-pzcor`.
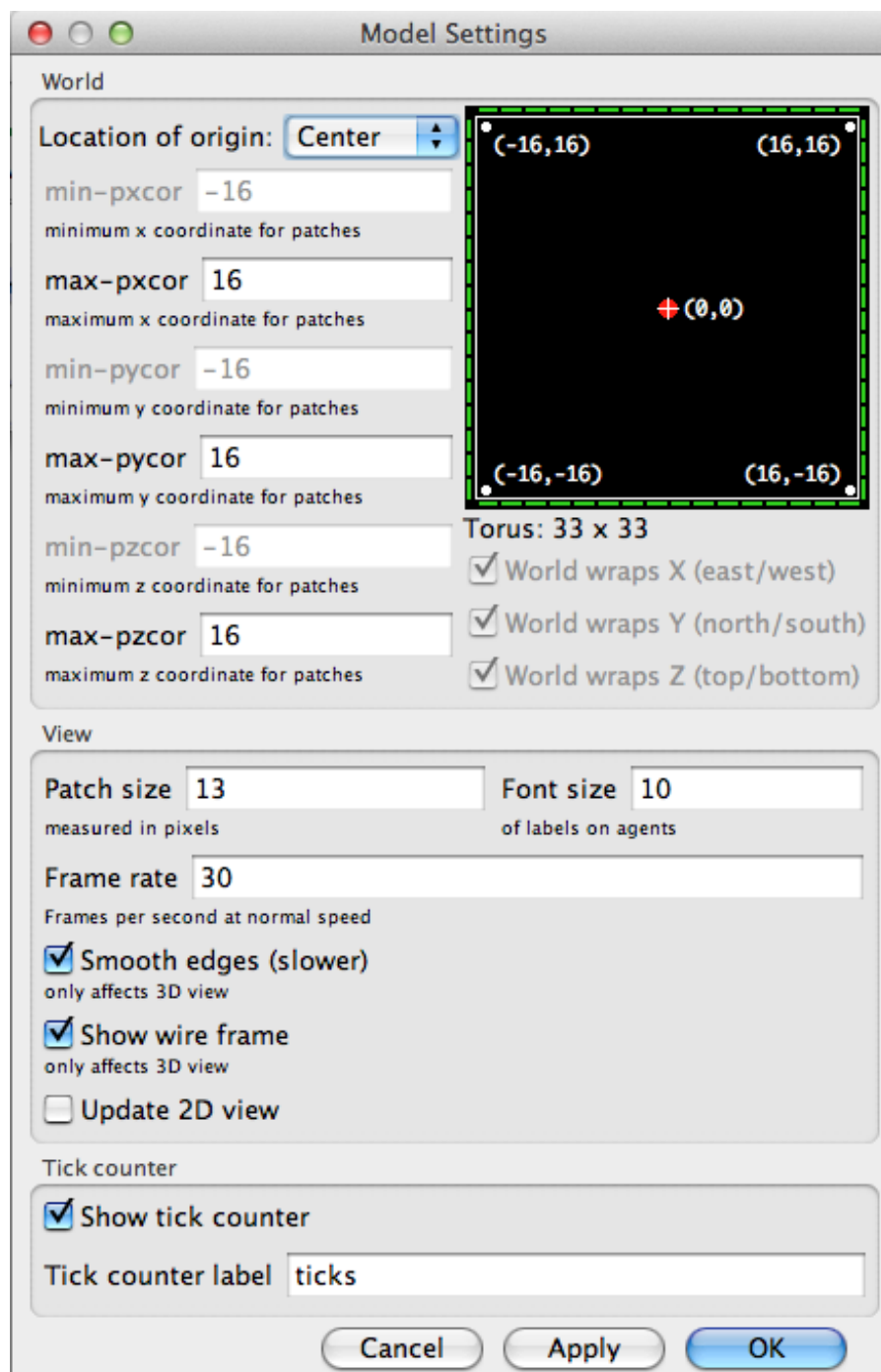
The z-axis is perpendicular to both the x-axis and the y-axis, when you `reset-perspective` it is the axis that comes straight out of the screen. In the default position `max-pzcor` is the face of the cube nearest to you and `min-pzcor` is the face farthest from you. As always `min-pxcor` is on the left, `max-pxcor` on the right, `min-pycor` on the bottom, and `max-pycor` on the top.

You'll also notice on the left side of the Model Settings that there are options for wrapping in all three directions, however, they are all checked and grayed out. Topologies are not yet supported in NetLogo 3D, so the world always wraps in all dimensions.

- Move to the Command Center and type `print count patches.`

  Is the number smaller or larger than you expected?

In a 3D world the number of patches grows very quickly since `count patches = world-width * world-height * world-depth`. It's important to keep this in mind when you are building your model. Lots of patches can slow your model down or even cause NetLogo to run out of memory.

- Type `ask patch 1 2 3 [ set pcolor red ]` into the Command Center.
- Use the mouse in the 3D view to rotate the world.

Notice the shape of the patch and its position in relation to the edges of the world. You'll also notice that you now need three coordinates to address patches in a 3D world.

Step 2: Turtle Movement

- Open the Models Library in the File menu. (If you are on a Mac and you don't have a File menu, click on the main NetLogo window first and it should reappear.)
- Open Turtle and Observer Motion Example 3D in 3D/Code Examples

Take a moment to look for the controls and monitors. In the bottom left you'll notice a group of monitors that describe the location and orientation of the turtle, though until you press the setup button they'll all say "N/A".

- Press the "setup" button

Heading, pitch, and roll are turtle variables that represent the orientation of the turtle. Heading is absolute in relation to the x/y plane; it is the rotation of the turtle around the z-axis.
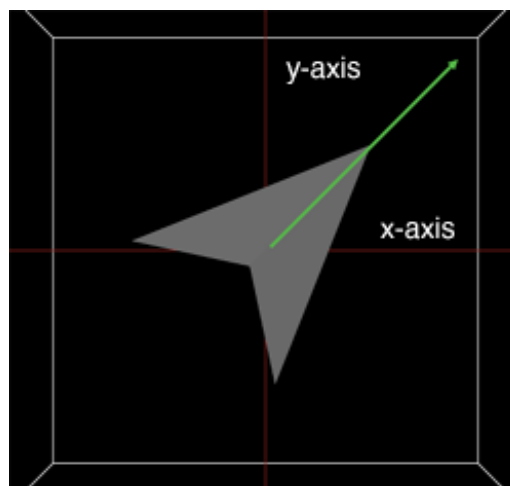
Pitch is the angle between the nose of the turtle and the xy-plane. It is relative to heading.
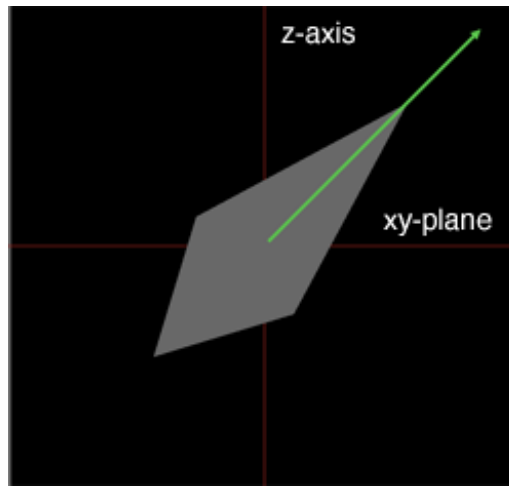
Roll is the rotation around the turtle's forward vector. It is relative to heading and pitch.
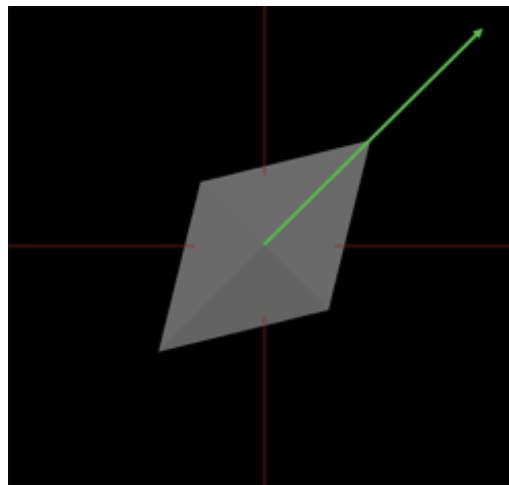
When turtles are created with `create-turtles` or `create-ordered-turtles`, their initial headings vary but their initial pitch and roll are always zero.

Take a look at the "Turtle Movement" buttons.

- Press the "left 1" button.

  How does the turtle move? Is is the same or different from 2D NetLogo? Which of the turtle variables change?

- Press the "pitch-down 1" button.

  How does the turtle move? Which of the turtle variables change?

- Press the "left 1" button again.

  How does the turtle move? Is it different than the last time you pressed the "left 1" button?

- Take a little time to play with the Turtle Movement buttons, watching both how the turtle moves and which of the turtle variables change.

You probably noticed that often more than one of the turtle variables may change for a single turn. For this reason we suggest that you use the turtle commands rather than

setting the orientation variables directly.

Step 3: Observer Movement

At the bottom of the interface you will see Orbit, Zoom, and Move buttons. If you have ever used the 3D view in NetLogo 2D or if you have been using the mouse controls in the 3D view through this tutorial you have been moving the observer. Changing the point of view in the 3D view is actually moving and changing the orientation of the observer. The observer has x, y and z coordinates, just like a turtle or patch, while turtles and patches are constrained to be inside the world the observer can be anywhere. Like a turtle the observer has a heading, pitch and roll, these variables control where the observer is looking, that is, what you see in the view.

- Move to the 3D view, and make sure "Orbit" is selected in the bottom left corner of the view.
- Click and hold the mouse button in the middle of the view, move the mouse left, right, up, and down.

   How does the position and orientation of the observer change?

- Press the reset-perspective button in the lower right corner of the view and select "Zoom" in the lower left corner.
- Click and hold the mouse button in the middle of the view and move the mouse up and down.

   Which of the observer variables change? Which stay the same?

- Try rotating the world a bit and then zoom again.
- Press the "Move" button in the lower left corner of the view.
- Click and hold the mouse button in the middle of the view and move the mouse up, down, left and right.

   How does the view change? How do the observer variables change?

After you are done exploring the world using the mouse controls you can take a look at the observer control buttons in the lower left portion of the interface.

You may already be familiar with the first three buttons in the observer group from your experience with NetLogo 2D. Watch, follow, and ride, are special modes that automatically update the position and orientation of the observer. When in follow or ride mode, the observer position and orientation are the same as the turtle's. Note that follow and ride are functionally exactly the same, the difference is only visual in the 3D view. When in watch mode the observer does not move but updates to face the target agent.

- Press the "setup" button again so you are back to the default orientation.
- Press the "orbit-right" button.

   How did the view change? Was it what you expected? How is it similar or different from using the mouse controls?

- Take a little time to experiment with orbit, roll and zoom buttons; notice

similarities and differences to the mouse controls.

The direction of the orbit commands refer to the direction that the observer moves. That is, imagine that the observer is on the surface of a sphere, the center of the sphere is the point that the observer is facing represented by the blue cross, by default (0,0,0). The observer will always face the center of the sphere and the radius of the sphere will remain constant. The directions, up, down, left, and right, refer to moving along the lines of latitude and the lines of longitude of the sphere. When you zoom the radius of the sphere changes but the center and the observer's orientation in relation to the center of the sphere will remain the same.

- Press one of the "setxyz" buttons.

  How does the view change? How do the observer variables change?

- Press the "facexyz" button.

  How does the view change? How do the observer variables change?

When you `setxyz` the center of the sphere remains the same (so the observer automatically keeps that point in the center of the view.) However, the radius of the sphere may change as well as the observer's orientation in relation to the center. When you `facexyz` or `face`, the center of the sphere changes but the observer does not move. The radius of the sphere may change, as well as the orientation of the observer.

Dictionary

Commands and Reporters

Turtle-related primitives

distancexyz distancexyz-nowrap dz left patch-at patch-at-heading-pitch-and-distance tilt-down tilt-up right roll-left roll-right setxyz towards-pitch towards-pitch-nowrap towards-pitch-xyz towards-pitch-xyz-nowrap turtles-at

Patch-related primitives

distancexyz distancexyz-nowrap neighbors neighbors6 patch patch-at patch-at-heading-pitch-and-distance

Agentset primitives

at-points breeds-at turtles-at

World primitives

[max-pzcor](#) [min-pzcor](#) [random-pzcor](#) [random-zcor](#) [world-depth](#) [load-shapes-3d](#)

Observer primitives

[face](#) [facexyz](#) [orbit-down](#) [orbit-left](#) [orbit-right](#) [orbit-up](#) [__oxcor](#) [__oycor](#) [__ozcor](#) [setxyz](#) [zoom](#)

Link primitives

[link-pitch](#)

Built-In Variables

Turtles

[zcor](#) [pitch](#) [roll](#)

Patches

[pzcor](#)

Primitives

at-points

agentset at-points [[*x1 y1 z1*] [*x2 y2 z2*] ...]

Reports a subset of the given agentset that includes only the agents on the patches the given distances away from this agent. The distances are specified as a list of three-item lists, where the three items are the x, y, and z offsets.

If the caller is the observer, then the points are measured relative to the origin, in other words, the points are taken as absolute patch coordinates.

If the caller is a turtle, the points are measured relative to the turtle's exact location, and not from the center of the patch under the turtle.

```
ask turtles at-points [[2 4 0] [1 2 1] [10 15 10]]
[ fd 1 ]  ;; only the turtles on the patches at the
          ;; distances (2,4,0), (1,2,1) and (10,15,10),
          ;; relative to the caller, move
```

distancexyz
distancexyz-nowrap

distancexyz *xcor ycor zcor*
distancexyz-nowrap *xcor ycor zcor*

3D versions of [distancexy](#).

Reports the distance from this agent to the point (*xcor*, *ycor*, *zcor*).

The distance from a patch is measured from the center of the patch.

distancexyz-nowrap always reports the in world distance, never a distance that would require wrapping around the edges of the world. With distancexyz the wrapped distance (around the edges of the world) is used if that distance is shorter than the in world distance.

```
if (distancexyz 0 0 0) < 10
  [ set color green ]
;; all turtles less than 10 units from
;; the center of the screen turn green.
```

dz

dz

Reports the z-increment (the amount by which the turtle's zcor would change) if the turtle were to take one step forward at its current heading and pitch.

NOTE: dz is simply the sine of the turtle's pitch. Both dx and dy have changed in this case. So, dx = cos(pitch) * sin(heading) and dy = cos(pitch) * cos(heading).

See also [dx](#), [dy](#).

face
facexyz

face *agent*
facexyz *x y z*

Set the caller's heading and pitch towards *agent* or towards the point *(x,y,z)*.

If the caller and the target are at the same x and y coordinates the caller's heading will not change. If the caller and the target are also at the same z coordinate the pitch will not change either.

left

left *number*

The turtle turns left by *number* degrees, relative to its current orientation. While left in a 2D world only modifies the turtle's heading, left in a 3D world may also modify the turtle's pitch and roll.

See also [left](#), [tilt-up](#), [tilt-down](#)

link-pitch

link-pitch

Reports the pitch from end1 to end2 of this link.

```
ask link 0 1 [ print link-pitch ]
;; prints [[towards-pitch other-end] of end1] of link 0 1
```

See also [link-heading](#), [pitch](#)

load-shapes-3d

load-shapes-3d *filename*

Loads custom 3D shapes from the given file. See the [3D guide](#) for more details. You must also add a 2D shape of the same name to the model using the Turtle Shapes Editor. Custom shapes override built-in 3D shapes and converted 2D shapes.

max-pzcor
min-pzcor

max-pzcor
min-pzcor

These reporters give the maximum and minimum z-coordinates (respectively) for patches, which determines the size of the world.

Unlike in older versions of NetLogo the origin does not have to be at the center of the world. However, the minimum z-coordinate has to be less than or equal to 0 and the maximum z-coordinate has to be greater than or equal to 0.

Note: You can set the size of the world only by editing the view -- these are reporters which cannot be set.

See also [max-pxcor](#), [max-pycor](#), [min-pxcor](#), [min-pycor](#), and [world-depth](#).

neighbors
neighbors6

neighbors
neighbors6

3D versions of [neighbors](#) and [neighbors4](#).

Reports an agentset containing the 26 surrounding patches (neighbors) or 6 surrounding patches (neighbors6).

```
show sum values-from neighbors [count turtles-here]
   ;; prints the total number of turtles on the twenty-six
   ;; patches around this turtle or patch
ask neighbors6 [ set pcolor red ]
   ;; turns the six neighboring patches red
```

orbit-down
orbit-left
orbit-right
orbit-up

orbit-down *number*
orbit-left *number*
orbit-right *number*
orbit-up *number*

Rotate the observer around the last point faced. Imagine the observer is on the surface of a sphere, the last point face is the center of that sphere. Up and down orbit along the lines of longitude and right and left orbit along the lines of latitude. The observer will remain facing the last point faced so the heading and pitch may change as result of orbiting. However, because we assume an absolute north pole (parallel to the positive z-axis) the roll will never change.

See also [setxyz](#), [face](#) and [zoom](#)

__oxcor
__oycor
__ozcor

__oxcor
__oycor
__ozcor

Reports the x-, y-, or z-coordinate of the observer.

See also [setxyz](#)

patch

patch *pxcor pycor pzcor*

3D version of [patch](#).

Given three integers, reports the single patch with the given pxcor, pycor and pzcor. *pxcor*, *pycor* and *pzcor* must be integers.

```
ask (patch 3 -4 2) [ set pcolor green ]
;; patch with pxcor of 3 and pycor of -4 and pzcor of 2 turns green
```

See also patch

patch-at

patch-at *dx dy dz*

3D version of patch-at.

Reports the single patch at (dx, dy, dz) from the caller, that is, dx patches east, dy patches north and dz patches up from the caller.

```
ask patch-at 1 -1 1 [ set pcolor green ]
;; turns the patch just southeast and up from the caller green
```

patch-at-heading-pitch-and-distance

patch-at-heading-pitch-and-distance *heading pitch distance*

3D version of patch-at-heading-and-distance.

patch-at-heading-pitch-and-distance reports the single patch that is the given distance from this turtle or patch, along the given absolute heading and pitch. (In contrast to patch-left-and-ahead and patch-right-and-ahead, this turtle's current heading is not taken into account.)

```
ask patch-at-heading-pitch-and-distance 0 90 1 [ set pcolor green ]
;; turns the patch directly above the caller green.
```

pitch

pitch

This is a built-in turtle variable. Pitch is the angle between the "nose" of the turtle and the xy-plane. Heading and pitch together define the forward vector of the turtle or the direction that the turtle is facing.

This is a number greater than or equal to 0 and less than 360. 0 is parallel to the xy-plane, 90 is parallel to the z-axis. While you can set pitch we recommend that you use the primitives to turn the turtle. Depending on the position more than one relative angle (heading, pitch and roll) may change at once.

Example:

```
;; assume roll and heading are 0
set pitch 45        ;; turtle is now north and up
set heading heading + 10 ;; same effect as "tilt-up 10"
```

See also [heading](#), [roll](#), [tilt-up](#), [tilt-down](#), [right](#), [left](#)

pzcor

pzcor

This is a built-in patch variable. It holds the z coordinate of the patch. It is always an integer. You cannot set this variable, because patches don't move.

pzcor is greater than or equal to min-pzcor and less than or equal to max-pzcor.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [pxcor, pycor](#), [zcor](#).

random-pzcor

random-pzcor

Reports a random integer ranging from min-pzcor to max-pxcor inclusive.

```
ask turtles [
  ;; move each turtle to the center of a random patch
  setxyz random-pxcor random-pycor random-pzcor
]
```

See also [random-pxcor](#), [random-pycor](#).

random-zcor

random-zcor

Reports a random floating point number from the allowable range of turtle coordinates along the z axis.

Turtle coordinates range from min-pzcor - 0.5 (inclusive) to max-pzcor + 0.5 (exclusive).

```
ask turtles [
  ;; move each turtle to a random point
  setxyz random-xcor random-ycor random-zcor
]
```

See also [random-xcor](#), [random-ycor](#).

right

right *number*

The turtle turns right by *number* degrees, relative to its current orientation. While right in a 2D world only modifies the turtle's heading, right in a 3D world may also modify the turtle's pitch and roll.

See also [right](#) and [left](#)

roll

roll

This is a built-in turtle variable. Roll is the angle between the "wing-tip" of the turtle and the xy-plane.

This is a number greater than or equal to 0 and less than 360. You can set this variable to make a turtle roll. Since roll is always from the turtle's point of view, rolling right and left only only change roll regardless of turtle orientation.

Example:

```
set roll 45       ;; turtle rotated right
set roll roll + 10 ;; same effect as "roll-right 10"
```

See also [heading](#), [pitch](#), [roll-left](#), [roll-right](#).

roll-left

roll-left *number*

The wingtip of the turtle rotates to the left *number* degrees with respect to the current heading and pitch.

roll-right

roll-right *number*

The wingtip of the turtle rotates to the right *number* degrees with respect to the current heading and pitch.

setxyz

setxyz *x y z*

3D version of [setxy](#).

The agent, a turtle or the observer, sets its x-coordinate to *x*, its y-coordinate to *y* and its z-coordinate to *z*. When the observer uses `setxyz` it remains facing the same point so the heading, pitch, and roll, may also change.

For turtles equivalent to `set xcor x set ycor y set zcor z`, except it happens in one time step instead of three.

```
setxyz 0 0 0
;; agent moves to the middle of the center patch
```

See also [face](#)

tilt-down
tilt-up

tilt-down *number*
tilt-up *number*

The nose of the turtle rotates by *number* degrees, relative to its current orientation. Depending on the orientation of the turtle more than one of the relative angles (heading, pitch, and roll) may change when a turtle turns.

towards-pitch
towards-pitch-nowrap

towards-pitch *agent*
towards-pitch-nowrap *agent*

Reports the pitch from this agent to the given agent.

If the wrapped distance (around the edges of the screen) is shorter than the on-screen distance, towards-pitch will report the pitch of the wrapped path. towards-pitch-nowrap never uses the wrapped path.

Note: In order to get one turtle to face another you need to use both towards-pitch and towards.

Note: asking for the pitch from an agent to itself, or an agent on the same location, will cause a runtime error.

See also [towards](#)

towards-pitch-xyz

towards-pitch-xyz-nowrap

towards-pitch-xyz *x y z*
towards-pitch-xyz-no-wrap *x y z*

Reports the pitch from this agent to the coordinates x, y, z

If the wrapped distance (around the edges of the screen) is shorter than the on-screen distance, towards-pitch will report the pitch of the wrapped path. towards-pitch-nowrap never uses the wrapped path.

Note: In order to get a turtle to face a given location you need to use both towards-pitch-xyz and towardsxy.

Note: asking for the pitch from an agent to the location it is standing on will cause a runtime error.

See also towardsxy

turtles-at
<breeds>-at

turtles-at *dx dy dz*
<breeds>-at *dx dy dz*

3D versions of turtles-at and breeds-at.

Reports an agentset containing the turtles on the patch (dx, dy, dz) from the caller (including the caller itself if it's a turtle).

```
;; suppose I have 40 turtles at the origin
show [count turtles-at 0 0 0] of turtle 0
=> 40
```

world-depth

world-depth

Reports the total depth of the NetLogo world.

The depth of the world is the same as max-pzcor - min-pzcor + 1.

See also max-pzcor, min-pzcor, world-width, and world-height

zcor

zcor

This is a built-in turtle variable. It holds the current z coordinate of the turtle. This is a floating point number, not an integer. You can set this variable to change the turtle's location.

This variable is always greater than or equal to (- screen-edge-z) and strictly less than screen-edge-z.

See also setxy, xcor, ycor, pxcor, pycor, pzcor

zoom

zoom *number*

Move the observer toward the point it is facing, *number* steps. The observer will never move beyond the point it is facing so if *number* is greater than the distance to that point it will only move as far as the point it is facing.

Extensions Guide

NetLogo allows users to write new commands and reporters in Java and other languages and use them in their models. This section of the User Manual introduces this facility and shows how to use an extension in your model once you have obtained or made one.

Extensions created by members of the NetLogo community are available from https://github.com/NetLogo/NetLogo/wiki/Extensions.

For information on creating your own extensions, go here.

Using Extensions

To use an extension in a model, add the `extensions` keyword at the beginning of the Code tab, before declaring any breeds or variables.

After `extensions` comes a list of extension names in square brackets. For example:

```
extensions [sound speech]
```

Using `extensions` tells NetLogo to find and open the specified extension and makes the custom commands and reporters found in the extension available to the current model. You can use these commands and reporters just as if they were built-in NetLogo primitives.

Where extensions are located

NetLogo will look for extensions in several places:

1. In the folder of the current model.
2. The `extensions` folder in the same location as the NetLogo application.

Each NetLogo extension consists of a folder with the same name as the extension, entirely in lower case. This folder must contain a JAR file with the same name as the folder. For example the `sound` extension is stored in a folder called `sound` with a file inside called `sound.jar`.

To install a NetLogo extension for use by any model, put the extension's folder in the `extensions` directory in the NetLogo directory. Or, you can just keep the extension's folder in the same folder as the model that uses it.

Some extensions depend on additional files. These files will be in the extension's folder along with the JAR file. The folder may also contain other files such as documentation and example models.

Array and Table Extensions

These extensions add two new data structures to NetLogo, arrays and hash tables.

When to use

In general, anything you can do with an array or table, you could also just use a list for. But you may want to consider using an array or table instead for speed reasons. All three data structures (list, array, and table) have different performance characteristics, so you may be able to make your model run faster by selecting the appropriate data structure.

Arrays are useful when you need a collection of values whose size is fixed. You can quickly access or alter any item in an array if you know its position.

Tables are useful when you need to do associate values with other values. For example, you might make a table of words and their definitions. Then you can look up the definition of any word. Here, the words are the "keys". You can easily retrieve the value for any key in the table, but not vice versa.

Unlike NetLogo's lists and strings, arrays and tables are "mutable". That means that you can actually modify them directly, rather than constructing an altered copy as with lists. If the array or table is used in more than one place in your code, any changes you make will show up everywhere. It's tricky to write code involving mutable structures and it's easy to make subtle errors or get surprising results, so we suggest sticking with lists and strings unless you're certain you want and need mutability.

Lists and arrays in NetLogo 5.0

In NetLogo 4.1 and earlier, common list operations such as `last`, `lput`, `item`, and `replace-item` took linear time (proportional to the size of the list). As a result, some users chose to use arrays instead of lists in order to get good performance. But in the current NetLogo, these operations now run in nearly constant time. So arrays are now less often needed.

How to use

Both extensions come preinstalled.

To use the array extension in your model, add a line to the top of your Code tab:

```
extensions [array]
```

To use the table extension in your model, add a line to the top of your Code tab:

```
extensions [table]
```

You can use both extensions in the same model if you want, as follows:

```
extensions [array table]
```

If your model already uses other extensions, then it already has an extensions line in it, so just add `array` and/or `table` to the list.

For more information on using NetLogo extensions, see the [Extensions Guide](#).

Limitation on table keys

Table keys may only be strings, numbers, booleans, or lists. (Lists may be arbitrarily nested lists as long as all the items inside are strings, numbers, or booleans.)

Array example

```
let a array:from-list n-values 5 [0]
print a
=>
print array:length a
=> 5
foreach n-values 5 [?] [ array:set a ? ? * ? ]
print a
=>
print array:item a 0
=> 0
print array:item a 3
=> 9
array:set a 3 50
print a
=>
```

Table example

```
let dict table:make
table:put dict "turtle" "cute"
table:put dict "bunny" "cutest"
print dict
=>
print table:length dict
=> 2
print table:get dict "turtle"
=> "cute"
print table:get dict "leopard"
=> (error)
print table:keys dict
=> ["turtle" "bunny"]
```

> **Code Example:** Table Example

Array primitives

[array:from-list](#) [array:item](#) [array:set](#) [array:length](#) [array:to-list](#)

array:from-list

array:from-list *list*

Reports a new array containing the same items in the same order as the input list.

array:item

array:item *array index*

Reports the item in the given array with the given index (ranging from zero to the length of the array minus one).

array:set

array:set *array index value*

Sets the item in the given array with the given index (ranging from zero to the length of the array minus one) to the given value.

Note that unlike the `replace-item` primitive for lists, a new array is not created. The given array is actually modified.

array:length

array:length *array*

Reports the length of the given array, that is, the number of items in the array.

array:to-list

array:to-list *array*

Reports a new list containing the same items in the same order as the given array.

Table Primitives

table:clear table:from-list table:get table:has-key? table:keys table:length table:make table:put table:remove table:to-list

table:clear

table:clear *table*

Removes all key-value pairs from *table*.

table:from-list

table:from-list *list*

Reports a new table with the contents of *list*. *list* must be a list of two element lists, or pairs. The first element in the pair is the key and the second element is the value.

table:get

table:get *table key*

Reports the value that *key* is mapped to in the table. Causes an error if there is no entry for the key.

table:has-key?

table:has-key? *table key*

Reports true if *key* has an entry in *table*.

table:keys

table:keys *table*

Reports a list of all the keys in *table*, in the same order the keys were inserted.

table:length

table:length *table*

Reports the number of entries in *table*.

table:make

table:make

Reports a new, empty table.

table:put

table:put *table key value*

Maps *key* to *value* in *table*. If an entry already exists in the table for the given key, it is replaced.

table:remove

table:remove *table key*

Removes the mapping in *table* for *key*.

table:to-list

table:to-list *table*

Reports a list with the content of *table*. The list will be a list of two element lists, or pairs. The first element in the pair is the key and the second element is the value. The keys appear in the same order they were inserted.

Matrix Extension

The matrix extension adds a new matrix data structure to NetLogo. A matrix is a mutable 2-dimensional array containing only numbers.

When to use

Although matrices store numbers, much like a list of lists, or an array of arrays, the primary reason to use the matrix data type is to take advantage of special mathematical operations associated with matrices. For instance, matrix multiplication is a convenient way to perform geometric transformations, and the repeated application of matrix multiplication can also be used to simulate other dynamic processes (for instance, processes on graph/network structures).

If you'd like to know more about matrices and how they can be used, you might consider a course on linear algebra, or search the web for tutorials. The matrix extension also allows you to solve linear algebraic equations (specified in a matrix format), and even to identify trends in your data and perform linear (ordinary least squares) regressions on data sets with multiple explanatory variables.

How to use

The matrix extension comes preinstalled.

To use the matrix extension in your model, add a line to the top of your Code tab:

```
extensions [matrix]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `matrix` to the list.

For more information on using NetLogo extensions, see the Extensions Guide.

Matrix example

```
let m matrix:from-row-list [[1 2 3] [4 5 6]]
print m
=>
print matrix:pretty-print-text m
=>
[[ 1  2  3 ]
 [ 4  5  6 ]]

print matrix:dimensions m
=> [2 3]
;;(NOTE: row & column indexing starts at 0, not 1)
print matrix:get m 1 2 ;; what number is in row 1, column 2?
=> 6
matrix:set m 1 2 10 ;; change the 6 to a 10
print m
=>

let m2 matrix:make-identity 3
print m2
=>
print matrix:times m m2 ;; multiplying by the identity changes nothing
=>

;; make a new matrix with the middle 1 changed to -1
let m3 (matrix:set-and-report m2 1 1 -1)
print m3
=>
print matrix:times m m3
=>

print matrix:to-row-list (matrix:plus m2 m3)
=> [[2 0 0] [0 0 0] [0 0 2]]
```

Code Example: Matrix Example

Credits:

The Matrix extension for NetLogo was originally written by Forrest Stonedahl, with significant contributions from Charles Staelin (in particular, the forecast & regression primitives). The matrix extension provides a wrapper around Jama, which is a free/open-source matrix library for Java.

The source code for the Matrix extension is hosted on GitHub You are more than welcome to modify/improve upon the existing functionality and features and we encourage you to contribute your changes back to the NetLogo community.

Matrix Primitives

Matrix creation & conversion to/from lists

matrix:make-constant matrix:make-identity matrix:from-row-list matrix:from-column-list matrix:to-row-list matrix:to-column-list matrix:copy matrix:pretty-print-text

Matrix data retrieval and manipulation

matrix:get matrix:get-row matrix:get-column matrix:set matrix:set-row matrix:set-column matrix:swap-rows matrix:swap-columns matrix:set-and-report matrix:dimensions matrix:submatrix matrix:map

Math operations

matrix:times-scalar matrix:times matrix:* matrix:times-element-wise matrix:plus-scalar matrix:plus matrix:+ matrix:minus matrix:- matrix:inverse matrix:transpose matrix:real-eigenvalues matrix:imaginary-eigenvalues matrix:eigenvectors matrix:det matrix:rank matrix:cond matrix:trace

Advanced features

matrix:solve matrix:forecast-linear-growth matrix:forecast-compound-growth matrix:forecast-continuous-growth matrix:regress

Matrix creation & conversion to/from lists

matrix:make-constant

matrix:make-constant *n-rows n-cols number*

Reports a new n-rows by n-cols matrix object, with all entries in the matrix containing the same value (number).

matrix:make-identity

matrix:make-identity *n-size*

Reports a new square matrix object (with dimensions n-size x n-size), consisting of the identity matrix (1s along the main diagonal, 0s elsewhere).

matrix:from-row-list

matrix:from-row-list *nested-list*

Reports a new matrix object, created from a NetLogo list, where each item in that list is another list (corresponding to each of the rows of the matrix.)

```
print matrix:from-row-list [[1 2] [3 4]]
=>
;; Corresponds to this matrix:
;; 1 2
;; 3 4
```

matrix:from-column-list

matrix:from-column-list *nested-list*

Reports a new matrix object, created from a NetLogo list containing each of the *columns* of the matrix.

matrix:to-row-list

matrix:to-row-list *matrix*

Reports a list of lists, containing each row of the matrix.

matrix:to-column-list

matrix:to-column-list *matrix*

Reports a list of lists, containing each column of the matrix.

matrix:copy

matrix:copy matrix

Reports a new matrix that is an exact copy of the given matrix. This primitive is important because the matrix type is mutable (changeable). Here's a code example:

```
let m1 matrix:from-column-list [[1 4 7][2 5 8][3 6 9]] ; a 3x3 matrix
print m1
=>
let m2 m1 ;; m2 refers to the same matrix object as m1
let m3 matrix:copy m1 ;; m3 is a new copy containing m1's data
matrix:set m1 0 0 100 ;; now m1 is changed

print m1
=>

print m2
=>
;;Notice that m2 was also changed, when m1 was changed!

print m3
=>
```

matrix:pretty-print-text

matrix:pretty-print-text *matrix*

Reports a string that is a textual representation of the matrix, in a format that is reasonably human-readable when displayed.

Matrix modification and math operations

matrix:get

matrix:get *matrix row-i col-j*

Reports the (numeric) value at location *row-i,col-j* in the given *matrix*.

matrix:get-row

matrix:get-row *matrix row-i*

Reports a simple (not nested) NetLogo list containing the elements of *row-i* of the given *matrix*.

matrix:get-column

matrix:get-column *matrix col-j*

Reports a simple (not nested) NetLogo list containing the elements of *col-j* of the given *matrix*.

matrix:set

matrix:set *matrix row-i col-j new-value*

Changes the given *matrix*, by setting the location *row-i,col-j* to *new-value*

matrix:set-row

matrix:set-row *matrix row-i simple-list*

Changes the given matrix *matrix* by replacing the row at *row-i* with the contents of the simple (not nested) NetLogo list *simple-list*. The *simple-list* must have a length equal to the number of columns in the matrix, i.e., the matrix row length.

matrix:set-column

matrix:set-column *matrix col-j simple-list*

Changes the given matrix *matrix* by replacing the column at *col-j* with the contents of the simple (not nested) NetLogo list *simple-list*. The *simple-list* must have a length equal to the number of rows in the matrix, i.e., the matrix column length.

matrix:swap-rows

matrix:swap-rows *matrix row1 row2*

Changes the given matrix *matrix* by swapping the rows at *row1* and *row2* with each other.

matrix:swap-columns

matrix:swap-columns *matrix col1 col2*

Changes the given matrix *matrix* by swapping the columns at *col1* and *col2* with each other.

matrix:set-and-report

matrix:set-and-report *matrix row-i col-j new-value*

Reports a new matrix, which is a copy of the given matrix except that the value at *row-i,col-j* has been changed to *new-value*. A NetLogo statement such as "set mat matrix:set-and-report mat 2 3 10" will result in mat pointing to this new matrix, a copy of the old version of mat with the element at row 2, column 3 being set to 10. The old version of mat will be "lost".

matrix:dimensions

matrix:dimensions *matrix*

Reports a 2-element list ([num-rows,num-cols]), containing the number of rows and number of columns in the given *matrix*

matrix:submatrix

matrix:submatrix *matrix r1 c1 r2 c2*

Reports a new matrix object, consisting of a rectangular subsection of the given matrix. The rectangular region is from row *r1* up to (but not including) row *r2*, and from column *c1* up to (but not including) column *c2*. Here is an example:

```
let m matrix:from-row-list [[1 2 3][4 5 6][7 8 9]]
print matrix:submatrix m 0 1 2 3 ; matrix, row-start, col-start, row-end, col-end
                                 ; rows from 0 (inclusive) to 2 (exclusive),
                                 ; columns from 1 (inclusive) to 3 (exclusive)
=>
```

matrix:map

matrix:map *reporter-task matrix*
(matrix:map *reporter-task m1 m2 ...*)

Reports a new matrix which results from applying *reporter-task* to each of the elements of the given matrix. For example,

```
matrix:map sqrt matrix
```

would take the square root of each element of *matrix*. If more than one matrix argument is provided, the task is given the elements of each matrix as arguments. Thus,

```
(matrix:map + matrix1 matrix2)
```

would add *matrix1* and *matrix2*.

This reporter is meant to be the same as *map*, but for matrices instead of lists.

matrix:times-scalar

matrix:times-scalar *matrix factor*

As of NetLogo 5.1, *matrix:times* can multiply matrices by scalars making this function obsolete. Use *matrix:times* instead.

Reports a new matrix, which is the result of multiplying every entry in the original *matrix* by the given scaling *factor*.

matrix:times

matrix:times *m1 m2...*
(matrix:times *m1 m2 ...***)**

Reports a matrix, which is the result of multiplying the given matrices and scalars (using standard matrix multiplication -- make sure your matrix dimensions match up.) Without parentheses, it takes two arguments. With parentheses it takes two or more. The arguments may either be numbers or matrices, but at least one must be a matrix.

matrix:*

m1 **matrix:*** *m2*

Reports a matrix, which is the result of multiplying the given matrices and/or scalars (using standard matrix multiplication -- make sure your matrix dimensions match up.) This is exactly the same as

```
matrix:times m1 m2
```

Takes precedence over *matrix:+* and *matrix:-*, same as normal multiplication.

matrix:times-element-wise

matrix:times-element-wise *m1 m2*
(matrix:times-element-wise *m1 m2 ...***)**

Reports a matrix, which is the result of multiplying the given matrices together, element-wise. All elements are multiplied by scalar arguments as well. Note that all matrix arguments must have the same dimensions. Without parentheses, it takes two arguments. With parentheses it takes two or more. The arguments may either be numbers or matrices, but at least one must be a matrix.

matrix:plus-scalar

matrix:plus-scalar *matrix number*

As of NetLogo 5.1, *matrix:plus* can add matrices and scalars making this function obsolete. Use *matrix:plus* instead.

Reports a matrix, which is the result of adding the constant *number* to each element of the given *matrix*.

matrix:plus

matrix:plus *m1 m2*
(matrix:plus *m1 m2 ...***)**

Reports a matrix, which is the result of adding the given matrices and scalars. Scalars are added to each element. Without parentheses, it takes two arguments. With parentheses it takes two or more. The arguments may either be numbers or matrices, but at least one must be a matrix.

matrix:+

m1 **matrix:+** *m2*

Reports a matrix, which is the result of adding the given matrices and/or scalars. This is exactly the same as

```
matrix:plus m1 m2
```

Takes precedence after *matrix:**, same as normal addition.

matrix:minus

matrix:minus *m1 m2*
(matrix:minus *m1 m2 ...*)

Reports a matrix, which is the result of subtracting all arguments besides *m1* from *m1*. Scalar arguments are treated as matrices of the same size as the matrix arguments with every element equal to that scalar. Without parentheses, it takes two arguments. With parentheses it takes two or more. The arguments may either be numbers or matrices, but at least one must be a matrix.

matrix:-

m1 **matrix:-** *m2*

Reports a matrix, which is the result of subtracting the given matrices and/or scalars. This is exactly the same as

```
matrix:minus m1 m2
```

Takes precedence after *matrix:**, same as normal subtraction.

matrix:inverse

matrix:inverse *matrix*

Reports the inverse of the given *matrix*, or results in an error if the matrix is not invertible.

matrix:transpose

matrix:transpose *matrix*

Reports the transpose of the given *matrix*.

matrix:real-eigenvalues

matrix:real-eigenvalues *matrix*

Reports a list containing the real eigenvalues of the given *matrix*.

matrix:imaginary-eigenvalues

matrix:imaginary-eigenvalues *matrix*

Reports a list containing the imaginary eigenvalues of the given *matrix*.

matrix:eigenvectors

matrix:eigenvectors *matrix*

Reports a matrix that contains the eigenvectors of the given *matrix*. (Each eigenvector as a column of the resulting matrix.)

matrix:det

matrix:det *matrix*

Reports the determinant of the *matrix*.

matrix:rank

matrix:rank *matrix*

Reports the effective numerical rank of the *matrix*,obtained from SVD (Singular Value Decomposition).

matrix:cond

matrix:cond *matrix*

Reports the matrix condition (2 norm), which is the ratio of largest to smallest singular value (obtained from SVD).

matrix:trace

matrix:trace *matrix*

Reports the trace of the *matrix*, which is simply the sum of the main diagonal elements.

Advanced features

matrix:solve

matrix:solve *A C*

Reports the solution to a linear system of equations, specified by the *A* and *C* matrices. In general, solving a set of linear equations is akin to matrix division. That is, the goal is to find a matrix B such that $A * B = C$. (For simple linear systems, *C* and B can both be 1-dimensional matrices -- i.e. vectors). If A is not a square matrix, then a "least squares" solution is returned.

```
;; To solve the set of equations x + 3y = 10 and 7x - 4y = 20
;; We make our A matrix [[1 3][7 -4]], and our C matrix [[10][20]]
let A matrix:from-row-list [[1 3][7 -4]]
let C matrix:from-row-list [[10][20]]
print matrix:solve A C
=>
;; NOTE: as you can see, the results may be only approximate
;; (In this case, the true solution should be x=4 and y=2.)
```

matrix:forecast-linear-growth

matrix:forecast-linear-growth *data-list*

Reports a four-element list of the form: [*forecast constant slope R^2*]. The *forecast* is the predicted next value that would follow in the sequence given by the *data-list* input, based on a linear trend-line. Normally *data-list* will contain observations on some variable, Y, from time t = 0 to time t = (n-1) where n is the number of observations. The *forecast* is the predicted value of Y at t = n. The *constant* and *slope* are the parameters of the trend-line

Y = *constant* + *slope* * t.

The R^2 value measures the goodness of fit of the trend-line to the data, with an R^2 = 1 being a perfect fit and an R^2 of 0 indicating no discernible trend. Linear growth assumes that the variable Y grows by a constant absolute amount each period.

```
;; a linear extrapolation of the next item in the list.
print matrix:forecast-linear-growth [20 25 28 32 35 39]
=> [42.733333333333334 20.619047619047638 3.6857142857142824 0.9953743395474031]
;; These results tell us:
;; * the next predicted value is roughly 42.7333
;; * the linear trend line is given by Y = 20.6190 + 3.6857 * t
;; * Y grows by approximately 3.6857 units each period
;; * the R^2 value is roughly 0.9954 (a good fit)
```

matrix:forecast-compound-growth

matrix:forecast-compound-growth *data-list*

Reports a four-element list of the form: [*forecast constant growth-proportion R^2*]. Whereas matrix:forecast-linear-growth assumes growth by a constant absolute amount each period, matrix:forecast-compound-growth assumes that Y grows by a constant **proportion** each period. The *constant* and *growth-proportion* are the parameters of the trend-line

Y = *constant* * *growth-proportion*^{t}.

Note that the growth proportion is typically interpreted as *growth-proportion = (1.0 + growth-rate)*. Therefore, if matrix:forecast-compound-growth returns a *growth-proportion* of 1.10, that implies that Y grows by (1.10 - 1.0) = 10% each period. Note that if growth is negative, matrix:forecast-compound-growth will return a *growth-proportion* of less than one. E.g., a *growth-proportion* of 0.90 implies a growth rate of -10%.

NOTE: The compound growth forecast is achieved by taking the ln of Y. (See matrix:regress, below.) Because it is impossible to take the natural log of zero or a negative number, matrix:forecast-compound-growth will result in an error if it finds a zero or negative number in *data-list*.

```
;; a compound growth extrapolation of the next item in the list.
print matrix:forecast-compound-growth [20 25 28 32 35 39]
=> [45.60964465307147 21.15254147944863 1.136621034423892 0.9760867518334806]
;; These results tell us:
;; * the next predicted value is approximately 45.610
;; * the compound growth trend line is given by Y = 21.1525 * 1.1366 ^ t
;; * Y grows by approximately 13.66% each period
;; * the R^2 value is roughly 0.9761 (a good fit)
```

matrix:forecast-continuous-growth

matrix:forecast-continuous-growth *data-list*

Reports a four-element list of the form: [*forecast constant growth-rate R^2*]. Whereas matrix:forecast-compound-growth assumes discrete time with Y growing by a given proportion each finite period of time (e.g., a month or a year), matrix:forecast-continuous-growth assumes that Y is compounded **continuously** (e.g., each second or fraction of a second). The *constant* and *growth-rate* are the parameters of the trend-line

Y = *constant* * $e^{(growth\text{-}rate * t)}$.

matrix:forecast-continuous-growth is the "calculus" analog of matrix:forecast-compound-growth. The two will normally yield similar (but not identical) results, as shown in the example below. *growth-rate* may, of course, be negative.

NOTE: The continuous growth forecast is achieved by taking the ln of Y. (See matrix:regress, below.) Because it is impossible to take the natural log of zero or a negative number, matrix:forecast-continuous-growth will result in an error if it finds a zero or negative number in *data-list*.

```
;; a continuous growth extrapolation of the next item in the list.
print matrix:forecast-continuous-growth [20 25 28 32 35 39]
=> [45.60964465307146 21.15254147944863 0.12805985615332668 0.9760867518334806]
;; These results tell us:
;; * the next predicted value is approximately 45.610
;; * the compound growth trend line is given by Y = 21.1525 * e ^ (0.1281 * t)
;; * Y grows by approximately 12.81% each period if compounding takes place continuously
;; * the R^2 value is roughly 0.9761 (a good fit)
```

matrix:regress

matrix:regress *data-matrix*

All three of the forecast primitives above are just special cases of performing an OLS (ordinary-least-squares) linear regression -- the matrix:regress primitive provides a flexible/general-purpose approach. The input is a matrix *data-matrix*, with the first column being the observations on the dependent variable and each subsequent column being the observations on the (1 or more) independent variables. Thus each row consists of an observation of the dependent variable followed by the corresponding observations for each independent variable.

The output is a Logo nested list composed of two elements. The first element is a list containing the regression constant followed by the coefficients on each of the independent variables. The second element is a 3-element list containing the R^2 statistic, the total sum of squares, and the residual sum of squares. The following code example shows how the matrix:regress primitive can be used to perform the same function as the code examples shown in the matrix:forecast-*-growth primitives above. (However, keep in mind that the matrix:regress primitive is more powerful than this, and can have many more independent variables in the regression, as indicated in the fourth example below.)

```
;; this is equivalent to what the matrix:forecast-linear-growth does
let data-list [20 25 28 32 35 39]
let indep-var (n-values length data-list [?]) ; 0,1,2...,5
let lin-output matrix:regress matrix:from-column-list (list data-list indep-var)
let lincnst item 0 (item 0 lin-output)
let linslpe item 1 (item 0 lin-output)
let linR2   item 0 (item 1 lin-output)
;;Note the "6" here is because we want to forecast the value at time t=6.
print (list (lincnst + linslpe * 6) (lincnst) (linslpe) (linR2))

;; this is equivalent to what the matrix:forecast-compound-growth does
let com-log-data-list  (map [ln ?] [20 25 28 32 35 39])
let com-indep-var2 (n-values length com-log-data-list [?]) ; 0,1,2...,5
let com-output matrix:regress matrix:from-column-list (list com-log-data-list com-indep-var2)
let comcnst exp item 0 (item 0 com-output)
let comprop exp item 1 (item 0 com-output)
let comR2       item 0 (item 1 com-output)
;;Note the "6" here is because we want to forecast the value at time t=6.
print (list (comcnst * comprop ^ 6) (comcnst) (comprop) (comR2))

;; this is equivalent to what the matrix:forecast-continuous-growth does
let con-log-data-list  (map [ln ?] [20 25 28 32 35 39])
let con-indep-var2 (n-values length con-log-data-list [?]) ; 0,1,2...,5
let con-output matrix:regress matrix:from-column-list (list con-log-data-list con-indep-var2)
let concnst exp item 0 (item 0 con-output)
let conrate     item 1 (item 0 con-output)
let conR2       item 0 (item 1 con-output)
print (list (concnst * exp (conrate * 6)) (concnst) (conrate) (conR2))
```

```
;; example of a regression with two independent variables:
;; Pretend we have a dataset, and we want to know how well happiness
;; is correlated to snack-food consumption and accomplishing goals.
let happiness [2 4 5 8 10]
let snack-food-consumed [3 4 3 7 8]
let goals-accomplished [2 3 5 8 9]
print matrix:regress matrix:from-column-list (list happiness snack-food-consumed goals-accomplished)
=> [[-0.14606741573033788 0.3033707865168543 0.8202247191011234] [0.9801718440185063 40.8 0.8089887640449439]
;; linear regression: happiness = -0.146 + 0.303*snack-food-consumed + 0.820*goals-accomplished
;; (Since the 0.820 coefficient is higher than the 0.303 coefficient, it appears that each goal
;; accomplished yields more happiness than does each snack consumed, although both are positively
;; correlated with happiness.)
;; Also, we see that R^2 = 0.98, so the two factors together provide a good fit.
```

```
;; example of a regression with two independent variables:
;; Pretend we have a dataset, and we want to know how well happiness
;; is correlated to snack-food consumption and accomplishing goals.
let happiness [2 4 5 8 10]
let snack-food-consumed [3 4 3 7 8]
let goals-accomplished [2 3 5 8 9]
print matrix:regress matrix:from-column-list (list happiness snack-food-consumed goals-accomplished)
=> [[-0.14606741573033788 0.3033707865168543 0.8202247191011234] [0.9801718440185063 40.8 0.8089887640449439]
;; linear regression: happiness = -0.146 + 0.303*snack-food-consumed + 0.820*goals-accomplished
;; (Since the 0.820 coefficient is higher than the 0.303 coefficient, it appears that each goal
;; accomplished yields more happiness than does each snack consumed, although both are positively
;; correlated with happiness.)
;; Also, we see that R^2 = 0.98, so the two factors together provide a good fit.
```

Sound Extension

The Sound Extension lets NetLogo models make two kinds of sounds: MIDI sounds and playback of pre-recorded sound files.

The Java API's used are javax.sound.midi and java.applet.AudioClip.

Using the Sound Extension

The sound extension comes preinstalled. To use the extension in your model, add this at the top of your Code tab:

```
extensions [sound]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `sound` to the list.

For more information on using NetLogo extensions, see the Extensions Guide.

For examples that use the sound extension, see the Sound section under Code Examples in the NetLogo Models Library.

MIDI support

The MIDI part of the extension simulates a 128-key electronic keyboard with 47 drums and 128 melodic instruments, as provided by General MIDI Level 1 specification.

It supports 15 polyphonic instrument channels and a single percussion channel. Using more than 15 different melodic instruments simultaneously in a model will cause some sounds to be lost or cut off.

The pitch of a melodic instrument is specified by a key number. The keys on the keyboard are numbered consecutively from 0 to 127, where 0 is the left-most key. Middle C is key number 60.

The loudness of an instrument is specified by a velocity, which represents the force with which the keyboard key is depressed. Velocity ranges from 0 to 127, where 64 is the standard velocity. A higher velocity results in a louder sound.

Primitives

sound:drums sound:instruments sound:play-drum sound:play-note sound:play-note-later sound:play-sound sound:play-sound-and-wait sound:play-sound-later sound:start-note sound:stop-note sound:stop-instrument sound:stop-music

sound:drums

sound:drums

Reports a list of the names of the 47 drums for use with "sound:play-drum".

sound:instruments

Reports a list of the names of the 128 instruments for use with "sound:play-note", "sound:play-note-later", "sound:start-note" and "sound:stop-note".

sound:play-drum

sound:play-drum *drum velocity*

Plays a drum.

```
sound:play-drum "ACOUSTIC SNARE" 64
```

sound:play-note

sound:play-note *instrument keynumber velocity duration*

Plays a note for a specified duration, in seconds. The agent does not wait for the note to finish before continuing to next command.

```
;; play a trumpet at middle C for two seconds
sound:play-note "TRUMPET" 60 64 2
```

sound:play-note-later

sound:play-note-later *delay instrument keynumber velocity duration*

Waits for the specified delay before playing the note for a specified duration, in seconds. The agent does not wait for the note to finish before continuing to next command.

```
;; in one second, play a trumpet at middle C for two seconds
sound:play-note-later 1 "TRUMPET" 60 64 2
```

sound:play-sound

sound:play-sound *filename*

Plays a sound file. It does not wait for the sound file to finish before moving to the next command. It supports WAV, AIFF, and AU files.

```
;; plays the beep.wav sample file
sound:play-sound "beep.wav"
```

sound:play-sound-and-wait

sound:play-sound-and-wait *filename*

Plays a sound file, waiting for it to finish playing before moving to the next command. It supports WAV, AIFF, and AU files.

```
;; plays the beep.wav sample file, waiting for it to finish before
;; playing boop.wav
sound:play-sound-and-wait "beep.wav"
sound:play-sound-and-wait "boop.wav"
```

sound:play-sound-later

sound:play-sound-later *filename delay*

Plays a sound file after the specified delay, in seconds. It does not wait for the sound file to play or finish before moving to the next command. It supports WAV, AIFF, and AU files.

```
;; plays the beep.wav sample file one second from now
sound:play-sound-later "beep.wav" 1
```

sound:start-note

sound:start-note *instrument keynumber velocity*

Starts a note.

The note will continue until "sound:stop-note", "sound:stop-instrument" or "sound:stop-music" is called.

```
;; play a violin at middle C
sound:start-note "VIOLIN" 60 64
```

```
;; play a C-major scale on a xylophone
foreach [60 62 64 65 67 69 71 72] [
  sound:start-note "XYLOPHONE" ? 65
  wait 0.2
  sound:stop-note "XYLOPHONE" ?
]
```

sound:stop-note

sound:stop-note *instrument keynumber*

Stops a note.

```
;; stop a violin note at middle C
```

```
sound:stop-note "VIOLIN" 60
```

sound:stop-instrument

sound:stop-instrument *instrument*

Stops all notes of an instrument.

```
;; stop all cello notes
sound:stop-instrument "CELLO"
```

sound:stop-music

sound:stop-music

Stops all notes.

Sound names

Drums

```
35. Acoustic Bass Drum          59. Ride Cymbal 2
36. Bass Drum 1                 60. Hi Bongo
37. Side Stick                  61. Low Bongo
38. Acoustic Snare              62. Mute Hi Conga
39. Hand Clap                   63. Open Hi Conga
40. Electric Snare              64. Low Conga
41. Low Floor Tom               65. Hi Timbale
42. Closed Hi Hat               66. Low Timbale
43. Hi Floor Tom                67. Hi Agogo
44. Pedal Hi Hat                68. Low Agogo
45. Low Tom                     69. Cabasa
47. Open Hi Hat                 70. Maracas
47. Low Mid Tom                 71. Short Whistle
48. Hi Mid Tom                  72. Long Whistle
49. Crash Cymbal 1              73. Short Guiro
50. Hi Tom                      74. Long Guiro
51. Ride Cymbal 1               75. Claves
52. Chinese Cymbal              76. Hi Wood Block
53. Ride Bell                   77. Low Wood Block
54. Tambourine                  78. Mute Cuica
55. Splash Cymbal               79. Open Cuica
56. Cowbell                     80. Mute Triangle
57. Crash Cymbal 2              81. Open Triangle
58. Vibraslap
```

Instruments

```
Piano                           Reed
1. Acoustic Grand Piano         65. Soprano Sax
2. Bright Acoustic Piano        66. Alto Sax
3. Electric Grand Piano         67. Tenor Sax
4. Honky-tonk Piano             68. Baritone Sax
5. Electric Piano 1             69. Oboe
6. Electric Piano 2             70. English Horn
7. Harpsichord                  71. Bassoon
```

8. Clavi

Chromatic Percussion
9. Celesta
10. Glockenspiel
11. Music Box
12. Vibraphone
13. Marimba
14. Xylophone
15. Tubular Bells
16. Dulcimer

Organ
17. Drawbar Organ
18. Percussive Organ
19. Rock Organ
20. Church Organ
21. Reed Organ
22. Accordion
23. Harmonica
24. Tango Accordion

Guitar
25. Nylon String Guitar
26. Steel Acoustic Guitar
27. Jazz Electric Guitar
28. Clean Electric Guitar
29. Muted Electric Guitar
30. Overdriven Guitar
31. Distortion Guitar
32. Guitar harmonics

Bass
33. Acoustic Bass
34. Fingered Electric Bass
35. Picked Electric Bass
36. Fretless Bass
37. Slap Bass 1
38. Slap Bass 2
39. Synth Bass 1
40. Synth Bass 2

Strings
41. Violin
42. Viola
43. Cello
44. Contrabass
45. Tremolo Strings
47. Pizzicato Strings
47. Orchestral Harp
48. Timpani

Ensemble
49. String Ensemble 1
50. String Ensemble 2
51. Synth Strings 1
52. Synth Strings 2
53. Choir Aahs
54. Voice Oohs
55. Synth Voice
56. Orchestra Hit

Brass
57. Trumpet
58. Trombone
59. Tuba
60. Muted Trumpet
61. French Horn
62. Brass Section
63. Synth Brass 1
64. Synth Brass 2

72. Clarinet

Pipe
73. Piccolo
74. Flute
75. Recorder
76. Pan Flute
77. Blown Bottle
78. Shakuhachi
79. Whistle
80. Ocarina

Synth Lead
81. Square Wave
82. Sawtooth Wave
83. Calliope
84. Chiff
85. Charang
86. Voice
87. Fifths
88. Bass and Lead

Synth Pad
89. New Age
90. Warm
91. Polysynth
92. Choir
93. Bowed
94. Metal
95. Halo
96. Sweep

Synth Effects
97. Rain
98. Soundtrack
99. Crystal
100. Atmosphere
101. Brightness
102. Goblins
103. Echoes
104. Sci-fi

Ethnic
105. Sitar
106. Banjo
107. Shamisen
108. Koto
109. Kalimba
110. Bag pipe
111. Fiddle
112. Shanai

Percussive
113. Tinkle Bell
114. Agogo
115. Steel Drums
116. Woodblock
117. Taiko Drum
118. Melodic Tom
119. Synth Drum
120. Reverse Cymbal

Sound Effects
121. Guitar Fret Noise
122. Breath Noise
123. Seashore
124. Bird Tweet
125. Telephone Ring
126. Helicopter
127. Applause
128. Gunshot

NetLogoLab and the GoGo Board Extension

What is NetLogoLab?

NetLogoLab is the technological infrastructure that connects NetLogo and the physical world. It can be used for robotics, interactive art, scientific investigations, and model validation. This infrastructure was created at the CCL by Paulo Blikstein and Uri Wilensky in 2005 as part of the Bifocal Modeling project. For more information, please check the new and old websites, where you will find academic papers, models, and demos.

NetLogoLab is comprised of the following software and hardware components:

1. A NetLogo extension to control a robotics or data-logging board.
2. A robotics or data-logging board (also know as a serial interface board, or analog-to-digital board).
3. Sensor and actuator toolkits.
4. NetLogo models.

NetLogo's robotics/data-logging board of choice is the GoGo Board, an open-source, easy-to-build, low-cost interface designed by Arnan Sipitakiat and Paulo Blikstein, first at the MIT Media Lab, then at Northwestern's CCL, and is in continuous refinement. Other robotics hardware can be used with NetLogo, including those that are commercially available, such as Arduino boards, Vernier and Pasco sensors and actuators, Phidgets, Lego robotics kits, and VEX kits, but specific extensions have not yet been developed for each of those platforms. So far, only the GoGo Board extension is available with NetLogo's standard distribution.

The GoGo Board NetLogo extension

The GoGo Extension for NetLogo provides primitives to communicate with a GoGo Board. This enables the user to connect NetLogo with the physical world using sensors, motors, light bulbs, LEDs, relays and other devices.

GoGo Board: a low-cost robotics and data-logging board

A GoGo Board is an open source, easy-to-build, low cost, general purpose serial interface board especially designed to be used in school and for educational projects. It was created by Arnan Sipitakiat and Paulo Blikstein at the MIT Media Lab in 2001, and has been actively developed since then. It is currently used in over 10 countries, such as the United States, China, Thailand, South Korea, Brazil, Portugal, Mexico, Malaysia, and Egypt.

Up to 8 sensors (i.e., temperature, light, pressure) and 4 output devices (i.e., motors, light bulbs, LEDs, relays) can be connected to the board simultaneously. The board also has a connector for add-on devices (such as displays, Bluetooth or ZigBee wireless modules, voice recorders, real-time clock, and GPS).

Sensor and actuator toolkits

NetLogo models can interact with the physical world in two ways. First, they can gather data from the environment. This information can be used by the model to change or calibrate its behavior. This data is gathered using electronic sensors, which can measure a wide range of phenomena: temperature, light, touch (see pictures below), pH, chemical concentration, pressure, etc.
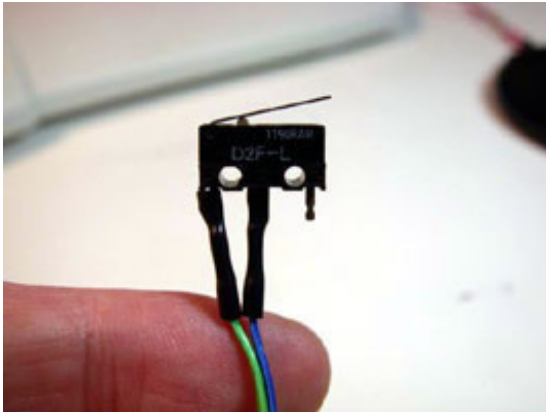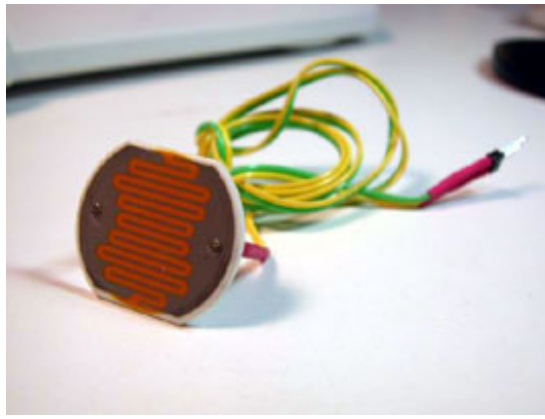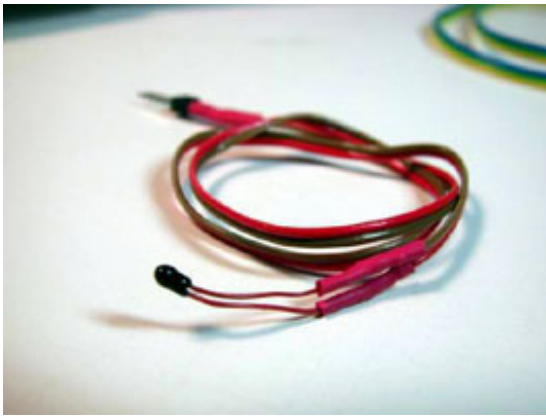
The second mode of interaction between NetLogo and the physical world is the control of output devices, or "actuators" - motors, light bulbs (see pictures below), LEDs, and more complex devices that include these outputs such as toys, remote controlled cars, electrical appliances, and automated laboratory equipment.

For educators willing to start robotics or sensing projects, there are some important considerations regarding the exact type of sensors and actuators to use - for example, the sturdiness, reliability, "openness", and cost of these devices. Both vary greatly in price and complexity. For example, for most educational projects, off-the-shelf, generic, low-cost

sensors can be used with very reliable results. A generic temperature sensor with a precision of 0.5 degrees Celsius can be purchased for approximately US$ 1.00 at most sensor and electronics retailers. Using generic, low-cost sensors requires very basic knowledge of electronics. For example, some basic soldering might be needed to attach a piece of wire to the sensors. While this is feasible in schools and has been tried in several educational settings, some educators might prefer to buy proprietary sensors and actuators, which come assembled and ready to be used. Some companies offer proprietary systems for educational sensing which are more sturdy and reliable than generic sensors, but are also much more expensive. As a comparison, a proprietary temperature sensor could cost as much as US$ 50.00. Actuators follow the same rule: for example, a generic geared motor could cost from US$ 3 to 10, while a proprietary version would retail for US$ 30 or 40.

Sensors and actuators can be found through online retailers such as Digikey, Mouser, Phidgets, Spark Fun, and Solarbotics. More information about how to make sensors is available from the "How to Make Sensors" web page on the GoGo board web site.

NetLogo models

To make use of the GoGo Board extension and the NetLogoLab framework, users need to create NetLogo models using the special primitives made available by the extension. Later in this document, we will provide examples of models that do this.

How to get a GoGo Board?

The GoGo Board is open source hardware (see the Open Hardware License here), so anyone can freely make one or even sell them. To get a GoGo Board, you have to build one yourself, ask/hire someone to do it for you, or buy from some of the existing retailers such as the SEED Foundation (or see a list here.) Many electronics assembly companies also can assemble boards, but they normally require a minimum quantity, which can range from 5 to 50. The main resource about the GoGo Board is the web site www.gogoboard.org, where you will find step-by-step instructions on how to buy components, design the printed circuit board, and assemble it - the board was especially designed to be easy and cheap to build, even if you don't have electronics skills. The GoGo Board mailing list is gogoboard@yahoogroups.com, and you can join it by going to Yahoo groups.

Installing and testing the GoGo Extension

The GoGo Board connects with the computer via the USB port (older versions used the serial port which required a USB-to-Serial adapter -- one model that worked quite well was the Keyspan USA-19HS). To connect, make sure to plug the USB cable into the computer. Turn the GoGo Board on using the switch behind the power connector: the board will beep twice and the red light will turn on.

Windows

- **Windows 7 and Vista:**
  - When you start a NetLogo model that uses the GoGo extension, it will check to see if you have the proper drivers installed. If it detects that you do not, NetLogo will offer to launch a small program that will install the drivers for you (but note that you will need to have "administrator" access to the computer in order to do so (basically, Windows will ask you to click a button to confirm that you really want to run the program). You will also need to accept an "unsigned" driver (this does not mean you are installing anything harmful, it just means that the driver file was not

officially recognized by Microsoft as an "official" driver, but the file comes from the GoGo board creators and is 100% safe.)

- If you already have the drivers installed and *still* receive the popup, please select "Halt and Don't Remind Me Again" to prevent any further notices.
- If installing the GoGo Board drivers through NetLogo does not work correctly for you, please view the instructions for Windows advanced installation and see [Contacting Us](#).

- **Windows advanced installation:** Unfortunately, the driver installer does not always work properly on Windows.
  1. One possible solution is to download and install the [GoGo Monitor for Windows](#).
  2. If you don't want to download the GoGo Monitor, you can also install the driver more manually. To do so, please follow these steps:
     - Go to your Windows Control Panel and open Device Manager
     - Look for a device called "SERIAL DEMO", right-click it, and select "Update driver software"
     - Choose to browse for the driver manually
     - Navigate to the "Windows" folder in your GoGo extension installation (likely something along the lines of "C:\Program Files\NetLogo 5.x\extensions\gogo\Windows"), and select the file named "gogo_cdc_NTXPVista.inf"
     - Proceed through the driver installation wizard to complete the installation
     - **Note**: Depending on your computer, this process may install the driver for *only* the current USB port that the GoGo board is plugged into. That is, if the GoGo board is later connected to a different USB port, you may have to repeat this process in order for it to be recognized from that port.

Mac OS X

The GoGo extension requires no special installation on Mac OS X. The only issue for now, due to a bug in the open source software we use for controlling the serial port, is that you CANNOT use the "gogo:close" command, or execute the "gogo:open" twice. This will be fixed as soon as we get the updated software from the RXTX team. If you are using any of the example models (GoGo Monitor and GoGo Monitor Simple), DO NOT click on "setup" twice -- click once and wait some seconds for the connection to be made.

Linux

Many versions of Linux require no special installation. However, if you face problems, ensure that you are able to write to the serial devices (usually, this means `/dev/ttyS*`). In most Linux distributions this can be set up through the User Manager.

Using the GoGo Extension

The GoGo Extension comes preinstalled when you download and install NetLogo. To use the extension in your model, add this line to the top of your Code tab:

```
extensions [gogo]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `gogo` to the list.

After loading the extension, see what ports are available by typing the following into the command center:

```
print gogo:ports
```

You can open the serial port that the GoGo Board is connected to so that commands can begin to be sent to the board by using the `gogo:open` command. To make sure the board is properly connected, check that it is responding with the `ping` reporter. Note that in order to communicate with the board, you need to know which communications port it is connected to. If you are not sure which port is being used, you can use the `gogo:ports` primitive (see below), or you can find out by using the Device Manager on a Windows computer (in the Control Panel, click on the System icon), or the System Profiler on Mac OS X.

On Windows:

```
gogo:open "COM1"
print gogo:ping
```

On Linux:

```
gogo:open "/dev/ttyS01"
print gogo:ping
```

On Mac:

```
gogo:open "/dev/tty.KeySerial1"
print gogo:ping
```

For more information on NetLogo extensions, see the Extensions Guide.

For examples that use the GoGo extension, see the Robotics/NetLogoLab section under Sample Models in NetLogo's Models Library.

Examples of NetLogoLab models

Controlling a car

The first step when creating a NetLogoLab model is to add the `extensions` keyword to NetLogo's Code tab. Just go to the Code tab and add this line:

```
extensions [gogo]
```

The second step is to create a button to connect NetLogo to the GoGo board using the correct serial port for your operating system as described above.

```
gogo:open "COM1" ;; (for Windows machines)
```

When you are done creating the button, the "edit" dialog should look like this:
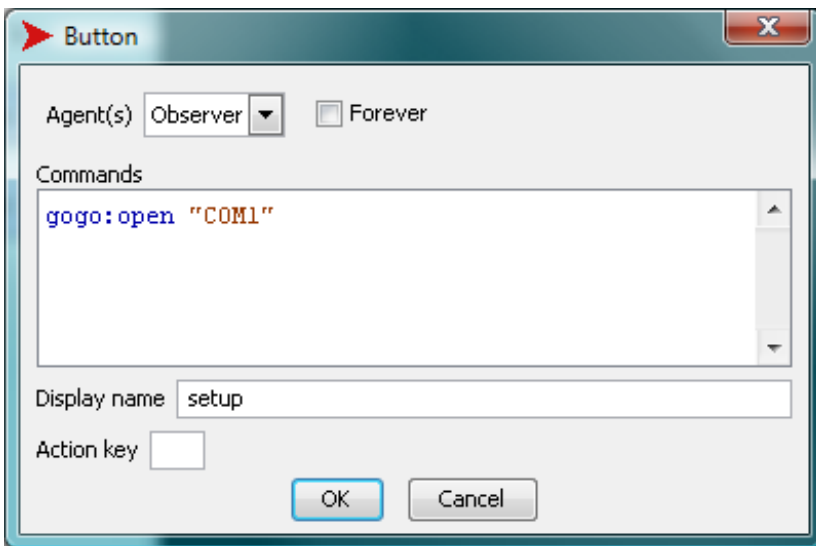
Now let's actually start the model. Imagine that we want to control a car with four wheels and two motors attached to the back wheels. We will assume that you have built such as car and connected the motors to the output ports "a" and "b" on the GoGo board. One very simple approach could be to create two buttons for each motor, "on" and "off":
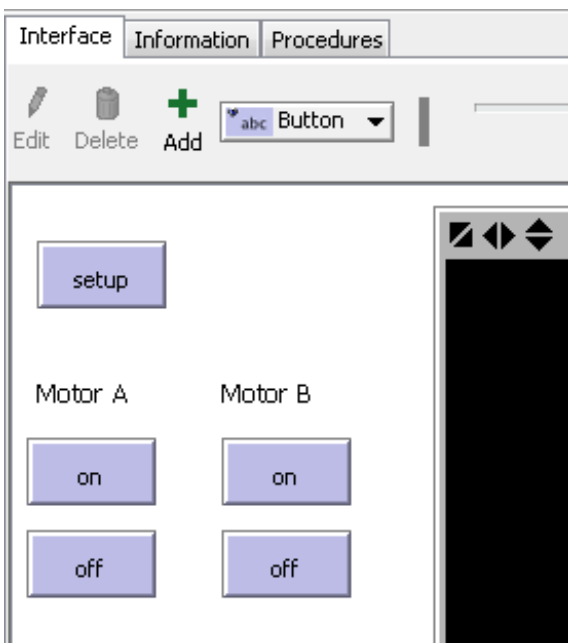
The code associated with these buttons is very simple: for the "on" button, we could simply have

```
gogo:talk-to-output-ports ["a"]
gogo:output-port-on
```

For the off button, it would be very similar:

```
gogo:talk-to-output-ports ["a"]
gogo:output-port-off
```

The other set of "on" and "off" buttons, used to control the second motor, would have very similar code, except that we would use the second output port ("b"), so:

```
gogo:talk-to-output-ports ["b"]
```

We could make our model more interesting by adding a "toggle direction" button, adding a button with the following code, which would reverse the direction of motors "a" and "b":

```
gogo:talk-to-output-ports ["a" "b"]
gogo:output-port-reverse
```

A simple sensing project

To create a simple sensing project, we will assume that you have added the GoGo extension to your model and successfully opened a connection to the GoGo board, i.e., adding the "extensions" command to the Code Tab and adding a "setup" button as previously described. For this sensing project we do not need motors, but we will need another device: a temperature sensor (click to see more information about a typical temperature sensor at Digikey's web site). Instructions on how to purchase and assemble a temperature sensor can be found in the "Making Sensors" tutorial on the GoGo Board's web site. This is how a temperature sensor will look after it has been assembled:

The simplest use of a temperature sensor, obviously, is to display the temperature. We could achieve this by adding a monitor to the NetLogo interface with this code:

```
gogo:sensor 1
```
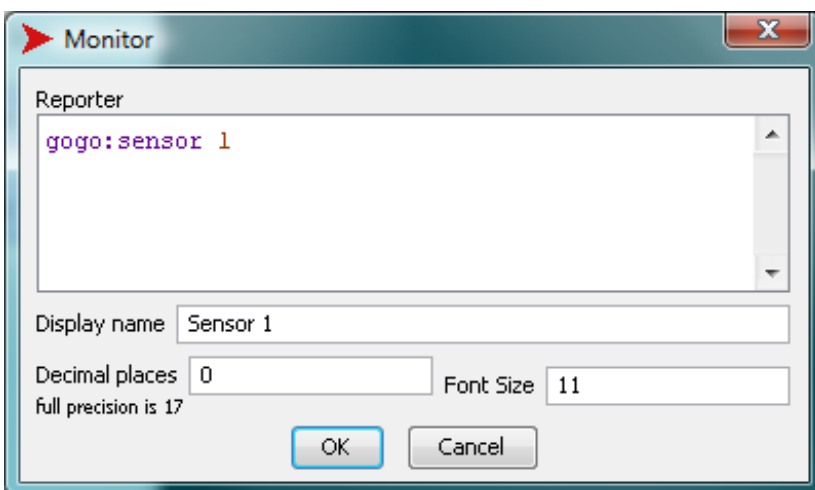

The sensor values shown are arbitrary numbers within a given range, and need to be properly converted to actual temperature units such as degrees Celsius or Fahrenheit. For all sensors, the GoGo Board's reporting range is from 0 to 1023. Every sensor comes with a data sheet with a converting formula or table that will transform the arbitrary 0-1023 range into an actual physical unit. Let us imagine that the sensor's data sheet contains a conversion formula that

looks like this: *degrees = arbitrary value / 30.* The monitor on the NetLogo interface could be changed to:

```
gogo:sensor 1 / 30
```

The sensor value could also be used to control on-screen objects, such as turtles. For example, let us create two buttons: a "create one turtle" button, which will clear the world and create a turtle, and a "move with heat" button, that will cause the turtle to move forwards depending on the temperature reading from the sensor. The code would look like this:

```
to create-one-turtle
  clear-all
  create-turtle
end

to move-with-heat
  if gogo:sensor 1 < 500
    [ forward 1 ]
end
```

If the "move with heat" forever button is activated and the user heats up the sensor (by rubbing it, or *slowly* bringing a flame near it), the heat threshold will be achieved (< 500) and the turtle will move. (Note that we are using one kind of temperature sensor for which resistance *decreases* with temperature, and so the sensor readings will go down as we heat the sensor up. This is a very common and inexpensive off-the-shelf sensor).

A more elaborate use of this sensor apparatus would be to control output devices, such as motors. The user could, for example, turn a motor on when the value from the temperature sensor reaches 500, using the following code:

```
to turn-motor-on-with-heat
  if gogo:sensor 1 < 500
  [
    gogo:talk-to-output-ports ["a"]
    gogo:output-port-on
  ]
end
```

Another possible use of the sensing primitives is to plot and log data. Logging could be useful for more elaborate data analysis and comparison, and can be achieved with NetLogo's list commands. For example, if the user wants to log sensor values from sensor 1 every 0.5 seconds, the code could look like this:

```
to log-data-from-sensor
  set data-vector lput (gogo:sensor 1) data-vector
  wait 0.5
end
```

Finally, plotting data is straightforward. The following code, for example, would create a graph for the value of sensor 1:

```
plot (gogo:sensor 1)
```

For more information on the GoGo Board's extensions functionalities and primitives, please refer to these two sample models: GoGoMonitor.nlogo and GoGoMonitorSimple.nlogo.

Primitives

gogo:beep

gogo:beep

Sends a signal to the GoGo Board to make it emit an audible beep. Useful for a testing purposes.

Example:

```
gogo:beep
```

gogo:burst-value

gogo:burst-value *sensor*

Reads the most recent value that was received by the gogo board from a sensor set to send burst data.

See also gogo:set-burst-mode and gogo:stop-burst-mode.

gogo:close

gogo:close

Close the connection to the GoGo Board.

See also `gogo:open` and `gogo:open?`.

gogo:install

gogo:install

If on a Windows system, attempt to force-run the Windows driver installer.

Example:

```
gogo:install
```

gogo:led-off

gogo:led-off

Sends the signal to the GoGo Board to turn off the LED.

Example:

```
gogo:led-off
```

See also `gogo:led-on`.

gogo:led-on

gogo:led-on

Sends the signal to the GoGo Board to turn on the LED.

Example:

```
gogo:led-on
```

See also `gogo:led-off`.

gogo:open

gogo:open *port-name*

Open a connection to the GoGo Board connected to serial port named *port-name*. See `gogo:ports` for more information about port names.

If the GoGo Board is not responding, or you attempt to open a port without a GoGo Board connected to it, an error will be generated.

Example:

```
gogo:open "COM1"
```

See also `gogo:open?` and `gogo:close`.

gogo:open?

gogo:open?

Reports true if there is a connection to a GoGo Board open. Reports false otherwise.

gogo:ports

gogo:ports

Reports a list of serial port names that a GoGo Board **may** be connected to. On certain computers, you might get a list of two or three different serial ports. In that case, try to open each of them until the connection is successful.

gogo:output-port-coast

gogo:output-port-coast

Turns off the power of the active ports. When attached to motors, does not apply a braking force as `gogo:output-port-off` does. Therefore, the motor will gradually slow down before stopping completely. This will have the same effect as `gogo:output-port-off` on most output devices other than motors. The output-ports affected by this command are determined by the `gogo:talk-to-output-ports` command.

The following code will turn on an output port a for 1 second, and then stop the motor gradually:

```
gogo:talk-to-output-ports ["a"]
gogo:output-port-on
wait 1
gogo:output-port-coast
```

gogo:output-port-off

gogo:output-port-off

Turns off power to the output ports. If using motors, a braking force is applied. The output ports affected by this command are determined by the `gogo:talk-to-output-ports` command.

gogo:output-port-on

gogo:output-port-on

Turns on power to the output ports. The output ports affected by this command are determined by the `gogo:talk-to-output-ports` command.

gogo:output-port-reverse

gogo:output-port-reverse

Reverses the direction of the output ports. The output ports affected by this command are determined by the `gogo:talk-to-output-ports` command.

gogo:output-port-[that/this]way

gogo:output-port-thatway

gogo:output-port-thisway

Apply power to the output port in a given direction. Output ports can be powered in two directions, arbitrarily called *thisway* and *thatway*. The output-ports affected by the command are determined by the `gogo:talk-to-output-ports` command. Note that this is different from `gogo:output-port-reverse` because *thisway* and *thatway* will always be the same direction provided the connector's polarity is the same.

gogo:talk-to-output-ports

gogo:talk-to-output-ports *output-portlist*

This command will set the corresponding output ports as active. They will be the ones affected by the commands such as `gogo:output-port-on` and `gogo:output-port-off`. The user can "talk" to one or multiple ports at the same time. Output ports are typically connected to motors, but you could also use bulbs, LEDs and relays. Output ports are identified by one letter names: "a", "b", "c", and "d".

Examples:

```
;; talk to all output-ports
gogo:talk-to-output-ports [ "a" "b" "c" "d" ]
;; will give power to all output-ports
gogo:output-port-on

;; talk to output-ports A and D
gogo:talk-to-output-ports [ "a" "d" ]
;; will turn off output-ports A and D.
;; The other output-ports will keep
;; their current state
gogo:output-port-off

gogo:talk-to-output-ports [ "c" "b" ]
;; turn off remaining output-ports
gogo:output-port-off
```

gogo:ping

gogo:ping

Checks the status of GoGo board. This is mostly used to make sure the board is connected to the correct serial port. It reports true if the GoGo Board responds to a diagnostic message, and false otherwise.

Example:

```
print gogo:ping
```

gogo:sensor

gogo:sensor *sensor*

Reports the numeric value of the sensor named *sensor*. Sensors are identified by numbers 1 to 8. Values range between 0-1023. 1023 is returned when there is no sensor attached to the port (highest resistance), or when the sensor is an "open" state. Zero is returned when the sensor is short circuited (no resistance).

Examples:

```
print gogo:sensor 1
;; prints the value of sensor 1

foreach [ 1 2 3 4 5 6 7 8 ]
  [print (word "Sensor " ? " = " gogo:sensor ?)]
;; prints the value of all sensors

if gogo:sensor 1 < 500 [ ask turtles [ fd 10 ]]
;; will move all turtles 10 steps forward if sensor 1's value is less than 500.

loop [if gogo:sensor 1 < 500 [ ask turtles [ fd 10 ] ] ]
;; will continuously check sensor 1's value and
;; move all turtles 10 steps forward every time
;; that the sensor value is less than 500.
```

gogo:set-burst-mode

gogo:set-burst-mode *sensor-list high-speed-mode?*

Turns on "burst mode" for the sensors in *sensor-list*. If *high-speed-mode?* is `true` then high speed burst mode will be used. If it is false, then low speed burst mode will be used.

See also [gogo:burst-value](#) and [gogo:stop-burst-mode](#).

Examples:

```
gogo:set-burst-mode [ 1 2 3 ] true
;; turns on high speed burst mode for sensors 1, 2 and 3

gogo:set-burst-mode [ 4 ] false
;; turns on low speed burst mode for sensor 4
;; this will also override any previous set-burst-mode calls
;; so there will be no new burst mode data for sensors 1, 2 and 3
```

gogo:set-output-port-power

gogo:set-output-port-power *power-level*

Sets the power level of the active output ports. *power-level* is a number between 0 (off) and 7 (full-power). The output-ports affected by this command are determined by the `gogo:talk-to-output-ports` command. Note that for many practical applications it is more efficient to use mechanical devices, such as gears and pulleys, to control the torque of motors.

Example:

```
gogo:talk-to-motors ["a" "b" "c" "d"]
gogo:set-motor-power 4
;; will lower the power of all output ports by half of the full power .
```

gogo:set-servo

gogo:set-servo *servo-position*

Set the servo position to the value given by *servo-position*.

servo-position must be between 20 and 40 (inclusive).

Example:

```
gogo:set-servo 25
```

gogo:stop-burst-mode

gogo:stop-burst-mode

Turns off "burst mode" for all sensors

See also [gogo:burst-value](#) and [gogo:set-burst-mode](#).

Profiler Extension

If you'd like your model to run faster, the profiler extension may be useful to you. It includes primitives that measure how many times the procedures in your model are called during a run and how long each call takes. You can use this information to where to focus your speedup efforts.

Caution

The profiler extension is experimental. It is not yet well tested or user friendly. Nonetheless, we think some users will find it useful.

Usage

The profiler extension comes preinstalled. To use the extension in your model, add a line to the top of your Code tab:

```
extensions [profiler]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `profiler` to the list.

For more information on using NetLogo extensions, see the Extensions Guide.

Example

```
setup                  ;; set up the model
profiler:start         ;; start profiling
repeat 20 [ go ]       ;; run something you want to measure
profiler:stop          ;; stop profiling
print profiler:report  ;; view the results
profiler:reset         ;; clear the data
```

> **Code Example:** Profiler Example

Primitives

profiler:calls profiler:exclusive-time profiler:inclusive-time profiler:start profiler:stop profiler:reset profiler:report

profiler:calls

profiler:calls *procedure-name*

Reports the number of times that *procedure-name* was called. If *procedure-name* is not defined, then reports 0.

profiler:exclusive-time

profiler:exclusive-time *procedure-name*

Reports the exclusive time, in milliseconds, that *procedure-name* was running for. Exclusive time is the time from when the procedure was entered, until it finishes, but does not include any time spent in other user-defined procedures which it calls.

If *procedure-name* is not defined, then reports 0.

profiler:inclusive-time

profiler:inclusive-time *procedure-name*

Reports the inclusive time, in milliseconds, that *procedure-name* was running for. Inclusive time is the time from when the procedure was entered, until it finishes.

If *procedure-name* is not defined, then reports 0.

profiler:start

profiler:start

Instructs the profiler to begin recording user-defined procedure calls.

profiler:stop

profiler:stop

Instructs the profiler to stop recording user-defined procedure calls.

profiler:reset

profiler:reset

Instructs the profiler to erase all collected data.

profiler:report

profiler:report

Reports a string containing a breakdown of all user-defined procedure calls. The `Calls` column contains the number of times a user-defined procedure was called. The `Incl T(ms)` column is the total time, in milliseconds, it took for the call to complete, including the time spent in other user-defined procedures. The `Excl T(ms)` column is the total time, in milliseconds, spent within that user-defined procedure, not counting other user-define procedures it called. The `Excl/calls` column is an estimate of the time, in milliseconds, spent in that user-defined

procedure for each call.

Here is example output:

```
Sorted by Exclusive Time
Name                          Calls Incl T(ms) Excl T(ms) Excl/calls
CALLTHEM                        13     26.066     19.476      1.498
CALLME                          13      6.413      6.413      0.493
REPORTME                        13      0.177      0.177      0.014

Sorted by Inclusive Time
Name                          Calls Incl T(ms) Excl T(ms) Excl/calls
CALLTHEM                        13     26.066     19.476      1.498
CALLME                          13      6.413      6.413      0.493
REPORTME                        13      0.177      0.177      0.014

Sorted by Number of Calls
Name                          Calls Incl T(ms) Excl T(ms) Excl/calls
CALLTHEM                        13     26.066     19.476      1.498
```

GIS Extension

This extension adds GIS (Geographic Information Systems) support to NetLogo. It provides the ability to load vector GIS data (points, lines, and polygons), and raster GIS data (grids) into your model.

The extension supports vector data in the form of ESRI shapefiles. The shapefile (.shp) format is the most common format for storing and exchanging vector GIS data. The extension supports raster data in the form of ESRI ASCII Grid files. The ASCII grid file (.asc or .grd) is not as common as the shapefile, but is supported as an interchange format by most GIS platforms.

How to use it

In general, you first define a transformation between GIS data space and NetLogo space, then load datasets and perform various operations on them. The easiest way to define a transformation between GIS space and NetLogo space is to take the union of the "envelopes" or bounding rectangles of all of your datasets in GIS space and map that directly to the bounds of the NetLogo world. See GIS General Examples for an example of this technique.

You may also optionally define a projection for the GIS space, in which case datasets will be re-projected to match that projection as they are loaded, as long as each of your data files has an associated .prj file that describes the projection or geographic coordinate system of the data. If no associated .prj file is found, the extension will assume that the dataset already uses the current projection, regardless of what that projection is.

Once the coordinate system is defined, you can load datasets using [gis:load-dataset](). This primitive reports either a VectorDataset or a RasterDataset, depending on what type of file you pass it.

A VectorDataset consists of a collection of VectorFeatures, each one of which is a point, line, or polygon, along with a set of property values. A single VectorDataset may contain only one of the three possible types of features.

There are several things you can do with a VectorDataset: ask it for the names of the properties of its features, ask it for its "envelope" (bounding rectangle), ask for a list of all VectorFeatures in the dataset, search for a single VectorFeature or list of VectorFeatures whose value for a particular property is less than or greater than a particular value, or lies within a given range, or matches a given string using wildcard matching ("*", which matches any number of occurrences of any characters). If the VectorFeatures are polygons, you can also apply the values of a particular property of the dataset's features to a given patch variable.

There are also several things you can do with a VectorFeature from a VectorDataset: ask it for a list of vertex lists, ask it for a property value by name, ask it for its centroid (center of gravity), and ask for a subset of a given agentset whose agents intersect the given VectorFeature. For point data, each vertex list will be a one-element list. For line data, each vertex list will represent the vertices of a line that makes up that feature. For polygon data, each vertex list will represent one "ring" of the polygon, and the first and last vertex of the list will be the same. The vertex lists are made up of values of type Vertex, and the centroid will be a value of type Vertex as well.

There are a number of operations defined for RasterDatasets as well. Mostly these involve sampling the values in the dataset, or re-sampling a raster to a different resolution. You can also apply a raster to a given patch variable, and convolve a raster using an arbitrary

convolution matrix.

> **Code Example:** GIS General Examples has general examples of how to use the extension
>
> **Code Example:** GIS Gradient Example is a more advanced example of raster dataset analysis.

Known issues

Values of type RasterDataset, VectorDataset, VectorFeature, and Vertex are not handled properly by `export-world` and `import-world`. To save datasets, you must use the `gis:store-dataset` primitive.

There is currently no way to distinguish positive-area "shell" polygons from negative-area "hole" polygons, or to determine which holes are associated with which shells.

Credits

The primary developer of the GIS extension was Eric Russell.

The GIS extension makes use of several open-source software libraries. For copyright and license information on those, see the copyright section of the manual. The extension also contains elements borrowed from My World GIS.

This documentation and the example NetLogo models are in the public domain. The GIS extension itself is free and open source software. See the README.md file in the extension/gis directory for details.

We would love to hear your suggestions on how to improve the GIS extension, or just about what you're using it for. Post questions and comments at the NetLogo Users Group, or write directly to Eric Russell and the NetLogo team at ccl-gis@ccl.northwestern.edu.

GIS primitives

Coordinate System Primitives

set-transformation set-transformation-ds set-world-envelope set-world-envelope-ds world-envelope envelope-of envelope-union-of load-coordinate-system set-coordinate-system

Dataset Primitives

load-dataset store-dataset type-of patch-dataset turtle-dataset link-dataset

VectorDataset Primitives

shape-type-of property-names feature-list-of vertex-lists-of centroid-of location-of property-value find-features find-one-feature find-less-than find-greater-than find-range property-minimum property-maximum apply-coverage coverage-minimum-threshold set-coverage-

RasterDataset Primitives

Drawing Primitives

Coordinate System Primitives

gis:set-transformation

gis:set-transformation *gis-envelope netlogo-envelope*

Defines a mapping between GIS coordinates and NetLogo coordinates. The *gis-envelope* and *netlogo-envelope* parameters must each be four-element lists consisting of:

```
[minimum-x maximum-x minimum-y maximum-y]
```

The scale of the transformation will be equal to the minimum of the scale necessary to make the mapping between the ranges of x values and the scale necessary to make the mapping between the ranges of y values. The GIS space will be centered in NetLogo space.

For example, the following two lists would map all of geographic (latitude and longitude) space in degrees to NetLogo world space, regardless of the current dimensions of the NetLogo world:

```
(list -180 180 -90 90)
(list min-pxcor max-pxcor min-pycor max-pycor)
```

However, if you're setting the envelope of the NetLogo world, you should probably be using [set-world-envelope](#) below.

gis:set-transformation-ds

gis:set-transformation-ds *gis-envelope netlogo-envelope*

Does the same thing as [set-transformation](#) above, except that it allows the scale for mapping the range of x values to be different than the scale for y values. The "-ds" on the end stands for "different scales". Using different scales will cause distortion of the shape of GIS features, and so it is generally not recommended, but it may be useful for some models.

Here is an example of the difference between set-transformation and set-transformation-ds:

Using set-transformation, the scale along the x and y axis is the same, preserving the round shape of the Earth in this Orthographic projection.	Using set-transformation-ds, the scale along the x axis is stretched so that the earth covers the entire NetLogo View, which in this case distorts the shape of the Earth.
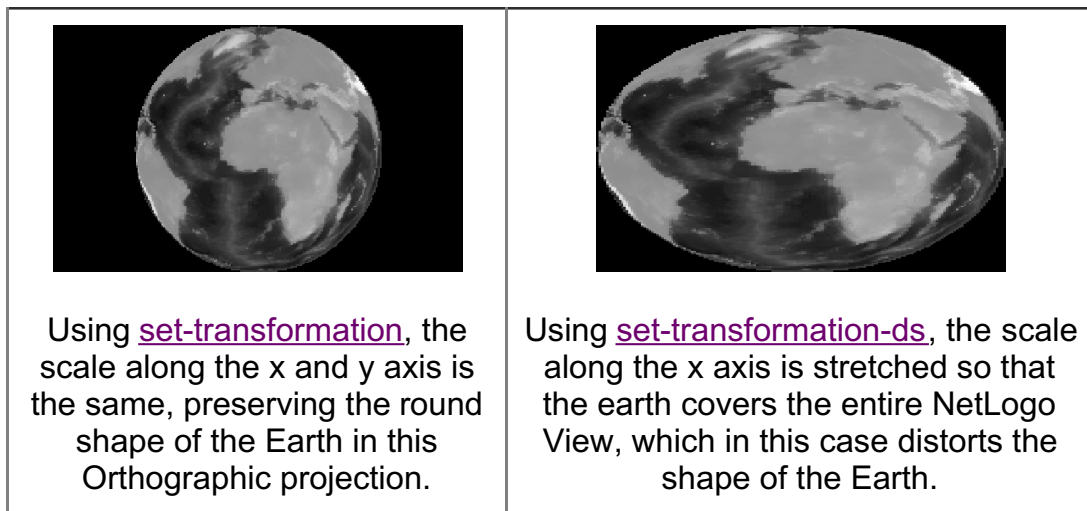
gis:set-world-envelope

gis:set-world-envelope *gis-envelope*

A shorthand for setting the transformation by mapping the envelope of the NetLogo world to the given envelope in GIS space, while keeping the scales along the x and y axis the same. It is equivalent to:

```
set-transformation gis-envelope (list min-pxcor max-pxcor min-pycor max-pycor)
```

This primitive is supplied because most of the time you'll want to set the envelope of the entire NetLogo world, rather than just a part of it.

gis:set-world-envelope-ds

gis:set-world-envelope-ds *gis-envelope*

A shorthand for setting the transformation by mapping the envelope of the NetLogo world to the given envelope in GIS space, using different scales along the x and y axis if necessary. It is equivalent to:

```
set-transformation-ds gis-envelope (list min-pxcor max-pxcor min-pycor max-pycor)
```

See the pictures above for the difference between using equal scales for x and y coordinates and using different scales.

gis:world-envelope

gis:world-envelope

Reports the envelope (bounding rectangle) of the NetLogo world, transformed into GIS space.

An envelope consists of a four-element list of the form:

```
[minimum-x maximum-x minimum-y maximum-y]
```

gis:envelope-of

gis:envelope-of *thing*

Reports the envelope (bounding rectangle) of *thing* in GIS coordinates. The *thing* may be an Agent, an AgentSet, a RasterDataset, a VectorDataset, or a VectorFeature. An envelope consists of a four-element list of the form:

```
[minimum-x maximum-x minimum-y maximum-y]
```

gis:envelope-union-of

gis:envelope-union-of *envelope1 envelope2*
(gis:envelope-union-of *envelope1* ...)

Reports an envelope (bounding rectangle) that entirely contains the given envelopes. An envelope consists of a four-element list of the form

```
[minimum-x maximum-x minimum-y maximum-y]
```

No assumption is made about the coordinate system of the arguments, though if they are not in the same coordinate system, results will be unpredictable.

gis:load-coordinate-system

gis:load-coordinate-system *file*

Loads a new global projection used for projecting or re- projecting GIS data as it is loaded from a file. The file must contain a valid [Well-Known Text (WKT)](#) projection description.

WKT projection files are frequently distributed alongside GIS data files, and usually have a ".prj" filename extension.

Relative paths are resolved relative to the location of the current model, or the user's home directory if the current model hasn't been saved yet.

The GIS extension does not support all WKT coordinate systems and projections. Only geographic (`"GEOGCS"`) and projected (`"PROJCS"`) coordinate systems are supported. For projected coordinate systems, only the following projections are supported:

| Albers_Conic_Equal_Area | Lambert_Conformal_Conic_2SP Polyconic |

Lambert_Azimuthal_Equal_Area	Mercator_1SP	Robinson
Azimuthal_Equidistant	Miller	Stereographic
Cylindrical_Equal_Area	Oblique_Mercator	Transverse_Mercator
Equidistant_Conic	hotine_oblique_mercator	
Gnomonic	Orthographic	

See remotesensing.org for a complete list of WKT projections and their parameters.

gis:set-coordinate-system

gis:set-coordinate-system *system*

Sets the global projection used for projecting or re- projecting GIS data as it is loaded. The *system* must be either a string in Well-Known Text (WKT) format, or a NetLogo list that consists of WKT converted to a list by moving each keyword inside its associated brackets and putting quotes around it. The latter is preferred because it makes the code much more readable.

The same limitations on WKT support apply as described above in the documentation for load-coordinate-system

Dataset Primitives

gis:load-dataset

gis:load-dataset *file*

Loads the given data file, re-projecting the data as necessary if a global projection is defined and if the data file itself has an associated .prj file, then reports the resulting dataset.

If no ".prj" file is present, then `load-dataset` assumes that the projection of the data being loaded is the same as the current global coordinate system.

Relative paths are resolved relative to the location of the current model, or the user's home directory if the current model hasn't been saved yet.

Currently, two types of data file are supported:

- "**.shp**" (ESRI shapefile): contains vector data, consisting of points, lines, or polygons. When the target file is a shapefile, `load-dataset` reports a VectorDataset.
- "**.asc**" or "**.grd**" (ESRI ASCII grid): contains raster data, consisting of a grid of values. When the target file is an ASCII grid file, `load-dataset` reports a RasterDataset.

gis:store-dataset

gis:store-dataset *dataset file*

Saves the given dataset to the given file. If the name of the file does not have the proper file extension, the extension will be automatically appended to the name. Relative paths are

resolved relative to the location of the current model, or the user's home directory if the current model hasn't been saved yet.

Currently, this primitive only works for RasterDatasets, and it can only save those datasets as ESRI ASCII grid files.

gis:type-of

gis:type-of *dataset*

Reports the type of the given GIS dataset: either "VECTOR" or "RASTER".

gis:patch-dataset

gis:patch-dataset *patch-variable*

Reports a new raster whose cells correspond directly to NetLogo patches, and whose cell values consist of the values of the given patch variable. This primitive is basically the inverse of [apply-raster](); `apply-raster` copies values from a raster dataset to a patch variable, while this primitive copies values from a patch variable to a raster dataset.

gis:turtle-dataset

gis:turtle-dataset *turtle-set*

Reports a new, point VectorDataset built from the turtles in the given agentset. The points are located at locations of the turtles, translated from NetLogo space into GIS space using the current coordinate transformation. And the dataset's properties consist of all of the turtle variables common to every turtle in the agentset.

gis:link-dataset

gis:link-dataset *link-set*

Reports a new, line VectorDataset built from the links in the given agentset. The endpoints of each line are at the location of the turtles connected by each link, translated from NetLogo space into GIS space using the current coordinate transformation. And the dataset's properties consist of all of the link variables common to every link in the agentset.

VectorDataset Primitives

gis:shape-type-of

gis:shape-type-of *VectorDataset*

Reports the shape type of the given dataset. The possible output values are "POINT", "LINE", and "POLYGON".

gis:property-names

gis:property-names *VectorDataset*

Reports a list of strings where each string is the name of a property possessed by each VectorFeature in the given VectorDataset, suitable for use in gis:property-value.

gis:feature-list-of

gis:feature-list-of *VectorDataset*

Reports a list of all VectorFeatures in the given dataset.

gis:vertex-lists-of

gis:vertex-lists-of *VectorFeature*

Reports a list of lists of Vertex values. For point datasets, each vertex list will contain exactly one vertex: the location of a point. For line datasets, each vertex list will contain at least two points, and will represent a "polyline", connecting each adjacent pair of vertices in the list. For polygon datasets, each vertex list will contain at least three points, representing a polygon connecting each vertex, and the first and last vertices in the list will be the same.

gis:centroid-of

gis:centroid-of *VectorFeature*

Reports a single Vertex representing the centroid (center of gravity) of the given feature. For point datasets, the centroid is defined as the average location of all points in the feature. For line datasets, the centroid is defined as the average of the locations of the midpoints of all line segments in the feature, weighted by segment length. For polygon datasets, the centroid is defined as the weighted sum of the centroids of a decomposition of the area into (possibly overlapping) triangles. See this FAQ for more details on the polygon centroid algorithm.

gis:location-of

gis:location-of *Vertex*

Reports a two-element list containing the x and y values (in that order) of the given vertex translated into NetLogo world space using the current transformation, or an empty list if the given vertex lies outside the NetLogo world.

gis:property-value

gis:property-value *VectorFeature property-name*

Reports the value of the property with the given name for the given VectorDataset. The reported value may be a number, a string, or a boolean value, depending on the type of the field in the underlying data file.

For shapefiles, values from dBase CHARACTER and DATE fields are returned as strings, values from NUMBER and FLOAT fields are returned as numbers, and values from LOGICAL fields are returned as boolean values. MEMO fields are not supported. DATE values are converted to strings using ISO 8601 format (YYYY-MM-DD).

gis:find-features

gis:find-features *VectorDataset property-name string*

Reports a list of all VectorFeatures in the given dataset whose value for the property *property-name* matches the given string. Value comparison is not case sensitive, and the wildcard character "*" will match any number of occurrences (including zero) of any character.

gis:find-one-feature

gis:find-one-feature *VectorDataset property-name string*

Reports the first VectorFeature in the dataset whose value for the property *property-name* matches the given string. Value comparison is not case sensitive, and the wildcard character "*" will match any number of occurrences (including zero) of any character. Features are searched in the order that they appear in the data file that was the source of the dataset, and searching stops as soon as a match is found. Reports nobody if no matching VectorFeature is found.

gis:find-less-than

gis:find-less-than *VectorDataset property-name value*

Reports a list of all VectorFeatures in the given dataset whose value for the property *property-name* is less than the given *value*. String values are compared using case-sensitive lexicographic order as defined in the [Java Documentation](). Using a string value for a numeric property or a numeric value for a string property will cause an error.

gis:find-greater-than

gis:find-greater-than *VectorDataset property-name value*

Reports a list of all VectorFeatures in the given dataset whose value for the property *property-name* is greater than the given *value*. String values are compared using case-sensitive lexicographic order as defined in the [Java Documentation](). Using a string value for a numeric property or a numeric value for a string property will cause an error.

gis:find-range

gis:find-range *VectorDataset property-name minimum-value maximum-value*

Reports a list of all VectorFeatures in the given dataset whose value for the property *property-name* is strictly greater than *minimum-value* and strictly less than *maximum-value*. String values are compared using case-sensitive lexicographic order as defined in the [Java Documentation](). Using a string value for a numeric property or a numeric value for a string property will cause an error.

gis:property-minimum

gis:property-minimum *VectorDataset property-name*

Reports the smallest value for the given property over all of the VectorFeatures in the given dataset. String values are compared using case-sensitive lexicographic order as defined in the [Java Documentation]().

gis:property-maximum

gis:property-maximum *VectorDataset property-name*

Reports the largest value for the given property over all of the VectorFeatures in the given dataset. String values are compared using case-sensitive lexicographic order as defined in the [Java Documentation]().

gis:apply-coverage

gis:apply-coverage *VectorDataset property-name patch-variable*

Copies values from the given property of the VectorDataset's features to the given patch variable. The dataset must be a `polygon` dataset; points and lines are not supported.

For each patch, it finds all VectorFeatures that intersect that patch. Then, if the property is a string property, it computes the majority value by computing the total area of the patch covered by VectorFeatures having each possible value of the property, then returning the value which represents the largest proportion of the patch area. If the property is a numeric property, it computes a weighted average of property values from all VectorFeatures which intersect the patch, weighted by the proportion of the patch area they cover.

There are two exceptions to this default behavior:

- If a percentage of a patches' area greater than the coverage-maximum-threshold is covered by a single VectorFeature, then the property value from that VectorFeature is copied directly. If more than one VectorFeature covers a percentage of area greater than the threshold, only the first will be used.
- If the total percentage of a patches' area covered by VectorFeatures is less than the coverage-minimum-threshold, the target patch variable is set to Not A Number.

By default, the minimum threshold is 10% and the maximum threshold is 33%. These values may be modified using the four primitives that follow.

gis:coverage-minimum-threshold

gis:coverage-minimum-threshold

Reports the current coverage minimum threshold used by gis:apply-coverage.

gis:set-coverage-minimum-threshold

gis:set-coverage-minimum-threshold *new-threshold*

Sets the current coverage minimum threshold to be used by gis:apply-coverage.

gis:coverage-maximum-threshold

gis:coverage-maximum-threshold

Reports the current coverage maximum threshold used by gis:apply-coverage.

gis:set-coverage-maximum-threshold

gis:set-coverage-maximum-threshold *new-threshold*

Sets the current coverage maximum threshold to be used by gis:apply-coverage.

gis:intersects?

gis:intersects? *x y*

Reports true if the given objects' spatial representations share at least one point in common, and false otherwise. The objects x and y may be any one of

- a VectorDataset, in which case the object's spatial representation is the union of all the points, lines, or polygons the dataset contains.
- a VectorFeature, in which case the object's spatial representation is defined by the point, line, or polygon the feature contains.
- A turtle, in which case the spatial representation is a point.
- A link, whose spatial representation is a line segment connecting the two points represented by the turtles the link is connecting.
- A patch, whose spatial representation is a rectangular polygon.
- An agentset, whose spatial representation is the union of the representations of all of the agents it contains.
- A list containing of any of the items listed here, including another list. The spatial representation of such a list is the union of the spatial representations of its contents.

gis:contains?

gis:contains? *x y*

Reports true if every point of *y*'s spatial representation is also a part of *x*'s spatial representation. Note that this means that polygons do contain their boundaries. The objects x and y may be any one of

- a VectorDataset, in which case the object's spatial representation is the union of all the points, lines, or polygons the dataset contains.
- a VectorFeature, in which case the object's spatial representation is defined by the point, line, or polygon the feature contains.
- A turtle, in which case the spatial representation is a point.
- A link, whose spatial representation is a line segment connecting the two points represented by the turtles the link is connecting.
- A patch, whose spatial representation is a rectangular polygon.
- An agentset, whose spatial representation is the union of the representations of all of the agents it contains.
- A list containing of any of the items listed here, including another list. The spatial representation of such a list is the union of the spatial representations of its contents.

gis:contained-by?

gis:contained-by? *x y*

Reports true if every point of *x*'s spatial representation is also a part of *y*'s spatial representation. The objects x and y may be any one of

- a VectorDataset, in which case the object's spatial representation is the union of all the points, lines, or polygons the dataset contains.
- a VectorFeature, in which case the object's spatial representation is defined by the point, line, or polygon the feature contains.
- A turtle, in which case the spatial representation is a point.
- A link, whose spatial representation is a line segment connecting the two points represented by the turtles the link is connecting.
- A patch, whose spatial representation is a rectangular polygon.
- An agentset, whose spatial representation is the union of the representations of all of the agents it contains.
- A list containing of any of the items listed here, including another list. The spatial representation of such a list is the union of the spatial representations of its contents.

gis:have-relationship?

gis:have-relationship? *x y relationship*

Reports true if the spatial representations of the two objects have the given spatial relationship, and false otherwise. The spatial relationship is specified using a **Dimensionally Extended Nine- Intersection Model (DE-9IM)** matrix. The matrix consists of 9 elements, each of which specifies the required relationship between the two objects' interior space, boundary space, or exterior space. The elements must have one of six possible values:

- "T", meaning the spaces must intersect in some way
- "F", meaning the spaces must not intersect in any way
- "0", meaning the dimension of the spaces' intersection must be zero (i.e., it must be a

point or non-empty set of points).
- "1", meaning the dimension of the spaces' intersection must be one (i.e., it must be a line or non-empty set of line segments).
- "2", meaning the dimension of the spaces' intersection must be two (i.e., it must be a polygon or set of polygons whose area is greater than zero).
- "*", meaning that the two spaces may have any relationship.

For example, this matrix:

		x		
		Interior	Boundary	Exterior
	Interior	T	*	*
y	Boundary	*	*	*
	Exterior	F	F	*

would return true if and only if some part of object *x*'s interior lies inside object *y*'s interior, and no part of object *x*'s interior or boundary intersects object *y*'s exterior. This is essentially a more restrictive form of the `contains?` primitive; one in which polygons are not considered to contain their boundaries.

The matrix is given to the `have-relationship?` primitive as a string, whose elements are given in the following order:

1	2	3
4	5	6
7	8	9

So to use the example matrix above, you would write:

```
gis:have-relationship? x y "T*****FF*"
```

A much more detailed and formal description of the DE-9IM matrix and the associated point-set theory can be found in the OpenGIS Simple Features Specification for SQL.

The objects x and y may be any one of

- a VectorDataset, in which case the object's spatial representation is the union of all the points, lines, or polygons the dataset contains.
- a VectorFeature, in which case the object's spatial representation is defined by the point, line, or polygon the feature contains.
- A turtle, in which case the spatial representation is a point.
- A link, whose spatial representation is a line segment connecting the two points represented by the turtles the link is connecting.
- A patch, whose spatial representation is a rectangular polygon.
- An agentset, whose spatial representation is the union of the representations of all of the agents it contains.
- A list containing of any of the items listed here, including another list. The spatial representation of such a list is the union of the spatial representations of its contents.

gis:relationship-of

gis:relationship-of *x y*

Reports the **Dimensionally Extended Nine-Intersection Model (DE-9IM)** matrix that describes the spatial relationship of the two objects. The matrix consists of 9 elements, each of which describes the relationship between the two objects' interior space, boundary space, or exterior space. Each element will describe the dimension of the intersection of two spaces, meaning that it may have one of four possible values:

- "-1", meaning the spaces do not intersect
- "0", meaning the dimension of the spaces' intersection is zero (i.e., they intersect at a point or set of points).
- "1", meaning the dimension of the spaces' intersection is one (i.e., they intersect along one or more lines).
- "2", meaning the dimension of the spaces' intersection is two (i.e., their intersection is a non-empty polygon).

For example, the two polygons x and y shown here:

(x)   (y)
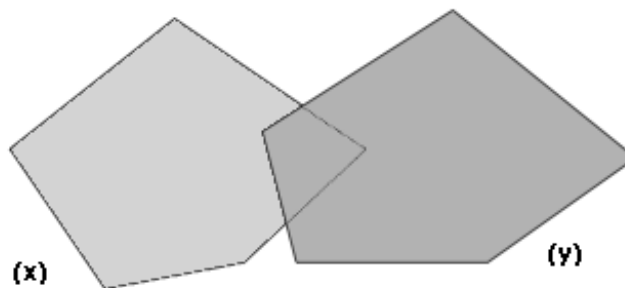
have the following DE-9IM matrix:

		x		
		Interior	Boundary	Exterior
	Interior	2	1	2
y	Boundary	1	0	1
	Exterior	2	1	2

Which would be reported by the `relationship-of` primitive as the string "212101212".

A much more detailed and formal description of the DE-9IM matrix and the associated point-set theory can be found in the OpenGIS Simple Features Specification for SQL.

The objects x and y may be any one of

- a VectorDataset, in which case the object's spatial representation is the union of all the points, lines, or polygons the dataset contains.
- a VectorFeature, in which case the object's spatial representation is defined by the point, line, or polygon the feature contains.
- A turtle, in which case the spatial representation is a point.
- A link, whose spatial representation is a line segment connecting the two points represented by the turtles the link is connecting.
- A patch, whose spatial representation is a rectangular polygon.
- An agentset, whose spatial representation is the union of the representations of all of the agents it contains.
- A list containing of any of the items listed here, including another list. The spatial representation of such a list is the union of the spatial representations of its contents.

gis:intersecting

patch-set gis:intersecting *data*

Reports a new agent set containing only those members of the given agent set which intersect given GIS *data*, which may be any one of: a VectorDataset, a VectorFeature, an Agent, an Agent Set, or a list containing any of the above.

RasterDataset Primitives

gis:width-of

gis:width-of *RasterDataset*

Reports the number of columns in the dataset. Note that this is the number of cells from left to right, not the width of the dataset in GIS space.

gis:height-of

gis:height-of *RasterDataset*

Reports the number of rows in the dataset. Note that this is the number of cells from top to bottom, not the height of the dataset in GIS space.

gis:raster-value

gis:raster-value *RasterDataset x y*

Reports the value of the given raster dataset in the given cell. Cell coordinates are numbered from left to right, and from top to bottom, beginning with zero. So the upper left cell is (0, 0), and the bottom right cell is (`gis:width-of dataset` - 1, `gis:height-of dataset` - 1).

gis:set-raster-value

gis:set-raster-value *RasterDataset x y value*

Sets the value of the given raster dataset at the given cell to a new value. Cell coordinates are numbered from left to right, and from top to bottom, beginning with zero. So the upper left cell is (0, 0), and the bottom right cell is (`gis:width-of dataset` - 1, `gis:height-of dataset` - 1).

gis:minimum-of

gis:minimum-of *RasterDataset*

Reports the highest value in the given raster dataset.

gis:maximum-of

gis:maximum-of *RasterDataset*

Reports the lowest value in the given raster dataset.

gis:sampling-method-of

gis:sampling-method-of *RasterDataset*

Reports the sampling method used to compute the value of the given raster dataset at a single point, or over an area smaller than a single raster cell. Sampling is performed by the GIS extension primitives raster-sample, resample, convolve, and apply-raster. The sampling method will be one of the following:

- `"NEAREST_NEIGHBOR"`: the value of the cell nearest the sampling location is used.
- `"BILINEAR"`: the value of the four nearest cells are sampled by linear weighting, according to their proximity to the sampling site.
- `"BICUBIC"`: the value of the sixteen nearest cells are sampled, and their values are combined by weight according to a piecewise cubic polynomial recommended by Rifman (see *Digital Image Warping*, George Wolberg, 1990, pp 129-131, IEEE Computer Society Press).
- `"BICUBIC_2"`: the value is sampled using the same procedure and the same polynomial as with `BICUBIC` above, but using a different coefficient. This method may produce somewhat sharper results than `BICUBIC`, but that result is data dependent.

For more information on these sampling methods and on raster sampling in general, see this wikipedia article.

gis:set-sampling-method

gis:set-sampling-method *RasterDataset sampling-method*

Sets the sampling method used by the given raster dataset at a single point, or over an area smaller than a single raster cell. Sampling is performed by the GIS extension primitives raster-sample, resample, convolve, and apply-raster. The sampling method must be one of the following:

- `"NEAREST_NEIGHBOR"`
- `"BILINEAR"`
- `"BICUBIC"`
- `"BICUBIC_2"`

See sampling-method-of above for a more specific description of each sampling method.

gis:raster-sample

gis:raster-sample *RasterDataset sample-location*

Reports the value of the given raster over the given location. The location may be any of the following:

- A list of length 2, which is taken to represent a point in netlogo space (`[xcor ycor]`) of the sort reported by [location-of](#) Vertex. The raster dataset is sampled at the point of that location.
- A list of length 4, which is taken to represent an envelope in GIS space, of the sort reported by [envelope-of](#). The raster dataset is sampled over the area of that envelope.
- A patch, in which case the raster dataset is sampled over the area of the patch.
- A turtle, in which case the raster dataset is sampled at the location of that turtle.
- A Vertex, in which case the raster dataset is sampled at the location of that Vertex.

If the requested location is outside the area covered by the raster dataset, this primitive reports the special value representing "not a number", which is printed by NetLogo as "NaN". Using the special "not a number" value as an argument to primitives that expect a number may cause an error, but you can test the value reported by this primitive to filter out "not a number" values. A value that is not a number will be neither less than nor greater than a number value, so you can detect "not a number" values using the following:

```
let value gis:raster-sample dataset turtle 0
; set color to blue if value is a number, red if value is "not a number"
ifelse (value <= 0) or (value >= 0)
[ set color blue ]
[ set color red ]
```

If the requested location is a point, the sample is always computed using the method set by [set-sampling-method](#). If the requested location is an area (i.e., an envelope or patch), the sample is computed by taking the average of all raster cells covered by the requested area.

gis:raster-world-envelope

gis:raster-world-envelope *RasterDataset x y*

Reports the GIS envelope needed to match the boundaries of NetLogo patches with the boundaries of cells in the given raster dataset. This envelope could then be used as an argument to [set-transformation-ds](#).

There may be more cells in the dataset than there are patches in the NetLogo world. In that case, you will need to select a subset of cells in the dataset by specifying which cell in the dataset you want to match with the upper-left corner of the NetLogo world. Cells are numbered from left to right, and from top to bottom, beginning with zero. So the upper left cell is (0, 0), and the bottom right cell is (`gis:width-of dataset` - 1, `gis:height-of dataset` - 1).

gis:create-raster

gis:create-raster *width height envelope*

Creates and reports a new, empty raster dataset with the given number of columns and rows, covering the given envelope.

gis:resample

gis:resample *RasterDataset envelope width height*

Reports a new dataset that consists of the given RasterDataset resampled to cover the given envelope and to contain the given number of columns and rows. If the new raster's cells are smaller than the existing raster's cells, they will be resampled using the method set by set-sampling-method. If the new cells are larger than the original cells, they will be sampled using the `"NEAREST_NEIGHBOR"` method.

gis:convolve

gis:convolve *RasterDataset kernel-rows kernel-columns kernel key-column key-row*

Reports a new raster whose data consists of the given raster convolved with the given kernel.

A convolution is a mathematical operation that computes each output cell by multiplying elements of a kernel with the cell values surrounding a particular source cell. A kernel is a matrix of values, with one particular value defined as the "key element", the value that is centered over the source cell corresponding to the destination cell whose value is being computed.

The values of the kernel matrix are given as a list, which enumerates the elements of the matrix from left to right, top to bottom. So the elements of a 3-by-3 matrix would be listed in the following order:

1	2	3
4	5	6
7	8	9

The key element is specified by column and row within the matrix. Columns are numbered from left to right, beginning with zero. Rows are numbered from top to bottom, also beginning with zero. So, for example, the kernel for the horizontal Sobel operator, which looks like this:

1	0	-1
2	0 (key)	-2
1	0	-1

would be specified as follows:

```
let horizontal-gradient gis:convolve dataset 3 3 [1 0 -1 2 0 -2 1 0 -1] 1 1
```

gis:apply-raster

gis:apply-raster *RasterDataset patch-variable*

Copies values from the given raster dataset to the given patch variable, resampling the raster as necessary so that its cell boundaries match up with NetLogo patch boundaries. This resampling is done as if using resample rather than raster-sample, for the sake of efficiency.

However, patches not covered by the raster are assigned values of "not a number" in the same way that raster-sample reports values for locations outside the raster.

Drawing Primitives

gis:drawing-color

gis:drawing-color

Reports the color used by the GIS extension to draw vector features into the NetLogo drawing layer. Color can be represented either as a NetLogo color (a single number between zero and 140) or an RGB color (a list of 3 numbers). See details in the Colors section of the Programming Guide.

gis:set-drawing-color

gis:set-drawing-color *color*

Sets the color used by the GIS extension to draw vector features into the NetLogo drawing layer. *Color* can be represented either as a NetLogo color (a single number between zero and 140) or an RGB color (a list of 3 numbers). See details in the Colors section of the Programming Guide.

gis:draw

gis:draw *vector-data line-thickness*

Draws the given vector data to the NetLogo drawing layer, using the current GIS drawing color, with the given line thickness. The data may consist either of an entire VectorDataset, or a single VectorFeature. This primitive draws only the boundary of polygon data, and for point data, it fills a circle with a radius equal to the line thickness.

gis:fill

gis:fill *vector-data line-thickness*

Fills the given vector data in the NetLogo drawing layer using the current GIS drawing color, using the given line thickness around the edges. The data may consist either of an entire VectorDataset, or a single VectorFeature. For point data, it fills a circle with a radius equal to the line thickness.

gis:paint

gis:paint *RasterDataset transparency*

Paints the given raster data to the NetLogo drawing layer. The highest value in the dataset is painted white, the lowest is painted in black, and the other values are painted in shades of

gray scaled linearly between white and black.

The *transparency* input determines how transparent the new image in the drawing will be. Valid inputs range from 0 (completely opaque) to 255 (completely transparent).

gis:import-wms-drawing

gis:import-wms-drawing *server-url spatial-reference layers transparency*

Imports an image into the NetLogo drawing layer using the Web Mapping Service protocol, as defined by the Open Geospatial Consortium.

The *spatial reference* and *layers* inputs should be given as strings. The *spatial reference* input corresponds to the **SRS** parameter to the **GetMap** request as defined in section 7.2.3.5 of version 1.1.1 of the WMS standard. The *layers* input corresponds to the **LAYERS** parameter to the as defined in 7.2.3.3 of version 1.1.1 of the WMS standard.

You can find the list of valid spatial reference codes and layer names by examining the response to a **GetCapabilities** request to the WMS server. Consult the relevant standard for instructions on how to issue a **GetCapabilities** request to the server and how to interpret the results.

The *transparency* input determines how transparent the new image in the drawing will be. Valid inputs range from 0 (completely opaque) to 255 (completely transparent).

The NetLogo NW Extension for Network Analysis

This is the new extension for network analysis in NetLogo. It replaces the old Network Extension that came bundled with NetLogo (the old extension is here.) This extension comes bundled with NetLogo as of NetLogo 5.1.

While this extension is stable and we encourage using this instead of the old extension, we're continuing to add new features to it. Users are invited to experiment with it and report any issues they might find here on GitHub or by submitting bugs to bugs@ccl.northwestern.edu. If there are any features you would like to see added, feel free to request them!

The source code for the extension is hosted online at https://github.com/NetLogo/NW-Extension.

A much shorter version of this documentation, that can be useful as a cheat sheet, is available as a PDF file.

Index of Primitives

General

- set-context, get-context, with-context

Path and Distance

- turtles-in-radius, turtles-in-reverse-radius, distance-to, weighted-distance-to, path-to, turtles-on-path-to, weighted-path-to, turtles-on-weighted-path-to, mean-path-length, mean-weighted-path-length

Centrality

- betweenness-centrality, eigenvector-centrality, page-rank, closeness-centrality, weighted-closeness-centrality

Clustering Measures

- clustering-coefficient

Clusterers

- bicomponent-clusters, weak-component-clusters

Cliques

- maximal-cliques, biggest-maximal-cliques

Generators

- generate-preferential-attachment, generate-random, generate-small-world, generate-lattice-2d, generate-ring, generate-star, generate-wheel, generate-wheel-inward, generate-wheel-outward

Import and Export

- save-matrix, load-matrix, save-graphml, load-graphml
- load, load-dl, load-gdf, load-gexf, load-gml, load-vna
- save, save-dl, save-gdf, save-gexf, save-gml, save-vna

Changes

Compared to the previous extension, this new version offers:

- **Improved functionality of existing features**: pathfinding primitives now allow taking edge weights into account.
- **Centrality measures**: calculate the betweenness centrality, closeness centrality, PageRank, and eigenvector centrality of the nodes in your network.
- **Clusterers**: find bicomponent and weak component clusters in your network.
- **Clique finder**: find all maximal cliques or the biggest maximal clique in your network.
- **Generators**: generate many different kinds of networks, namely, preferential attachment, random, small world, 2D lattice, ring, star, and wheel networks.
- **Import/Export**: save and load your networks using plain text matrix files, or export them to GraphML.

To provide some of this functionality, the Network Extension is relying on two external, popular and well-tested network libraries: Jung and JGraphT.

Usage

The first thing that one needs to understand in order to work with the network extension is how to tell the extension

which network to work with. Consider the following example situation:

```
breed [ bankers banker ]
breed [ clients client ]

undirected-link-breed [ friendships friendship ]
directed-link-breed [ accounts account ]
```

Basically, you have bankers and clients. Clients can have accounts with bankers. Bankers can probably have account with other bankers, and anyone can be friends with anyone.

Now we might want to consider this whole thing as one big network. If that is the case, there is nothing special to do: by default, the NW extension primitives consider all turtles and all links to be part of the current network.

We could also, however, be only interested in a subset of the network. Maybe we want to consider only friendship relations. Furthermore, maybe we want to consider only the friendships *between bankers.* After all, having a very high centrality in a network of banker friendships is very different from having a high centrality in a network of client friendships.

To specify such networks, we need to tell the extension*both* which turtles *and* which links we are interested in. All the turtles from the specified set of turtles will be included in the network, and only the links from the specified set of links that are between turtles of the specified set will be included. For example, if you ask for `bankers` and `friendships`, even the lonely bankers with no friends will be included, but friendship links between bankers and clients will **not** be included. The way to tell the extension about this is with the `nw:set-context` primitive, which you must call *prior* to doing any operations on a network.

Some examples:

- `nw:set-context turtles links` will give you everything: bankers and clients, friendships and accounts, as one big network.
- `nw:set-context turtles friendships` will give you all the bankers and clients and friendships between any of them.
- `nw:set-context bankers friendships` will give you all the bankers, and only friendships between bankers.
- `nw:set-context bankers links` will give you all the bankers, and any links between them, whether these links are friendships or accounts.
- `nw:set-context clients accounts` will give you all the clients, and accounts between each other, but since in our fictional example clients can only have accounts with bankers, this will be a completely disconnected network.

Special agentsets vs normal agentsets

It must be noted that NetLogo has two types of agentsets that behave slightly differently, and that this has an impact on the way `nw:set-context` works. We will say a few words about these concepts here but, for a thorough understanding, it is highly recommended that you read the section on agentsets in the NetLogo programming guide

The "special" agentsets in NetLogo are`turtles`, `links` and the different "breed" agentsets. What is special about them is that they can grow: if you create a new turtle, it will be added to the `turtles` agentset. If you have a `bankers` breed and you create a new banker, it will be added to the `bankers` agentset and to the`turtles` agentset. Same goes for links. Other agentsets, such as those created with the `with` primitive (e.g., `turtles with [ color = red ]`) or the`turtle-set` and `link-set` primitives) are never added to. The content of normal agentsets will only change if the agents that they contain die.

To show how different types of agentsets interact with`nw:set-context`, let's create a very simple network:

```
clear-all
create-turtles 3 [ create-links-with other turtles ]
```

Let's set the context to`turtles` and `links` (which is the default anyway) and use`nw:get-context` to see what we have:

```
nw:set-context turtles links
show map sort nw:get-context
```

We get all three turtles and all three links:

```
[[(turtle 0) (turtle 1) (turtle 2)] [(link 0 1) (link 0 2) (link 1 2)]]
```

Now let's kill one turtle:

```
ask one-of turtles [ die ]
show map sort nw:get-context
```

As expected, the context is updated to reflect the death of the turtle and of the two links that died with it:

```
[[(turtle 0) (turtle 1)] [(link 0 1)]]
```

What if we now create a new turtle?

```
create-turtles 1
show map sort nw:get-context
```

Since our context is using the special `turtles` agentset, the new turtle is automatically added:

```
[[(turtle 0) (turtle 1) (turtle 3)] [(link 0 1)]]
```

Now let's demonstrate how it works with normal agentsets. We start over with a new network of red turtles:

```
clear-all
create-turtles 3 [
  create-links-with other turtles
  set color red
]
```

And we set the context to `turtles with [ color = red ])` and `links`

```
nw:set-context (turtles with [ color = red ]) links
show map sort nw:get-context
```

Since all turtles are red, we get everything in our context:

```
[[(turtle 0) (turtle 1) (turtle 2)] [(link 0 1) (link 0 2) (link 1 2)]]
```

But what if we ask one of them to turn blue?

```
ask one-of turtles [ set color blue ]
show map sort nw:get-context
```

No change. The agentset used in our context remains unaffected:

```
[[(turtle 0) (turtle 1) (turtle 2)] [(link 0 1) (link 0 2) (link 1 2)]]
```

If we kill one of them, however...

```
ask one-of turtles [ die ]
show map sort nw:get-context
```

It gets removed from the set:

```
[[(turtle 0) (turtle 2)] [(link 0 2)]]
```

What if we add a new red turtle?

```
create-turtles 1 [ set color red ]
show map sort nw:get-context
```

Nope:

```
[[(turtle 0) (turtle 2)] [(link 0 2)]]
```

A final note regarding the different types of agentsets: because of how they are implemented in NetLogo and handled in the NW extension, it is likely that special agentsets will perform better for some of the extension's primitives. So, if you can, prefer a context referring to `turtles` and `links` or simple breeds, instead of agentsets built with `with`, `turtle-set`, etc.

Primitives

General

set-context

```
nw:set-context turtleset linkset
```

Specifies the set of turtles and the set of links that the extension will consider to be the current graph. All the turtles from *turtleset* and all the links from *linkset* that connect two turtles from *turtleset* will be included.

This context is used by all other primitives (unless specified otherwise) until a new context is specified. (At the moment, only the [generator primitives](#) and the file input primitives (`nw:load-matrix` and `nw:load-graphml`) are exceptions to this rule.)

See [the usage section](#) for a much more detailed explanation of `nw:set-context`.

get-context

```
nw:get-context
```

Reports the content of the current graph context as a list containing two agentsets: the agentset of turtles that are part of the context and the agentset of links that are part of the context.

Let's say with start with a blank slate and the default context consisting of `turtles` and `links`, `nw:get-context` will report a list the special `turtles` and `links` breed agentsets:

```
observer> clear-all
observer> show nw:get-context
observer: [turtles links]
```

If we add some turtles and links to our context, we'll still see the same thing, even though `turtles` and `links` have internally grown:

```
observer> crt 2 [ create-links-with other turtles ]
observer> show nw:get-context
observer: [turtles links]
```

If you had set your context to normal agentsets instead (built with `turtle-set`, `link-set` or `with`) here is what you would see:

```
observer> clear-all
observer> nw:set-context turtle-set turtles link-set links
observer> show nw:get-context
observer: [(agentset, 0 turtles) (agentset, 0 links)]
```

If you then create new turtles and links, they are not added to the context because normal agentsets don't grow (see [Special agentsets vs normal agentsets](#)):

```
observer> crt 2 [ create-links-with other turtles ]
observer> show nw:get-context
observer: [(agentset, 0 turtles) (agentset, 0 links)]
```

But if you construct new agentsets and set the context to them, your new agents will be there:

```
observer> nw:set-context turtle-set turtles link-set links
observer> show nw:get-context
observer: [(agentset, 2 turtles) (agentset, 1 link)]
```

If you want to see the actual content of your context, it is easy to turn your agentsets into lists that can be nicely displayed. Just use a combination of [map](#) and [sort](#):

```
observer> show map sort nw:get-context
observer: [[(turtle 0) (turtle 1)] [(link 0 1)]]
```

Finally, you can use `nw:get-context` to store a context that you eventually want to restore:

```
extensions [ nw ]
to store-and-restore-context
```

```
      clear-all
      crt 2 [
        set color red
        create-links-with other turtles with [ color = red ] [
          set color yellow
        ]
      ]
      crt 2 [
        set color blue
        create-links-with other turtles with [ color = blue ] [
          set color green
        ]
      ]
      nw:set-context turtles with [ color = red ] links with [ color = yellow ]
      show map sort nw:get-context
      let old-turtles item 0 nw:get-context
      let old-links item 1 nw:get-context
      nw:set-context turtles with [ color = blue ] links with [ color = green ]
      show map sort nw:get-context
      nw:set-context old-turtles old-links
      show map sort nw:get-context
    end
```

Here is the result:

```
    observer> store-and-restore-context
    observer: [[(turtle 0) (turtle 1)] [(link 0 1)]]
    observer: [[(turtle 2) (turtle 3)] [(link 2 3)]]
    observer: [[(turtle 0) (turtle 1)] [(link 0 1)]]
```

with-context

```
nw:with-context turtleset linkset command-block
```

Executes the *command-block* with the context temporarily set to *turtleset* and *linkset*. After *command-block* finishes running, the previous context will be restored.

For example:

```
    observer> create-turtles 3 [ create-links-with other turtles ]
    observer> nw:with-context (turtle-set turtle 0 turtle 1) (link-set link 0 1) [ show nw:get-context ]
    observer: [(agentset, 2 turtles) (agentset, 1 link)
    observer> show nw:get-context
    observer: [turtles links]
```

If you have NW extension code running in two forever buttons or `loop` blocks that each need to use different contexts, you should use `nw:with-context` in each to make sure they are operating in the correct context.

Path and Distance

turtles-in-radius, turtles-in-reverse-radius

🧩 nw:turtles-in-radius *radius*

🧩 nw:turtles-in-reverse-radius *radius*

Returns the set of turtles within the given distance (number of links followed) of the calling turtle in the current context. Both forms include the calling turtle, whom you can exclude with `other` if need be.

The `turtles-in-radius` form will follow both undirected links and directed **out** links. The `turtles-in-reverse-radius` form will follow both undirected links and directed **in** links. You can think of `turtles-in-radius` as "turtles **who I can get to** in *radius* steps" and of `turtles-in-reverse-radius` as "turtles **who can get to me** in *radius* steps".

If you want the primitive to follow only undirected links or only directed links, you can do it by setting the context appropriately. For example: `nw:set-context turtles undir-links` (assuming `undir-links` is an undirected link breed) or `nw:set-context turtles dir-links` (assuming `dir-links` is a directed link breed).

Example:

```
    clear-all
    create-turtles 5
    ask turtle 0 [ create-link-with turtle 1 ]
    ask turtle 0 [ create-link-with turtle 2 ]
    ask turtle 1 [ create-link-with turtle 3 ]
    ask turtle 2 [ create-link-with turtle 4 ]
    ask turtle 0 [
      show sort nw:turtles-in-radius 1
```

```
    ]
```

Will output:

```
(turtle 0): [(turtle 0) (turtle 1) (turtle 2)]
```

As you may have noticed, the result includes the calling turtle. This mimics the behavior of the regular NetLogo `in-radius` primitive.

distance-to, weighted-distance-to

🧩 nw:distance-to *target-turtle*

🧩 nw:weighted-distance-to *target-turtle weight-variable-name*

Finds the shortest path to the target turtle and reports the total distance for this path, or false if no path exists in the current context.

The `nw:distance-to` version of the primitive assumes that each link counts for a distance of one. The `nw:weighted-distance-to` version accepts a *weight-variable-name* parameter, which must be **a string** naming the link variable to use as the weight of each link in distance calculations. The weights cannot be negative numbers.

Example:

```
links-own [ weight ]
to go
  clear-all
  create-turtles 5
  ask turtle 0 [ create-link-with turtle 1 [ set weight 2.0 ] ]
  ask turtle 1 [ create-link-with turtle 2 [ set weight 2.0 ] ]
  ask turtle 0 [ create-link-with turtle 3 [ set weight 0.5 ] ]
  ask turtle 3 [ create-link-with turtle 4 [ set weight 0.5 ] ]
  ask turtle 4 [ create-link-with turtle 2 [ set weight 0.5 ] ]
  ask turtle 0 [ show nw:distance-to turtle 2 ]
  ask turtle 0 [ show nw:weighted-distance-to turtle 2 "weight" ]
end
```

Will output:

```
(turtle 0): 2
(turtle 0): 1.5
```

path-to, turtles-on-path-to, weighted-path-to, turtles-on-weighted-path-to

🧩 nw:path-to *target-turtle*

🧩 nw:turtles-on-path-to *target-turtle*

🧩 nw:weighted-path-to *target-turtle weight-variable-name*

🧩 nw:turtles-on-weighted-path-to *target-turtle weight-variable-name*

Finds the shortest path to the target turtle and reports the actual path between the source and the target turtle. The `nw:path-to` and `nw:weighted-path-to` variants will report the list of links that constitute the path, while the `nw:turtles-on-path-to` and `nw:turtles-on-weighted-path-to` variants will report the list of turtles along the path, including the source and destination turtles.

As with the link distance primitives, the `nw:weighted-path-to` and `nw:turtles-on-weighted-path-to` accept a *weight-variable-name* parameter, which must be **a string** naming the link variable to use as the weight of each link in distance calculations. The weights cannot be negative numbers.

If no path exist between the source and the target turtles, all primitives will report an empty list.

Note that the NW-Extension remembers paths that its calculated previously unless the network changes. Thus, you don't need to store paths to efficiently move across the network; you can just keep re-calling one of the path primitives. If the network changes, however, the stored answers are forgotten.

Example:

```
links-own [ weight ]
to go
  clear-all
  create-turtles 5
  ask turtle 0 [ create-link-with turtle 1 [ set weight 2.0 ] ]
  ask turtle 1 [ create-link-with turtle 2 [ set weight 2.0 ] ]
  ask turtle 0 [ create-link-with turtle 3 [ set weight 0.5 ] ]
  ask turtle 3 [ create-link-with turtle 4 [ set weight 0.5 ] ]
  ask turtle 4 [ create-link-with turtle 2 [ set weight 0.5 ] ]
  ask turtle 0 [ show nw:path-to turtle 2 ]
  ask turtle 0 [ show nw:turtles-on-path-to turtle 2 ]
  ask turtle 0 [ show nw:weighted-path-to turtle 2 "weight" ]
  ask turtle 0 [ show nw:turtles-on-weighted-path-to turtle 2 "weight" ]
end
```

Will output:

```
(turtle 0): [(link 0 1) (link 1 2)]
(turtle 0): [(turtle 0) (turtle 1) (turtle 2)]
(turtle 0): [(link 0 3) (link 3 4) (link 2 4)]
(turtle 0): [(turtle 0) (turtle 3) (turtle 4) (turtle 2)]
```

mean-path-length, mean-weighted-path-length

`nw:mean-path-length`

`nw:mean-weighted-path-length` *weight-variable-name*

Reports the average shortest-path length between all distinct pairs of nodes in the current context. If the `nw:mean-weighted-path-length` is used, the distances will be calculated using *weight-variable-name*. The weights cannot be negative numbers.

Reports false unless paths exist between all pairs.

Example:

```
links-own [ weight ]
to go
  clear-all
  create-turtles 3
  ask turtle 0 [ create-link-with turtle 1 [ set weight 2.0 ] ]
  ask turtle 1 [ create-link-with turtle 2 [ set weight 2.0 ] ]
  show nw:mean-path-length
  show nw:mean-weighted-path-length "weight"
  create-turtles 1 ; create a new, disconnected turtle
  show nw:mean-path-length
  show nw:mean-weighted-path-length "weight"
end
```

Will ouput:

```
observer: 1.3333333333333333
observer: 2.6666666666666665
observer: false
observer: false
```

Centrality

betweenness-centrality

🧩 `nw:betweenness-centrality`

To calculate the betweenness centrality of a turtle, you take every other possible pairs of turtles and, for each pair, you calculate the proportion of shortest paths between members of the pair that passes through the current turtle. The betweenness centrality of a turtle is the sum of these.

As of now, link weights are not taken into account.

eigenvector-centrality

🧩 `nw:eigenvector-centrality`

The Eigenvector centrality of a node can be thought of as the amount of influence a node has on a network. In practice,

turtles that are connected to a lot of other turtles that are themselves well-connected (and so on) get a higher Eigenvector centrality score.

Eigenvector centrality is only defined for connected networks, and the primitive will report `false` for disconnected graphs. (Just like `distance-to` does when there is no path to the target turtle.)

In this implementation, the eigenvector centrality is normalized such that the highest eigenvector centrality a node can have is 1. This implementation is designed to agree with Gephi's implementation out to at least 3 decimal places. If you discover that it disagrees with Gephi on a particular network, please report it.

The primitive respects link direction, even in mixed-directed networks. This is the one place where it should disagree with Gephi; Gephi refuses to treat directed links as directed in mixed-networks.

As of now, link weights are not taken into account.

page-rank

🧩 `nw:page-rank`

The page rank of a node can be thought of as the proportion of time that an agent walking forever at random on the network would spend at this node. The agent has an equal chance of taking any of a nodes edges, and will jump around the network completely randomly 15% of the time. In practice, like with eigenvector centrality, turtles that are connected to a lot of other turtles that are themselves well-connected (and so on) get a higher page rank.

Page rank is one of the several algorithms that search engines use to determine the importance of a website.

The sum of all page rank values should be approximately one. Unlike eigenvector centrality, page rank is defined for all networks, no matter the connectivity. Currently, it treats all links as undirected links.

As of now, link weights are not taken into account.

closeness-centrality

🧩 `nw:closeness-centrality`

The closeness centrality of a turtle is defined as the inverse of the average of it's distances to all other turtles. (Some people use the sum of distances instead of the average, but the extension uses the average.)

Note that this primitive reports the *intra-component* closeness of a turtle, that is, it takes into account only the distances to the turtles that are part of the same component as the current turtle, since distance to turtles in other components is undefined. The closeness centrality of an isolated turtle is defined to be zero.

weighted-closeness-centrality

🧩 `nw:weighted-closeness-centrality` *weight-variable-name*

This is identical to closeness-centrality, except that weights provided by the given variable are treated as the distances of links.

Clustering Measures

clustering-coefficient

🧩 `nw:clustering-coefficient`

Reports the local clustering coefficient of the turtle. The clustering coefficient of a node measures how connected its neighbors are. It is defined as the number of links between the node's neighbors divided by the total number of possible links between its neighbors.

`nw:clustering-coefficient` takes the directedness of links into account. A directed link counts as a single link whereas an undirected link counts as two links (one going one-way, one going the other).

The global clustering coefficient measures how much nodes tend to cluster together in the network in general. It is defined based on the types of triplets in the network. A triplet consists of a central node and two of its neighbors. If its neighbors are also connected, it's a closed triplet. If its neighbors are not connected, it's an open triplet. The global clustering coefficient is simply the number of closed triplets in a network divided by the total number of triplets. It can be calculated from the local clustering coefficient quite easily with the following code

```
to-report global-clustering-coefficient
  let closed-triplets sum [ nw:clustering-coefficient * count my-links * (count my-links - 1) ] of turtle
  let triplets sum [ count my-links * (count my-links - 1) ] of turtles
  report closed-triplets / triplets
end
```

Note that the above will only work with the default context, and may need to tweaked if you've set the turtles or links in the network to something other than `turtles` and `links`.

The average local clustering coefficient is another popular method for measuring the amount of clustering in the network as a whole. It may be calculated with

```
mean [ nw:clustering-coefficient ] of turtles
```

Clusterers

bicomponent-clusters

`nw:bicomponent-clusters`

Reports the list of bicomponent clusters in the current network context. A bicomponent (also known as a maximal biconnected subgraph) is a part of a network that cannot be disconnected by removing only one node (i.e. you need to remove at least two to disconnect it). The result is reported as a list of agentsets, in random order. Note that one turtle can be a member of more than one bicomponent at once.

weak-component-clusters

`nw:weak-component-clusters`

Reports the list of "weakly" connected components in the current network context. A weakly connected component is simply a group of nodes where there is a path from each node to every other node. A "strongly" connected component would be one where there is a *directed* path from each node to every other. The extension does not support the identification of strongly connected components at the moment.

The result is reported as a list of agentsets, in random order. Note that one turtle *cannot* be a member of more than one weakly connected component at once.

Cliques

maximal-cliques

`nw:maximal-cliques`

A clique is a subset of a network in which every node has a direct link to every other node. A maximal clique is a clique that is not, itself, contained in a bigger clique.

The result is reported as a list of agentsets, in random order. Note that one turtle can be a member of more than one maximal clique at once.

The primitive uses the Bron–Kerbosch algorithm and only works with undirected links.

biggest-maximal-cliques

`nw:biggest-maximal-cliques`

The biggest maximal cliques are, as the name implies, the biggest cliques in the current context. Often, more than one clique are tied for the title of biggest clique, so the result is reported as a list of agentsets, in random order. If you want only one clique, use `one-of nw:biggest-maximal-cliques`.

The primitive uses the Bron–Kerbosch algorithm and only works with undirected links.

Generators

The generators are amongst the only primitives that do not operate on the current network context. Instead, all of them take a turtle breed and a link breed as inputs and generate a new network using the given breeds.

generate-preferential-attachment

`nw:generate-preferential-attachment` `turtle-breed link-breed nb-nodes optional-command-block`

Generates a new network using the Barabási–Albert algorithm. This network will have the property of being "scale free": the distribution of degrees (i.e. the number of links for each turtle) should follow a power law.

In this version of the primitive, turtles are added, one by one, each forming one link to a previously added turtle, until *nb-nodes* is reached. The more links a turtle already has, the greater the probability that new turtles form links with it when they are added. Future versions of the primitive might provide more flexibility in the way the network is generated.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-preferential-attachment turtles links 100 [ set color red ]
```

generate-random

`nw:generate-random` `turtle-breed link-breed nb-nodes connection-probability optional-command-block`

Generates a new random network of *nb-nodes* turtles in which each one has a *connection-probability* (between 0 and 1) of being connected to each other turtles. The algorithm uses the *G(n, p)* variant of the Erdős–Rényi model.

The algorithm is $O(n^2)$ for directed networks and $O(n^2/2)$ for undirected networks, so generating more than a couple thousand nodes will likely take a very long time.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-random turtles links 100 0.5 [ set color red ]
```

generate-small-world

`nw:generate-small-world` `turtle-breed link-breed row-count column-count clustering-exponent is-toroidal optional-command-block`

Generates a new small-world network using the Kleinberg Model.

The algorithm proceeds by generating a lattice of the given number of rows and columns (the lattice will wrap around itself if *is-toroidal* is `true`). The "small world effect" is created by adding additional links between the nodes in the lattice. The higher the *clustering-exponent*, the more the algorithm will favor already close-by nodes when adding new links. A clustering exponent of `2.0` is typically used.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-small-world turtles links 10 10 2.0 false [ set color red ]
```

generate-lattice-2d

`nw:generate-lattice-2d` `turtle-breed link-breed row-count column-count is-toroidal optional-command-block`

Generates a new 2D lattice network (basically, a grid) of *row-count* rows and *column-count* columns. The grid will wrap around itself if *is-toroidal* is `true`.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-lattice-2d turtles links 10 10 false [ set color red ]
```

generate-ring

`nw:generate-ring` `turtle-breed link-breed nb-nodes optional-command-block`

Generates a ring network of *nb-nodes* turtles, in which each turtle is connected to exactly two other turtles.

The number of nodes must be at least three.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-ring turtles links 100 [ set color red ]
```

generate-star

```
nw:generate-star turtle-breed link-breed nb-nodes optional-command-block
```

Generates a [star network](#) in which there is one central turtle and every other turtle is connected only to this central node. The number of turtles can be as low as one, but it won't look much like a star.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-star turtles links 100 [ set color red ]
```

generate-wheel, generate-wheel-inward, generate-wheel-outward

```
nw:generate-wheel turtle-breed link-breed nb-nodes optional-command-block
```

```
nw:generate-wheel-inward turtle-breed link-breed nb-nodes optional-command-block
```

```
nw:generate-wheel-outward turtle-breed link-breed nb-nodes optional-command-block
```

Generates a [wheel network](#), which is basically a [ring network](#) with an additional "central" turtle that is connected to every other turtle.

The number of nodes must be at least four.

The `nw:generate-wheel` only works with undirected link breeds. The `nw:generate-wheel-inward` and `nw:generate-wheel-outward` versions only work with directed *link-breed*. The `inward` and `outward` part of the primitive names refer to the direction that the "spokes" of the wheel point to relative to the central turtle.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:generate-wheel turtles links 100 [ set color red ]
```

Import and Export

save-matrix

```
nw:save-matrix file-name
```

Saves the current network, as defined by `nw:set-context`, to *file-name*, as a text file, in the form of a simple connection matrix.

Here is, for example, a undirected ring network with four nodes:

```
0.00 1.00 0.00 1.00
1.00 0.00 1.00 0.00
0.00 1.00 0.00 1.00
1.00 0.00 1.00 0.00
```

And here is the directed version:

```
0.00 1.00 0.00 0.00
0.00 0.00 1.00 0.00
0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00
```

At the moment, `nw:save-matrix` does not support link weights. Every link is represented as a "1.00" in the connection matrix. This will change in a future version of the extension.

load-matrix

```
nw:load-matrix file-name turtle-breed link-breed optional-command-block
```

Generates a new network according to the connection matrix saved in *file-name*, using *turtle-breed* and *link-breed* to create the new turtles and links.

At the moment, `nw:load-matrix` does not support link weights.

Please be aware that the breeds that use use to load the matrix may be different from those that you used when you saved it.

For example:

```
extensions [ nw ]
directed-link-breed [ dirlinks dirlink ]
to go
  clear-all
  crt 5 [ create-dirlinks-to other turtles ]
  nw:set-context turtles dirlinks
  nw:save-matrix "matrix.txt"
  clear-all
  nw:load-matrix "matrix.txt" turtles links
  layout-circle turtles 10
end
```

...*will* give you back **undirected** links, even if you saved directed links into the matrix.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:load-matrix "matrix.txt" turtles links [ set color red ]
```

save-graphml

`nw:save-graphml` *file-name*

You can save the current graph to GraphML. The following NetLogo code:

```
extensions [ nw ]

breed [ bankers banker ]
bankers-own [ bank-name ]
breed [ clients client ]
clients-own [ hometown ]

undirected-link-breed [ friendships friendship ]

directed-link-breed [ accounts account ]
accounts-own [ amount ]

to go
  clear-all
  create-bankers 1 [
    set bank-name "The Bank"
  ]
  create-clients 1 [
    set hometown "Turtle City"
    create-friendship-with banker 0
    create-account-to banker 0 [
      set amount 9999.99
    ]
  ]
  nw:set-context turtles links
  nw:save-graphml "example.graphml"
end
```

Will produce the following GraphML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml">
<key id="PEN-MODE" for="node" attr.name="PEN-MODE" attr.type="string"/>
<key id="YCOR" for="node" attr.name="YCOR" attr.type="double"/>
<key id="PEN-SIZE" for="node" attr.name="PEN-SIZE" attr.type="double"/>
<key id="LABEL" for="node" attr.name="LABEL" attr.type="string"/>
<key id="SHAPE" for="node" attr.name="SHAPE" attr.type="string"/>
<key id="BREED" for="node" attr.name="BREED" attr.type="string"/>
<key id="WHO" for="node" attr.name="WHO" attr.type="double"/>
<key id="HIDDEN?" for="node" attr.name="HIDDEN?" attr.type="boolean"/>
<key id="LABEL-COLOR" for="node" attr.name="LABEL-COLOR" attr.type="double"/>
<key id="HEADING" for="node" attr.name="HEADING" attr.type="double"/>
<key id="BANK-NAME" for="node" attr.name="BANK-NAME" attr.type="string"/>
<key id="HOMETOWN" for="node" attr.name="HOMETOWN" attr.type="string"/>
<key id="COLOR" for="node" attr.name="COLOR" attr.type="double"/>
<key id="XCOR" for="node" attr.name="XCOR" attr.type="double"/>
<key id="SIZE" for="node" attr.name="SIZE" attr.type="double"/>
<key id="END1" for="edge" attr.name="END1" attr.type="string"/>
<key id="TIE-MODE" for="edge" attr.name="TIE-MODE" attr.type="string"/>
<key id="END2" for="edge" attr.name="END2" attr.type="string"/>
<key id="LABEL-COLOR" for="edge" attr.name="LABEL-COLOR" attr.type="double"/>
<key id="THICKNESS" for="edge" attr.name="THICKNESS" attr.type="double"/>
```

```
<key id="LABEL" for="edge" attr.name="LABEL" attr.type="string"/>
<key id="SHAPE" for="edge" attr.name="SHAPE" attr.type="string"/>
<key id="BREED" for="edge" attr.name="BREED" attr.type="string"/>
<key id="COLOR" for="edge" attr.name="COLOR" attr.type="double"/>
<key id="AMOUNT" for="edge" attr.name="AMOUNT" attr.type="double"/>
<key id="HIDDEN?" for="edge" attr.name="HIDDEN?" attr.type="boolean"/>
<graph edgedefault="undirected">
<node id="client 1">
<data key="PEN-MODE">up</data>
<data key="YCOR">0</data>
<data key="PEN-SIZE">1</data>
<data key="LABEL"></data>
<data key="SHAPE">default</data>
<data key="BREED">clients</data>
<data key="WHO">1</data>
<data key="HIDDEN?">false</data>
<data key="LABEL-COLOR">9.9</data>
<data key="HEADING">356</data>
<data key="HOMETOWN">Turtle City</data>
<data key="COLOR">115</data>
<data key="XCOR">0</data>
<data key="SIZE">1</data>
</node>
<node id="banker 0">
<data key="PEN-MODE">up</data>
<data key="YCOR">0</data>
<data key="PEN-SIZE">1</data>
<data key="LABEL"></data>
<data key="SHAPE">default</data>
<data key="BREED">bankers</data>
<data key="WHO">0</data>
<data key="HIDDEN?">false</data>
<data key="LABEL-COLOR">9.9</data>
<data key="HEADING">32</data>
<data key="BANK-NAME">The Bank</data>
<data key="COLOR">85</data>
<data key="XCOR">0</data>
<data key="SIZE">1</data>
</node>
<edge source="client 1" target="banker 0">
<data key="END1">(client 1)</data>
<data key="TIE-MODE">none</data>
<data key="END2">(banker 0)</data>
<data key="LABEL-COLOR">9.9</data>
<data key="THICKNESS">0</data>
<data key="LABEL"></data>
<data key="SHAPE">default</data>
<data key="BREED">accounts</data>
<data key="COLOR">5</data>
<data key="AMOUNT">9999.99</data>
<data key="HIDDEN?">false</data>
</edge>
<edge source="banker 0" target="client 1">
<data key="END1">(banker 0)</data>
<data key="TIE-MODE">none</data>
<data key="END2">(client 1)</data>
<data key="LABEL-COLOR">9.9</data>
<data key="THICKNESS">0</data>
<data key="LABEL"></data>
<data key="SHAPE">default</data>
<data key="BREED">friendships</data>
<data key="COLOR">5</data>
<data key="HIDDEN?">false</data>
</edge>
</graph>
</graphml>
```

A few things to notice:

- The breed is stored as data field, both for nodes and edges.
- The data includes both NetLogo's internal variables and the variables that were defined as either `breeds-own`, `turtles-own`, `linkbreeds-own` or `links-own`.
- Each key gets an `attr.type` based on the actual types of the values contained in the agent variables. The three possible types are `"string"`, `"double"` and `"boolean"`. To determine the attribute type of a particular agent variable, the extension will look at the first agent in the graph. To see which agent is first, you can look at the result of `nw:get-context`. Note that variables containing other types of values, such as turtles, patches, lists, etc., will be stored as strings.
- This example only has a directed link, and you will notice the `<graph edgedefault="directed">` element. If we had only undirected links, we would have `<graph edgedefault="undirected">`. What if we try to mix both kinds of link? At the moment, the extension will save such a "mixed" graph as if it were an undirected graph (see this issue for more details). The order of the `source` and `target` will be respected, however, so if you know which breeds represent directed links, you can figure it out *a posteriori*.

load-graphml

```
nw:load-graphml file-name optional-command-block
```

Loading a GraphML file into NetLogo with the network extension should be as simple as calling `nw:load-graphml "example.graphml"`, but there is a bit of preparation involved.

The key idea is that `nw:load-graphml` will try to assign the attribute values defined in the GraphML file to NetLogo agent variables of the same names (this is *not* case sensitive). The first one it tries to set is `breed` if it is there, so the turtle or link will get the right breed and, hence, the right breed variables.

One special case is the `who` number, which is ignored by the importer if it is present as a GraphML attribute: NetLogo does not allow you to modify this number once a turtle is created and, besides, there could already be an existing turtle with that number.

The simplest case to handle is when the original GraphML file has been saved from NetLogo by using `nw:save-graphml`. In this case, all you should have to do is to make sure that you have the same breed and variables definition as when you saved the file and you should get back your original graph. For example, if you want to load the file from the `nw:save-graphml` example above, you should have the following definitions:

```
breed [ bankers banker ]
bankers-own [ bank-name ]
breed [ clients client ]
clients-own [ hometown ]

undirected-link-breed [ friendships friendship ]

directed-link-breed [ accounts account ]
accounts-own [ amount ]
```

Loading a graph that was saved from a different program than NetLogo is quite possible as well, but it may take a bit of tinkering to get all the attribute-variable match up right. If you encounter major problems, please do not hesitate to open an issue.

The extension will try to assign the type defined by `attr.type` to each variable that it loads. If it's unable to convert it to that type, it will load it as a string. If `attr.type` is not defined, or is set to an unknown value, the extension will first try to load the value as a double, then try it as a boolean, and finally fall back on a string.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. For example:

```
nw:load-graphml "example.graphml" [ set color red ]
```

Note that this command block can be used to build a list or an agentset containing the newly created nodes:

```
let node-list []
nw:load-graphml "example.graphml" [
  set node-list lput self node-list
]
let node-set turtle-set node-list
```

load

nw:load *file-name default-turtle-breed default-link-breed optional-command-block*

nw:load-dl *file-name default-turtle-breed default-link-breed optional-command-block*

nw:load-gdf *file-name default-turtle-breed default-link-breed optional-command-block*

nw:load-gexf *file-name default-turtle-breed default-link-breed optional-command-block*

nw:load-gml *file-name default-turtle-breed default-link-breed optional-command-block*

nw:load-vna *file-name default-turtle-breed default-link-breed optional-command-block*

Import the given file into NetLogo. Like `nw:load-graphml`, the importer will do its best to match node and edge attributes in the file with turtle and link variables in NetLogo. If `breed` is specified for nodes and edges in the file and exists in NetLogo, it will be used. Otherwise, the default turtle and link breeds are used.

Limitations:

- Multigraphs are not supported in importing. Even if the file format supports it (and many don't), only the first link will be used on import. This is due to a limitation in the parsing libraries NW uses. `nw:load-graphml` does support multigraphs with the normal NetLogo limitation that two turtles can share more than one link only if all the links are of different breeds.

nw:load determines the file-type of given file based on the extension and calls the corresponding `load-*` primitive on it. Note that GraphML must be imported with `nw:load-graphml`.

save

nw:save *file-name default-turtle-breed default-link-breed optional-command-block*

```
nw:save-dl file-name default-turtle-breed default-link-breed optional-command-block
```

```
nw:save-gdf file-name default-turtle-breed default-link-breed optional-command-block
```

```
nw:save-gexf file-name default-turtle-breed default-link-breed optional-command-block
```

```
nw:save-gml file-name default-turtle-breed default-link-breed optional-command-block
```

```
nw:save-vna file-name default-turtle-breed default-link-breed optional-command-block
```

Export the network context in the given format to the given file. Turtle and link attributes will be exported to formats that support node and edge properties.

Limitations:

- x and y (not xcor and ycor) can only be numbers. x and y are commonly used in formats pertaining to position and behind the scenes NW uses Gephi's libraries for exporting. Furthermore, x and y will be added even if they didn't exist in the model. Again, this is because NW uses Gephi's libraries which assume that nodes have positions stored in x and y. If you wish to export to Gephi specifically, we recommend creating x and y turtles variables and setting them to xcor and ycor before export.
- Color will be exported in a standard RGB format. This should hopefully increase compatibility with other programs.
- Turtle and link variables that contain values of different types will be stored as strings. Unfortunately, most network formats require that node and attributes have a single type.
- Many programs use label to store the id of nodes. Thus, if you're having trouble importing data exported from NetLogo into another program, you might try setting turtles' labels to their who number.
- Multigraphs are not supported. Thus, two turtles can share at most one link. nw:save-graphml does support multigraphs, so use that if turtles can have more than one type of link connecting them.

nw:save determines the file-type of the given file based on the extension and calls the corresponding save-* primitive on it. Note that GraphML must be exported with nw:save-graphml.

A note regarding floating point calculations

Neither JGraphT nor Jung, the two network libraries that we use internally, use strictfp floating point calculations. This does mean that exact reproducibility of results involving floating point calculations *between different hardware architectures* is not fully guaranteed. (NetLogo itself always uses strict math so this only applies to some primitives of the NW extension.)

Performance

In order to be fast in as many circumstances as possible, the NW extension tries hard to never calculate things twice. It remembers all paths, distances, and centralities that it calculates. So, while the first time you ask for the distance between turtle 0 and turtle 3782 may take some time, after that, it should be almost instantaneous. Furthermore, it keeps track of values it just happened to calculate along the way. For example, if turtle 297 is closer to turtle 0 than turtle 3782 is, it may just happen to figure out the distance between turtle 0 and turtle 297 while it figures out the distance between turtle 0 and turtle 3782. It will remember this value, so that if you ask it for the distance between turtle 0 and turtle 297, it doesn't have to do all that work again.

There are a few circumstances where the NW extension has to forget things. If the network changes at all (you add turtles or links, or remove turtles or links), it has to forget everything. For weighted primitives, if the value of the weight variable changes for any of the links in the network, it will forget the values associated with that weight variable.

If you're working on a network that can change regularly, try to do all your network calculations at once, then all your network changes at once. The more your interweave network calculations and network changes, the more the NW extension will have to recalculate things. For example, if you have a traffic model, and cars need to figure out the shortest path to their destination based on the traffic each tick, have all the cars find their shortest paths, then change the network weights to account for how traffic has changed.

There may be rare occasions in which you don't want the NW extension to remember values. For example, if you're working on an extremely large network, remembering all those values may take more memory than you have. In that case, you can just call nw:set-context (first nw:get-context) (last nw:get-context) to force the NW extension to immediately forget everything.

Using the extension with applets

If you want to use the extension with applets, you will find that the distributed nw.zip contains a folder named alternate-netlogolite with NetLogoLite.jar and NetLogoLite.jar.pack.gz. You should use these *instead* of the jar files that come with the regular NetLogo distribution. (Reasons for this are explained here.) Applet support is, however, brittle and experimental, and might not be supported in the final release of the extension. Nonetheless, please report issues if you find some.

General information on applets is available in the NetLogo documentation.

Building

The extension is written in Scala (version 2.9.2).

Run `./sbt package` to build the extension.

Unless they are already present, sbt will download the needed Jung and JGraphT jar files from the Internet.

If the build succeeds, `nw.jar` will be created. To use the extension, this file and all the other jars will need to be in the `extensions/nw` folder under your NetLogo installation.

Terms of Use

Jung is licensed under the BSD license and JGraphT is licensed under the LGPL license.

CF-Extension

This NetLogo extension adds somewhat experimental control-flow primitives. This extension is a place where we can try out new control-flow structures before possibly moving them into core NetLogo. In particular, the CF extension currently includes primitives that allow you to do things similar to `if-else if-else` chains you see in other languages, as well as things similar to `switch`es in other languages. However, it does it in a more flexible way than many languages. A few quick examples to get you started:

```
let x 5
let y 7
cf:when
cf:case [ x > y ] [ print "x is bigger than y!" ]
cf:case [ x < y ] [ print "x is less than y!" ]
cf:else           [ print "x is the same as y!" ]
```

```
let x 5
cf:match x
cf:case [ ? > 7 ] [ print "x is greater than 7!" ]
cf:case [ ? < 3 ] [ print "x is less than 3!" ]
cf:else           [ print "x is somewhere in between 3 and 7!" ]
```

Cases

Central to this extension is the concept of a case. A case is simply a list of two elements, where the first element is a reporter and the second element is either a reporter or a command. The first element, called the condition, *must* report either `true` or `false`. The second argument is called the consequent. If you're curious about how CF works, keep reading this section. Otherwise, feel free to skip to the list of primitives; you don't need to understand the internals to start using CF.

Almost all primitives in the CF extension take a list of cases as an argument. Typically, they go through the cases, looking for a true condition. When they find one, they then run the consequent. Although CF provides primitives that make constructing a list of cases easy, you could actually just build such a list with primitives already in NetLogo, like so:

```
let x 5
let y 7
let list-of-cases (list
  (list task [ x > y ] task [ print "x is greater than y!" ])
  (list task [ x < y ] task [ print "x is less than y!" ])
  (list task [ true  ] task [ print "x is the same as y!" ])
)
```

However, this is not very nice looking. With CF, you can write it like this instead:

```
let x 5
let y 7

let list-of-cases
cf:case [ x > y ] [ print "x is greater than y!" ]
cf:case [ x < y ] [ print "x is less than y!" ]
cf:else           [ print "x is the same as y!" ]
```

Primitives

when

```
cf:when <list of cases>
```

`cf:when` runs the command task from the first case in the list with a true condition. For instance:

```
let x 3
cf:when
cf:case [ x < 2 ] [ print "x is less than 2!" ]
cf:case [ x < 4 ] [ print "x is less than 4!" ]
cf:case [ x < 6 ] [ print "x is less than 6!" ]
cf:else           [ print "x is greater than or equal to 6!" ]
```

The above code will print out `x is less than 4!` since that's the first case with a true condition.

If no true case is found, and no `cf:else` given, `cf:when` will error with a suggestion for a fix.

select

```
cf:select <list of cases>
```

`cf:select` picks the first case in the list with a true condition and reports the result of its consequent. The consequents of the cases in a `cf:select` must be reporter tasks. Thus, `cf:select` is exactly like `cf:when`, except that it reports the value from the true case, rather than just running it. For example:

```
let x 3
print cf:select
cf:case [ x < 2 ] [ "x is less than 2!" ]
cf:case [ x < 4 ] [ "x is less than 4!" ]
cf:case [ x < 6 ] [ "x is less than 6!" ]
cf:else           [ "x is greater than or equal to 6!" ]
```

The above code will print out `x is less than 4!` since that's the first case with a true condition.

If no true case is found, and no `cf:else` given, `cf:select` will error with a suggestion for a fix.

match

```
cf:match <value> <list of cases>
```

`cf:match` is like `cf:when`, except that it applies the conditions in its cases to the given value. For instance:

```
let x 3
cf:match x
cf:case [ ? < 2 ] [ print "x is less than 2!" ]
cf:case [ ? < 4 ] [ print "x is less than 4!" ]
cf:case [ ? < 6 ] [ print "x is less than 6!" ]
cf:else           [ print "x is greater than or equal to 6!" ]
```

The above code will print out `x is less than 4!` since that's the first case with a true condition.

The value is also passed to the consequent of the case. For instance:

```
cf:match one-of turtles
cf:case [ [color] of ? = red  ] [ ask ? [ show "I'm red!" ] ]
cf:case [ [color] of ? = blue ] [ ask ? [ show "I'm blue!" ] ]
cf:else                         [ ask ? [ show "I'm some other color!" ] ]
```

If no matching case is found, `cf:match` will error with a suggestion for a fix.

matching

```
<value> cf:matching  <list of cases>
```

`cf:matching` is like `cf:match`, except that it reports the result of the matching case. `cf:matching` is to `cf:match` as `cf:select` is to `cf:when`. For instance:

```
let x 3
print x cf:matching
cf:case [ ? < 2 ] [ "x is less than 2!" ]
cf:case [ ? < 4 ] [ "x is less than 4!" ]
cf:case [ ? < 6 ] [ "x is less than 6!" ]
cf:else           [ "x is greater than or equal to 6!" ]
```

The above code will print out `x is less than 4!` since that's the first case with a true condition. `cf:matching` also applies the consequent of the matching case to the given value, just like `cf:match`:

```
print (one-of turtles) cf:matching
cf:case [ [color] of ? = red  ] [ [ "I'm red!" ] of ? ]
cf:case [ [color] of ? = blue ] [ [ "I'm blue!" ] of ? ]
cf:else                         [ [ "I'm some other color!" ] of ? ]
```

If no matching case is found, `cf:matching` will error with a suggestion for a fix.

case

```
cf:case <reporter task> <reporter task or command task> <list of remaining cases>
```

`cf:case` allows you to construct a list of cases that the other primitives will then pick from. It constructs a new case from the the two given tasks and adds it to the front of the list of remaining cases. Thus, you can chain it together with other instances of `cf:case` to create an arbitrarily long list of cases.

Note that because the condition in a case is just a reporter task, you can check for many common conditions in a very concise manner. For instance, if we want to do something depending on the breed of a turtle, you can do:

```
cf:match my-turtle
cf:case is-wolf? [ show "Growl!" ]
cf:case is-a-sheep? [ show "Baah!" ]
cf:case is-dog? [ show "Bark!" ]
cf:case is-cat? [ show "Meow!" ]
cf:else [ show "I'm not sure what sound to make..." ]
```

case-is

`cf:case-is <reporter task> <value> <reporter task or command task> <list of remaining cases>`

`cf:case-is` allows you to write some common uses of `cf:case` in `cf:match` or `cf:matching` in a more concise, readable way. The given reporter task should be a relationship such as `=`, `<`, or `member?`. `cf:case-is` then fills in the second argument of the task with the given value. This is much easier to understand in an example:

```
let x 5
print x cf:matching
cf:case-is = 0 [ "x is 0!" ]
cf:case-is = 1 [ "x is 1!" ]
cf:case-is > 2 [ "x is greater than 2!" ]
cf:case-is member? [ -1 -2 -3] [ "x is either -1, -2, or -3" ]
cf:else [ "x is something else" ]
```

Thus, `cf:case-is` allows you to do something quite similar to `switch` in some other languages, but is also much more flexible.

Internally, `cf:case-is` is equivalent to the following:

```
cf:case [(runresult <reporter task> ? <value>)] <reporter task or command task> <list of remaining cases>
```

else

`cf:else <reporter task or command task>`

`cf:else` creates a case where the condition is always true. Thus, it allows you to create a case that will be run if all the other cases fail. You should almost always finish up a chain of cases with `cf:else`. However, if you'd prefer to error rather than have a default case, you can replace `cf:else` with `[]`, like so:

```
let x -5
cf:when
cf:case [ 0 < x and x < 10 ] [ print "x is between 0 and 10!" ]
cf:case [ x < 100 ] [ print "x is less than 100!" ]
[]
```

The above code will error, since no matching case will be found.

CSV-Extension

This NetLogo extension adds CSV parsing capabilities to models.

Common use cases and examples

Read a file all at once

Just use `csv:from-file "/path/to/myfile.csv"`! See [from-file](#) for more information.

Read a file one line at a time

For really big files, you may not want to store the entire file in memory, but rather just process it a line at a time. For instance, if you want to sum each of the columns of a numeric CSV file, you can do:

```
to-report sum-columns [ file ]
  file-open file
  set result csv:from-row file-read-line
  while [ not file-at-end? ] [
    let row csv:from-row file-read-line
    set result (map [?1 + ?2] result row)
  ]
  file-close
  report result
end
```

You can also use this technique to...

Read a file one line per tick

Here's an example model that reads in a file one line per tick:

```
globals [ data ]

to setup
  clear-all
  file-close-all ; Close any files open from last run
  file-open "data.csv"
  ; other setup goes here
  reset-ticks
end

to go
  if file-at-end? [ stop ]
  set data csv:from-row file-read-line
  ; model update goes here
  tick
end
```

Write a file

Just use `csv:to-file "/path/to/myfile.csv" my-data`! See [to-file](#) for more information.

Primitives

Reading

from-row

```
csv:from-row <string>
```

```
(csv:from-row <string> <delimiter>)
```

Parses the given string as though it were a row from a CSV file and returns it as a list of values. For example:

```
observer> show csv:from-row "one,two,three"
observer: ["one" "two" "three"]
```

Quotes can be used when items contain commas:

```
observer> show csv:from-row "there's,a,comma,\"in,here\""
observer: ["there's" "a" "comma" "in,here"]
```

You can put two quotes in a row to put an actual quote in an entry. If the entry is not quoted, you can just use one quote:

```
observer> foreach (csv:from-row "he said \"hi there\",\"afterwards, she said \"\"hello\"\"\"") print
he said "hi there"
afterwards, she said "hello"
```

Number-like-entries will be parsed as numbers:

```
observer> show csv:from-row "1,-2.5,1e3"
observer: [1 -2.5 1000]
```

`true` and `false` with any capitalization will be parsed as booleans:

```
observer> show csv:from-row "true,TRUE,False,falsE"
observer: [true true false false]
```

To use a different delimiter, you can specify a second, optional argument. Only single character delimiters are supported:

```
observer> show (csv:from-row "1;2;3" ";")
observer: [1 2 3]
```

Different types of values can be mixed freely:

```
observer> show csv:from-row "one,2,true"
observer: ["one" 2 true]
```

from-string

`csv:from-string <string>`

`(csv:from-string <string> <delimiter>)`

Parses a string representation of one or more CSV rows and returns it as a list of lists of values. For example:

```
observer> show csv:from-string "1,two,3\nfour,5,true"
observer: [[1 "two" 3] ["four" 5 true]]
```

from-file

`csv:from-file <string>`

`(csv:from-file <string> <delimiter)`

Parses an entire CSV file to a list of lists of values. For example, if we have a file `example.csv` that contains:

```
1,2,3
4,5,6
7,8,9
10,11,12
```

Then, we get:

```
observer> show csv:from-file "example.csv"
observer: [[1 2 3] [4 5 6] [7 8 9] [10 11 12]]
```

The parser doesn't care if the rows have different numbers of items on them. The number of items in the rows list will always be `<number of delimiters> + 1`, though blank lines are skipped. This makes handling files with headers quite easy. For instance, if we have `header.csv` that contains:

```
My Data
2/1/2015

Parameters:
start,stop,resolution,population,birth?
0,4,1,100,true

Data:
time,x,y
0,0,0
1,1,1
2,4,8
3,9,27
```

This gives:

```
observer> foreach csv:from-file "header.csv" show
observer: ["My Data"]
observer: ["2/1/2015"]
observer: ["Parameters:"]
observer: ["start" "stop" "resolution" "population" "birth?"]
observer: [0 4 1 100 true]
observer: ["Data:"]
observer: ["time" "x" "y"]
observer: [0 0 0]
observer: [1 1 1]
observer: [2 4 8]
observer: [3 9 27]
```

Writing

to-row

`csv:to-row <list>`

`(csv:to-row <list> <delimiter>)`

Reports the given list as a CSV row. For example:

```
observer> show csv:to-row ["one" 2 true]
observer: "one,2,true"
```

to-string

`csv:to-string <list>`

`(csv:to-string <list> <delimiter>)`

Reports the given list of lists as a CSV string. For example:

```
observer> show csv:to-string [[1 "two" 3] [4 5]]
observer: "1,two,3\n4,5"
```

to-file

`csv:to-file <string> <list>`

`(csv:to-file <string> <list> <delimiter>)`

Writes the given list of lists to a new CSV file. For example:

```
observer> csv:to-file "myfile.csv" [[1 "two" 3] [4 5]]
```

will result in a file `myfile.csv` containing:

```
1,two,3
4,5
```

NetLogo Palette Extension

The NetLogo palette extension allows to map values to colors. The colors go beyond NetLogo colors, including ColorBrewer color schemes or arbitrary RGB colors. Additionally, it provides a primitive to map to color gradients and a primitive to launch a ColorBrewer dialog for easy scheme selection.

Getting Started

To get started with palettes add to the top of your Code tab:

```
extensions [palette]
```

you can then call any of the primitives by adding`palette:` before the primitive:

```
palette:scale-gradient
palette:scale-scheme

palette:scheme-color
palette:scheme-dialog
```

The palette extension primitives return a list containing RGB colors `[[r g b][r g b]...[r g b]]`, except for `palette:scheme-dialog` which opens a dialog.

What colors should I use ?

ColorBrewer has many colors where to start. ColorBrewer has three schemes Sequential, Divergent and Qualitative. The use of ColorBrewer for maps is discussed at length in this paper (Harrower, Brewer 2003). Choosing the right colors is a design problem, thus, there are many acceptable solution. However, these guidelines might be useful for choosing colors in Agent Based Models:

- Sequential colors are best for continuous natural phenomena models such as as heat diffusion in physics or fire in earth sciences.
- Divergent colors are useful for highlighting a middle value in a model. It can be also applied to the heat diffusion model if the goal is to highlight the middle temperature.
- Qualitative colors are best for choosing colors in models where color denotes category and not value.
- For agents that cover large areas avoid strong colors and try to use pastel colors. However, for a low number of small isolated agents try to use strong colors such as such a accent.
- The main goal is to avoid having a large area covered with agents with a bright color and or having small areas having a muted pastel color.
- If you are coloring both turtles and patches, make sure they have different ranges of hue, saturation and value. E.g. Use different hues of pastel for patches and accent for turtles

Should I use a continuous color gradient or just a discrete color set ?

The answer depends on the task that your will be asking from your user.

For example, gradients are more aesthetic thus are more memorable than discrete colors. Consequently, a gradient can be a better choice for presentations where the main goal of the image is to be attractive and memorable. However, binning values in a discrete set of colors simplifies tasks such as estimation and counting by removing unnecessary detail to display the big picture. Thus, discrete colors can be a better choice for a paper where the user will have the time and interest to study the visualization.

In order to see the difference you can turn on and off the gradient in the Heat Diffusion model. You can observe that turning gradient on makes the model more aesthetic, but it becomes harder to estimate the value of a patch at a given position.

Example Models

There is an example of using the palette primitives in the Code Examples section of the models library:

- Palette Example

And one Sample Model that uses the extension:

- Heat Diffusion - Alternative Gradient

Documentation

Color mapping using NetLogo Built-in Primitives Colors mapping using NetLogo Palette Extension

Be sure to check to ColorBrewer web page. To get a deeper understanding of how to use the color schemes read the ColorBrewer paper (Harrower, Brewer 2003)

Palette Extension Dictionary

palette:scale-gradient

`scale-gradient` *rgb-color-list number range1 range2*

Reports an RGB color proportional to *number* using a gradient generated with *rgb-color-list*. An *rgb-color-list* consist of a list containing RGB list with three values between 0 and 255: [[r1 g1 b1] [r2 g2 b2] [r3 g3 b3] ...]

If *range1* is less than *range2*, the color will be directly mapped to gradient colors. While, if *range2* is less than *range1*, the color gradient is inverted.

If *number* is less than *range1*, then the first color of is *RGB-color-list* is chosen.

If *number* is grater than *range2*, then the last color of is *RGB-color-list* is chosen.

```
ask patches
[
  set pcolor palette:scale-gradient [[255 0 0] [0 0 255]] pxcor min-pxcor max-pxcor
]

;; colors each patch with a color proportional to the gradient
```

palette:scale-scheme

`scale-scheme` *scheme-type scheme-color number-of-classes number range1 range2*

Reports an RGB color proportional to *number* using the color brewer schemes. It takes six arguments the first three arguments define the ColorBrewer legend. Fir the user should select a scheme-type which can be "Sequential", "Divergent, Qualitative". Then it should select a variety of scheme-colors which depending on the scheme-color can have names such as "Reds", "Divergent", "Set1". Finally the user should select the number of classes with a minimum of 3 and a maximum between 9 and 11. For more information go to http://www.colorbrewer.org or consult the scheme-dialog primitive.

If *range1* is less than *range2*, the color will be directly mapped to scheme colors. While, if *range2* is less than *range1*, the color scheme selection is inverted.

If *number* is less than *range1*, then the first color of the resulting ColorBrewer legend is chosen.

If *number* is grater than *range2*, then the last color of the resulting ColorBrewer legend is chosen.

```
ask patches
[
  set pcolor palette:scale-scheme [[255 0 0] [0 0 255]] pxcor min-pxcor max-pxcor
]

;; colors each patch with a color from the Color Brewer Schemes
```

palette:scheme-colors

`scheme-colors` *scheme-type scheme-color number-of-classes*

report a list of RGB colors with the size specified in the a number of classes

```
show palette:scheme-colors "Divergent" "Spectral" 3
=> [[252 141 89] [255 255 191] [153 213 148]]


; The schemes-color primitive can be used with the scale-gradient primitive
ask patches
  [set pcolor palette:scale-gradient palette:scheme-colors "Divergent" "Spectral" 9 pxcor min-pxcor max-pxcor
```

palette:scale-gradient

`scale-gradient` *rgb-color-list number range1 range2*

Reports an RGB color proportional to number using a gradient generated with *rgb-color-list*. An *rgb-color-list* consist of a list containing RGB list with three values between 0 and 255: [[r1 g1 b1] [r2 g2 b2] [r3 g3 b3] ...]

If *range1* is less than *range2*, the color will be directly mapped to gradient colors. While, if *range2* is less than *range1*, the color gradient is inverted.

If *number* is less than *range1*, then the first color of is *RGB-color-list* is chosen.

If *number* is grater than *range2*, then the last color of is *RGB-color-list* is chosen.

```
ask patches
[
  set pcolor palette:scale-gradient [[255 0 0] [0 0 255]] pxcor min-pxcor max-pxcor
]

;; colors each patch with a color proportional to the gradient
```

References

ColorBrewer http://www.colorbrewer.org

HARROWER, M. and C. BREWER (2003). ColorBrewer: An online tool for selecting color schemes for maps. The Cartographic Journal 40(1): 27-37.)

HEALEY, C G (2006) Perception in Visualization, (comprehensive review updated regularly).

HEALEY, C G, BOOTH K S, and ENNS, J T (1995). Visualizing Real-Time Multivariate Data Using Preattentive Processing ACM Transactions on Modeling and Computer Simulation 5, 3, 190-221.

TUFTE, E (1983) The Visual Display of Quantitative Information , Graphics Press.

WARE, C (2004) Information Visualization, 2nd Ed., Morgan Kaufmann. Feedback

Terms of Use

FAQ (Frequently Asked Questions)

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. (See Contacting Us.)

Questions

General

- **Why is it called NetLogo?**
- **How do I cite NetLogo or HubNet in a publication?**
- **How do I cite a model from the Models Library in a publication?**
- **Where and when was NetLogo created?**
- **What programming language was NetLogo written in?**
- **What's the difference between StarLogo and NetLogo?**
- **Under what license is NetLogo released? Is the source code available?**
- **Do you offer any workshops or other training opportunities for NetLogo?**
- **Are there any NetLogo textbooks?**
- **Is NetLogo available in other languages besides English?**
- **Is NetLogo compiled or interpreted?**
- **Has anyone built a model of <x>?**
- **Are NetLogo models runs scientifically reproducible?**
- **Will NetLogo and NetLogo 3D remain separate applications?**
- **Can I run NetLogo on my tablet or phone?**

Downloading

- **Can I have multiple versions of NetLogo installed at the same time?**
- **I'm on a UNIX system and I can't untar the download. Why?**
- **How do I install NetLogo unattended?**
- **On Windows, how do I configure the Java installation that the without-Java installer uses?**

Applets

- **Should I use applets?**
- **Why don't NetLogo applets work for me?**

Running

- **Can I run NetLogo from a CD, a network drive, or a USB drive?**
- **Why is NetLogo so much slower when I unplug my Windows laptop?**
- **Why does NetLogo bundle Java?**
- **How come NetLogo won't start up on my Linux machine?**
- **When I try to start NetLogo on Windows I get an error "The JVM could not be started". Help!**
- **Can I run NetLogo from the command line, without the GUI?**
- **Does NetLogo take advantage of multiple processors/cores?**
- **Can I distribute NetLogo model runs across a cluster or grid of computers?**
- **Is there any way to recover lost work if NetLogo crashes or freezes?**

Usage

- **When I move the speed slider all the way to the right, why does my model seem to stop?**
- **Can I use the mouse to "paint" in the view?**
- **How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?**

- [Can I use GIS data in NetLogo?](#)
- [My model runs slowly. How can I speed it up?](#)
- [Can I have more than one model open at a time?](#)
- [Can I change the choices in a chooser on the fly?](#)
- [Can I divide the code for my model up into several files?](#)

Programming

- [How does the NetLogo language differ from other Logos?](#)
- [How come my model from an earlier NetLogo doesn't work right?](#)
- [How do I take the negative of a number?](#)
- [My turtle moved forward 1, but it's still on the same patch. Why?](#)
- [How do I keep my turtles on patch centers?](#)
- [`patch-ahead 1` is reporting the same patch my turtle is already standing on. Why?](#)
- [How do I give my turtles "vision"?](#)
- [Can agents sense what's in the drawing layer?](#)
- [I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?](#)
- [The documentation says that `random-float 1.0` might return 0.0 but will never return 1.0. What if I want 1.0 to be included?](#)
- [How can I use different patch "neighborhoods" (circular, Von Neumann, Moore, etc.)?](#)
- [How can I keep two turtles from occupying the same patch?](#)
- [How can I find out if a turtle is dead?](#)
- [Does NetLogo have arrays?](#)
- [Does NetLogo have hash tables or associative arrays?](#)
- [How can I convert an agentset to a list, or vice versa?](#)
- [How do I stop foreach?](#)
- [I'm trying to make a list. Why do I keep getting the error "Expected a constant"?](#)

BehaviorSpace

- [Why are the rows in my BehaviorSpace table results out of order?](#)
- [How do I gather data every n ticks?](#)
- [I'm varying a global variable I declared in the Code tab, but it doesn't work. Why?](#)
- [Why are some of my results cut off in Excel?](#)

NetLogo 3D

- [Can I make a 3D applet?](#)
- [Does NetLogo 3D work with my stereoscopic device?](#)

Extensions

- [I'm writing an extension. Why does the compiler say it can't find org.nlogo.api?](#)

General

Why is it called NetLogo?

The "Logo" part is because NetLogo is a dialect of the Logo language.

"Net" is meant to evoke the decentralized, interconnected nature of the phenomena you can model with NetLogo, including network phenomena. It also refers to HubNet, the multiuser participatory simulation environment included in NetLogo.

How do I cite NetLogo or HubNet in a publication?

If you use or refer to NetLogo, HubNet or a model from the NetLogo models library, we ask that you cite it as follows:

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

HubNet: Wilensky, U. & Stroup, W., 1999. HubNet. http://ccl.northwestern.edu/netlogo/hubnet.html. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

How do I cite a model from the Models Library in a publication?

The correct citation is included in the "Credits and References" section of each model's Info tab.

Where and when was NetLogo created?

NetLogo was first created in 1999 by Uri Wilensky at the Center for Connected Learning and Computer-Based Modeling, then at Tufts University in the Boston area. NetLogo grew out of StarLogoT, which was authored by Wilensky in 1997. In 2000, the CCL moved to Northwestern University, in the Chicago area. NetLogo 1.0 came out in 2002, 2.0 in 2003, 3.0 in 2005, 4.0 in 2007, 4.1 in 2009, and 5.0 in 2012.

What programming language was NetLogo written in?

NetLogo is written mostly in Scala, with some parts in Java. (Scala code compiles to Java byte code and is fully interoperable with Java and other JVM languages.)

What's the relationship between StarLogo and NetLogo?

The original StarLogo began at the MIT Media Lab in 1989 and ran on the Connection Machine. Later versions were developed for Macintosh computers: MacStarLogo (1994, MIT) and StarLogoT (1997, Tufts).

Today there are two StarLogo descendants under active development: NetLogo (from the CCL at Northwestern University) and StarLogo TNG (from MIT). NetLogo is the most widely used agent-based modeling environment in both education and research. StarLogo TNG is distinguished by its programming interface based on colored blocks.

Under what license is NetLogo released? Is the source code available?

NetLogo is free, open source software under the GPL (GNU General Public License), version 2, or (at your option) any later version.

Commercial licenses are also available. To inquire about commercial licenses, please contact Uri Wilensky at uri@northwestern.edu.

The source code is on GitHub, here. Development discussion is on the netlogo-devel group.

The User Manual is published under a Creative Commons Attribution-ShareAlike license (CC BY-SA 3.0).

Source code for all of the extensions bundled with NetLogo is on GitHub, here. Most of the extensions are in the public domain (CC0 notice). Other extensions are released under open source licenses. See each extension's README for details.

The Code Examples in the Models Library are in the public domain (CC0 notice).

The rest of the models in the Models Library are provided under a variety of licenses. Some are public domain and some are open source, but most are under the Creative Commons Attribution-ShareAlike license (CC BY-NC-SA), which is not an open source license, though the models are free for noncommercial distribution and use.

See each model's Info tab to check its particular license.

The models are in a public Git repository here.

Do you offer any workshops or other training opportunities for NetLogo?

We offer workshops from time to time. If a workshop has been scheduled, we will announce it on the NetLogo Users Group.

Are there any NetLogo textbooks?

The CCL has published a textbook (written by the author of NetLogo) that gives an introduction to agent-based modeling methods using NetLogo. It goes step by step with coding examples how to design, build, revise, and analyze models. And it presents some advanced techniques.

See www.intro-to-abm.com for more information on that textbook.

See the Textbooks section of our Resources page.

We at the CCL have hoped to write several more NetLogo textbooks. These could be aimed at different audiences, such as: middle school, high school, undergraduate course in modeling or complexity, practical guide for interested adults.

Unfortunately, we have not yet been able to find the time to make these happen. If anyone from the user community would like to collaborate on such a venture, please let us know. We would welcome it.

Is NetLogo available in other languages besides English?

Volunteers have translated the user manual into Chinese and Czech. The translated versions are available from the NetLogo web site.

So far, the NetLogo user interface has been localized in English, Spanish, Chinese, and Russian. All four languages are included in the standard download.

We are seeking volunteers to complete and improve these localizations and to translate the NetLogo software and manual into as many other languages as possible. If you're able to help in this endeavor, please contact us.

Is NetLogo compiled or interpreted?

Short answer: some of both.

Long answer: NetLogo does include a compiler that generates Java byte code. However, this compiler does not yet support the entire language, so some parts of user code remain interpreted. Note that our compiler generates Java byte code, and Java virtual machines have "just-in-time" compilers that in turn compile Java byte code all the way to native code, so much user code is ultimately translated to native code.

Has anyone built a model of <x>?

Try looking at the NetLogo Models Library, the NetLogo Modeling Commons, our Community Models page, and our list of references to NetLogo in outside works.

You might also ask the question on the NetLogo Users Group and/or search past messages on the group.

Are NetLogo models runs scientifically reproducible?

Yes. NetLogo's pseudorandom number generator and agent scheduling algorithms are deterministic, and NetLogo always uses Java's "strict math" library, which gives bit-for-bit identical results regardless of platform. But keep the following cautions in mind:

- If your model uses random numbers, then in order to get reproducible behavior, you must use the `random-seed` command to set the random seed in advance, so that your model will receive the exact same sequence of random numbers every time. Remember that agentsets are always in random order, so anything you do with agentsets uses random numbers.
- If your model uses the `every` or `wait` commands in such a way that affects the outcome of the model, then you may get different results on different computers, or even on the same computer, since the model may run at a different speed.
- In order to reproduce model runs exactly, you must be using the exact same version of NetLogo. The details of the agent scheduling mechanism and the random number generator may change between NetLogo versions, and other changes (bugfixes in the engine, language changes, and so forth) may also affect the behavior of your model. (Then again, they may not.)
- We have expended every effort to make NetLogo model runs fully reproducible, but of course this can never truly be an iron-clad guarantee, due not only to the possibility of random hardware failure, but also the possibility of human error in the design of: your model, NetLogo, your Java VM, your hardware, and so on.

Will NetLogo and NetLogo 3D remain separate?

For now, yes. NetLogo 3D is included with NetLogo, but it is still a separate application.

Ideally a single unified application would support both 2D and 3D modeling. We would design the 3D world support so it doesn't get in the way when you are building 2D models. Models built in NetLogo 3D might require changes in order to run in the hypothetical unified application.

Can I run NetLogo on my phone or tablet?

No. Neither iOS, nor Android, nor Windows RT supports running Java applications such as NetLogo.

We are working on an alternate implementation of NetLogo on a JavaScript and HTML5 base, instead of Java. It will work on a variety of tablets and phones. We don't know yet when it will be ready, and we expect that for a long time it will only support a subset of the features in desktop NetLogo. The many person-years of development effort that have gone into the Java version can't cheaply or easily be replicated on another platform.

For technical details on this new project, go [here](#).

Downloading

Can I have multiple versions of NetLogo installed at the same time?

Yes. When you install NetLogo, the folder that is created contains has the version number in its name, so multiple versions can coexist.

On Windows systems, whichever version you installed last will be the version that opens when you double click a model file in Windows Explorer. On Macs, you can control what version opens via "Get Info" in the Finder.

I'm on a UNIX system and I can't untar the download. Why?

Some of the files in the tarball have long pathnames, too long for the standard tar format. You must use the GNU version of tar instead (or another program which understands the GNU tar extensions). On some systems, the GNU version of tar is available under the name "gnutar". You can find out if you are already using the GNU version by typing `tar --version` and seeing if the output says "tar (GNU tar)".

How do I install NetLogo unattended

It depends on which platform you are using.

- **Linux:** Untar NetLogo into the appropriate place.

  Mac: Copy the NetLogo directory from the disk image into the Applications folder.

- **Windows:**

  Run the installer from the command line using the -q option:

  ```
  NetLogo5.3.1Installer.exe -q
  ```

On Windows, how do I configure the Java installation that the without-Java installer uses?

The without-Java installer automatically searches for appropriate Java installations on your machine. If you wish to direct NetLogo to a particular installation you may do so by setting the environment variable NETLOGO_JAVA to the full path of the installation.

- Go to the Start Menu -> Control Panel -> System.
- Switch to the "Advanced" panel
- Press the "Environment Variables" button.

- Add NETLOGO_JAVA as a variable and set the value to the path of the Java installation.

Applets

Should I use applets?

No. We don't recommend it. [Details here](#).

Why don't NetLogo applets work for me?

In Java 7 Update 51, released in 2014, Oracle changed applet security requirements such that NetLogo applets no longer work for most people. [Details here](#).

Running

Can I run NetLogo from a CD, a network drive, or a USB drive?

Yes. NetLogo runs fine from any file system, including read-only file systems.

Why is NetLogo so much slower when I unplug my Windows laptop?

Your computer is switching to power saving mode when unplugged. It's normal for this to reduce speed a little, but unfortunately there is a bug in Java that drastically slows down Swing applications, including NetLogo.

One workaround is to change the power settings on your computer so it doesn't go into power saving mode when you unplug it. (If you do this, your battery won't last as long.)

Another workaround is to run NetLogo with an option recommended by Oracle, by editing the NetLogo 5.3.1.vmoptions file, found in the NetLogo directory (under Program Files on your hard drive, unless you installed NetLogo in a different location). Add on a new line:

```
-Dsun.java2d.ddoffscreen=false
```

You can see the details of the Java bug and vote for Oracle to fix it [here](#).

Why does NetLogo bundle Java?

For backwards compatibility, NetLogo 5.0.0-5.2.1 on Mac OS X required that Java 6 from Apple be installed. As OS X continued to update, java 6 was removed from the default distribution and only available for download from Apple, adding to the complexity of installing NetLogo.

The decision was made to bundle Java 8 with 5.3, reducing installation complexity and hopefully promoting stability. To minimize any disruption, the upgrade NetLogo 5.3 avoided any changes not directly related to bundling java 8. If you're running 5.2.1 happily on Java 6, or stuck on a version of OS X older than 10.8.3, there are **no** features that you need to upgrade in order to get.

How come NetLogo won't start up on my Linux machine?

We bundle Oracle's Java runtime when using NetLogo on Linux. If you would like to change the version of linux used, you can modify the .cfg files to point to a different version of java (see How big can my model be? for more information).

In theory, any Java 6 or later runtime will run NetLogo. In practice, some Java implementations aren't high enough quality. Recent versions of OpenJDK should work; older ones may not. GNU libgcj does not work.

Ubuntu users should consult http://help.ubuntu.com/community/Java.

When I try to start NetLogo on Windows I get an error "The JVM could not be started". Help!

A nearly certain fix is to use a text editor to edit the NetLogo.cfg file, changing 1024m to a smaller number, like 512m. This should permit NetLogo to start, although the lower heap size limit may affect your ability to run models with many agents. See How big can my model be? for information on model size and how to edit the cfg file.

If running with the lower heap size limit is unacceptable, read on.

Some Windows systems have trouble allocating large amounts of contiguous virtual memory. Upgrading to a newer version of Windows may help.

Running Windows in 64-bit mode instead of 32-bit mode may also help. Double check that Windows is actually running in 64-bit mode; see Microsoft's FAQ page on 64-bit Windows.

Can I run NetLogo from the command line, without the GUI?

Yes. The easiest way is to set up your model run or runs as a BehaviorSpace experiment. No additional programming is required. See the BehaviorSpace Guide for details.

Another option is to use our Controlling API. Some light Java programming is required. See the Controlling API Guide for details and sample code.

Does NetLogo take advantage of multiple processors?

Only when using BehaviorSpace. BehaviorSpace does parallel runs, one per processor.

For a single model run, only one processor is used.

We are seeking funding to make it possible to split a single model run across multiple processors or multiple computers.

Can I distribute NetLogo model runs across a cluster or grid of computers?

Many of the same comments in the previous answer apply. It is not possible to split a single model run across multiple computers, but you can have each machine in a cluster doing one or more separate, independent model runs, using either BehaviorSpace or our Controlling API. We don't have automated support for splitting the runs across clusters, so you'll need to

arrange that yourself.

Various users are already using NetLogo on clusters, with a variety of hardware and software. You can seek them out on the NetLogo Users Group.

Is there any way to recover lost work if NetLogo crashes or freezes?

Yes. NetLogo auto-saves files as you are working on them. The auto-save files can be found in your OS-specific temporary directory. On most Unix-like systems (including MacOS) that is `/tmp`. On Windows Vista the logs can be found in `C:\Users\<user>\AppData\Local\Temp`, where `<user>` is the logged in user. The files are named according to the following format: `autosave_yyyy-MM-dd.HH_mm_ss.nlogo` where the time and date are the time and date the model was opened.

Usage

When I move the speed slider all the way to the right, why does my model seem to stop?

The only way NetLogo can make your model run faster is by updating the view less frequently. As you move the speed slider to the right, view updates become less and less frequent. Since view updates take time, that means more speed.

However, fewer updates also means that the updates come farther apart. When several seconds pass between updates, it may seem like your model has stopped. It hasn't. It's running at full speed. Watch the tick counter! (If your model uses it. If it doesn't, watch something else, like a plot.)

To get a feel for what's going on, try moving the slider to the right gradually rather than suddenly. If you find the updates too infrequent at the rightmost position, just don't push the slider that far.

Can I use the mouse to "paint" in the view?

NetLogo does not have a built-in set of painting tools for painting in the view. But with only a few lines of code, you can add painting capability to your model. To see how it's done, look at Mouse Example, in the Code Examples section of the Models Library. The same techniques can be used to let the user interact with your model using the mouse in other ways, too.

Another possibility is to create an image in another program and import it using the import items on the File menu or the `import-*` primitives.

How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?

We have tested NetLogo with models that use hundreds of megabytes of RAM and they work fine. We haven't tested models that use gigabytes of RAM, though. Theoretically it should work, but you might hit some limits that are inherent in the underlying Java VM and/or operating system (either designed-in limits, or bugs).

The NetLogo engine has no fixed limits on size. By default, though, NetLogo ships with a one-gigabyte ceiling on how much total RAM it can use. If your model exceeds that limit, you'll get

an OutOfMemoryError dialog.

If you are using BehaviorSpace, note that doing runs in parallel will multiply your RAM usage accordingly.

Each platform contains ".cfg" files containing JVM settings. There is one cfg file for each sub-application (NetLogo, NetLogo 3D, HubNet Client, etc.) Although the file location varies by platform, the process for changing it is the same. Locate the section of the file that looks like the following:

```
[JVMOptions]
# there may be one or more lines, leave them unchanged
-Xmx1024m
# there may be one or more lines, leave them unchanged
```

Modify the value immediately following

```
-Xmx
```

, changing it to the amount of space you need, save the file, and restart NetLogo. Platform specific notes follow:

- **Windows:** The cfg files will be in

  ```
  C:\Program Files (x86)\NetLogo 5.3.1\app
  ```

  if you are running 64-bit windows, and

  ```
  C:\Program Files\NetLogo 5.3.1\app
  ```

  otherwise.
- **Mac OS X:** The file for NetLogo will be located at:

  ```
  /Application/NetLogo 5.3.1/NetLogo.app/Contents/Java/NetLogo.cfg
  ```

  For NetLogo 3D and the other applications, you will find the file in the corresponding location for each application package. You can reach the cfg file by control-clicking the application in the Finder and choosing "Show Package Contents" from the popup menu.

  Please note that depending on your version of Mac OS X, changing the cfg file *may* break application signing. If this happens, follow Apple's directions on this page to temporarily allow apps from "Anywhere", run NetLogo once, then restore the setting to "Mac App Store and Identified Developers".

- **Other:** The cfg files will be located in the

  ```
  netlogo-5.3.1/app
  ```

  folder after untarring.

By default, Mac builds of NetLogo bundle a 64-bit JVM, which should be able to make use of as much memory as the lesser of available system memory and the value following

```
-Xmx
```

. Windows and Linux will bundle a 32-bit or 64-bit JVM, depending on which version you have downloaded. It is recommended that you install 64-bit NetLogo on all 64-bit operating systems for best performance.

Can I use GIS data in NetLogo?

Yes, many users are using GIS data with NetLogo. The most complete way to do that is with the GIS extension. See the [GIS Extension Guide](#).

A simpler way is to use `import-pcolors`, but that only works for importing maps that are images, not maps in other formats.

It is also possible to write NetLogo code that reads GIS data using our file I/O primitives such as `file-open`. For example, see the Grand Canyon model in the Earth Science section of Sample Models.

My model runs slowly. How can I speed it up?

Here's some ways to make it run faster without changing the structure of the code:

- Use tick-based view updates, not continuous updates.
- Decrease the frequency of view updates by pushing the speed slider to the right, or turn updates off using the checkbox.
- If your model is using all available RAM on your computer, then installing more RAM should help. If your hard drive makes a lot of noise while your model is running, you probably need more RAM.
- Use turtle size 1, 1.5, or 2, as the 2D renderings for these sizes are cached by NetLogo. (This only affects graphics speed in the 2D view, not computation speed.)

In many cases, though, if you want your model to run faster, you may need to make some changes to the code. Usually the most obvious opportunity for speedup is that you're doing too many computations that involve all the turtles or all the patches. Often this can be reduced by reworking the model so that it does less computation per time step. The members of the [NetLogo Users Group](#) may be able to help with this.

The [profiler extension](#) is useful for identifying which parts of your code are taking the most time.

Unless you are running the exact same strings over and over, using `run` and `runresult` are much slower than running code directly; you should avoid using these primitives on fresh strings in performance-critical code.

Can I have more than one model open at a time?

One instance of NetLogo can only have one model open at a time. (Unfortunately, it is unlikely that this will change in a future version, due to the engineering difficulties involved.)

You can have multiple models open by opening multiple instances of NetLogo, though. On Windows and Linux, simply start the application again. On a Mac, you'll need to duplicate the application (not the whole folder, just the application itself) in the Finder, then open the copy. (The copy takes up only a very small amount of additional disk space.)

Can I change the choices in a chooser on the fly?

No.

Can I divide the code for my model up into several files?

On an experimental basis, this is available using the `__includes` keyword.

Programming

How does the NetLogo language differ from other Logos?

This is answered in detail at the end of the [Programming Guide](#).

How come my model from an earlier NetLogo doesn't work right?

See the [Transition Guide](#) for help.

How do I take the negative of a number?

Any of these ways:

```
(- x)
-1 * x
0 - x
```

With the first way, the parentheses are required.

My turtle moved forward 1, but it's still on the same patch. Why?

If you have disabled wrapping at the world edges in your model, then the turtle might be at a world edge and unable to move any further. You can test for this using `can-move?`.

Assuming the turtle isn't hitting a world edge, moving forward 1 is only guaranteed to take a turtle to a new patch if the turtle's heading is a multiple of 90 (that is, exactly north, south, east, or west).

It's because the turtle might not be standing in the center of a patch. It might be near the corner of a patch. For example, suppose your turtle is close to the southwest corner of a patch and is facing northeast. The length of the patch diagonal is 1.414... (the square root of two), so `fd 1` will leave the turtle near the northeast corner of the same patch.

If you don't want to have to think about these issues, one possibility is to write your model in such a way that your turtles always come to rest on patch centers. See next question.

How do I keep my turtles on patch centers?

A turtle is on a patch center when its `xcor` and `ycor` are integers.

You can move a turtle to the center of its current patch with either of these two equivalent commands:

```
move-to patch-here
setxy pxcor pycor
```

But you'll never need to do that if you never allow turtles off of patch centers in the first place.

The `sprout` command creates turtles on patch centers. For example:

```
ask n-of 50 patches [ sprout 1 [ face one-of neighbors4 ] ]
```

Another way for a turtle to start on a patch center is with a command such as this line of turtle code, which moves it to the center of a random patch:

```
move-to one-of patches
```

Once a turtle is on a patch center, as long as its heading always stays an exact multiple of 90 (that is to say, due north, east, south, or west), and as it long as it moves forward or back by integer amounts, it will always land on patch centers.

See Random Grid Walk Example, in the Code Examples section of the Models Library, to see these code snippets in use.

`patch-ahead 1` is reporting the same patch my turtle is already standing on. Why?

See the answer two answers up. It's the same issue.

This might not be the meaning of "ahead" you were expecting. With `patch-ahead`, you must specify the distance ahead that you want to look. If you want to know the next patch a turtle would cross into if it moved forward continuously, it is possible to find that out. See Next Patch Example, in the Code Examples section of the Models Library.

How do I give my turtles "vision"?

You can use `in-radius` to let a turtle see a circular area around it.

Several primitives let the turtle "look" at specific points. The `patch-ahead` primitive is useful for letting a turtle see what is directly in front of it. If you want the turtle to look in another direction besides straight ahead, try `patch-left-and-ahead` and `patch-right-and-ahead`.

If you want the turtle to have a full "cone" of vision, use the `in-cone` primitive.

You can also find out the next patch a turtle would cross into if it moved forward continuously. See Next Patch Example, in the Code Examples section of the Models Library.

Can agents sense what's in the drawing layer?

No. If you want to make marks that agents can sense, use patch colors.

I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?

See the ["Math" section of the Programming Guide](#) for a discussion of this issue.

The documentation says that `random-float 1` might return 0 but will never return 1. What if I want 1 to be included?

It really doesn't matter. Even if 1 were a possible result, it would only come up approximately 1 in 2^64 tries, which means you'd be waiting hundreds of years before it ever came up exactly 1.

Nonetheless, if you are convinced that it really must be possible to get 1, you can use `precision` to round your answer to a certain number of decimal places. For example:

```
print precision (random-float 1) 10
0.2745173723
```

(If you use this method, note that 0 and 1 are only half as likely to come up as other answers. To see why this is so, consider the case where you only keep one digit after the decimal point. Results between 0 and 0.5 get rounded to 0, but results between 0.5 and 1.5 get rounded to 1; the latter range is twice as large. If you want 0, 0.1, 0.2, ..., 0.9, and 1 to all be equally likely, an alternative is to write `random 11 / 10`; this gives all 11 answers with equal probability.)

How can I keep two turtles from occupying the same patch?

See One Turtle Per Patch Example, in the Code Examples section of the Models Library.

How can I find out if a turtle is dead?

When a turtle dies, it turns into `nobody`. `nobody` is a special value used in NetLogo used to indicate the absence of a turtle or patch. So for example:

```
if turtle 0 != nobody [ ... ]
```

You could also use `is-turtle?`:

```
if is-turtle? turtle 0 [ ... ]
```

Does NetLogo have arrays?

Nearly all models should just use lists for this.

The usual motivation for using arrays in other programming languages is that they provide fast random access (`item`) and mutation (`replace-item`). But NetLogo's lists, even though they are immutable, now provide near constant time performance on these operations.

Lists in earlier versions of NetLogo (4.1 and 4.0) were simple singly-linked lists and therefore

these operations took linear time. The data structure underlying NetLogo's lists now is the immutable Vector class from the Scala collections library. It uses 32-wide [hash array mapped tries](#), as implemented by Tiark Rompf, based in part on work by Phil Bagwell and Rich Hickey.

If you are certain you want to use raw, mutable JVM arrays in your model, they are provided by the array extension. See the Arrays & Tables section of the User Manual.

Does NetLogo have hash tables or associative arrays?

Yes, using the table extension. See the Arrays & Tables section of the User Manual.

How can I use different patch "neighborhoods" (circular, Von Neumann, Moore, etc.)?

The `in-radius` primitives lets you access circular neighborhoods of any radius.

The `neighbors` primitive gives you a Moore neighborhood of radius 1, and the `neighbors4` primitive gives you a Von Neumann neighborhood of radius 1.

For Moore or Von Neumann neighborhoods of a larger radius, see Moore & Von Neumann Example in the Code Examples section of the Models Library.

How can I convert an agentset to a list of agents, or vice versa?

If you want the list in a particular order, use the `sort` or `sort-by` primitives. The Lists section of the Programming Guide explains how to do this. See also Ask Ordering Example, in the Code Examples section of the Models Library.

If you want the list in a random order, here's how:

```
[self] of <agentset>
```

Because all operations on agentsets are in random order, the resulting list is in random order.

To convert a list of agents to an agentset, use the `turtle-set`, `patch-set`, or `link-set` primitives.

How do I stop foreach?

To stop a foreach from executing you need to define a separate procedure that contains only the foreach, for example:

```
to test
  foreach [1 2 3] [
    if ? = 2 [ stop ]
    print ?
  ]
end
```

This code will only print the number 1. The `stop` returns from the current procedure so nothing after the foreach will be executed either. (If the procedure is a reporter procedure, use `report` instead of `stop`.)

I'm trying to make a list. Why do I keep getting the error "Expected a constant"?

If a list contains only constants, you can write it down just by putting square brackets around it, like `[1 2 3]`.

If you want your list to contain items that may vary at runtime, the list cannot be written down directly. Instead, you build it using the [list](#) primitive.

BehaviorSpace

Why are the rows in my BehaviorSpace table results out of order?

This is normal when doing multiple runs in parallel. For a discussion of the issue, see the section on parallel runs in the [BehaviorSpace Guide](#) section of the User Manual.

How do I measure runs every n ticks?

Use `repeat` in your experiment's go commands, e.g.:

```
repeat 10 [ go ]
```

to measure the run after every 10 ticks. Essentially you are making one experiment step equal 10 ticks.

I'm varying a global variable I declared in the Code tab, but it doesn't work. Why?

It's probably because your setup commands or setup procedure are using `clear-all`, causing the values set by BehaviorSpace to be cleared.

One possible workaround is to change your experiment's setup commands to preserve the value of the variable, e.g.:

```
let old-var1 var1
setup
set var1 old-var1
```

This works because even `clear-all` doesn't clear the values of local variables made with `let`

Another possible workaround is to change your model's setup procedure to use more specific clearing commands to clear only what you want cleared.

Why are some of my results cut off in Excel?

In some versions of Excel, spreadsheets can't have more than 256 columns. (See [a Microsoft support article](#) on the subject.)

Possible workarounds include:

- Use a newer version of Excel, such as Excel 2007 (Windows) or Excel 2008 (Mac).
- Use a different program besides Excel.
- Ask BehaviorSpace to generate results in table format instead of, or in addition to, spreadsheet format. (Excel can read our table format, too.)
- Change your experiment so the result has fewer columns.

NetLogo 3D

Can I make a 3D applet?

No.

Does NetLogo work with my stereoscopic device?

NetLogo supports fullscreen exclusive mode. If that is all your device needs then, possibly yes. However, it can be tricky to get it working. We don't have any such devices so it is difficult for us to make the process easier. If your device needs something else, for example, quadbuffers enabled, the answer is probably no.

Extensions

I'm writing an extension. Why does the compiler say it can't find org.nlogo.api?

You need to add NetLogo.jar to your classpath when compiling. NetLogo.jar is included with NetLogo.

NetLogo Dictionary

Categories

This is an approximate grouping. Remember that a turtle-related primitive might still be used by patches or the observer, and vice versa. To see which agents (turtles, patches, links, observer) can actually run a primitive, consult its dictionary entry.

Turtle-related

back (bk) *<breeds>*-at *<breeds>*-here *<breeds>*-on can-move? clear-turtles (ct) create-*<breeds>* create-ordered-*<breeds>* create-ordered-turtles (cro) create-turtles (crt) die distance distancexy downhill downhill4 dx dy face facexy forward (fd) hatch hatch-*<breeds>* hide-turtle (ht) home inspect is-*<breed>*? is-turtle? jump layout-circle left (lt) move-to myself nobody no-turtles of other patch-ahead patch-at patch-at-heading-and-distance patch-here patch-left-and-ahead patch-right-and-ahead pen-down (pd) pen-erase (pe) pen-up (pu) random-xcor random-ycor right (rt) self set-default-shape __set-line-thickness setxy shapes show-turtle (st) sprout sprout-*<breeds>* stamp stamp-erase stop-inspecting subject subtract-headings tie towards towardsxy turtle turtle-set turtles turtles-at turtles-here turtles-on turtles-own untie uphill uphill4

Patch-related

clear-patches (cp) diffuse diffuse4 distance distancexy import-pcolors import-pcolors-rgb inspect is-patch? myself neighbors neighbors4 nobody no-patches of other patch patch-at patch-ahead patch-at-heading-and-distance patch-here patch-left-and-ahead patch-right-and-ahead patch-set patches patches-own random-pxcor random-pycor self sprout sprout-*<breeds>* stop-inspecting subject turtles-here

Agentset

all? any? ask ask-concurrent at-points *<breeds>*-at *<breeds>*-here *<breeds>*-on count in-cone in-radius is-agent? is-agentset? is-patch-set? is-turtle-set? link-set max-n-of max-one-of member? min-n-of min-one-of n-of neighbors neighbors4 no-links no-patches no-turtles of one-of other patch-set patches sort sort-by sort-on turtle-set turtles turtles-at turtles-here turtles-on with with-max with-min

Color

approximate-hsb approximate-rgb base-colors color extract-hsb extract-rgb hsb import-pcolors import-pcolors-rgb pcolor rgb scale-color shade-of? wrap-color

Control flow and logic

and ask ask-concurrent carefully end error error-message every if ifelse ifelse-value let loop not or repeat report run runresult ; (semicolon) set stop startup to to-report wait while with-local-randomness without-interruption xor

Task

filter foreach is-command-task? is-reporter-task? map n-values reduce run runresult sort-by task

World

clear-all (ca) clear-drawing (cd) clear-patches (cp) clear-ticks clear-turtles (ct) display import-drawing import-pcolors import-pcolors-rgb no-display max-pxcor max-pycor min-pxcor min-pycor patch-size reset-ticks resize-world set-patch-size stop-inspecting-dead-agents tick tick-advance ticks world-width world-height

Perspective

follow follow-me reset-perspective (rp) ride ride-me subject watch watch-me

HubNet

hubnet-broadcast hubnet-broadcast-clear-output hubnet-broadcast-message hubnet-clear-override hubnet-clear-

overrides hubnet-clients-list hubnet-enter-message? hubnet-exit-message? hubnet-kick-all-clients hubnet-kick-client hubnet-fetch-message hubnet-message hubnet-message-source hubnet-message-tag hubnet-message-waiting? hubnet-reset hubnet-reset-perspective hubnet-send hubnet-send-clear-output hubnet-send-follow hubnet-send-message hubnet-send-override hubnet-send-watch hubnet-set-client-interface

Input/output

beep clear-output date-and-time export-view export-interface export-output export-plot export-all-plots export-world import-drawing import-pcolors import-pcolors-rgb import-world mouse-down? mouse-inside? mouse-xcor mouse-ycor output-print output-show output-type output-write print read-from-string reset-timer set-current-directory show timer type user-directory user-file user-new-file user-input user-message user-one-of user-yes-or-no? write

File

file-at-end? file-close file-close-all file-delete file-exists? file-flush file-open file-print file-read file-read-characters file-read-line file-show file-type file-write user-directory user-file user-new-file

List

but-first but-last empty? filter first foreach fput histogram is-list? item last length list lput map member? modes n-of n-values of position one-of reduce remove remove-duplicates remove-item replace-item reverse sentence shuffle sort sort-by sort-on sublist

String

Operators (<, >, =, !=, <=, >=)but-first but-last empty? first is-string? item last length member? position remove remove-item read-from-string replace-item reverse substring word

Mathematical

Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=)abs acos asin atan ceiling cos e exp floor int is-number? ln log max mean median min mod modes new-seed pi precision random random-exponential random-float random-gamma random-normal random-poisson random-seed remainder round sin sqrt standard-deviation subtract-headings sum tan variance

Plotting

autoplot? auto-plot-off auto-plot-on clear-all-plots clear-plot create-temporary-plot-pen export-plot export-all-plots histogram plot plot-name plot-pen-exists? plot-pen-down plot-pen-reset plot-pen-up plot-x-max plot-x-min plot-y-max plot-y-min plotxy set-current-plot set-current-plot-pen set-histogram-num-bars set-plot-pen-color set-plot-pen-interval set-plot-pen-mode set-plot-x-range set-plot-y-range setup-plots update-plots

Links

both-ends clear-links create-<breed>-from create-<breeds>-from create-<breed>-to create-<breeds>-to create-<breed>-with create-<breeds>-with create-link-from create-links-from create-link-to create-links-to create-link-with create-links-with die hide-link in-<breed>-neighbor? in-<breed>-neighbors in-<breed>-from in-link-neighbor? in-link-neighbors in-link-from is-directed-link? is-link? is-link-set? is-undirected-link? layout-radial layout-spring layout-tutte <breed>-neighbor? <breed>-neighbors <breed>-with link-heading link-length link-neighbor? link links links-own <link-breeds>-own link-neighbors link-with my-<breeds> my-in-<breeds> my-in-links my-links my-out-<breeds> my-out-links no-links other-end out-<breed>-neighbor? out-<breed>-neighbors out-<breed>-to out-link-neighbor? out-link-neighbors out-link-to show-link tie untie

Movie

movie-cancel movie-close movie-grab-view movie-grab-interface movie-set-frame-rate movie-start movie-status

BehaviorSpace

behaviorspace-experiment-name behaviorspace-run-number

System

netlogo-applet? netlogo-version netlogo-web?

Built-In Variables

Turtles

breed color heading hidden? label label-color pen-mode pen-size shape size who xcor ycor

Patches

pcolor plabel plabel-color pxcor pycor

Links

breed color end1 end2 hidden? label label-color shape thickness tie-mode

Other

?

Keywords

breed directed-link-breed end extensions globals __includes links-own patches-own to to-report turtles-own undirected-link-breed

Constants

Mathematical Constants

e = 2.718281828459045
pi = 3.141592653589793

Boolean Constants

false
true

Color Constants

black = 0
gray = 5
white = 9.9
red = 15
orange = 25
brown = 35
yellow = 45
green = 55
lime = 65
turquoise = 75
cyan = 85
sky = 95
blue = 105
violet = 115
magenta = 125
pink = 135

See the Colors section of the Programming Guide for more details.

A

abs

abs *number*

Reports the absolute value of *number*.

```
show abs -7
=> 7
show abs 5
=> 5
```

acos

acos *number*

Reports the arc cosine (inverse cosine) of the given number. The input must be in the range -1 to 1. The result is in degrees, and lies in the range 0 to 180.

all?

all? *agentset* [*reporter*]

Reports true if all of the agents in the agentset report true for the given reporter. Otherwise reports false as soon as a counterexample is found.

If the agentset is empty, reports true.

The reporter must report a boolean value for every agent (either true or false), otherwise an error occurs.

```
if all? turtles [color = red]
  [ show "every turtle is red!" ]
```

See also any?.

and

condition1 and *condition2*

Reports true if both *condition1* and *condition2* are true.

Note that if *condition1* is false, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) and (pycor > 0)
  [ set pcolor blue ]  ;; the upper-right quadrant of
                       ;; patches turn blue
```

any?

any? *agentset*

Reports true if the given agentset is non-empty, false otherwise.

Equivalent to "count *agentset* > 0", but more efficient (and arguably more readable).

```
if any? turtles with [color = red]
  [ show "at least one turtle is red!" ]
```

Note: nobody is not an agentset. You only get nobody back in situations where you were expecting a single agent, not a whole agentset. If any? gets nobody as input, an error results.

See also all?, nobody.

approximate-hsb

approximate-hsb *hue saturation brightness*

Reports a number in the range 0 to 140, not including 140 itself, that represents the given color, specified in the HSB spectrum, in NetLogo's color space.

The first value (hue) should be in the range of 0 to 360, the second and third (saturation and brightness) in the range between 0 and 100.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors.

```
show approximate-hsb 0 0 0
=> 0  ;; (black)
show approximate-hsb 180 57.143 76.863
=> 85 ;; (cyan)
```

See also [extract-hsb](#), [approximate-rgb](#), [extract-rgb](#).

approximate-rgb

approximate-rgb *red green blue*

Reports a number in the range 0 to 140, not including 140 itself, that represents the given color, specified in the RGB spectrum, in NetLogo's color space.

All three inputs should be in the range 0 to 255.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (See [approximate-hsb](#) for a description of what parts of the HSB color space NetLogo colors cover; this is difficult to characterize in RGB terms.)

```
show approximate-rgb 0 0 0
=> 0  ;; black
show approximate-rgb 0 255 255
=> 85.2 ;; cyan
```

See also [extract-rgb](#), [approximate-hsb](#), and [extract-hsb](#).

Arithmetic Operators

+
*
-
/
^
<
>
=
!=
<=
>=

All of these operators take two inputs, and all act as "infix operators" (going between the two inputs, as in standard mathematical use). NetLogo correctly supports order of operations for infix operators.

The operators work as follows: + is addition, * is multiplication, - is subtraction, / is division, ^ is exponentiation, < is less than, > is greater than, = is equal to, != is not equal to, <= is less than or equal, >= is greater than or equal.

Note that the subtraction operator (-) always takes two inputs unless you put parentheses around it, in which case it can take one input. For example, to take the negative of x, write (- x), with the parentheses.

All of the comparison operators also work on strings.

All of the comparison operators work on agents. Turtles are compared by who number. Patches are compared top to bottom left to right, so patch 0 10 is less than patch 0 9 and patch 9 0 is less than patch 10 0. Links are ordered by end points and in case of a tie by breed. So link 0 9 is before link 1 10 as the end1 is smaller, and link 0 8 is less than link 0 9. If there are multiple breeds of links unbreeded links will come before breeded links of the same end points and breeded links will be sorted in the order they are declared in the Code tab.

Agentsets can be tested for equality or inequality. Two agentsets are equal if they are the same type (turtle or patch) and contain the same agents.

If you are not sure how NetLogo will interpret your code, you should add parentheses.

```
show 5 * 6 + 6 / 3
=> 32
show 5 * (6 + 6) / 3
=> 20
```

asin

asin *number*

Reports the arc sine (inverse sine) of the given number. The input must be in the range -1 to 1. The result is in

degrees, and lies in the range -90 to 90.

ask

ask *agentset* [*commands*]
ask *agent* [*commands*]

The specified agent or agentset runs the given commands.

```
ask turtles [ fd 1 ]
  ;; all turtles move forward one step
ask patches [ set pcolor red ]
  ;; all patches turn red
ask turtle 4 [ rt 90 ]
  ;; only the turtle with id 4 turns right
```

Note: only the observer can ask all turtles or all patches. This prevents you from inadvertently having all turtles ask all turtles or all patches ask all patches, which is a common mistake to make if you're not careful about which agents will run the code you are writing.

Note: Only the agents that are in the agentset *at the time the ask begins* run the commands.

ask-concurrent

ask-concurrent *agentset* [*commands*]

This primitive exists only for backwards compatibility. We don't recommend using it new models.

The agents in the given agentset run the given commands, using a turn-taking mechanism to produce simulated concurrency. See the Ask-Concurrent section of the Programming Guide for details on how this works.

Note: Only the agents that are in the agentset *at the time the ask begins* run the commands.

See also without-interruption.

at-points

agentset **at-points** [[*x1 y1*] [*x2 y2*] ...]

Reports a subset of the given agentset that includes only the agents on the patches the given distances away from this agent. The distances are specified as a list of two-item lists, where the two items are the x and y offsets.

If the caller is the observer, then the points are measured relative to the origin, in other words, the points are taken as absolute patch coordinates.

If the caller is a turtle, the points are measured relative to the turtle's exact location, and not from the center of the patch under the turtle.

```
ask turtles at-points [[2 4] [1 2] [10 15]]
  [ fd 1 ]  ;; only the turtles on the patches at the
            ;; distances (2,4), (1,2) and (10,15),
            ;; relative to the caller, move
```

atan

atan *x y*

Converts x and y offsets to a turtle heading in degrees (from 0 to 360).

Note that this version of atan is designed to conform to the geometry of the NetLogo world, where a heading of 0 is straight up, 90 is to the right, and so on clockwise around the circle. (Normally in geometry an angle of 0 is right, 90 is up, and so on, counterclockwise around the circle, and atan would be defined accordingly.)

When y is 0: if x is positive, it reports 90; if x is negative, it reports 270; if x is zero, you get an error.

```
show atan 1 -1
=> 135
show atan -1 1
=> 315
crt 1 [ set heading 30  fd 1  print atan xcor ycor ]
=> 30
```

In the final example, note that the result of `atan` equals the turtle's heading.

If you ever need to convert a turtle heading (obtained with atan or otherwise) to a normal mathematical angle, the following should be helpful:

```
to-report heading-to-angle [ h ]
  report (90 - h) mod 360
end
```

autoplot?

autoplot?

Reports true if auto-plotting is on for the current plot, false otherwise.

auto-plot-off
auto-plot-on

auto-plot-off
auto-plot-on

This pair of commands is used to control the NetLogo feature of auto-plotting in the current plot. Auto-plotting will automatically update the x and y axes of the plot whenever the current pen exceeds these boundaries. It is useful when wanting to show all plotted values in the current plot, regardless of the current plot ranges.

B

back
bk

back *number*

The turtle moves backward by *number* steps. (If *number* is negative, the turtle moves forward.)

Turtles using this primitive can move a maximum of one unit per time increment. So `bk 0.5` and `bk 1` both take one unit of time, but `bk 3` takes three.

If the turtle cannot move backward *number* steps because it is not permitted by the current topology the turtle will complete as many steps of 1 as it can and stop.

See also <u>forward</u>, <u>jump</u>, <u>can-move?</u>.

base-colors

base-colors

Reports a list of the 14 basic NetLogo hues.

```
print base-colors
=> [5 15 25 35 45 55 65 75 85 95 105 115 125 135]
ask turtles [ set color one-of base-colors ]
;; each turtle turns a random base color
ask turtles [ set color one-of remove gray base-colors ]
;; each turtle turns a random base color except for gray
```

beep

beep

Emits a beep. Note that the beep sounds immediately, so several beep commands in close succession may produce only one audible sound.

Example:

```
beep                    ;; emits one beep
repeat 3 [ beep ]       ;; emits 3 beeps at once,
                        ;; so you only hear one sound
repeat 3 [ beep wait 0.1 ] ;; produces 3 beeps in succession,
                        ;; separated by 1/10th of a second
```

When running headless, this command has no effect.

behaviorspace-experiment-name

behaviorspace-experiment-name

Reports the current experiment name in the current experiment.

If no BehaviorSpace experiment is running, reports "".

behaviorspace-run-number

behaviorspace-run-number

Reports the current run number in the current BehaviorSpace experiment, starting at 1.

If no BehaviorSpace experiment is running, reports 0.

both-ends

both-ends

Reports the agentset of the 2 nodes connected by this link.

```
crt 2
ask turtle 0 [ create-link-with turtle 1 ]
ask link 0 1 [
  ask both-ends [ set color red ] ;; turtles 0 and 1 both turn red
]
```

breed

breed

This is a built-in turtle and link variable. It holds the agentset of all turtles or links of the same breed as this turtle or link. (For turtles or links that do not have any particular breed, this is the turtles agentset of all turtles or the links agentset of all links respectively.)

You can set this variable to change a turtle or link's breed. (When a turtle changes breeds, its shape is reset to the default shape for that breed. See set-default-shape.)

See also breed, directed-link-breed, undirected-link-breed

Example:

```
breed [cats cat]
breed [dogs dog]
;; turtle code:
if breed = cats [ show "meow!" ]
set breed dogs
show "woof!"
```

```
directed-link-breed [ roads road ]
;; link code
if breed = roads [ set color gray ]
```

breed

breed [<*breeds*> <*breed*>]

This keyword, like the globals, turtles-own, and patches-own keywords, can only be used at the beginning of the Code tab, before any procedure definitions. It defines a breed. The first input defines the name of the agentset associated with the breed. The second input defines the name of a single member of the breed.

Any turtle of the given breed:

- is part of the agentset named by the breed name
- has its breed built-in variable set to that agentset

Most often, the agentset is used in conjunction with ask to give commands to only the turtles of a particular breed.

```
breed [mice mouse]
breed [frogs frog]
to setup
  clear-all
  create-mice 50
  ask mice [ set color white ]
  create-frogs 50
  ask frogs [ set color green ]
  show [breed] of one-of mice     ;; prints mice
  show [breed] of one-of frogs    ;; prints frogs
end

show mouse 1
;; prints (mouse 1)
show frog 51
;; prints (frog 51)
show turtle 51
;; prints (frog 51)
```

See also [globals](), [patches-own](), [turtles-own](), [*<breeds>*-own](), [create-*<breeds>*](), [*<breeds>*-at](), [*<breeds>*-here]().

but-first
butfirst
bf
but-last
butlast
bl

but-first *list*
but-first *string*
but-last *list*
but-last *string*

When used on a list, but-first reports all of the list items of *list* except the first, and but-last reports all of the list items of *list* except the last.

On strings, but-first and but-last report a shorter string omitting the first or last character of the original string.

```
;; mylist is [2 4 6 5 8 12]
set mylist but-first mylist
;; mylist is now [4 6 5 8 12]
set mylist but-last mylist
;; mylist is now [4 6 5 8]
show but-first "string"
;; prints "tring"
show but-last "string"
;; prints "strin"
```

C

can-move?

can-move? *distance*

Reports true if this turtle can move *distance* in the direction it is facing without violating the topology; reports false otherwise.

It is equivalent to:

```
patch-ahead distance != nobody
```

carefully

carefully [*commands1*] [*commands2*]

Runs *commands1*. If a runtime error occurs inside *commands1*, NetLogo won't stop and alert the user that an error occurred. It will suppress the error and run *commands2* instead.

The error-message reporter can be used in *commands2* to find out what error was suppressed in *commands1*. See error-message.

```
carefully [ print one-of [1 2 3] ] [ print error-message ]
=> 3
observer> carefully [ print one-of [] ] [ print error-message ]
=> ONE-OF got an empty list as input.
```

ceiling

ceiling *number*

Reports the smallest integer greater than or equal to *number*.

```
show ceiling 4.5
=> 5
show ceiling -4.5
=> -4
```

See also floor, round, precision.

clear-all
ca

clear-all

Combines the effects of clear-globals, clear-ticks, clear-turtles, clear-patches, clear-drawing, clear-all-plots, and clear-output.

clear-all-plots

clear-all-plots

Clears every plot in the model. See clear-plot for more information.

clear-drawing
cd

clear-drawing

Clears all lines and stamps drawn by turtles.

clear-globals

clear-globals

Sets all global variables to 0.

clear-links

clear-links

Kills all links.

See also [die](#).

clear-output

clear-output

Clears all text from the model's output area, if it has one. Otherwise does nothing.

clear-patches
cp

clear-patches

Clears the patches by resetting all patch variables to their default initial values, including setting their color to black.

clear-plot

clear-plot

In the current plot only, resets all plot pens, deletes all temporary plot pens, resets the plot to its default values (for x range, y range, etc.), and resets all permanent plot pens to their default values. The default values for the plot and for the permanent plot pens are set in the plot Edit dialog, which is displayed when you edit the plot. If there are no plot pens after deleting all temporary pens, that is to say if there are no permanent plot pens, a default plot pen will be created with the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Name: "default"
- Interval: 1

See also [clear-all-plots](#).

clear-ticks

clear-ticks

Clears the tick counter.

Does not set the counter to zero. After this command runs, the tick counter has no value. Attempting to access or update it is an error until [reset-ticks](#) is called.

See also [reset-ticks](#).

clear-turtles
ct

clear-turtles

Kills all turtles.

Also resets the who numbering, so the next turtle created will be turtle 0.

See also [die](#).

color

color

This is a built-in turtle or link variable. It holds the color of the turtle or link. You can set this variable to make the turtle or link change color. Color can be represented either as a NetLogo color (a single number), or an RGB color (a list of 3 numbers). See details in the Colors section of the Programming Guide.

See also pcolor.

cos

cos *number*

Reports the cosine of the given angle. Assumes the angle is given in degrees.

```
show cos 180
=> -1
```

count

count *agentset*

Reports the number of agents in the given agentset.

```
show count turtles
;; prints the total number of turtles
show count patches with [pcolor = red]
;; prints the total number of red patches
```

create-ordered-turtles
cro

create-ordered-turtles *number*
create-ordered-turtles *number* **[** *commands* **]**
create-ordered<*breeds***>** *number*
create-ordered<*breeds***>** *number* **[** *commands* **]**

Creates *number* new turtles. New turtles start at position (0, 0), are created with the 14 primary colors, and have headings from 0 to 360, evenly spaced.

If the create-ordered-<*breeds*> form is used, the new turtles are created as members of the given breed.

If *commands* are supplied, the new turtles immediately run them. This is useful for giving the new turtles a different color, heading, or whatever. (The new turtles are created all at once then run one at a time, in random order.)

```
cro 100 [ fd 10 ]  ;; makes an evenly spaced circle
```

create-<breed>-to
create-<breeds>-to
create-<breed>-from
create-<breeds>-from
create-<breed>-with
create-<breeds>-with
create-link-to
create-links-to
create-link-from
create-links-from
create-link-with
create-links-with

create-<breed>-to *turtle*
create-<breed>-to *turtle* **[** *commands* **]**
create-<breed>-from *turtle*
create-<breed>-from *turtle* **[** *commands* **]**
create-<breed>-with *turtle*
create-<breed>-with *turtle* **[** *commands* **]**
create-<breeds>-to *turtleset*

create-\<breeds\>-to *turtleset* **[commands]**
create-\<breeds\>-from *turtleset*
create-\<breeds\>-from *turtleset* **[commands]**
create-\<breeds\>-with *turtleset*
create-\<breeds\>-with *turtleset* **[commands]**
create-link-to *turtle*
create-link-to *turtle* **[commands]**
create-link-from *turtle*
create-link-from *turtle* **[commands]**
create-link-with *turtle*
create-link-with *turtle* **[commands]**
create-links-to *turtleset*
create-links-to *turtleset* **[commands]**
create-links-from *turtleset*
create-links-from *turtleset* **[commands]**
create-links-with *turtleset*
create-links-with *turtleset* **[commands]**

Used for creating breeded and unbreeded links between turtles.

`create-link-with` creates an undirected link between the caller and *agent*. `create-link-to` creates a directed link from the caller to *agent*. `create-link-from` creates a directed link from *agent* to the caller.

When the plural form of the breed name is used, an *agentset* is expected instead of an agent and links are created between the caller and all agents in the agentset.

The optional command block is the set of commands each newly formed link runs. (The links are created all at once then run one at a time, in random order.)

A node cannot be linked to itself. Also, you cannot have more than one undirected link of the same breed between the same two nodes, nor can you have more than one directed link of the same breed going in the same direction between two nodes.

If you try to create a link where one (of the same breed) already exists, nothing happens. If you try to create a link from a turtle to itself you get a runtime error.

```
to setup
  clear-all
  create-turtles 5
  ;; turtle 1 creates links with all other turtles
  ;; the link between the turtle and itself is ignored
  ask turtle 0 [ create-links-with other turtles ]
  show count links ;; shows 4
  ;; this does nothing since the link already exists
  ask turtle 0 [ create-link-with turtle 1 ]
  show count links ;; shows 4 since the previous link already existed
  ask turtle 2 [ create-link-with turtle 1 ]
  show count links ;; shows 5
end
```

```
directed-link-breed [red-links red-link]
undirected-link-breed [blue-links blue-link]

to setup
  clear-all
  create-turtles 5
  ;; create links in both directions between turtle 0
  ;; and all other turtles
  ask turtle 0 [ create-red-links-to other turtles ]
  ask turtle 0 [ create-red-links-from other turtles ]
  show count links ;; shows 8
  ;; now create undirected links between turtle 0 and other turtles
  ask turtle 0 [ create-blue-links-with other turtles ]
  show count links ;; shows 12
end
```

create-turtles
crt

create-turtles *number*
create-turtles *number* **[commands]**
create-\<breeds\> *number*
create-\<breeds\> *number* **[commands]**

Creates *number* new turtles at the origin. New turtles have random integer headings and the color is randomly

selected from the 14 primary colors.

If the create-*<breeds>* form is used, the new turtles are created as members of the given breed.

If *commands* are supplied, the new turtles immediately run them. This is useful for giving the new turtles a different color, heading, or whatever. (The new turtles are created all at once then run one at a time, in random order.)

```
crt 100 [ fd 10 ]     ;; makes a randomly spaced circle
```

```
breed [canaries canary]
breed [snakes snake]
to setup
  clear-all
  create-canaries 50 [ set color yellow ]
  create-snakes 50 [ set color green ]
end
```

See also hatch, sprout.

create-temporary-plot-pen

create-temporary-plot-pen *string*

A new temporary plot pen with the given name is created in the current plot and set to be the current pen.

Few models will want to use this primitive, because all temporary pens disappear when clear-plot or clear-all-plots are called. The normal way to make a pen is to make a permanent pen in the plot's Edit dialog.

If a pen with that name already exists in the current plot, no new pen is created, and the existing pen is set to the current pen.

The new temporary plot pen has the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Interval: 1

See: clear-plot, clear-all-plots, and set-current-plot-pen.

D

date-and-time

date-and-time

Reports a string containing the current date and time. The format is shown below. All fields are fixed width, so they are always at the same locations in the string. The potential resolution of the clock is milliseconds. (Whether you get resolution that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.)

```
show date-and-time
=> "01:19:36.685 PM 19-Sep-2002"
```

die

die

The turtle or link dies.

```
if xcor > 20 [ die ]
;; all turtles with xcor greater than 20 die
ask links with [color = blue] [ die ]
;; all the blue links will die
```

A dead agent ceases to exist. The effects of this include:

- The agent will not execute any further code. So if you write `ask turtles [ die print "last words?" ]`, no last words will be printed, because the turtles are already dead before they have a chance to print anything.
- The agent will disappear from any agentsets it was in, reducing the size of those agentsets by one.
- Any variable that was storing the agent will now instead have `nobody` in it. So for example `let x one-of turtles ask x [ die ] print x` prints `nobody`.
- If the dead agent was a turtle, every link connected to it also dies.
- If the observer was watching or following the agent, the observer's perspective resets, as if `reset-perspective` had been run.

See also: <u>clear-turtles</u> <u>clear-links</u>

diffuse

diffuse *patch-variable number*

Tells each patch to give equal shares of (*number* * 100) percent of the value of *patch-variable* to its eight neighboring patches. *number* should be between 0 and 1. Regardless of topology the sum of *patch-variable* will be conserved across the world. (If a patch has fewer than eight neighbors, each neighbor still gets an eighth share; the patch keeps any leftover shares.)

Note that this is an observer command only, even though you might expect it to be a patch command. (The reason is that it acts on all the patches at once -- patch commands act on individual patches.)

```
diffuse chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 8 patches. Thus,
;; each patch gets 1/8 of 50% of the chemical
;; from each neighboring patch.)
```

diffuse4

diffuse4 *patch-variable number*

Like diffuse, but only diffuses to the four neighboring patches (to the north, south, east, and west), not to the diagonal neighbors.

```
diffuse4 chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 4 patches. Thus,
;; each patch gets 1/4 of 50% of the chemical
;; from each neighboring patch.)
```

directed-link-breed

directed-link-breed [<*link-breeds*> <*link-breed*>]

This keyword, like the globals and breeds keywords, can only be used at the beginning of the Code tab, before any procedure definitions. It defines a directed link breed. Links of a particular breed are always all directed or all undirected The first input defines the name of the agentset associated with the link breed. The second input defines the name of a single member of the breed. Directed links can be created using <u>create-link(s)-to</u>, and <u>create-link(s)-from</u>, but not `create-link(s)-with`

Any link of the given link breed:

- is part of the agentset named by the link breed name
- has its built-in variable `breed` set to that agentset
- is directed or undirected as declared by the keyword

Most often, the agentset is used in conjunction with ask to give commands to only the links of a particular breed.

```
directed-link-breed [streets street]
directed-link-breed [highways highway]
to setup
  clear-all
  crt 2
  ;; create a link from turtle 0 to turtle 1
  ask turtle 0 [ create-street-to turtle 1 ]
  ;; create a link from turtle 1 to turtle 0
  ask turtle 0 [ create-highway-from turtle 1 ]
end
```

```
ask turtle 0 [ show one-of my-in-links ]
;; prints (street 0 1)
ask turtle 0 [ show one-of my-out-links ]
;; prints (highway 1 0)
```

See also [breed](#), [undirected-link-breed](#)

display

display

Causes the view to be updated immediately. (Exception: if the user is using the speed slider to fast-forward the model, then the update may be skipped.)

Also undoes the effect of the no-display command, so that if view updates were suspended by that command, they will resume.

```
no-display
ask turtles [ jump 10 set color blue set size 5 ]
display
;; turtles move, change color, and grow, with none of
;; their intermediate states visible to the user, only
;; their final state
```

Even if no-display was not used, "display" can still be useful, because ordinarily NetLogo is free to skip some view updates, so that fewer total updates take place, so that models run faster. This command lets you force a view update, so whatever changes have taken place in the world are visible to the user.

```
ask turtles [ set color red ]
display
ask turtles [ set color blue]
;; turtles turn red, then blue; use of "display" forces
;; red turtles to appear briefly
```

Note that display and no-display operate independently of the switch in the view control strip that freezes the view.

See also [no-display](#).

distance

distance *agent*

Reports the distance from this agent to the given turtle or patch.

The distance to or a from a patch is measured from the center of the patch. Turtles and patches use the wrapped distance (around the edges of the world) if wrapping is allowed by the topology and the wrapped distance is shorter.

```
ask turtles [ show max-one-of turtles [distance myself] ]
;; each turtle prints the turtle farthest from itself
```

distancexy

distancexy *xcor ycor*

Reports the distance from this agent to the point (*xcor*, *ycor*).

The distance from a patch is measured from the center of the patch. Turtles and patches use the wrapped distance (around the edges of the world) if wrapping is allowed by the topology and the wrapped distance is shorter.

```
if (distancexy 0 0) > 10
  [ set color green ]
;; all turtles more than 10 units from
;; the center of the world turn green.
```

downhill
downhill4

downhill *patch-variable*
downhill4 *patch-variable*

Moves the turtle to the neighboring patch with the lowest value for *patch-variable*. If no neighboring patch has a smaller value than the current patch, the turtle stays put. If there are multiple patches with the same lowest value, the turtle picks one randomly. Non-numeric values are ignored.

downhill considers the eight neighboring patches; downhill4 only considers the four neighbors.

Equivalent to the following code (assumes variable values are numeric):

```
move-to patch-here  ;; go to patch center
let p min-one-of neighbors [patch-variable]  ;; or neighbors4
if [patch-variable] of p < patch-variable [
  face p
  move-to p
]
```

Note that the turtle always ends up on a patch center and has a heading that is a multiple of 45 (downhill) or 90 (downhill4).

See also uphill, uphill4.

dx
dy

dx
dy

Reports the x-increment or y-increment (the amount by which the turtle's xcor or ycor would change) if the turtle were to take one step forward in its current heading.

Note: dx is simply the sine of the turtle's heading, and dy is simply the cosine. (If this is the reverse of what you expected, it's because in NetLogo a heading of 0 is north and 90 is east, which is the reverse of how angles are usually defined in geometry.)

Note: In earlier versions of NetLogo, these primitives were used in many situations where the new `patch-ahead` primitive is now more appropriate.

E

empty?

empty? *list*
empty? *string*

Reports true if the given list or string is empty, false otherwise.

Note: the empty list is written `[]`. The empty string is written `""`.

end

end

Used to conclude a procedure. See to and to-report.

end1

end1

This is a built-in link variable. It indicates the first endpoint (turtle) of a link. For directed links this will always be the source for undirected links it will always be the turtle with the lower who number. You cannot set end1.

```
crt 2
ask turtle 0
```

```
[ create-link-to turtle 1 ]
ask links
[ show end1 ] ;; shows turtle 0
```

end2

end2
⚭

This is a built-in link variable. It indicates the second endpoint (turtle) of a link. For directed links this will always be the destination for undirected links it will always be the turtle with the higher who number. You cannot set end2.

```
crt 2
ask turtle 1
[ create-link-with turtle 0 ]
ask links
[ show end2 ] ;; shows turtle 1
```

error

error *value*

Causes a runtime error to occur.

The given value is converted to a string (if it isn't one already) and used as the error message.

See also error-message, carefully.

error-message

error-message

Reports a string describing the error that was suppressed by carefully.

This reporter can only be used in the second block of a carefully command.

See also error, carefully.

every

every *number* **[*commands*]**

Runs the given commands only if it's been more than *number* seconds since the last time this agent ran them in this context. Otherwise, the commands are skipped.

By itself, every doesn't make commands run over and over again. You need to use every inside a loop, or inside a forever button, if you want the commands run over and over again. every only limits how often the commands run.

Above, "in this context" means during the same ask (or button press or command typed in the Command Center). So it doesn't make sense to write ask turtles [every 0.5 [...]], because when the ask finishes the turtles will all discard their timers for the "every". The correct usage is shown below.

```
every 0.5 [ ask turtles [ fd 1 ] ]
;; twice a second the turtles will move forward 1
every 2 [ set index index + 1 ]
;; every 2 seconds index is incremented
```

See also wait.

exp

exp *number*

Reports the value of e raised to the *number* power.

Note: This is the same as e ^ *number*.

export-view

export-interface
export-output
export-plot
export-all-plots
export-world

export-view *filename*
export-interface *filename*
export-output *filename*
export-plot *plotname filename*
export-all-plots *filename*
export-world *filename*

export-view writes the current contents of the current view to an external file given by the string *filename*. The file is saved in PNG (Portable Network Graphics) format, so it is recommended to supply a filename ending in ".png".

export-interface is similar, but for the whole interface tab.

Note that export-view still works when running NetLogo in headless mode, but export-interface doesn't.

export-output writes the contents of the model's output area to an external file given by the string *filename*. (If the model does not have a separate output area, the output portion of the Command Center is used.)

export-plot writes the x and y values of all points plotted by all the plot pens in the plot given by the string *plotname* to an external file given by the string *filename*. If a pen is in bar mode (mode 0) and the y value of the point plotted is greater than 0, the upper-left corner point of the bar will be exported. If the y value is less than 0, then the lower-left corner point of the bar will be exported.

export-all-plots writes every plot in the current model to an external file given by the string *filename*. Each plot is identical in format to the output of export-plot.

export-world writes the values of all variables, both built-in and user-defined, including all observer, turtle, and patch variables, the drawing, the contents of the output area if one exists, the contents of any plots and the state of the random number generator, to an external file given by the string *filename*. (The result file can be read back into NetLogo with the import-world primitive.) export-world does not save the state of open files.

export-plot, export-all-plots and export-world save files in in plain-text, "comma-separated values" (`.csv`) format. CSV files can be read by most popular spreadsheet and database programs as well as any text editor.

If you wish to export to a file in a location other than the model's location, you should include the full path to the file you wish to export. (Use the forward-slash "/" as the folder separator.)

Note that the functionality of these primitives is also available directly from NetLogo's File menu.

```
export-world "fire.csv"
;; exports the state of the model to the file fire.csv
;; located in the NetLogo folder
export-plot "Temperature" "c:/My Documents/plot.csv"
;; exports the plot named
;; "Temperature" to the file plot.csv located in
;; the C:\My Documents folder
export-all-plots "c:/My Documents/plots.csv"
;; exports all plots to the file plots.csv
;; located in the C:\My Documents folder
```

If the file already exists, it is overwritten. To avoid this you may wish to use some method of generating fresh names. Examples:

```
export-world user-new-file
export-world (word "results " date-and-time ".csv") ;; Colon characters in the time cause errors on Window
export-world (word "results " random-float 1.0 ".csv")
```

extensions

extensions [*name ...***]**

Allows the model to use primitives from the extensions with the given names. See the Extensions guide for more information.

extract-hsb

extract-hsb *color*

Reports a list of three values, the first (hue) in the range of 0 to 360, the second and third (brightness and saturation) in the range of in the range 0 to 100, of the given NetLogo *color* in the range 0 to 140, not including 140 itself.

```
show extract-hsb red
=> [3.103 80.93 84.314]
show extract-hsb cyan
=> [180 57.143 76.863]
```

See also approximate-hsb, approximate-rgb, extract-rgb.

extract-rgb

extract-rgb *color*

Reports a list of three values in the range 0 to 255 representing the levels of red, green, and blue, respectively, of the given NetLogo *color* in the range 0 to 140, not including 140 itself.

```
show extract-rgb red
=> [215 50 41]
show extract-rgb cyan
=> [84 196 196]
```

See also approximate-rgb, approximate-hsb, extract-hsb.

F

face

face *agent*

Set the caller's heading towards *agent*.

If wrapping is allowed by the topology and the wrapped distance (around the edges of the world) is shorter, face will use the wrapped path.

If the caller and the agent are at the exact same position, the caller's heading won't change.

facexy

facexy *number number*

Set the caller's heading towards the point (x,y).

If wrapping is allowed by the topology and the wrapped distance (around the edges of the world) is shorter and wrapping is allowed, facexy will use the wrapped path.

If the caller is on the point (x,y), the caller's heading won't change.

file-at-end?

file-at-end?

Reports true when there are no more characters left to read in from the current file (that was opened previously with file-open). Otherwise, reports false.

```
file-open "my-file.txt"
print file-at-end?
=> false ;; Can still read in more characters
print file-read-line
=> This is the last line in file
print file-at-end?
=> true ;; We reached the end of the file
```

See also file-open, file-close-all.

file-close

file-close

Closes a file that has been opened previously with file-open.

Note that this and file-close-all are the only ways to restart to the beginning of an opened file or to switch between file modes.

If no file is open, does nothing.

See also file-close-all, file-open.

file-close-all

file-close-all

Closes all files (if any) that have been opened previously with file-open.

See also file-close, file-open.

file-delete

file-delete *string*

Deletes the file specified as *string*

string must be an existing file with writable permission by the user. Also, the file cannot be open. Use the command file-close to close an opened file before deletion.

Note that the string can either be a file name or an absolute file path. If it is a file name, it looks in whatever the current directory is. This can be changed using the command set-current-directory. It is defaulted to the model's directory.

file-exists?

file-exists? *string*

Reports true if *string* is the name of an existing file on the system. Otherwise it reports false.

Note that the string can either be a file name or an absolute file path. If it is a file name, it looks in whatever the current directory is. This can be changed using the command set-current-directory. It defaults to to the model's directory.

file-flush

file-flush

Forces file updates to be written to disk. When you use file-write or other output commands, the values may not be immediately written to disk. This improves the performance of the file output commands. Closing a file ensures that all output is written to disk.

Sometimes you need to ensure that data is written to disk without closing the file. For example, you could be using a file to communicate with another program on your machine and want the other program to be able to see the output immediately.

file-open

file-open *string*

This command will interpret *string* as a path name to a file and open the file. You may then use the reporters file-read, file-read-line, and file-read-characters to read in from the file, or file-write, file-print, file-type, or file-show to write out to the file.

Note that you can only open a file for reading or writing but not both. The next file i/o primitive you use after this command dictates which mode the file is opened in. To switch modes, you need to close the file using file-close.

Also, the file must already exist if opening a file in reading mode.

When opening a file in writing mode, all new data will be appended to the end of the original file. If there is no original file, a new blank file will be created in its place. (You must have write permission in the file's directory.) (If you don't want to append, but want to replace the file's existing contents, use file-delete to delete it first, perhaps inside a carefully if you're not sure whether it already exists.)

Note that the string can either be a file name or an absolute file path. If it is a file name, it looks in whatever the current directory is. This can be changed using the command set-current-directory. It is defaulted to the model's directory.

```
file-open "my-file-in.txt"
print file-read-line
=> First line in file ;; File is in reading mode
file-open "C:\\NetLogo\\my-file-out.txt"
;; assuming Windows machine
file-print "Hello World" ;; File is in writing mode
```

Opening a file does not close previously opened files. You can use file-open to switch back and forth between multiple open files.

See also file-close See also file-close-all.

file-print

file-print *value*

Prints *value* to an opened file, followed by a carriage return.

This agent is *not* printed before the value, unlike file-show.

Note that this command is the file i/o equivalent of print, and file-open needs to be called before this command can be used.

See also file-show, file-type, and file-write.

file-read

file-read

This reporter will read in the next constant from the opened file and interpret it as if it had been typed in the Command Center. It reports the resulting value. The result may be a number, list, string, boolean, or the special value nobody.

Whitespace separates the constants. Each call to file-read will skip past both leading and trailing whitespace.

Note that strings need to have quotes around them. Use the command file-write to have quotes included.

Also note that the file-open command must be called before this reporter can be used, and there must be data remaining in the file. Use the reporter file-at-end? to determine if you are at the end of the file.

```
file-open "my-file.data"
print file-read + 5
;; Next value is the number 1
=> 6
print length file-read
;; Next value is the list [1 2 3 4]
=> 4
```

See also file-open and file-write.

file-read-characters

file-read-characters *number*

Reports the given *number* of characters from an opened file as a string. If there are fewer than that many characters left, it will report all of the remaining characters.

Note that it will return every character including newlines and spaces.

Also note that the file-open command must be called before this reporter can be used, and there must be data remaining in the file. Use the reporter file-at-end? to determine if you are at the end of the file.

```
file-open "my-file.txt"
print file-read-characters 5
;; Current line in file is "Hello World"
=> Hello
```

See also [file-open](#).

file-read-line

file-read-line

Reads the next line in the file and reports it as a string. It determines the end of the file by a carriage return, an end of file character or both in a row. It does not return the line terminator characters.

Also note that the [file-open](#) command must be called before this reporter can be used, and there must be data remaining in the file. Use the reporter [file-at-end?](#) to determine if you are at the end of the file.

```
file-open "my-file.txt"
print file-read-line
=> Hello World
```

See also [file-open](#).

file-show

file-show *value*

Prints *value* to an opened file, preceded by this agent agent, and followed by a carriage return. (This agent is included to help you keep track of what agents are producing which lines of output.) Also, all strings have their quotes included similar to [file-write](#).

Note that this command is the file i/o equivalent of [show](#), and [file-open](#) needs to be called before this command can be used.

See also [file-print](#), [file-type](#), and [file-write](#).

file-type

file-type *value*

Prints *value* to an opened file, *not* followed by a carriage return (unlike [file-print](#) and [file-show](#)). The lack of a carriage return allows you to print several values on the same line.

This agent is *not* printed before the value. unlike [file-show](#).

Note that this command is the file i/o equivalent of [type](#), and [file-open](#) needs to be called before this command can be used.

See also [file-print](#), [file-show](#), and [file-write](#).

file-write

file-write *value*

This command will output *value*, which can be a number, string, list, boolean, or nobody to an opened file, *not* followed by a carriage return (unlike [file-print](#) and [file-show](#)).

This agent is *not* printed before the value, unlike [file-show](#). Its output also includes quotes around strings and is prepended with a space. It will output the value in such a manner that [file-read](#) will be able to interpret it.

Note that this command is the file i/o equivalent of [write](#), and [file-open](#) needs to be called before this command can be used.

```
file-open "locations.txt"
ask turtles
  [ file-write xcor file-write ycor ]
```

See also [file-print](#), [file-show](#), and [file-type](#).

filter

filter *reporter-task list*

Reports a list containing only those items of *list* for which the task reports true -- in other words, the items satisfying the given condition.

```
show filter is-number? [1 "2" 3]
=> [1 3]
show filter [? < 3] [1 3 2]
=> [1 2]
show filter [first ? != "t"] ["hi" "there" "everyone"]
=> ["hi" "everyone"]
```

See also map, reduce, ?.

first

first *list*
first *string*

On a list, reports the first (0th) item in the list.

On a string, reports a one-character string containing only the first character of the original string.

floor

floor *number*

Reports the largest integer less than or equal to *number*.

```
show floor 4.5
=> 4
show floor -4.5
=> -5
```

See also ceiling, round, precision.

follow

follow *turtle*

Similar to ride, but, in the 3D view, the observer's vantage point is behind and above *turtle*.

See also follow-me, ride, reset-perspective, watch, subject.

follow-me

follow-me

Asks the observer to follow this turtle.

See also follow.

foreach

foreach *list command-task*
(foreach *list1 ... command-task***)**

With a single list, runs the task for each item of *list*.

```
foreach [1.1 2.2 2.6] show
=> 1.1
=> 2.2
=> 2.6
foreach [1.1 2.2 2.6] [ show (word ? " -> " round ?) ]
```

```
=> 1.1 -> 1
=> 2.2 -> 2
=> 2.6 -> 3
```

With multiple lists, runs *command-task* for each group of items from each list. So, they are run once for the first items, once for the second items, and so on. All the lists must be the same length.

Some examples make this clearer:

```
(foreach [1 2 3] [2 4 6]
    [ show word "the sum is: " (?1 + ?2) ])
=> "the sum is: 3"
=> "the sum is: 6"
=> "the sum is: 9"
(foreach list (turtle 1) (turtle 2) [3 4]
   [ ask ?1 [ fd ?2 ] ])
;; turtle 1 moves forward 3 patches
;; turtle 2 moves forward 4 patches
```

See also [map](#), [?](#).

forward
fd

forward *number*

The turtle moves forward by *number* steps, one step at a time. (If *number* is negative, the turtle moves backward.)

`fd 10` is equivalent to `repeat 10 [ jump 1 ]`. `fd 10.5` is equivalent to `repeat 10 [ jump 1 ] jump 0.5`.

If the turtle cannot move forward *number* steps because it is not permitted by the current topology the turtle will complete as many steps of 1 as it can, then stop.

See also [jump](#), [can-move?](#).

fput

fput *item list*

Adds *item* to the beginning of a list and reports the new list.

```
;; suppose mylist is [5 7 10]
set mylist fput 2 mylist
;; mylist is now [2 5 7 10]
```

G

globals

globals [*var1* ...]

This keyword, like the breed, *<breeds>*-own, patches-own, and turtles-own keywords, can only be used at the beginning of a program, before any function definitions. It defines new global variables. Global variables are "global" because they are accessible by all agents and can be used anywhere in a model.

Most often, globals is used to define variables or constants that need to be used in many parts of the program.

H

hatch

hatch *number* [*commands*]
hatch-*<breeds>* *number* [*commands*]

This turtle creates *number* new turtles. Each new turtle inherits of all its variables, including its location, from its parent. (Exceptions: each new turtle will have a new who number, and it may be of a different breed than its parent if

the `hatch-<breeds>` form is used.)

The new turtles then run *commands*. You can use the commands to give the new turtles different colors, headings, locations, or whatever. (The new turtles are created all at once, then run one at a time, in random order.)

If the hatch-*<breeds>* form is used, the new turtles are created as members of the given breed. Otherwise, the new turtles are the same breed as their parent.

```
hatch 1 [ lt 45 fd 1 ]
;; this turtle creates one new turtle,
;; and the child turns and moves away
hatch-sheep 1 [ set color black ]
;; this turtle creates a new turtle
;; of the sheep breed
```

See also create-turtles, sprout.

heading

heading
🐢

This is a built-in turtle variable. It indicates the direction the turtle is facing. This is a number greater than or equal to 0 and less than 360. 0 is north, 90 is east, and so on. You can set this variable to make a turtle turn.

See also right, left, dx, dy.

Example:

```
set heading 45       ;; turtle is now facing northeast
set heading heading + 10 ;; same effect as "rt 10"
```

hidden?

hidden?
🐢 🔗

This is a built-in turtle or link variable. It holds a boolean (true or false) value indicating whether the turtle or link is currently hidden (i.e., invisible). You can set this variable to make a turtle or link disappear or reappear.

See also hide-turtle, show-turtle, hide-link, show-link

Example:

```
set hidden? not hidden?
;; if turtle was showing, it hides, and if it was hiding,
;; it reappears
```

hide-link

hide-link
🔗

The link makes itself invisible.

Note: This command is equivalent to setting the link variable "hidden?" to true.

See also show-link.

hide-turtle
ht

hide-turtle
🐢

The turtle makes itself invisible.

Note: This command is equivalent to setting the turtle variable "hidden?" to true.

See also <u>show-turtle</u>.

histogram

histogram *list*

Histograms the values in the given list

Draws a histogram showing the frequency distribution of the values in the list. The heights of the bars in the histogram represent the numbers of values in each subrange.

Before the histogram is drawn, first any previous points drawn by the current plot pen are removed.

Any non-numeric values in the list are ignored.

The histogram is drawn on the current plot using the current plot pen and pen color. Auto scaling does not affect a histogram's horizontal range, so set-plot-x-range should be used to control the range, and the pen interval can then be set (either directly with set-plot-pen-interval, or indirectly via set-histogram-num-bars) to control how many bars that range is split up into.

Be sure that if you want the histogram drawn with bars that the current pen is in bar mode (mode 1).

For histogramming purposes the plot's X range is not considered to include the maximum X value. Values equal to the maximum X will fall outside of the histogram's range.

```
histogram [color] of turtles
;; draws a histogram showing how many turtles there are
;; of each color
```

home

home

This turtle moves to the origin (0,0). Equivalent to `setxy 0 0`.

hsb

hsb *hue saturation brightness*

Reports a RGB list when given three numbers describing an HSB color. Hue, saturation, and brightness are integers in the range 0-360, 0-100, 0-100 respectively. The RGB list contains three integers in the range of 0-255.

See also <u>rgb</u>

hubnet-broadcast

hubnet-broadcast *tag-name value*

This broadcasts *value* from NetLogo to the interface element with the name *tag-name* on the clients.

See the <u>HubNet Authoring Guide</u> for details and instructions.

hubnet-broadcast-clear-output

hubnet-broadcast-clear-output

This clears all messages printed to the text area on every client.

See also: <u>hubnet-broadcast-message</u>, <u>hubnet-send-clear-output</u>

hubnet-broadcast-message

hubnet-broadcast-message *value*

This prints the value in the text area on each client. This is the same functionality as the "Broadcast Message" button in the HubNet Control Center.

See also: hubnet-send-message

hubnet-clear-override
hubnet-clear-overrides

hubnet-clear-override *client agent-or-set variable-name*
hubnet-clear-overrides *client*

Remove overrides from the override list on *client*. `hubnet-clear-override` removes only the override for the specified variable for the specified agent or agentset. `hubnet-clear-overrides` removes all overrides from the specified client.

See also: hubnet-send-override

hubnet-clients-list

hubnet-clients-list

Reports a list containing the names of all the clients currently connected to the HubNet server.

hubnet-enter-message?

hubnet-enter-message?

Reports true if a new client just entered the simulation. Reports false otherwise. hubnet-message-source will contain the user name of the client that just logged on.

See the HubNet Authoring Guide for details and instructions.

hubnet-exit-message?

hubnet-exit-message?

Reports true if a client just exited the simulation. Reports false otherwise. hubnet-message-source will contain the user name of the client that just logged off.

See the HubNet Authoring Guide for details and instructions.

hubnet-fetch-message

hubnet-fetch-message

If there is any new data sent by the clients, this retrieves the next piece of data, so that it can be accessed by hubnet-message, hubnet-message-source, and hubnet-message-tag. This will cause an error if there is no new data from the clients.

See the HubNet Authoring Guide for details.

hubnet-kick-client

hubnet-kick-client *client-name*

Kicks the client with the given client-name. This is equivalent to clicking the client name in the HubNet Control Center and pressing the Kick button.

hubnet-kick-all-clients

hubnet-kick-all-clients

Kicks out all currently connected HubNet clients. This is equivalent to selecting all clients in the HubNet Control Center and pressing the Kick button.

hubnet-message

hubnet-message

Reports the message retrieved by hubnet-fetch-message.

See the HubNet Authoring Guide for details.

hubnet-message-source

hubnet-message-source

Reports the name of the client that sent the message retrieved by hubnet-fetch-message.

See the HubNet Authoring Guide for details.

hubnet-message-tag

hubnet-message-tag

Reports the tag that is associated with the data that was retrieved by hubnet-fetch-message. The tag will be one of the Display Names of the interface elements in the client interface.

See the HubNet Authoring Guide for details.

hubnet-message-waiting?

hubnet-message-waiting?

This looks for a new message sent by the clients. It reports true if there is one, and false if there is not.

See the HubNet Authoring Guide for details.

hubnet-reset

hubnet-reset

Starts up the HubNet system. HubNet must be started to use any of the other hubnet primitives with the exception of `hubnet-set-client-interface`.

See the HubNet Authoring Guide for details.

hubnet-reset-perspective

hubnet-reset-perspective *tag-name*

Clears watch or follow sent directly to the client. The view perspective will revert to the server perspective.

See also: hubnet-send-watch hubnet-send-follow

hubnet-send

hubnet-send *string tag-name value*

hubnet-send *list-of-strings tag-name value*

For a *string*, this sends *value* from NetLogo to the tag *tag-name* on the client that has *string* for its user name.

For a *list-of-strings*, this sends *value* from NetLogo to the tag *tag-name* on all the clients that have a user name that is in the *list-of-strings*.

Sending a message to a non-existent client, using `hubnet-send`, generates a `hubnet-exit-message`.

See the HubNet Authoring Guide for details.

hubnet-send-clear-output

hubnet-send-clear-output *string*

hubnet-send-clear-output *list-of-strings*

This clears all messages printed to the text area on the given client or clients (specified in the *string* or *list-of-strings*.

See also: [hubnet-send-message](#), [hubnet-broadcast-clear-output](#)

hubnet-send-follow

hubnet-send-follow *client-name agent radius*

Tells the client associated with *client-name* to follow *agent* showing a *radius* sized Moore neighborhood around the agent.

See also: [hubnet-send-watch](#), [hubnet-reset-perspective](#)

hubnet-send-message

hubnet-send-message *string value*

This prints `value` in the text area on the client specified by `string`.

See also: [hubnet-broadcast-message](#)

hubnet-send-override

hubnet-send-override *client-name agent-or-set variable-name*
[*reporter*]

Evaluates *reporter* for the agent or agentset indicated then sends the values to the client to "override" the value of *variable-name* only on *client-name*. This is used to change the appearance of agents in the client view, hence, only built-in variables that affect the appearance of the agent may be selected. For example, you can override the color variable of a turtle:

```
ask turtles [ hubnet-send-override client-name self "color" [ red ] ]
```

In this example assume that there is a turtles-own variable client-name which is associated with a logged in client, and all the turtles are blue. This code makes the turtle associated with each client appear red in his or her own view but not on anyone else's or on the server.

See also: [hubnet-clear-overrides](#)

hubnet-send-watch

hubnet-send-watch *client-name agent*

Tells the client associated with *client-name* to watch *agent*.

See also: [hubnet-send-follow](#), [hubnet-reset-perspective](#)

hubnet-set-client-interface

hubnet-set-client-interface *client-type client-info*

If *client-type* is "COMPUTER", *client-info* is ignored.

```
hubnet-set-client-interface "COMPUTER" []
```

Future versions of HubNet will support other client types. Even for computer clients, the meaning of the second input to this command may change.

See the [HubNet Authoring Guide](#) for details.

I

if

if *condition* [*commands*]

Reporter must report a boolean (true or false) value.

If *condition* reports true, runs *commands*.

The reporter may report a different value for different agents, so some agents may run *commands* and others don't.

```
if xcor > 0[ set color blue ]
;; turtles in the right half of the world
;; turn blue
```

See also [ifelse](), [ifelse-value]().

ifelse

ifelse *reporter* **[** *commands1* **]** **[** *commands2* **]**

Reporter must report a boolean (true or false) value.

If *reporter* reports true, runs *commands1*.

If *reporter* reports false, runs *commands2*.

The reporter may report a different value for different agents, so some agents may run *commands1* while others run *commands2*.

```
ask patches
  [ ifelse pxcor > 0
      [ set pcolor blue ]
      [ set pcolor red ] ]
;; the left half of the world turns red and
;; the right half turns blue
```

See also [if](), [ifelse-value]().

ifelse-value

ifelse-value *reporter* **[***reporter1***]** **[***reporter2***]**

Reporter must report a boolean (true or false) value.

If *reporter* reports true, the result is the value of *reporter1*.

If *reporter* reports false, the result is the value of *reporter2*.

This can be used when a conditional is needed in the context of a reporter, where commands (such as [ifelse]()) are not allowed.

```
ask patches [
  set pcolor ifelse-value (pxcor > 0) [blue] [red]
]
;; the left half of the world turns red and
;; the right half turns blue
show n-values 10 [ifelse-value (? < 5) [0] [1]]
=> [0 0 0 0 0 1 1 1 1 1]
show reduce [ifelse-value (?1 > ?2) [?1] [?2]]
  [1 3 2 5 3 8 3 2 1]
=> 8
```

See also [if](), [ifelse]().

import-drawing

import-drawing *filename*

Reads an image file into the drawing, scaling it to the size of the world, while retaining the original aspect ratio of the image. The image is centered in the drawing. The old drawing is not cleared first.

Agents cannot sense the drawing, so they cannot interact with or process images imported by import-drawing. If you need agents to sense an image, use [import-pcolors]() or [import-pcolors-rgb]().

The following image file formats are supported: BMP, JPG, GIF, and PNG. If the image format supports transparency (alpha), that information will be imported as well.

import-pcolors

import-pcolors *filename*

Reads an image file, scales it to the same dimensions as the patch grid while maintaining the original aspect ratio of the image, and transfers the resulting pixel colors to the patches. The image is centered in the patch grid. The resulting patch colors may be distorted, since the NetLogo color space does not include all possible colors. (See the Color section of the Programming Guide.) import-pcolors may be slow for some images, particularly when you have many patches and a large image with many different colors.

Since import-pcolors sets the pcolor of patches, agents can sense the image. This is useful if agents need to analyze, process, or otherwise interact with the image. If you want to simply display a static backdrop, without color distortion, see import-drawing.

The following image file formats are supported: BMP, JPG, GIF, and PNG. If the image format supports transparency (alpha), then all fully transparent pixels will be ignored. (Partially transparent pixels will be treated as opaque.)

import-pcolors-rgb

import-pcolors-rgb *filename*

Reads an image file, scales it to the same dimensions as the patch grid while maintaining the original aspect ratio of the image, and transfers the resulting pixel colors to the patches. The image is centered in the patch grid. Unlike import-pcolors the exact colors in the original image are retained. The pcolor variable of all the patches will be an RGB list rather than an (approximated) NetLogo color.

The following image file formats are supported: BMP, JPG, GIF, and PNG. If the image format supports transparency (alpha), then all fully transparent pixels will be ignored. (Partially transparent pixels will be treated as opaque.)

import-world

import-world *filename*

Reads the values of all variables for a model, both built-in and user-defined, including all observer, turtle, and patch variables, from an external file named by the given string. The file should be in the format used by the export-world primitive.

Note that the functionality of this primitive is also directly available from NetLogo's File menu.

When using import-world, to avoid errors, perform these steps in the following order:

1. Open the model from which you created the export file.
2. Press the Setup button, to get the model in a state from which it can be run.
3. Import the file.
4. Re-open any files that the model had opened with the `file-open` command.
5. If you want, press Go button to continue running the model from the point where it left off.

If you wish to import a file from a location other than the model's location, you may include the full path to the file you wish to import. See export-world for an example.

in-cone

agentset **in-cone** *distance angle*

This reporter lets you give a turtle a "cone of vision" in front of itself. The cone is defined by the two inputs, the vision distance (radius) and the viewing angle. The viewing angle may range from 0 to 360 and is centered around the turtle's current heading. (If the angle is 360, then in-cone is equivalent to in-radius.)

in-cone reports an agentset that includes only those agents from the original agentset that fall in the cone. (This can include the agent itself.)

The distance to a patch is measured from the center of the patch.

```
ask turtles
  [ ask patches in-cone 3 60
      [ set pcolor red ] ]
;; each turtle makes a red "splotch" of patches in a 60 degree
;; cone of radius 3 ahead of itself
```

in-<breed>-neighbor?
in-link-neighbor?

in-<breed>-neighbor? *agent*
in-link-neighbor? *turtle*

Reports true if there is a directed link going from *turtle* to the caller.

```
crt 2
ask turtle 0 [
  create-link-to turtle 1
  show in-link-neighbor? turtle 1  ;; prints false
  show out-link-neighbor? turtle 1 ;; prints true
]
ask turtle 1 [
  show in-link-neighbor? turtle 0  ;; prints true
  show out-link-neighbor? turtle 0 ;; prints false
]
```

in-<breed>-neighbors
in-link-neighbors

in-<breed>-neighbors
in-link-neighbors

Reports the agentset of all the turtles that have directed links coming from them to the caller.

```
crt 4
ask turtle 0 [ create-links-to other turtles ]
ask turtle 1 [ ask in-link-neighbors [ set color blue ] ] ;; turtle 0 turns blue
```

in-<breed>-from
in-link-from

in-<breed>-from *turtle*
in-link-from *turtle*

Report the directed link from *turtle* to the caller. If no link exists then it reports nobody.

```
crt 2
ask turtle 0 [ create-link-to turtle 1 ]
ask turtle 1 [ show in-link-from turtle 0 ] ;; shows link 0 1
ask turtle 0 [ show in-link-from turtle 1 ] ;; shows nobody
```

See also: out-link-to link-with

__includes

__includes [*filename ...* **]**

Causes external NetLogo source files (with the `.nls` suffix) to be included in this model. Included files may contain breed, variable, and procedure definitions. `__includes` can only be used once per file.

in-radius

agentset **in-radius** *number*

Reports an agentset that includes only those agents from the original agentset whose distance from the caller is less than or equal to *number*. (This can include the agent itself.)

The distance to or a from a patch is measured from the center of the patch.

```
ask turtles
  [ ask patches in-radius 3
      [ set pcolor red ] ]
;; each turtle makes a red "splotch" around itself
```

inspect

inspect *agent*

Opens an agent monitor for the given agent (turtle or patch).

```
inspect patch 2 4
;; an agent monitor opens for that patch
inspect one-of sheep
;; an agent monitor opens for a random turtle from
;; the "sheep" breed
```

See stop-inspecting and stop-inspecting-dead-agents

int

int *number*

Reports the integer part of number -- any fractional part is discarded.

```
show int 4.7
=> 4
show int -3.5
=> -3
```

is-agent?
is-agentset?
is-boolean?
is-command-task?
is-directed-link?
is-link?
is-link-set?
is-list?
is-number?
is-patch?
is-patch-set?
is-reporter-task?
is-string?
is-turtle?
is-turtle-set?
is-undirected-link?

is-agent? *value*
is-agentset? *value*
is-boolean? *value*
is-*<breed>*? *value*
is-command-task? *value*
is-directed-link? *value*
is-link? *value*
is-link-set? *value*
is-list? *value*
is-number? *value*
is-patch? *value*
is-patch-set? *value*
is-reporter-task? *value*
is-string? *value*
is-turtle? *value*

is-turtle-set? *value*
is-undirected-link? *value*

Reports true if *value* is of the given type, false otherwise.

item

item *index list*
item *index string*

On lists, reports the value of the item in the given list with the given index.

On strings, reports the character in the given string at the given index.

Note that the indices begin from 0, not 1. (The first item is item 0, the second item is item 1, and so on.)

```
;; suppose mylist is [2 4 6 8 10]
show item 2 mylist
=> 6
show item 3 "my-shoe"
=> "s"
```

J

jump

jump *number*

The turtle moves forward by *number* units all at once (rather than one step at a time as with the `forward` command).

If the turtle cannot jump *number* units because it is not permitted by the current topology the turtle does not move at all.

See also forward, can-move?.

L

label

label

This is a built-in turtle or link variable. It may hold a value of any type. The turtle or link appears in the view with the given value "attached" to it as text. You can set this variable to add, change, or remove a turtle or link's label.

See also label-color, plabel, plabel-color.

Example:

```
ask turtles [ set label who ]
;; all the turtles now are labeled with their
;; who numbers
ask turtles [ set label "" ]
;; all turtles now are not labeled
```

label-color

label-color

This is a built-in turtle or link variable. It holds a number greater than or equal to 0 and less than 140. This number determines what color the turtle or link's label appears in (if it has a label). You can set this variable to change the color of a turtle or link's label.

See also label, plabel, plabel-color.

Example:

```
ask turtles [ set label-color red ]
;; all the turtles now have red labels
```

last

last *list*
last *string*

On a list, reports the last item in the list.

On a string, reports a one-character string containing only the last character of the original string.

layout-circle

layout-circle *agentset radius*
layout-circle *list-of-turtles radius*

Arranges the given turtles in a circle centered on the patch at the center of the world with the given radius. (If the world has an even size the center of the circle is rounded down to the nearest patch.) The turtles point outwards.

If the first input is an agentset, the turtles are arranged in random order.

If the first input is a list, the turtles are arranged clockwise in the given order, starting at the top of the circle. (Any non-turtles in the list are ignored.)

```
;; in random order
layout-circle turtles 10
;; in order by who number
layout-circle sort turtles 10
;; in order by size
layout-circle sort-by [[size] of ?1 < [size] of ?2] turtles 10
```

layout-radial

layout-radial *turtle-set link-set root-agent*

Arranges the turtles in *turtle-set* connected by links in *link-set*, in a radial tree layout, centered around the *root-agent* which is moved to the center of the world view.

Only links in the *link-set* will be used to determine the layout. If links connect turtles that are not in *turtle-set* those turtles will remain stationary.

Even if the network does contain cycles, and is not a true tree structure, this layout will still work, although the results will not always be pretty.

```
to make-a-tree
  set-default-shape turtles "circle"
  crt 6
  ask turtle 0 [
    create-link-with turtle 1
    create-link-with turtle 2
    create-link-with turtle 3
  ]
  ask turtle 1 [
    create-link-with turtle 4
    create-link-with turtle 5
  ]
  ; do a radial tree layout, centered on turtle 0
  layout-radial turtles links (turtle 0)
end
```

layout-spring

layout-spring *turtle-set link-set spring-constant spring-length repulsion-constant*

Arranges the turtles in *turtle-set*, as if the links in *link-set* are springs and the turtles are repelling each other. Turtles that are connected by links in *link-set* but not included in *turtle-set* are treated as anchors and are not moved.

spring-constant is a measure of the "tautness" of the spring. It is the "resistance" to change in their length. spring-constant is the force the spring would exert if it's length were changed by 1 unit.

spring-length is the "zero-force" length or the natural length of the springs. This is the length which all springs try to achieve either by pushing out their nodes or pulling them in.

repulsion-constant is a measure of repulsion between the nodes. It is the force that 2 nodes at a distance of 1 unit will exert on each other.

The repulsion effect tries to get the nodes as far as possible from each other, in order to avoid crowding and the spring effect tries to keep them at "about" a certain distance from the nodes they are connected to. The result is the laying out of the whole network in a way which highlights relationships among the nodes and at the same time is crowded less and is visually pleasing.

The layout algorithm is based on the Fruchterman-Reingold layout algorithm. More information about this algorithm can be obtained here.

```
to make-a-triangle
  set-default-shape turtles "circle"
  crt 3
  ask turtle 0
  [
    create-links-with other turtles
  ]
  ask turtle 1
  [
    create-link-with turtle 2
  ]
  repeat 30 [ layout-spring turtles links 0.2 5 1 ] ;; lays the nodes in a triangle
end
```

layout-tutte

layout-tutte *turtle-set link-set radius*

The turtles that are connected by links in *link-set* but not included in *turtle-set* are placed in a circle layout with the given *radius*. There should be at least 3 agents in this agentset.

The turtles in *turtle-set* are then laid out in the following manner: Each turtle is placed at centroid (or barycenter) of the polygon formed by its linked neighbors. (The centroid is like a 2-dimensional average of the coordinates of the neighbors.)

(The purpose of the circle of "anchor agents" is to prevent all the turtles from collapsing down to one point.)

After a few iterations of this, the layout will stabilize.

This layout is named after the mathematician William Thomas Tutte, who proposed it as a method for graph layout.

```
to make-a-tree
  set-default-shape turtles "circle"
  crt 6
  ask turtle 0 [
    create-link-with turtle 1
    create-link-with turtle 2
    create-link-with turtle 3
  ]
  ask turtle 1 [
    create-link-with turtle 4
    create-link-with turtle 5
  ]
  ; place all the turtles with just one
  ; neighbor on the perimeter of a circle
  ; and then place the remaining turtles inside
  ; this circle, spread between their neighbors.
  repeat 10 [ layout-tutte (turtles with [link-neighbors = 1]) links 12 ]
end
```

left
lt

left *number*

The turtle turns left by *number* degrees. (If *number* is negative, it turns right.)

length

length *list*

length *string*

Reports the number of items in the given list, or the number of characters in the given string.

let

let *variable value*

Creates a new local variable and gives it the given value. A local variable is one that exists only within the enclosing block of commands.

If you want to change the value afterwards, use set.

Example:

```
let prey one-of sheep-here
if prey != nobody
  [ ask prey [ die ] ]
```

link

link *end1 end2* **<breed>** *end1 end2*

Given the who numbers of the endpoints, reports the link connecting the turtles. If there is no such link reports nobody. To refer to breeded links you must use the singular breed form with the endpoints.

```
ask link 0 1 [ set color green ]
;; unbreeded link connecting turtle 0 and turtle 1 will turn green
ask directed-link 0 1 [ set color red ]
;; directed link connecting turtle 0 and turtle 1 will turn red
```

See also patch-at.

link-heading

link-heading
𝄜

Reports the heading in degrees (at least 0, less than 360) from end1 to end2 of the link. Throws a runtime error if the endpoints are at the same location.

```
ask link 0 1 [ print link-heading ]
;; prints [[towards other-end] of end1] of link 0 1
```

See also link-length

link-length

link-length
𝄜

Reports the distance between the endpoints of the link.

```
ask link 0 1 [ print link-length ]
;; prints [[distance other-end] of end1] of link 0 1
```

See also link-heading

link-set

link-set *value*
(link-set *value1 value2 ...*)

Reports an agentset containing all of the links anywhere in any of the inputs. The inputs may be individual links, link agentsets, nobody, or lists (or nested lists) containing any of the above.

```
link-set self
link-set [my-links] of nodes with [color = red]
```

See also [turtle-set](#), [patch-set](#).

link-shapes

link-shapes

Reports a list of strings containing all of the link shapes in the model.

New shapes can be created, or imported from other models, in the [Link Shapes Editor](#).

```
show link-shapes
=> ["default"]
```

links

links

Reports the agentset consisting of all links.

```
show count links
;; prints the number of links
```

links-own

links-own [*var1* ...]
<link-breeds>-own [*var1* ...]

The links-own keyword, like the globals, breed, *<breeds>*-own, turtles-own, and patches-own keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables belonging to each link.

If you specify a breed instead of "links", only links of that breed have the listed variables. (More than one link breed may list the same variable.)

```
undirected-link-breed [sidewalks sidewalk]
directed-link-breed [streets street]
links-own [traffic]    ;; applies to all breeds
sidewalks-own [pedestrians]
streets-own [cars bikes]
```

list

list *value1 value2*
(list *value1* ...)

Reports a list containing the given items. The items can be of any type, produced by any kind of reporter.

```
show list (random 10) (random 10)
=> [4 9]  ;; or similar list
show (list 5)
=> [5]
show (list (random 10) 1 2 3 (random 10))
=> [4 1 2 3 9]  ;; or similar list
```

ln

ln *number*

Reports the natural logarithm of *number*, that is, the logarithm to the base e (2.71828...).

See also [e](#), [log](#).

log

log *number base*

Reports the logarithm of *number* in base *base*.

```
show log 64 2
=> 6
```

See also <u>ln</u>.

loop

loop [*commands*]

Repeats the commands forever, or until the enclosing procedure exits through use of the <u>stop</u> or <u>report</u> commands.

```
to move-to-world-edge  ;; turtle procedure
  loop [
    if not can-move? 1 [ stop ]
    fd 1
  ]
end
```

In this example, `stop` exits not just the loop, but the entire procedure.

Note: in many circumstances, it is more appropriate to use a forever button to repeat something indefinitely. See <u>Buttons</u> in the Programming Guide.

lput

lput *value list*

Adds *value* to the end of a list and reports the new list.

```
;; suppose mylist is [2 7 10 "Bob"]
set mylist lput 42 mylist
;; mylist now is [2 7 10 "Bob" 42]
```

M

map

map *reporter-task list*
(map *reporter-task list1* ...)

With a single *list*, the given task is run for each item in the list, and a list of the results is collected and reported.

```
show map round [1.1 2.2 2.7]
=> [1 2 3]
show map [? * ?] [1 2 3]
=> [1 4 9]
```

With multiple lists, the given reporter is run for each group of items from each list. So, it is run once for the first items, once for the second items, and so on. All the lists must be the same length.

Some examples make this clearer:

```
show (map + [1 2 3] [2 4 6])
=> [3 6 9]
show (map [?1 + ?2 = ?3] [1 2 3] [2 4 6] [3 5 9])
=> [true false true]
```

See also <u>foreach</u>, <u>?</u>.

max

max *list*

Reports the maximum number value in the list. It ignores other types of items.

```
show max [xcor] of turtles
;; prints the x coordinate of the turtle which is
;; farthest right in the world
show max list a b
;; prints the larger of the two variables a and b
show max (list a b c)
;; prints the largest of the three variables a, b, and c
```

max-n-of

max-n-of *number agentset* [*reporter*]

Reports an agentset containing *number* agents from *agentset* with the highest values of *reporter*. The agentset is built by finding all the agents with the highest value of *reporter*, if there are not *number* agents with that value then agents with the second highest value are found, and so on. At the end, if there is a tie that would make the resulting agentset too large, the tie is broken randomly.

```
;; assume the world is 11 x 11
show max-n-of 5 patches [pxcor]
;; shows 5 patches with pxcor = max-pxcor
show max-n-of 5 patches with [pycor = 0] [pxcor]
;; shows an agentset containing:
;; (patch 1 0) (patch 2 0) (patch 3 0) (patch 4 0) (patch 5 0)
```

See also max-one-of, with-max.

max-one-of

max-one-of *agentset* [*reporter*]

Reports the agent in the agentset that has the highest value for the given reporter. If there is a tie this command reports one random agent with the highest value. If you want all such agents, use with-max instead.

```
show max-one-of patches [count turtles-here]

;; prints the first patch with the most turtles on it
```

See also max-n-of, with-max.

max-pxcor
max-pycor

max-pxcor
max-pycor

These reporters give the maximum x-coordinate and maximum y-coordinate, (respectively) for patches, which determines the size of the world.

Unlike in older versions of NetLogo the origin does not have to be at the center of the world. However, the maximum x- and y- coordinates must be greater than or equal to zero.

Note: You can set the size of the world only by editing the view -- these are reporters which cannot be set.

```
crt 100 [ setxy random-float max-pxcor
              random-float max-pycor ]
;; distributes 100 turtles randomly in the
;; first quadrant
```

See also min-pxcor, min-pycor, world-width, and world-height

mean

mean *list*

Reports the statistical mean of the numeric items in the given list. Errors on non-numeric items. The mean is the average, i.e., the sum of the items divided by the total number of items.

```
show mean [xcor] of turtles
;; prints the average of all the turtles' x coordinates
```

median

median *list*

Reports the statistical median of the numeric items of the given list. Ignores non-numeric items. The median is the item that would be in the middle if all the items were arranged in order. (If two items would be in the middle, the median is the average of the two.)

```
show median [xcor] of turtles
;; prints the median of all the turtles' x coordinates
```

member?

member? *value list*
member? *string1 string2*
member? *agent agentset*

For a list, reports true if the given value appears in the given list, otherwise reports false.

For a string, reports true or false depending on whether *string1* appears anywhere inside *string2* as a substring.

For an agentset, reports true if the given agent is appears in the given agentset, otherwise reports false.

```
show member? 2 [1 2 3]
=> true
show member? 4 [1 2 3]
=> false
show member? "bat" "abate"
=> true
show member? turtle 0 turtles
=> true
show member? turtle 0 patches
=> false
```

See also [position](#).

min

min *list*

Reports the minimum number value in the list. It ignores other types of items.

```
show min [xcor] of turtles
;; prints the lowest x-coordinate of all the turtles
show min list a b
;; prints the smaller of the two variables a and b
show min (list a b c)
;; prints the smallest of the three variables a, b, and c
```

min-n-of

min-n-of *number agentset* [*reporter*]

Reports an agentset containing *number* agents from *agentset* with the lowest values of *reporter*. The agentset is built by finding all the agents with the lowest value of *reporter*, if there are not *number* agents with that value then the agents with the second lowest value are found, and so on. At the end, if there is a tie that would make the resulting agentset too large, the tie is broken randomly.

```
;; assume the world is 11 x 11
show min-n-of 5 patches [pxcor]
;; shows 5 patches with pxcor = min-pxcor
show min-n-of 5 patches with [pycor = 0] [pxcor]
;; shows an agentset containing:
;; (patch -5 0) (patch -4 0) (patch -3 0) (patch -2 0) (patch -1 0)
```

See also [min-one-of](#), [with-min](#).

min-one-of

min-one-of *agentset* [*reporter*]

Reports a random agent in the agentset that reports the lowest value for the given reporter. If there is a tie, this command reports one random agent that meets the condition. If you want all such agents use with-min instead.

```
show min-one-of turtles [xcor + ycor]
;; reports the first turtle with the smallest sum of
;; coordinates
```

See also with-min, min-n-of.

min-pxcor
min-pycor

min-pxcor
min-pycor

These reporters give the minimum x-coordinate and minimum y-coordinate, (respectively) for patches, which determines the size of the world.

Unlike in older versions of NetLogo the origin does not have to be at the center of the world. However, the minimum x- and y- coordinates must be less than or equal to zero.

Note: You can set the size of the world only by editing the view -- these are reporters which cannot be set.

```
crt 100 [ setxy random-float min-pxcor
               random-float min-pycor ]
;; distributes 100 turtles randomly in the
;; third quadrant
```

See also max-pxcor, max-pycor, world-width, and world-height

mod

number1 **mod** *number2*

Reports *number1* modulo *number2*: that is, the residue of *number1* (mod *number2*). mod is is equivalent to the following NetLogo code:

```
number1 - (floor (number1 / number2)) * number2
```

Note that mod is "infix", that is, it comes between its two inputs.

```
show 62 mod 5
=> 2
show -8 mod 3
=> 1
```

See also remainder. mod and remainder behave the same for positive numbers, but differently for negative numbers.

modes

modes *list*

Reports a list of the most common item or items in *list*.

The input list may contain any NetLogo values.

If the input is an empty list, reports an empty list.

```
show modes [1 2 2 3 4]
=> [2]
show modes [1 2 2 3 3 4]
=> [2 3]
show modes [ [1 2 [3]] [1 2 [3]] [2 3 4] ]
=> [[1 2 [3]]]
```

```
show modes [pxcor] of turtles
;; shows which columns of patches have the most
;; turtles on them
```

mouse-down?

mouse-down?

Reports true if the mouse button is down, false otherwise.

Note: If the mouse pointer is outside of the current view , mouse-down? will always report false.

mouse-inside?

mouse-inside?

Reports true if the mouse pointer is inside the current view, false otherwise.

mouse-xcor
mouse-ycor

mouse-xcor
mouse-ycor

Reports the x or y coordinate of the mouse in the 2D view. The value is in terms of turtle coordinates, so it might not be an integer. If you want patch coordinates, use `round mouse-xcor` and `round mouse-ycor`.

Note: If the mouse is outside of the 2D view, reports the value from the last time it was inside.

```
;; to make the mouse "draw" in red:
if mouse-down?
  [ ask patch mouse-xcor mouse-ycor [ set pcolor red ] ]
```

move-to

move-to *agent*

The turtle sets its x and y coordinates to be the same as the given agent's.

(If that agent is a patch, the effect is to move the turtle to the center of that patch.)

```
move-to turtle 5
;; turtle moves to same point as turtle 5
move-to one-of patches
;; turtle moves to the center of a random patch
move-to max-one-of turtles [size]
;; turtle moves to same point as biggest turtle
```

Note that the turtle's heading is unaltered. You may want to use the face command first to orient the turtle in the direction of motion.

See also setxy.

movie-cancel

movie-cancel

Cancels the current movie.

movie-close

movie-close

Stops the recording of the current movie.

movie-grab-view
movie-grab-interface

movie-grab-view
movie-grab-interface

Adds an image of the current view (2D or 3D) or the interface panel to the current movie.

```
;; make a 20-step movie of the current view
setup
movie-start "out.mov"
repeat 20 [
  movie-grab-view
  go
]
movie-close
```

movie-set-frame-rate

movie-set-frame-rate *frame-rate*

Sets the frame rate of the current movie. The frame rate is measured in frames per second. (If you do not explicitly set the frame rate, it defaults to 15 frames per second.)

Must be called after movie-start, but before movie-grab-view or movie-grab-interface.

See also movie-status.

movie-start

movie-start *filename*

Creates a new movie. *filename* specifies a new QuickTime file where the movie will be saved, so it should end with ".mov".

See also movie-grab-view, movie-grab-interface, movie-cancel, movie-status, movie-set-frame-rate, movie-close.

movie-status

movie-status

Reports a string describing the current movie.

```
print movie-status
=> No movie.
movie-start
print movie-status
=> 0 frames; frame rate = 15.
movie-grab-view
print movie-status
1 frames; frame rate = 15; size = 315x315.
```

my-<breeds>
my-links

my-<breeds>
my-links

Reports an agentset of all links connected to the caller, regardless of directedness.

```
crt 5
ask turtle 0
[
  create-links-with other turtles
  show my-links ;; prints the agentset containing all links
              ;; (since all the links we created were with turtle 0 )
]
ask turtle 1
[
  show my-links ;; shows an agentset containing the link 0 1
```

```
    ]
  end
```

my-in-<breeds>
my-in-links

my-in-<breeds>
my-in-links

Reports an agentset of all the directed links coming in from other nodes to the caller.

```
crt 5
ask turtle 0
[
  create-links-to other turtles
  show my-in-links ;; shows an empty agentset
]
ask turtle 1
[
  show my-in-links ;; shows an agentset containing the link 0 1
]
```

my-out-<breeds>
my-out-links

my-out-<breeds>
my-out-links

Reports an agentset of all the directed links going out from the caller to other nodes.

```
crt 5
ask turtle 0
[
  create-links-to other turtles
  show my-out-links ;; shows agentset containing all the links
]
ask turtle 1
[
  show my-out-links ;; shows an empty agentset
]
```

myself

myself

"self" and "myself" are very different. "self" is simple; it means "me". "myself" means "the turtle, patch or link who asked me to do what I'm doing right now."

When an agent has been asked to run some code, using myself in that code reports the agent (turtle, patch or link) that did the asking.

myself is most often used in conjunction with `of` to read or set variables in the asking agent.

myself can be used within blocks of code not just in the ask command, but also hatch, sprout, of, with, all?, with-min, with-max, min-one-of, max-one-of, min-n-of, max-n-of.

```
ask turtles
  [ ask patches in-radius 3
      [ set pcolor [color] of myself ] ]
;; each turtle makes a colored "splotch" around itself
```

See the "Myself Example" code example for more examples.

See also self.

N

n-of

n-of *size agentset*
n-of *size list*

From an agentset, reports an agentset of size *size* randomly chosen from the input set, with no repeats.

From a list, reports a list of size *size* randomly chosen from the input set, with no repeats. The items in the result appear in the same order that they appeared in the input list. (If you want them in random order, use shuffle on the result.)

It is an error for *size* to be greater than the size of the input.

```
ask n-of 50 patches [ set pcolor green ]
;; 50 randomly chosen patches turn green
```

See also one-of.

n-values

n-values *size reporter-task*

Reports a list of length *size* containing values computed by repeatedly running the task.

If the task accepts inputs, the input will be the number of the item currently being computed, starting from zero.

```
show n-values 5 [1]
=> [1 1 1 1 1]
show n-values 5 [?]
=> [0 1 2 3 4]
show n-values 3 turtle
=> [(turtle 0) (turtle 1) (turtle 2)]
show n-values 5 [? * ?]
=> [0 1 4 9 16]
```

See also reduce, filter, ?.

neighbors
neighbors4

neighbors
neighbors4

Reports an agentset containing the 8 surrounding patches (neighbors) or 4 surrounding patches (neighbors4).

```
show sum [count turtles-here] of neighbors
   ;; prints the total number of turtles on the eight
   ;; patches around this turtle or patch
show count turtles-on neighbors
   ;; a shorter way to say the same thing
ask neighbors4 [ set pcolor red ]
   ;; turns the four neighboring patches red
```

<breed>-neighbors
link-neighbors

<breed>-neighbors
link-neighbors

Reports the agentset of all turtles found at the other end of undirected links connected to this turtle.

```
crt 3
ask turtle 0
[
  create-links-with other turtles
  ask link-neighbors [ set color red ] ;; turtles 1 and 2 turn red
]
ask turtle 1
[
  ask link-neighbors [ set color blue ] ;; turtle 0 turns blue
]
end
```

<breed>-neighbor?
link-neighbor?

<breed>-neighbor? *turtle*
link-neighbor? *turtle*

Reports true if there is an undirected link between *turtle* and the caller.

```
crt 2
ask turtle 0
[
  create-link-with turtle 1
  show link-neighbor? turtle 1  ;; prints true
]
ask turtle 1
[
  show link-neighbor? turtle 0    ;; prints true
]
```

netlogo-applet?

netlogo-applet?

Reports true if the model is running as an applet.

netlogo-version

netlogo-version

Reports a string containing the version number of the NetLogo you are running.

```
show netlogo-version
=> "5.3.1"
```

netlogo-web?

netlogo-web?

Reports true if the model is running in NetLogo Web.

new-seed

new-seed

Reports a number suitable for seeding the random number generator.

The numbers reported by new-seed are based on the current date and time in milliseconds and lie in the generator's usable range of seeds, -2147483648 to 2147483647.

new-seed never reports the same number twice in succession, even across parallel BehaviorSpace runs. (This is accomplished by waiting a millisecond if the seed for the current millisecond was already used.)

See also [random-seed](#).

no-display

no-display

Turns off all updates to the current view until the display command is issued. This has two major uses.

One, you can control when the user sees view updates. You might want to change lots of things on the view behind the user's back, so to speak, then make them visible to the user all at once.

Two, your model will run faster when view updating is off, so if you're in a hurry, this command will let you get results faster. (Note that normally you don't need to use no-display for this, since you can also use the on/off switch in view control strip to freeze the view.)

Note that display and no-display operate independently of the switch in the view control strip that freezes the view.

See also [display](#).

nobody

nobody

This is a special value which some primitives such as turtle, one-of, max-one-of, etc. report to indicate that no agent was found. Also, when a turtle dies, it becomes equal to nobody.

Note: Empty agentsets are not equal to nobody. If you want to test for an empty agentset, use [any?](#). You only get nobody back in situations where you were expecting a single agent, not a whole agentset.

```
set target one-of other turtles-here
if target != nobody
  [ ask target [ set color red ] ]
```

no-links

no-links

Reports an empty link agentset.

no-patches

no-patches

Reports an empty patch agentset.

not

not *boolean*

Reports true if *boolean* is false, otherwise reports false.

```
if not any? turtles [ crt 10 ]
```

no-turtles

no-turtles

Reports an empty turtle agentset.

O

of

[*reporter*] of *agent*
[*reporter*] of *agentset*

For an agent, reports the value of the reporter for that agent (turtle or patch).

```
show [pxcor] of patch 3 5
;; prints 3
show [pxcor] of one-of patches
;; prints the value of a random patch's pxcor variable
show [who * who] of turtle 5
=> 25
show [count turtles in-radius 3] of patch 0 0
;; prints the number of turtles located within a
;; three-patch radius of the origin
```

For an agentset, reports a list that contains the value of the reporter for each agent in the agentset (in random order).

```
crt 4
show sort [who] of turtles
=> [0 1 2 3]
show sort [who * who] of turtles
=> [0 1 4 9]
```

one-of

one-of *agentset*
one-of *list*

From an agentset, reports a random agent. If the agentset is empty, reports nobody.

From a list, reports a random list item. It is an error for the list to be empty.

```
ask one-of patches [ set pcolor green ]
;; a random patch turns green
ask patches with [any? turtles-here]
  [ show one-of turtles-here ]
;; for each patch containing turtles, prints one of
;; those turtles

;; suppose mylist is [1 2 3 4 5 6]
show one-of mylist
;; prints a value randomly chosen from the list
```

See also n-of.

or

boolean1* or *boolean2

Reports true if either *boolean1* or *boolean2*, or both, is true.

Note that if *condition1* is true, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) or (pycor > 0) [ set pcolor red ]
;; patches turn red except in lower-left quadrant
```

other

other *agentset*

Reports an agentset which is the same as the input agentset but omits this agent.

```
show count turtles-here
=> 10
show count other turtles-here
=> 9
```

other-end

other-end

If run by a turtle, reports the turtle at the other end of the asking link.

If run by a link, reports the turtle at the end of the link that isn't the asking turtle.

These definitions are difficult to understand in the abstract, but the following examples should help:

```
ask turtle 0 [ create-link-with turtle 1 ]
ask turtle 0 [ ask link 0 1 [ show other-end ] ] ;; prints turtle 1
ask turtle 1 [ ask link 0 1 [ show other-end ] ] ;; prints turtle 0
ask link 0 1 [ ask turtle 0 [ show other-end ] ] ;; prints turtle 1
```

As these examples hopefully make plain, the "other" end is the end that is neither asking nor being asked.

out-<breed>-neighbor?
out-link-neighbor?

out-<breed>-neighbor? *turtle*
out-link-neighbor? *turtle*
🐢

Reports true if there is a directed link going from the caller to *turtle*.

```
crt 2
ask turtle 0 [
  create-link-to turtle 1
  show in-link-neighbor? turtle 1  ;; prints false
  show out-link-neighbor? turtle 1 ;; prints true
]
ask turtle 1 [
  show in-link-neighbor? turtle 0  ;; prints true
  show out-link-neighbor? turtle 0 ;; prints false
]
```

out-<breed>-neighbors
out-link-neighbors

out-<breed>-neighbors
out-link-neighbors
🐢

Reports the agentset of all the turtles that have directed links from the caller.

```
crt 4
ask turtle 0
[
  create-links-to other turtles
  ask out-link-neighbors [ set color pink ] ;; turtles 1-3 turn pink
]
ask turtle 1
[
  ask out-link-neighbors [ set color orange ]  ;; no turtles change colors
                                               ;; since turtle 1 only has in-links
]
end
```

out-<breed>-to
out-link-to

out-<breed>-to *turtle*
out-link-to *turtle*
🐢

Reports the directed link from the caller to *turtle*. If no link exists then it reports nobody.

```
crt 2
ask turtle 0 [
  create-link-to turtle 1
  show out-link-to turtle 1 ;; shows link 0 1
]
ask turtle 1
[
  show out-link-to turtle 0 ;; shows nobody
]
```

See also: in-link-from link-with

output-print
output-show
output-type
output-write

output-print *value*
output-show *value*
output-type *value*
output-write *value*

These commands are the same as the [print](), [show](), [type](), and [write]() commands except that *value* is printed in the model's output area, instead of in the Command Center. (If the model does not have a separate output area, then the Command Center is used.)

P

patch

patch *xcor ycor*

Given the x and y coordinates of a point, reports the patch containing that point. (The coordinates are absolute coordinates; they are not computed relative to this agent, as with patch-at.)

If x and y are integers, the point is the center of a patch. If x or y is not an integer, rounding to the nearest integer is used to determine which patch contains the point.

If wrapping is allowed by the topology, the given coordinates will be wrapped to be within the world. If wrapping is not allowed and the given coordinates are outside the world, reports nobody.

```
ask patch 3 -4 [ set pcolor green ]
;; patch with pxcor of 3 and pycor of -4 turns green
show patch 1.2 3.7
;; prints (patch 1 4); note rounding
show patch 18 19
;; supposing min-pxcor and min-pycor are -17
;; and max-pxcor and max-pycor are 17,
;; in a wrapping topology, prints (patch -17 -16);
;; in a non-wrapping topology, prints nobody
```

See also [patch-at]().

patch-ahead

patch-ahead *distance*

Reports the single patch that is the given distance "ahead" of this turtle, that is, along the turtle's current heading. Reports nobody if the patch does not exist because it is outside the world.

```
ask patch-ahead 1 [ set pcolor green ]
;; turns the patch 1 in front of this turtle
;;   green; note that this might be the same patch
;;   the turtle is standing on
```

See also [patch-at](), [patch-left-and-ahead](), [patch-right-and-ahead](), [patch-at-heading-and-distance]().

patch-at

patch-at *dx dy*

Reports the patch at (dx, dy) from the caller, that is, the patch containing the point dx east and dy patches north of this agent.

Reports nobody if there is no such patch because that point is beyond a non-wrapping world boundary.

```
ask patch-at 1 -1 [ set pcolor green ]
;; if caller is a turtle or patch, turns the
;;   patch just southeast of the caller green
```

See also [patch](), [patch-ahead](), [patch-left-and-ahead](), [patch-right-and-ahead](), [patch-at-heading-and-distance]().

patch-at-heading-and-distance

patch-at-heading-and-distance *heading distance*

patch-at-heading-and-distance reports the single patch that is the given distance from this turtle or patch, along the

given absolute heading. (In contrast to patch-left-and-ahead and patch-right-and-ahead, this turtle's current heading is not taken into account.) Reports nobody if the patch does not exist because it is outside the world.

```
ask patch-at-heading-and-distance -90 1 [ set pcolor green ]
;; turns the patch 1 to the west of this patch green
```

See also patch, patch-at, patch-left-and-ahead, patch-right-and-ahead.

patch-here

patch-here

patch-here reports the patch under the turtle.

Note that this reporter isn't available to a patch because a patch can just say "self".

patch-left-and-ahead
patch-right-and-ahead

patch-left-and-ahead *angle distance*
patch-right-and-ahead *angle distance*

Reports the single patch that is the given distance from this turtle, in the direction turned left or right the given angle (in degrees) from the turtle's current heading. Reports nobody if the patch does not exist because it is outside the world.

(If you want to find a patch in a given absolute heading, rather than one relative to the current turtle's heading, use patch-at-heading-and-distance instead.)

```
ask patch-right-and-ahead 30 1 [ set pcolor green ]
;; this turtle "looks" 30 degrees right of its
;;    current heading at the patch 1 unit away, and turns
;;    that patch green; note that this might be the same
;;    patch the turtle is standing on
```

See also patch, patch-at, patch-at-heading-and-distance.

patch-set

patch-set *value1*
(patch-set *value1 value2 ...*)

Reports an agentset containing all of the patches anywhere in any of the inputs. The inputs may be individual patches, patch agentsets, nobody, or lists (or nested lists) containing any of the above.

```
patch-set self
patch-set patch-here
(patch-set self neighbors)
(patch-set patch-here neighbors)
(patch-set patch 0 0 patch 1 3 patch 4 -2)
(patch-set patch-at -1 1 patch-at 0 1 patch-at 1 1)
patch-set [patch-here] of turtles
patch-set [neighbors] of turtles
```

See also turtle-set, link-set.

patch-size

patch-size

Reports the size of the patches in the view in pixels. The size is typically an integer, but may also be a floating point number.

See also set-patch-size.

patches

patches

Reports the agentset consisting of all patches.

patches-own

patches-own [*var1* ...]

This keyword, like the globals, breed, *<breed>*-own, and turtles-own keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables that all patches can use.

All patches will then have the given variables and be able to use them.

All patch variables can also be directly accessed by any turtle standing on the patch.

See also globals, turtles-own, breed, *<breeds>*-own.

pcolor

pcolor

This is a built-in patch variable. It holds the color of the patch. You can set this variable to make the patch change color.

All patch variables can be directly accessed by any turtle standing on the patch. Color can be represented either as a NetLogo color (a single number) or an RGB color (a list of 3 numbers). See details in the Colors section of the Programming Guide.

See also color.

pen-down
pd
pen-erase
pe
pen-up
pu

pen-down
pen-erase
pen-up

The turtle changes modes between drawing lines, removing lines or neither. The lines will always be displayed on top of the patches and below the turtles. To change the color of the pen set the color of the turtle using `set color`.

Note: When a turtle's pen is down, all movement commands cause lines to be drawn, including jump, setxy, and move-to.

Note: These commands are equivalent to setting the turtle variable "pen-mode" to "down" , "up", and "erase".

Note: On Windows drawing and erasing a line might not erase every pixel.

pen-mode

This is a built-in turtle variable. It holds the state of the turtle's pen. You set the variable to draw lines, erase lines or stop either of these actions. Possible values are "up", "down", and "erase".

pen-size

This is a built-in turtle variable. It holds the width of the line, in pixels, that the turtle will draw (or erase) when the pen is down (or erasing).

plabel

plabel

This is a built-in patch variable. It may hold a value of any type. The patch appears in the view with the given value "attached" to it as text. You can set this variable to add, change, or remove a patch's label.

All patch variables can be directly accessed by any turtle standing on the patch.

See also plabel-color, label, label-color.

plabel-color

plabel-color

This is a built-in patch variable. It holds a number greater than or equal to 0 and less than 140. This number determines what color the patch's label appears in (if it has a label). You can set this variable to change the color of a patch's label.

All patch variables can be directly accessed by any turtle standing on the patch.

See also plabel, label, label-color.

plot

plot *number*

Increments the x-value of the plot pen by plot-pen-interval, then plots a point at the updated x-value and a y-value of *number*. (The first time the command is used on a plot, the point plotted has an x-value of 0.)

plot-name

plot-name

Reports the name of the current plot (a string)

plot-pen-exists?

plot-pen-exists? *string*

Reports true if a plot pen with the given name is defined in the current plot. Otherwise reports false.

plot-pen-down
plot-pen-up

plot-pen-down
plot-pen-up

Puts down (or up) the current plot-pen, so that it draws (or doesn't). (By default, all pens are down initially.)

plot-pen-reset

plot-pen-reset

Clears everything the current plot pen has drawn, moves it to (0,0), and puts it down. If the pen is a permanent pen, the color, mode, and interval are reset to the default values from the plot Edit dialog.

plotxy

plotxy *number1 number2*

Moves the current plot pen to the point with coordinates (*number1*, *number2*). If the pen is down, a line, bar, or point

will be drawn (depending on the pen's mode).

plot-x-min
plot-x-max
plot-y-min
plot-y-max

plot-x-min
plot-x-max
plot-y-min
plot-y-max

Reports the minimum or maximum value on the x or y axis of the current plot.

These values can be set with the commands set-plot-x-range and set-plot-y-range. (Their default values are set from the plot Edit dialog.)

position

position *item list*
position *string1 string2*

On a list, reports the first position of *item* in *list*, or false if it does not appear.

On strings, reports the position of the first appearance *string1* as a substring of *string2*, or false if it does not appear.

Note: The positions are numbered beginning with 0, not with 1.

```
;; suppose mylist is [2 7 4 7 "Bob"]
show position 7 mylist
=> 1
show position 10 mylist
=> false
show position "in" "string"
=> 3
```

See also member?.

precision

precision *number places*

Reports *number* rounded to *places* decimal places.

If *places* is negative, the rounding takes place to the left of the decimal point.

```
show precision 1.23456789 3
=> 1.235
show precision 3834 -3
=> 4000
```

See also round, ceiling, floor.

print

print *value*

Prints *value* in the Command Center, followed by a carriage return.

This agent is *not* printed before the value, unlike show.

See also show, type, and write.

See also output-print.

pxcor
pycor

pxcor
pycor

These are built-in patch variables. They hold the x and y coordinate of the patch. They are always integers. You cannot set these variables, because patches don't move.

pxcor is greater than or equal to min-pxcor and less than or equal to max-pxcor; similarly for pycor and min-pycor and max-pycor.

All patch variables can be directly accessed by any turtle standing on the patch.

See also xcor, ycor.

R

random

random *number*

If *number* is positive, reports a random integer greater than or equal to 0, but strictly less than *number*.

If *number* is negative, reports a random integer less than or equal to 0, but strictly greater than *number*.

If *number* is zero, the result is always 0 as well.

Note: In versions of NetLogo prior to version 2.0, this primitive reported a floating point number if given a non-integer input. This is no longer the case. If you want a floating point answer, you must now use random-float instead.

```
show random 3
;; prints 0, 1,  or 2
show random -3
;; prints 0, -1, or -2
show random 3.5
;; prints 0, 1, 2, or 3
```

See also random-float.

random-float

random-float *number*

If *number* is positive, reports a random floating point number greater than or equal to 0 but strictly less than *number*.

If *number* is negative, reports a random floating point number less than or equal to 0, but strictly greater than *number*.

If *number* is zero, the result is always 0.

```
show random-float 3
;; prints a number at least 0 but less than 3,
;; for example 2.589444906014774
show random-float 2.5
;; prints a number at least 0 but less than 2.5,
;; for example 1.0897423196760796
```

random-exponential
random-gamma
random-normal
random-poisson

random-exponential *mean*
random-gamma *alpha lambda*
random-normal *mean standard-deviation*
random-poisson *mean*

Reports an accordingly distributed random number with the *mean* and, in the case of the normal distribution, the *standard-deviation*. (The standard deviation may not be negative.)

random-exponential reports an exponentially distributed random floating point number. It is equivalent to `(- mean) *`

```
ln random-float 1.0.
```

random-gamma reports a gamma-distributed random floating point number as controlled by the floating point alpha and lambda parameters. Both inputs must be greater than zero. (Note: for results with a given mean and variance, use inputs as follows: alpha = mean * mean / variance; lambda = 1 / (variance / mean).)

random-normal reports a normally distributed random floating point number.

random-poisson reports a Poisson-distributed random integer.

```
show random-exponential 2
;; prints an exponentially distributed random floating
;; point number with a mean of 2
show random-normal 10.1 5.2
;; prints a normally distributed random floating point
;; number with a mean of 10.1 and a standard deviation
;; of 5.2
show random-poisson 3.4
;; prints a Poisson-distributed random integer with a
;; mean of 3.4
```

random-pxcor
random-pycor

random-pxcor
random-pycor

Reports a random integer ranging from min-pxcor (or -y) to max-pxcor (or -y) inclusive.

```
ask turtles [
  ;; move each turtle to the center of a random patch
  setxy random-pxcor random-pycor
]
```

See also random-xcor, random-ycor.

random-seed

random-seed *number*

Sets the seed of the pseudo-random number generator to the integer part of *number*. The seed must be in the range -2147483648 to 2147483647; note that this is smaller than the full range of integers supported by NetLogo (-9007199254740992 to 9007199254740992).

See the Random Numbers section of the Programming Guide for more details.

```
random-seed 47822
show random 100
=> 50
show random 100
=> 35
random-seed 47822
show random 100
=> 50
show random 100
=> 35
```

random-xcor
random-ycor

random-xcor
random-ycor

Reports a random floating point number from the allowable range of turtle coordinates along the given axis, x or y.

Turtle coordinates range from min-pxcor - 0.5 (inclusive) to max-pxcor + 0.5 (exclusive) horizontally; vertically, substitute -y for -x.

```
ask turtles [
  ;; move each turtle to a random point
  setxy random-xcor random-ycor
]
```

See also [random-pxcor](#), [random-pycor](#).

read-from-string

read-from-string *string*

Interprets the given string as if it had been typed in the Command Center, and reports the resulting value. The result may be a number, list, string, or boolean value, or the special value "nobody".

Useful in conjunction with the [user-input](#) primitive for converting the user's input into usable form.

```
show read-from-string "3" + read-from-string "5"
=> 8
show length read-from-string "[1 2 3]"
=> 3
crt read-from-string user-input "Make how many turtles?"
;; the number of turtles input by the user
;; are created
```

reduce

reduce *reporter-task list*

Reduces a list from left to right using the given task, resulting in a single value. This means, for example, that `reduce [?1 + ?2] [1 2 3 4]` is equivalent to *(((1 + 2) + 3) + 4)* If *list* has a single item, that item is reported. It is an error to reduce an empty list.

The first input passed to the task is the result so far, and the second input is the next item in the list.

Since it can be difficult to develop an intuition about what `reduce` does, here are some simple examples which, while not useful in themselves, may give you a better understanding of this primitive:

```
show reduce + [1 2 3]
=> 6
show reduce - [1 2 3]
=> -4
show reduce [?2 - ?1] [1 2 3]
=> 2
show reduce [?1] [1 2 3]
=> 1
show reduce [?2] [1 2 3]
=> 3
show reduce sentence [[1 2] [3 [4]] 5]
=> [1 2 3 [4] 5]
show reduce [fput ?2 ?1] (fput [] [1 2 3 4 5])
=> [5 4 3 2 1]
```

Here are some more useful examples:

```
;; find the longest string in a list
to-report longest-string [strings]
  report reduce
    [ifelse-value (length ?1 >= length ?2) [?1] [?2]]
    strings
end

show longest-string ["hi" "there" "!"]
=> "there"

;; count the number of occurrences of an item in a list
to-report occurrences [x the-list]
  report reduce
    [ifelse-value (?2 = x) [?1 + 1] [?1]] (fput 0 the-list)
end

show occurrences 1 [1 2 1 3 1 2 3 1 1 4 5 1]
=> 6

;; evaluate the polynomial, with given coefficients, at x
to-report evaluate-polynomial [coefficients x]
  report reduce [(x * ?1) + ?2] coefficients
end

;; evaluate 3x^2 + 2x + 1 at x = 4
show evaluate-polynomial [3 2 1] 4
=> 57
```

remainder

remainder *number1 number2*

Reports the remainder when *number1* is divided by *number2*. This is equivalent to the following NetLogo code:

```
number1 - (int (number1 / number2)) * number2
```

```
show remainder 62 5
=> 2
show remainder -8 3
=> -2
```

See also [mod](#). mod and remainder behave the same for positive numbers, but differently for negative numbers.

remove

remove *item list*
remove *string1 string2*

For a list, reports a copy of *list* with all instances of *item* removed.

For strings, reports a copy of *string2* with all the appearances of *string1* as a substring removed.

```
set mylist [2 7 4 7 "Bob"]
set mylist remove 7 mylist
;; mylist is now [2 4 "Bob"]
show remove "to" "phototonic"
=> "phonic"
```

remove-duplicates

remove-duplicates *list*

Reports a copy of *list* with all duplicate items removed. The first of each item remains in place.

```
set mylist [2 7 4 7 "Bob" 7]
set mylist remove-duplicates mylist
;; mylist is now [2 7 4 "Bob"]
```

remove-item

remove-item *index list*
remove-item *index string*

For a list, reports a copy of *list* with the item at the given index removed.

For strings, reports a copy of *string2* with the character at the given index removed.

Note that the indices begin from 0, not 1. (The first item is item 0, the second item is item 1, and so on.)

```
set mylist [2 7 4 7 "Bob"]
set mylist remove-item 2 mylist
;; mylist is now [2 7 7 "Bob"]
show remove-item 2 "string"
=> "sting"
```

repeat

repeat *number* [*commands*]

Runs *commands number* times.

```
 pd repeat 36 [ fd 1 rt 10 ]
 ;; the turtle draws a circle
```

replace-item

replace-item *index list value*
replace-item *index string1 string2*

On a list, replaces an item in that list. *index* is the index of the item to be replaced, starting with 0. (The 6th item in a list would have an index of 5.) Note that "replace-item" is used in conjunction with "set" to change a list.

Likewise for a string, but the given character of *string1* removed and the contents of *string2* spliced in instead.

```
show replace-item 2 [2 7 4 5] 15
=> [2 7 15 5]
show replace-item 1 "cat" "are"
=> "caret"
```

report

report *value*

Immediately exits from the current to-report procedure and reports *value* as the result of that procedure. report and to-report are always used in conjunction with each other. See to-report for a discussion of how to use them.

reset-perspective
rp

reset-perspective

The observer stops watching, following, or riding any turtles (or patches). (If it wasn't watching, following, or riding anybody, nothing happens.) In the 3D view, the observer also returns to its default position (above the origin, looking straight down).

See also follow, ride, watch.

reset-ticks

reset-ticks

Resets the tick counter to zero, sets up all plots, then updates all plots (so that the initial state of the world is plotted).

Normally reset-ticks goes at the end of a setup procedure.

See also clear-ticks, tick, ticks, tick-advance, setup-plots. update-plots.

reset-timer

reset-timer

Resets the timer to zero seconds. See also timer.

Note that the timer is different from the tick counter. The timer measures elapsed real time in seconds; the tick counter measures elapsed model time in ticks.

resize-world

resize-world *min-pxcor max-pxcor min-pycor max-pycor*

Changes the size of the patch grid.

As a side effect, all turtles and links die, and the existing patch grid is discarded and new patches created.

Retaining references to old patches or patch sets is inadvisable and may subsequently cause runtime errors or other unexpected behavior.

See also set-patch-size.

reverse

reverse *list*
reverse *string*

Reports a reversed copy of the given list or string.

```
show mylist
;; mylist is [2 7 4 "Bob"]
set mylist reverse mylist
;; mylist now is ["Bob" 4 7 2]
show reverse "live"
=> "evil"
```

rgb

rgb *red green blue*

Reports a RGB list when given three numbers describing an RGB color. The numbers are range checked to be between 0 and 255.

See also hsb

ride

ride *turtle*

Set the perspective to *turtle*.

Every time *turtle* moves the observer also moves. Thus, in the 2D View the turtle will stay at the center of the view. In the 3D view it is as if looking through the eyes of the turtle. If the turtle dies, the perspective resets to the default.

See also reset-perspective, watch, follow, subject.

ride-me

ride-me

Asks the observer to ride this turtle.

See also ride.

right
rt

right *number*

The turtle turns right by *number* degrees. (If *number* is negative, it turns left.)

round

round *number*

Reports the integer nearest to *number*.

If the decimal portion of *number* is exactly .5, the number is rounded in the **positive** direction.

Note that rounding in the positive direction is not always how rounding is done in other software programs. (In particular, it does not match the behavior of StarLogoT, which always rounded numbers ending in 0.5 to the nearest even integer.) The rationale for this behavior is that it matches how turtle coordinates relate to patch coordinates in NetLogo. For example, if a turtle's xcor is -4.5, then it is on the boundary between a patch whose pxcor is -4 and a patch whose pxcor is -5, but the turtle must be considered to be in one patch or the other, so the turtle is considered to be in the patch whose pxcor is -4, because we round towards the positive numbers.

```
show round 4.2
=> 4
show round 4.5
```

```
=> 5
show round -4.5
=> -4
```

See also <u>precision</u>, <u>ceiling</u>, <u>floor</u>.

run
runresult

run *command-task*
(run *command-task input1* **...)**
run *string*
runresult *reporter-task*
(runresult *reporter-task input1* **...)**
runresult *string*

The `run` form expects a command task or a string containing commands. This agent then runs them.

The `runresult` form expects a reporter task or a string containing a reporter. This agent runs it and reports the result.

Note that you can't use `run` to define or redefine procedures. If you care about performance, note that the code must be compiled first which takes time. However, compiled bits of code are cached by NetLogo and thus using `run` on the same string over and over is much faster than running different strings. The first run, though, will be many times slower than running the same code directly, or in a command task.

Tasks are recommended over strings whenever possible. (An example of when you must use strings is if you accept pieces of code from the user of your model.)

Tasks may freely read and/or set local variables and procedure inputs. Trying to do the same with strings may or may not work and should not be relied on.

When using tasks, you can provide them with inputs, if you surround the entire call with parentheses. For example:

```
(run task [ crt ?1 [ fd ?2 ] ] 10 5)
;; creates 10 turtles and move them forward 5 steps
show (runresult task [ ?1 + ?2 ] 10 5)
=> 15
;; adds 10 and 5
```

S

scale-color

scale-color *color number range1 range2*

Reports a shade of *color* proportional to the value of *number*.

If *range1* is less than *range2*, then the larger the number, the lighter the shade of *color*. But if *range2* is less than *range1*, the color scaling is inverted.

If *number* is less than *range1*, then the darkest shade of *color* is chosen.

If *number* is greater than *range2*, then the lightest shade of *color* is chosen.

Note: for *color* shade is irrelevant, e.g. green and green + 2 are equivalent, and the same spectrum of colors will be used.

```
ask turtles [ set color scale-color red age 0 50 ]
;; colors each turtle a shade of red proportional
;; to its value for the age variable
```

self

self
🟢🟥🔗

Reports this turtle, patch, or link.

"self" and "myself" are very different. "self" is simple; it means "me". "myself" means "the agent who asked me to do

what I'm doing right now."

Note that it is always redundant to write `[foo] of self`. This is always equivalent to simply writing `foo`.

See also [myself](#).

; (semicolon)

; *comments*

After a semicolon, the rest of the line is ignored. This is useful for adding "comments" to your code -- text that explains the code to human readers. Extra semicolons can be added for visual effect.

NetLogo's Edit menu has items that let you comment or uncomment whole sections of code.

sentence
se

sentence *value1 value2*
(sentence *value1 ...*)

Makes a list out of the values. If any value is a list, its items are included in the result directly, rather than being included as a sublist. Examples make this clearer:

```
show sentence 1 2
=> [1 2]
show sentence [1 2] 3
=> [1 2 3]
show sentence 1 [2 3]
=> [1 2 3]
show sentence [1 2] [3 4]
=> [1 2 3 4]
show sentence [[1 2]] [[3 4]]
=> [[1 2] [3 4]]
show (sentence [1 2] 3 [4 5] (3 + 3) 7)
=> [1 2 3 4 5 6 7]
```

set

set *variable value*

Sets *variable* to the given value.

Variable can be any of the following:

- A global variable declared using "globals"
- The global variable associated with a slider, switch, chooser, or input box.
- A variable belonging to this agent
- If this agent is a turtle, a variable belonging to the patch under the turtle.
- A local variable created by the [let](#) command.
- An input to the current procedure.

set-current-directory

set-current-directory *string*

Sets the current directory that is used by the primitives [file-delete](#), [file-exists?](#), and [file-open](#).

The current directory is not used if the above commands are given an absolute file path. This is defaulted to the user's home directory for new models, and is changed to the model's directory when a model is opened.

Note that in Windows file paths the backslash needs to be escaped within a string by using another backslash "C:\\"

The change is temporary and is not saved with the model.

Note: in applets, this command has no effect, since applets are only allowed to read files from the same directory on the server where the model is stored.

```
set-current-directory "C:\\NetLogo"
;; Assume it is a Windows Machine
file-open "my-file.txt"
;; Opens file "C:\\NetLogo\\my-file.txt"
```

set-current-plot

set-current-plot *plotname*

Sets the current plot to the plot with the given name (a string). Subsequent plotting commands will affect the current plot.

set-current-plot-pen

set-current-plot-pen *penname*

The current plot's current pen is set to the pen named *penname* (a string). If no such pen exists in the current plot, a runtime error occurs.

set-default-shape

set-default-shape turtles *string*
set-default-shape links *string*
set-default-shape *breed string*

Specifies a default initial shape for all turtles or links, or for a particular breed of turtles or links. When a turtle or link is created, or it changes breeds, it shape is set to the given shape.

This command doesn't affect existing agents, only agents you create afterwards.

The given breed must be either turtles, links, or the name of a breed. The given string must be the name of a currently defined shape.

In new models, the default shape for all turtles is "default".

Note that specifying a default shape does not prevent you from changing an agent's shape later. Agents don't have to be stuck with their breed's default shape.

```
create-turtles 1 ;; new turtle's shape is "default"
create-cats 1    ;; new turtle's shape is "default"

set-default-shape turtles "circle"
create-turtles 1 ;; new turtle's shape is "circle"
create-cats 1    ;; new turtle's shape is "circle"

set-default-shape cats "cat"
set-default-shape dogs "dog"
create-cats 1    ;; new turtle's shape is "cat"
ask cats [ set breed dogs ]
   ;; all cats become dogs, and automatically
   ;; change their shape to "dog"
```

See also shape.

set-histogram-num-bars

set-histogram-num-bars *number*

Set the current plot pen's plot interval so that, given the current x range for the plot, there would be *number* number of bars drawn if the histogram command is called.

See also histogram.

__set-line-thickness

__set-line-thickness *number*

Specifies the thickness of lines and outlined elements in the turtle's shape.

The default value is 0. This always produces lines one pixel thick.

Non-zero values are interpreted as thickness in patches. A thickness of 1, for example, produces lines which appear

one patch thick. (It's common to use a smaller value such as 0.5 or 0.2.)

Lines are always at least one pixel thick.

This command is experimental and may change in later releases.

set-patch-size

set-patch-size *size*

Sets the size of the patches of the view in pixels. The size is typically an integer, but may also be a floating point number.

See also patch-size, resize-world.

set-plot-pen-color

set-plot-pen-color *number*

Sets the color of the current plot pen to *number*.

set-plot-pen-interval

set-plot-pen-interval *number*

Tells the current plot pen to move a distance of *number* in the x direction during each use of the plot command. (The plot pen interval also affects the behavior of the histogram command.)

set-plot-pen-mode

set-plot-pen-mode *number*

Sets the mode the current plot pen draws in to *number*. The allowed plot pen modes are:

- 0 (line mode) the plot pen draws a line connecting two points together.
- 1 (bar mode): the plot pen draws a bar of width plot-pen-interval with the point plotted as the upper (or lower, if you are plotting a negative number) left corner of the bar.
- 2 (point mode): the plot pen draws a point at the point plotted. Points are not connected.

The default mode for new pens is 0 (line mode).

setup-plots

setup-plots

For each plot, runs that plot's setup commands, including the setup code for any pens in the plot.

reset-ticks has the same effect, so in models that use the tick counter, this primitive is not normally used.

See the Plotting section of the Programming Guide for more details.

See also update-plots.

set-plot-x-range
set-plot-y-range

set-plot-x-range *min max*
set-plot-y-range *min max*

Sets the minimum and maximum values of the x or y axis of the current plot.

The change is temporary and is not saved with the model. When the plot is cleared, the ranges will revert to their default values as set in the plot's Edit dialog.

setxy

setxy *x y*

The turtle sets its x-coordinate to *x* and its y-coordinate to *y*.

Equivalent to `set xcor x set ycor y`, except it happens in one time step instead of two.

If *x* or *y* is outside the world, NetLogo will throw a runtime error, unless wrapping is turned on in the relevant dimensions. For example, with wrapping turned on in both dimensions and the default world size where `min-pxcor = -16`, `max-pxcor = 16`, `min-pycor = -16` and `max-pycor = 16`, asking a turtle to `setxy 17 17` will move it to the center of patch (-16, -16).

```
setxy 0 0
;; turtle moves to the middle of the center patch
setxy random-xcor random-ycor
;; turtle moves to a random point
setxy random-pxcor random-pycor
;; turtle moves to the center of a random patch
```

See also [move-to](#).

shade-of?

shade-of? *color1 color2*

Reports true if both colors are shades of one another, false otherwise.

```
show shade-of? blue red
=> false
show shade-of? blue (blue + 1)
=> true
show shade-of? gray white
=> true
```

shape

shape

This is a built-in turtle and link variable. It holds a string that is the name of the turtle or link's current shape. You can set this variable to change the shape. New turtles and links have the shape "default" unless the a different shape has been specified using [set-default-shape](#).

Example:

```
ask turtles [ set shape "wolf" ]
;; assumes you have made a "wolf"
;; shape in NetLogo's Turtle Shapes Editor
ask links [ set shape "link 1" ]
;; assumes you have made a "link 1" shape in
;; the Link Shapes Editor
```

See also [set-default-shape](#), [shapes](#).

shapes

shapes

Reports a list of strings containing all of the turtle shapes in the model.

New shapes can be created, or imported from the shapes library or from other models, in the [Shapes Editor](#).

```
show shapes
=> ["default" "airplane" "arrow" "box" "bug" ...
ask turtles [ set shape one-of shapes ]
```

show

show *value*

Prints *value* in the Command Center, preceded by this agent, and followed by a carriage return. (This agent is included to help you keep track of what agents are producing which lines of output.) Also, all strings have their quotes included similar to write.

See also print, type, and write.

See also output-show.

show-turtle
st

show-turtle

The turtle becomes visible again.

Note: This command is equivalent to setting the turtle variable "hidden?" to false.

See also hide-turtle.

show-link

show-link

The link becomes visible again.

Note: This command is equivalent to setting the link variable "hidden?" to false.

See also hide-link.

shuffle

shuffle *list*

Reports a new list containing the same items as the input list, but in randomized order.

```
show shuffle [1 2 3 4 5]
=> [5 2 4 1 3]
show shuffle [1 2 3 4 5]
=> [1 3 5 2 4]
```

sin

sin *number*

Reports the sine of the given angle. Assumes angle is given in degrees.

```
show sin 270
=> -1
```

size

size

This is a built-in turtle variable. It holds a number that is the turtle's apparent size. The default size is 1, which means that the turtle is the same size as a patch. You can set this variable to change a turtle's size.

sort

sort *list*
sort *agentset*

Reports a sorted list of numbers, strings, or agents.

If the input contains no numbers, strings, or agents, the result is the empty list.

If the input contains at least one number, the numbers in the list are sorted in ascending order and a new list reported; non-numbers are ignored.

Or, if the input contains at least one string, the strings in the list are sorted in ascending order and a new list reported; non-strings are ignored.

Or, if the input is an agentset or a list containing at least one agent, a sorted list of agents (never an agentset) is reported; non-agents are ignored. Agents are sorted in the same order the < operator uses.

```
show sort [3 1 4 2]
=> [1 2 3 4]
let n 0
foreach sort patches [
  ask ? [
    set plabel n
    set n n + 1
  ]
]
;; patches are labeled with numbers in left-to-right,
;; top-to-bottom order
```

See also sort-by, sort-on.

sort-by

sort-by *reporter-task list*
sort-by *reporter-task agentset*

If the input is a list, reports a new list containing the same items as the input list, in a sorted order defined by the boolean reporter task.

The two inputs to the reporter task are the values being compared. The task should report true if ?1 comes strictly before ?2 in the desired sort order, and false otherwise.

If the input is an agentset or a list of agents, reports a list (never an agentset) of agents.

If the input is a list, the sort is stable, that is, the order of items considered equal by the reporter is not disturbed. If the input is an agentset, ties are broken randomly.

```
show sort-by < [3 1 4 2]
=> [1 2 3 4]
show sort-by > [3 1 4 2]
=> [4 3 2 1]
show sort-by [length ?1 < length ?2] ["Grumpy" "Doc" "Happy"]
=> ["Doc" "Happy" "Grumpy"]
```

See also sort, sort-on.

sort-on

sort-on [*reporter***]** *agentset*

Reports a list of agents, sorted according to each agent's value for *reporter*. Ties are broken randomly.

The values must be all numbers, all strings, or all agents of the same type.

```
crt 3
show sort-on [who] turtles
=> [(turtle 0) (turtle 1) (turtle 2)]
show sort-on [(- who)] turtles
=> [(turtle 2) (turtle 1) (turtle 0)]
foreach sort-on [size] turtles
  [ ask ? [ do-something ] ]
;; turtles run "do-something" one at a time, in
;; ascending order by size
```

See also sort, sort-by.

sprout

sprout *number* **[** *commands* **]**
sprout-<*breeds***>** *number* **[** *commands* **]**

Creates *number* new turtles on the current patch. The new turtles have random integer headings and the color is randomly selected from the 14 primary colors. The turtles immediately run *commands*. This is useful for giving the new turtles different colors, headings, or whatever. (The new turtles are created all at once then run one at a time, in random order.)

If the sprout-*<breeds>* form is used, the new turtles are created as members of the given breed.

```
sprout 5
sprout-wolves 10
sprout 1 [ set color red ]
sprout-sheep 1 [ set color black ]
```

See also create-turtles, hatch.

sqrt

sqrt *number*

Reports the square root of *number*.

stamp

stamp

This turtle or link leaves an image of its shape in the drawing at its current location.

Note: The shapes made by stamp may not be pixel-for-pixel identical from computer to computer.

stamp-erase

stamp-erase

This turtle or link removes any pixels below it in the drawing inside the bounds of its shape.

Note: The shapes made by stamp-erase may not be pixel-for-pixel identical from computer to computer.

standard-deviation

standard-deviation *list*

Reports the sample standard deviation of a *list* of numbers. Ignores other types of items.

(Note that this estimates the standard deviation for a *sample*, rather than for a whole *population*, using Bessel's correction.)

```
show standard-deviation [1 2 3 4 5 6]
=> 1.8708286933869707
show standard-deviation [energy] of turtles
;; prints the standard deviation of the variable "energy"
;; from all the turtles
```

startup

startup

User-defined procedure which, if it exists, will be called when a model is first loaded in the NetLogo application.

```
to startup
  setup
end
```

startup does not run when a model is run headless from the command line, or by parallel BehaviorSpace.

stop

stop

This agent exits immediately from the enclosing procedure, ask, or ask-like construct (e.g. crt, hatch, sprout). Only the enclosing procedure or construct stops, not all execution for the agent.

```
if not any? turtles [ stop ]
;; exits if there are no more turtles
```

Note: `stop` can also be used to stop a forever button. See [Buttons](#) in the Programming Guide for details.

`stop` can also be used to stop a BehaviorSpace model run. If the go commands directly call a procedure, then when that procedure calls *stop*, the run ends.

stop-inspecting

stop-inspecting *agent*

Closes the agent monitor for the given agent (turtle or patch). In the case that no agent monitor is open, `stop-inspecting` does nothing.

```
stop-inspecting patch 2 4
;; the agent monitor for that patch closes
ask sheep [ stop-inspecting self ]
;; close all agent monitors for sheep
```

See [inspect](#) and [stop-inspecting-dead-agents](#).

stop-inspecting-dead-agents

stop-inspecting-dead-agents

Closes all agent monitors for dead agents. See [inspect](#) and [stop-inspecting](#).

subject

subject

Reports the turtle (or patch) that the observer is currently watching, following, or riding. Reports [nobody](#) if there is no such turtle (or patch).

See also [watch](#), [follow](#), [ride](#).

sublist
substring

sublist *list position1 position2*
substring *string position1 position2*

Reports just a section of the given list or string, ranging between the first position (inclusive) and the second position (exclusive).

Note: The positions are numbered beginning with 0, not with 1.

```
show sublist [99 88 77 66] 1 3
=> [88 77]
show substring "apartment" 1 5
=> "part"
```

subtract-headings

subtract-headings *heading1 heading2*

Computes the difference between the given headings, that is, the number of degrees in the smallest angle by which heading2 could be rotated to produce heading1. A positive answer means a clockwise rotation, a negative answer

counterclockwise. The result is always in the range -180 to 180, but is never exactly -180.

Note that simply subtracting the two headings using the - (minus) operator wouldn't work. Just subtracting corresponds to always rotating clockwise from heading2 to heading1; but sometimes the counterclockwise rotation is shorter. For example, the difference between 5 degrees and 355 degrees is 10 degrees, not -350 degrees.

```
show subtract-headings 80 60
=> 20
show subtract-headings 60 80
=> -20
show subtract-headings 5 355
=> 10
show subtract-headings 355 5
=> -10
show subtract-headings 180 0
=> 180
show subtract-headings 0 180
=> 180
```

sum

sum *list*

Reports the sum of the items in the list.

```
show sum [energy] of turtles
;; prints the total of the variable "energy"
;; from all the turtles
```

T

tan

tan *number*

Reports the tangent of the given angle. Assumes the angle is given in degrees.

task

task [*commands*]
task [*reporter*]
task *command-name*
task *reporter-name*

Creates and reports a task, either a command task or a reporter task, depending on the input.

See the [Tasks section](#) of the Programming Guide for details.

thickness

thickness
⚭

This is a built-in link variable. It holds a number that is the link's apparent size as a fraction of the patch size. The default thickness is 0, which means that regardless of patch-size the links will always appear 1 pixel wide. You can set this variable to change a link's thickness.

tick

tick
🌐

Advances the tick counter by one and updates all plots.

If the tick counter has not been started yet with `reset-ticks`, an error results.

Normally `tick` goes at the end of a go procedure.

See also [ticks](#), [tick-advance](#), [reset-ticks](#), [clear-ticks](#), [update-plots](#).

tick-advance

tick-advance *number*

Advances the tick counter by *number*. The input may be an integer or a floating point number. (Some models divide ticks more finely than by ones.) The input may not be negative.

When using [tick-based view updates](#), the view is normally updated every 1.0 ticks, so using `tick-advance` with a number less then 1.0 may not always trigger an update. If you want to make sure that the view is updated, you can use the `display` command.

If the tick counter has not been started yet with `reset-ticks`, an error results.

Does not update plots.

See also [tick](#), [ticks](#), [reset-ticks](#), [clear-ticks](#).

ticks

ticks

Reports the current value of the tick counter. The result is always a number and never negative.

If the tick counter has not been started yet with `reset-ticks`, an error results.

Most models use the `tick` command to advance the tick counter, in which case `ticks` will always report an integer. If the `tick-advance` command is used, then `ticks` may report a floating point number.

See also [tick](#), [tick-advance](#), [reset-ticks](#), [clear-ticks](#).

tie

tie

Ties *end1* and *end2* of the link together. If the link is a directed link *end1* is the *root turtle* and *end2* is the *leaf turtle*. The movement of the *root turtle* affects the location and heading of the *leaf turtle*. If the link is undirected the tie is reciprocal so both turtles can be considered *root turtles* and *leaf turtles*. Movement or change in heading of either turtle affects the location and heading of the other turtle.

When the root turtle moves, the leaf turtles moves the same distance, in the same direction. The heading of the leaf turtle is not affected. This works with forward, jump, and setting the xcor or ycor of the root turtle.

When the root turtle turns right or left, the leaf turtle is rotated around the root turtle the same amount. The heading of the leaf turtle is also changed by the same amount.

If the link dies, the tie relation is removed.

```
crt 2 [ fd 3 ]
;; creates a link and ties turtle 1 to turtle 0
ask turtle 0 [ create-link-to turtle 1 [ tie ] ]
```

See also [untie](#)

tie-mode

tie-mode

This is a built-in link variable. It holds a string that is the name of the tie mode the link is currently in. Using the [tie](#) and [untie](#) commands changes the mode of the link. You can also set tie-mode to "free" to create a non-rigid joint between two turtles (see the [Tie section](#) of the Programming Guide for details). By default links are not tied.

See also: [tie](#), [untie](#)

timer

timer

Reports how many seconds have passed since the command [reset-timer](#) was last run (or since NetLogo started). The potential resolution of the clock is milliseconds. (Whether you get resolution that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.)

See also [reset-timer](#).

Note that the timer is different from the tick counter. The timer measures elapsed real time in seconds; the tick counter measures elapsed model time in ticks.

to

to *procedure-name*
to *procedure-name* [*input1* ...]

Used to begin a command procedure.

```
to setup
  clear-all
  crt 500
end

to circle [radius]
  crt 100 [ fd radius ]
end
```

to-report

to-report *procedure-name*
to-report *procedure-name* [*input1* ...]

Used to begin a reporter procedure.

The body of the procedure should use `report` to report a value for the procedure. See [report](#).

```
to-report average [a b]
  report (a + b) / 2
end

to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report (- number) ]
end

to-report first-turtle?
  report who = 0   ;; reports true or false
end
```

towards

towards *agent*

Reports the heading from this agent to the given agent.

If wrapping is allowed by the topology and the wrapped distance (around the edges of the world) is shorter, towards will use the wrapped path.

Note: asking for the heading from an agent to itself, or an agent on the same location, will cause a runtime error.

```
set heading towards turtle 1
;; same as "face turtle 1"
```

See also [face](#).

towardsxy

towardsxy *x y*

Reports the heading from the turtle or patch towards the point (*x*,*y*).

If wrapping is allowed by the topology and the wrapped distance (around the edges of the world) is shorter, towardsxy will use the wrapped path.

Note: asking for the heading to the point the agent is already standing on will cause a runtime error.

See also [facexy](#).

turtle

turtle *number* **<breed>** *number*

Reports the turtle with the given who number, or [nobody](#) if there is no such turtle. For breeded turtles you may also use the single breed form to refer to them.

```
ask turtle 5 [ set color red ]
;; turtle with who number 5 turns red
```

turtle-set

turtle-set *value1*
(turtle-set *value1 value2* **...)**

Reports an agentset containing all of the turtles anywhere in any of the inputs. The inputs may be individual turtles, turtle agentsets, nobody, or lists (or nested lists) containing any of the above.

```
turtle-set self
(turtle-set self turtles-on neighbors)
(turtle-set turtle 0 turtle 2 turtle 9)
(turtle-set frogs mice)
```

See also [patch-set](#), [link-set](#).

turtles

turtles

Reports the agentset consisting of all turtles.

```
show count turtles
;; prints the number of turtles
```

turtles-at

turtles-at *dx dy*
<breeds>-at *dx dy*

Reports an agentset containing the turtles on the patch (dx, dy) from the caller. (The result may include the caller itself if the caller is a turtle.)

```
create-turtles 5 [ setxy 2 3 ]
show count [turtles-at 1 1] of patch 1 2
=> 5
```

If the name of a breed is substituted for "turtles", then only turtles of that breed are included.

turtles-here

turtles-here
<breeds>-here

Reports an agentset containing all the turtles on the caller's patch (including the caller itself if it's a turtle).

```
crt 10
ask turtle 0 [ show count turtles-here ]
=> 10
```

If the name of a breed is substituted for "turtles", then only turtles of that breed are included.

```
breed [cats cat]
breed [dogs dog]
create-cats 5
create-dogs 1
ask dogs [ show count cats-here ]
=> 5
```

turtles-on

turtles-on *agent*
turtles-on *agentset*
<breeds>-on *agent*
<breeds>-on *agentset*

Reports an agentset containing all the turtles that are on the given patch or patches, or standing on the same patch as the given turtle or turtles.

```
ask turtles [
  if not any? turtles-on patch-ahead 1
    [ fd 1 ]
]
ask turtles [
  if not any? turtles-on neighbors [
    die-of-loneliness
  ]
]
```

If the name of a breed is substituted for "turtles", then only turtles of that breed are included.

turtles-own

turtles-own [*var1* ...]
<breeds>-own [*var1* ...]

The turtles-own keyword, like the globals, breed, *<breeds>*-own, and patches-own keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables belonging to each turtle.

If you specify a breed instead of "turtles", only turtles of that breed have the listed variables. (More than one turtle breed may list the same variable.)

```
breed [cats cat ]
breed [dogs dog]
breed [hamsters hamster]
turtles-own [eyes legs]   ;; applies to all breeds
cats-own [fur kittens]
hamsters-own [fur cage]
dogs-own [hair puppies]
```

See also globals, patches-own, breed, *<breeds>*-own.

type

type *value*

Prints *value* in the Command Center, *not* followed by a carriage return (unlike print and show). The lack of a carriage return allows you to print several values on the same line.

This agent is *not* printed before the value. unlike show.

```
type 3 type " " print 4
=> 3 4
```

See also print, show, and write.

See also output-type.

U

undirected-link-breed

undirected-link-breed [<*link-breeds*> <*link-breed*>]

This keyword, like the globals and breeds keywords, can only be used at the beginning of the Code tab, before any procedure definitions. It defines an undirected link breed. Links of a particular breed are always either all directed or all undirected. The first input defines the name of the agentset associated with the link breed. The second input defines the name of a single member of the breed.

Any link of the given link breed:

- is part of the agentset named by the link breed name
- has its built-in variable `breed` set to that agentset
- is directed or undirected as declared by the keyword

Most often, the agentset is used in conjunction with ask to give commands to only the links of a particular breed.

```
undirected-link-breed [streets street]
undirected-link-breed [highways highway]
to setup
  clear-all
  crt 2
  ask turtle 0 [ create-street-with turtle 1 ]
  ask turtle 0 [ create-highway-with turtle 1 ]
end

ask turtle 0 [ show sort my-links ]
;; prints [(street 0 1) (highway 0 1)]
```

See also breed, directed-link-breed

untie

untie

Unties *end2* from *end1* (sets tie-mode to "none") if they were previously tied together. If the link is an undirected link, then it will untie *end1* from *end2* as well. It does **not** remove the link between the two turtles.

See also tie

See the Tie section of the Programming Guide for more details.

update-plots

update-plots

For each plot, runs that plot's update commands, including the update code for any pens in the plot.

tick has the same effect, so in models that use the tick counter, this primitive is not normally used. Models that use fractional ticks may need `update-plots`, since tick-advance does not update the plots.

See the Plotting section of the Programming Guide for more details.

See also setup-plots.

uphill
uphill4

**uphill *patch-variable*
uphill4 *patch-variable***

Moves the turtle to the neighboring patch with the highest value for *patch-variable*. If no neighboring patch has a higher value than the current patch, the turtle stays put. If there are multiple patches with the same highest value, the turtle picks one randomly. Non-numeric values are ignored.

uphill considers the eight neighboring patches; uphill4 only considers the four neighbors.

Equivalent to the following code (assumes variable values are numeric):

```
move-to patch-here  ;; go to patch center
let p max-one-of neighbors [patch-variable]  ;; or neighbors4
if [patch-variable] of p > patch-variable [
  face p
  move-to p
]
```

Note that the turtle always ends up on a patch center and has a heading that is a multiple of 45 (uphill) or 90 (uphill4).

See also downhill, downhill4.

user-directory

user-directory

Opens a dialog that allows the user to choose an existing directory on the system.

It reports a string with the absolute path or false if the user cancels.

```
set-current-directory user-directory
;; Assumes the user will choose a directory
```

user-file

user-file

Opens a dialog that allows the user to choose an existing file on the system.

It reports a string with the absolute file path or false if the user cancels.

```
file-open user-file
;; Assumes the user will choose a file
```

user-new-file

user-new-file

Opens a dialog that allows the user to choose a location and name of a new file to be created. It reports a string with the absolute file path or false if the user cancels.

```
file-open user-new-file
;; Assumes the user will choose a file
```

Note that this reporter doesn't actually create the file; normally you would create the file using `file-open`, as in the example.

If the user chooses an existing file, they will be asked if they wish to replace it or not, but the reporter itself doesn't cause the file to be replaced. To do that you would use `file-delete`.

user-input

user-input *value*

Reports the string that a user types into an entry field in a dialog with title *value*.

value may be of any type, but is typically a string.

```
show user-input "What is your name?"
```

user-message

user-message *value*

Opens a dialog with *value* displayed as the message.

value may be of any type, but is typically a string.

```
user-message (word "There are " count turtles " turtles.")
```

user-one-of

user-one-of *value list-of-choices*

Opens a dialog with *value* displayed as the message and *list-of-choices* displayed as a popup menu for the user to select from.

Reports the item in *list-of-choices* selected by the user.

value may be of any type, but is typically a string.

```
if "yes" = user-one-of "Set up the model?" ["yes" "no"]
  [ setup ]
```

user-yes-or-no?

user-yes-or-no? *value*

Reports true or false based on the user's response to *value*.

value may be of any type, but is typically a string.

```
if user-yes-or-no? "Set up the model?"
  [ setup ]
```

V

variance

variance *list*

Reports the sample variance of a *list* of numbers. Ignores other types of items.

(Note that this computes an unbiased estimate of the variance for a *sample*, rather than for a whole *population*, using Bessel's correction.)

The sample variance is the sum of the squares of the deviations of the numbers from their mean, divided by one less than the number of numbers in the list.

```
show variance [2 7 4 3 5]
=> 3.7
```

W

wait

wait *number*

Wait the given number of seconds. (This needn't be an integer; you can specify fractions of seconds.) Note that you can't expect complete precision; the agent will never wait less than the given amount, but might wait slightly more.

```
repeat 10 [ fd 1 wait 0.5 ]
```

While the agent is waiting, no other agents can do anything. Everything stops until the agent is done.

See also every.

watch

watch *agent*

Puts a spotlight on *agent*. In the 3D view the observer will also turn to face the subject.

See also [follow](), [subject](), [reset-perspective](), [watch-me]().

watch-me

watch-me

Asks the observer to watch this agent.

See also [watch]().

while

while [*reporter*] [*commands*]

If *reporter* reports false, exit the loop. Otherwise run *commands* and repeat.

The reporter may have different values for different agents, so some agents may run *commands* a different number of times than other agents.

```
while [any? other turtles-here]
  [ fd 1 ]
;; turtle moves until it finds a patch that has
;; no other turtles on it
```

who

who

This is a built-in turtle variable. It holds the turtle's "who number" or ID number, an integer greater than or equal to zero. You cannot set this variable; a turtle's who number never changes.

Who numbers start at 0. A dead turtle's number will not be reassigned to a new turtle until you use the [clear-turtles]() or [clear-all]() commands, at which time who numbering starts over again at 0.

Example:

```
show [who] of turtles with [color = red]
;; prints a list of the who numbers of all red turtles
;; in the Command Center, in random order
crt 100
  [ ifelse who < 50
      [ set color red ]
      [ set color blue ] ]
;; turtles 0 through 49 are red, turtles 50
;; through 99 are blue
```

You can use the turtle reporter to retrieve a turtle with a given who number. See also [turtle]().

Note that who numbers aren't breed-specific. No two turtles can have the same who number, even if they are different breeds:

```
clear-turtles
create-frogs 1
create-mice 1
ask turtles [ print who ]
;; prints (in some random order):
;; (frog 0): 0
;; (mouse 1): 1
```

Even though we only have one mouse, it is `mouse 1` not `mouse 0`, because the who number 0 was already taken by the frog.

with

agentset with [*reporter*]

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a boolean reporter. Reports a new agentset containing only those agents that reported true -- in other words, the agents satisfying the given condition.

```
show count patches with [pcolor = red]
;; prints the number of red patches
```

<breed>-with
link-with

<breed>-with *turtle*
link-with *turtle*

Report the undirected link between *turtle* and the caller. If no link exists then it reports nobody.

```
crt 2
ask turtle 0 [
  create-link-with turtle 1
  show link-with turtle 1 ;; prints link 0 1
]
```

See also: in-link-from, out-link-to

with-max

agentset with-max [*reporter*]

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a reporter. Reports a new agentset containing all agents reporting the maximum value of the given reporter.

```
show count patches with-max [pxcor]
;; prints the number of patches on the right edge
```

See also max-one-of, max-n-of.

with-min

agentset with-min [*reporter*]

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a reporter. Reports a new agentset containing only those agents that have the minimum value of the given reporter.

```
show count patches with-min [pycor]
;; prints the number of patches on the bottom edge
```

See also min-one-of, min-n-of.

with-local-randomness

with-local-randomness [*commands*]

The commands are run without affecting subsequent random events. This is useful for performing extra operations (such as output) without changing the outcome of a model.

Example:

```
;; Run #1:
random-seed 50 setup repeat 10 [ go ]
;; Run #2:
random-seed 50 setup
with-local-randomness [ watch one-of turtles ]
```

```
repeat 10 [ go ]
```

Since `one-of` is used inside `with-local-randomness`, both runs will be identical.

Specifically how it works is, the state of the random number generator is remembered before the commands run, then restored afterwards. (If you want to run the commands with a fresh random state instead of the same random state that will be restored later, you can begin the commands with `random-seed new-seed`.)

The following example demonstrates that the random number generator state is the same both before the commands run and afterwards.

```
random-seed 10
with-local-randomness [ print n-values 10 [random 10] ]
;; prints [8 9 8 4 2 4 5 4 7 9]
print n-values 10 [random 10]
;; prints [8 9 8 4 2 4 5 4 7 9]
```

without-interruption

without-interruption [*commands*]

This primitive exists only for backwards compatibility. We don't recommend using it in new models.

The agent runs all the commands in the block without allowing other agents using `ask-concurrent` to "interrupt". That is, other agents are put "on hold" and do not run any commands until the commands in the block are finished.

Note: This command is only useful in conjunction with `ask-concurrent`.

See also [ask-concurrent](#).

word

word *value1 value2*
(word *value1 ...*)

Concatenates the inputs together and reports the result as a string.

```
show word "tur" "tle"
=> "turtle"
word "a" 6
=> "a6"
set directory "c:\\foo\\fish\\"
show word directory "bar.txt"
=> "c:\foo\fish\bar.txt"
show word [1 54 8] "fishy"
=> "[1 54 8]fishy"
show (word 3)
=> "3"
show (word "a" "b" "c" 1 23)
=> "abc123"
```

world-width
world-height

world-width
world-height

These reporters give the total width and height of the NetLogo world.

The width equals max-pxcor - min-pxcor + 1 and the height equals max-pycor - min-pycor + 1.

See also [max-pxcor](#), [max-pycor](#), [min-pxcor](#), and [min-pycor](#)

wrap-color

wrap-color *number*

wrap-color checks whether *number* is in the NetLogo color range of 0 to 140 (not including 140 itself). If it is not, wrap-color "wraps" the numeric input to the 0 to 140 range.

The wrapping is done by repeatedly adding or subtracting 140 from the given number until it is in the 0 to 140 range.

(This is the same wrapping that is done automatically if you assign an out-of-range number to the color turtle variable or pcolor patch variable.)

```
show wrap-color 150
=> 10
show wrap-color -10
=> 130
```

write

write *value*

This command will output *value*, which can be a number, string, list, boolean, or nobody to the Command Center, *not* followed by a carriage return (unlike print and show).

This agent is *not* printed before the value, unlike show. Its output also includes quotes around strings and is prepended with a space.

```
write "hello world"
=>  "hello world"
```

See also print, show, and type.

See also output-write.

X

xcor

xcor

This is a built-in turtle variable. It holds the current x coordinate of the turtle. You can set this variable to change the turtle's location.

This variable is always greater than or equal to (min-pxcor - 0.5) and strictly less than (max-pxcor + 0.5).

See also setxy, ycor, pxcor, pycor,

xor

boolean1 xor boolean2

Reports true if either *boolean1* or *boolean2* is true, but not when both are true.

```
if (pxcor > 0) xor (pycor > 0)
  [ set pcolor blue ]
;; upper-left and lower-right quadrants turn blue
```

Y

ycor

ycor

This is a built-in turtle variable. It holds the current y coordinate of the turtle. You can set this variable to change the turtle's location.

This variable is always greater than or equal to (min-pycor - 0.5) and strictly less than (max-pycor + 0.5).

See also setxy, xcor, pxcor, pycor,

?

?, ?1, ?2, ?3, ...

These special variable names refer to the inputs to a task, in order by number.

`?` is always equivalent to `?1`.

You may not set these variables, and you may not use them except in the context of a task.

Tasks are commonly used with the primitives foreach, map, reduce, filter, sort-by, and n-values. See those entries for example usage.

See the Tasks section of the Programming Guide for more details.