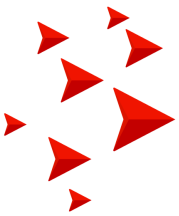


NetLogo

Version 7

The NetLogo 7.1.0-alpha1 User Manual

Table of Contents

1. The NetLogo 7.1.0-alpha1 User Manual

2. What is NetLogo?

2.1 Features

2.2 What is Agent-Based Modeling?

2.2.1 Why is NetLogo Useful for Agent-Based Modeling?

3. Copyright and License Information

3.1 How to reference

3.2 Acknowledgments

3.3 NetLogo license

3.4 Commercial licenses

3.5 NetLogo User Manual license

3.6 Open source

3.7 Third party licenses

3.7.1 Scala

3.7.2 MersenneTwisterFast

3.7.3 Colt

3.7.4 Config

3.7.5 Apache Commons Codec (TM)

3.7.6 Flexmark

3.7.7 JHotDraw

3.7.8 JOGL

3.7.9 Matrix3D

3.7.10 ASM

3.7.11 PicoContainer

3.7.12 Parboiled

3.7.13 RSyntaxTextArea

3.7.14 JCodec

3.7.15 Java-Objective-C Bridge

3.7.16 Webcam-capture

3.7.17 Guava

3.7.18 Gephi

3.7.19 JSSC

3.7.20 Apache Commons CSV

3.7.21 JTS Topology Suite

3.7.22 Apache Commons HTTPClient

3.7.23 Apache Commons Logging
3.7.24 ng-units
3.7.25 Tinfour
3.7.26 json-simple
3.7.27 hid4java
3.7.28 JAMA
3.7.29 Gson
4. Version History
4.1 Version 7.1.0-alpha1 (August 2026)
4.1.1 Biggest new feature
4.1.2 Major new features and changes
4.1.3 NetLogo Web Changes
4.1.4 Minor Features and Changes
4.1.5 Bugs Fixed
4.2 Models Library Changes
4.2.1 New Sample Models
4.2.2 Sample Model Changes
4.3 Version 7.0.4 (April 2026)
4.3.1 New Features
4.3.2 Bugs Fixed
4.4 Version 7.0.3 (November 2025)
4.5 Version 7.0.2 (October 2025)
4.5.1 Major bugs fixed
4.6 Version 7.0.1 (October 2025)
4.6.1 Bundled software to open multiple instances of NetLogo on Mac
4.6.2 Interface changes
4.6.3 Changes to NetLogo Web export
4.6.4 Major bugs fixed
4.6.5 Minor bugs fixed
4.7 Version 7.0.0 (September 2025)
4.7.1 Changes in NetLogo Functionality
4.7.2 Changes to the NetLogo GUI
4.7.3 Changes to the NetLogo File format
4.7.4 Changes in Primitive Functionality
4.7.5 Changes for Customizability
4.7.6 Features and Enhancements
4.7.7 Language Changes
4.7.8 Bug Fixes and Minor Changes
4.7.9 Extension Updates
4.7.10 Documentation Updates

4.7.11 Models Library Changes
4.8 Version 6.4.0 (November 2023)
4.8.1 BehaviorSpace New Features
4.8.2 Language Changes
4.8.3 Bug Fixes and Changes
4.8.4 Extension Updates
4.8.5 Documentation Updates
4.8.6 Models Library Changes
4.9 Version 6.3.0 (September 2022)
4.9.1 Bug Fixes and Changes
4.9.2 Extension Updates
4.9.3 Documentation Updates
4.9.4 Models Library Changes
4.10 Version 6.2.2 (December 2021)
4.10.1 Fixes and Changes
4.11 Version 6.2.1 (October 2021)
4.11.1 Features and Changes
4.11.2 Documentation Updates
4.11.3 Bugfixes
4.11.4 Models Library Changes
4.12 Version 6.2.0 (December 2020)
4.12.1 Features
4.12.2 Bugfixes
4.12.3 Logging Improvements
4.12.4 NetLogo 3D Bugfixes
4.12.5 Extension Changes
4.12.6 Documentation Changes
4.12.7 Model Changes
4.12.8 Alternative Visualization Updates:
4.13 Version 6.1.1 (September 2019)
4.13.1 Bugfixes
4.13.2 Features
4.13.3 Extension Changes
4.13.4 Documentation Changes
4.13.5 Model Changes
4.14 Version 6.1.0 (May 2019)
4.14.1 Feature Changes
4.14.2 Bugfixes
4.14.3 Extension Changes
4.14.4 Documentation Changes
4.14.5 Model Changes

4.15 Version 6.0.4 (June 2018)
4.15.1 Feature Changes
4.15.2 Bugfixes
4.15.3 Documentation Changes
4.15.4 Model Changes
4.16 Version 6.0.3 (March 2018)
4.16.1 Feature Changes
4.16.2 Documentation Changes
4.16.3 Bugfixes
4.16.4 Extension Changes
4.16.5 Documentation Changes
4.16.6 Model Changes
4.17 Version 6.0.2 (August 2017)
4.17.1 Feature Changes
4.17.2 Bugfixes
4.17.3 Extension Changes
4.17.4 Documentation Changes
4.17.5 Model Changes
4.18 Version 6.0.1 (March 2017)
4.18.1 Feature Changes
4.18.2 Bugfixes
4.18.3 Extension Changes
4.18.4 Documentation Changes
4.18.5 Models
4.19 Version 6.0 (December 2016)
4.19.1 Feature Changes
4.19.2 Bugfixes
4.19.3 Language Changes
4.19.4 Extension Changes
4.19.5 Operating System Support
4.19.6 Documentation Changes
4.19.7 Internationalization Changes
4.19.8 Models
4.20 Version 5.3.1 (February 2016)
4.20.1 Feature Changes
4.20.2 Extension Changes
4.20.3 Bugfixes
4.21 Version 5.3 (December 2015)
4.21.1 Feature Changes
4.21.2 Extension Changes
4.22 Version 5.2.1 (September 2015)

4.22.1 Extensions
4.22.2 New features
4.22.3 Bug fixes
4.22.4 Model changes
4.23 Version 5.2.0 (April 2015)
4.23.1 Extensions
4.23.2 New features
4.23.3 Bug fixes
4.23.4 Model changes
4.24 Version 5.1.0 (July 2014)
4.25 Version 5.0.4 (March 2013)
4.26 Version 5.0 (February 2012)
4.27 Version 4.1.3 (April 2011)
4.28 Version 4.1 (December 2009)
4.29 Version 4.0 (September 2007)
4.30 Version 3.1 (April 2006)
4.31 Version 3.0 (September 2005)
4.32 Version 2.1 (December 2004)
4.33 Version 2.0.2 (August 2004)
4.34 Version 2.0 (December 2003)
4.35 Version 1.3 (June 2003)
4.36 Version 1.2 (March 2003)
4.37 Version 1.1 (July 2002)
4.38 Version 1.0 (April 2002)
5. System Requirements
5.1 Application Requirements
5.1.1 Windows
5.1.2 Mac OS X
5.1.3 Linux
5.2 3D Requirements
5.3 32-bit or 64-bit?
6. Contact Information
6.1 Web site
6.2 Feedback, questions, etc.
6.3 Reporting bugs
6.4 Open source
7. Sample Model

7.1 At a Party
7.2 Challenge
7.3 Thinking with models
7.4 What's next?
8. Tutorial #1: Models
8.1 Sample Model: Wolf Sheep Predation
8.2 Controlling the Model: Buttons
8.3 Controlling speed: Speed Slider
8.4 Adjusting Settings: Choosers, Sliders, and Switches
8.5 Gathering Information: Plots and Monitors
8.5.1 Plots
8.5.2 Monitors
8.6 Controlling the View
8.7 Models Library
8.7.1 Sample Models
8.7.2 Curricular Models
8.7.3 Code Examples
8.7.4 HubNet Activities
8.8 What's Next?
9. Tutorial #2: Commands
9.1 Sample Model: Traffic Basic
9.2 Command Center
9.3 Working with colors
9.4 Agent Monitors and Agent Commanders
9.5 What's Next?
10. Tutorial #3: Procedures
10.1 Agents and procedures
10.2 Making the setup button
10.3 Switching to tick-based view updates
10.4 Making the go button
10.5 Experimenting with commands
10.6 Patches and variables
10.7 Turtle variables
10.8 Monitors
10.9 Switches and labels
10.10 More procedures

10.11 Plotting
10.12 Tick counter
10.13 Some more details
10.14 What's next?
10.15 Appendix: Complete code
11. Interface Guide
11.1 Menus
11.2 Tabs
11.3 International Usage
11.3.1 Character sets
11.3.2 Languages
11.3.3 Support for translators
12. Interface Tab
12.1 Overview: Working with interface elements (widgets)
12.2 Charts: Interface Widgets
12.2.1 Widget Appearance: Light and Dark Themes
12.2.2 Widget Icons and Functionality
12.3 Other Interface Toolbar controls
12.4 Interface Menus
12.4.1 Add Widgets Menu
12.4.2 Align Widgets Menu
12.5 Interface Tools
12.5.1 Interaction Tool
12.5.2 Selection, Editing and Deletion Tools
12.5.3 Selection Tool
12.5.4 Editing Tool
12.5.5 Deletion Tool
12.6 The 2D and 3D views
12.6.1 World Settings
12.6.2 View Settings
12.6.3 Tick Counter Settings
12.6.4 Interacting with Agents
12.6.5 Manipulating the 3D View
12.7 Command Center
12.8 Plots
12.8.1 Plot Pens
12.9 Agent Monitors
13. Info Tab Guide

13.1 Editing
13.2 Headings
13.3 Paragraphs
13.4 Italicized and bold text
13.5 Ordered lists
13.6 Unordered lists
13.7 Links
13.7.1 Automatic links
13.7.2 Links with text
13.7.3 Local links
13.8 Images
13.8.1 Local images
13.8.2 Bundled images
13.9 Block quotations
13.10 Code
13.11 Code blocks
13.12 Superscripts and subscripts
13.13 Notes on usage
13.14 Other features
14. Code Tab
14.1 Code Tab Toolbar
14.1.1 Procedures menu
14.2 Checking for Errors
14.3 Separate Code tab
14.4 Find & Replace
14.5 Automatic Indentation
14.6 Line Numbers
14.7 Code Completion
14.8 More Editing Options
14.9 Included Files
15. Programming Guide
15.1 Agents and Agentsets
15.1.1 Agents
15.1.2 Agentsets
15.1.3 Breeds
15.1.4 Variables
15.1.5 Links

15.1.6 Tie
15.1.7 Drawing
15.1.8 Turtle shapes
15.1.9 Link shapes
15.2 Programming
15.2.1 Procedures
15.2.2 ask
15.2.3 ask-concurrent
15.2.4 Syntax
15.2.5 Multiple source files
15.2.6 Buttons
15.2.7 Anonymous procedures
15.3 World
15.3.1 Tick counter
15.3.2 View updates
15.3.3 Topology
15.4 Observer Perspective
15.5 Input/Output
15.5.1 Output
15.5.2 File I/O
15.6 Lists
15.7 Strings
15.8 Mathematics
15.8.1 Math
15.8.2 Random numbers
15.9 Plotting
15.10 Color
15.11 Movies
16. NetLogo 7 Introduction
16.1 Motivation
16.2 Interface Tab
16.2.1 Overview of Visual Changes to the Interface Tab
16.2.2 Interface Elements (Widgets)
16.2.3 Widget Sizes
16.3 New File Format
16.4 Tools in the NetLogo 7 Interface
16.4.1 Overview: NetLogo Interface Tool Buttons and Dropdown menus
16.4.2 New Interface Tools
16.4.3 New Interface Menu

16.5 The Color Picker
16.6 Notifications
16.7 The Info Tab
16.8 Code Tab
16.8.1 Color Themes and Code Syntax Highlighting
16.8.2 Code Tab Toolbar
16.8.3 Code Tab Preferences
16.8.4 Included Files
16.9 Menu Items
16.9.1 Preferences Dialog
16.9.2 File → Resource Manager dialog
16.10 Tools Menu Changes
16.10.1 Turtle Shapes Editor and Link Shapes Editor
16.10.2 Help Menu
16.11 Collection of anonymous usage statistics
17. NetLogo Preferences
17.1 Overview
17.2 General Preferences
17.2.1 User Interface language
17.2.2 Reload the last used model on startup
17.2.3 Reload model on external changes
17.2.4 Enable remote commands
17.2.5 Bold widget text
17.2.6 Jump sliders to click location (instead of incrementing)
17.2.7 UI Scale
17.2.8 Send anonymous usage statistics
17.3 Code Preferences
17.3.1 Sort order of procedures menu
17.3.2 Always show “Included Files” menu
17.3.3 Auto-focus text or tab with errors when model is compiled
17.3.4 Open code tab in separate window on startup
17.3.5 Auto Indent
17.3.6 Show line numbers
17.3.7 Show auto-complete popup while typing
17.3.8 Code Font
17.4 Logging Preferences
17.4.1 Enable logging
17.4.2 Directory to store logs
17.4.3 Events to log:

17.5 Color Themes
17.5.1 Light
17.5.2 Dark
17.5.3 Classic
17.5.4 Automatic
18. Transition Guide
18.1 Changes for NetLogo 7.0.0
18.2 Changes for NetLogo 6.1.0
18.2.1 CF Extension Removal
18.2.2 ifelse-value Precedence Change with Infix Operators
18.3 Changes for NetLogo 6.0.3
18.3.1 Arduino Extension Changes
18.3.2 CF Extension Changes
18.4 Changes for NetLogo 6.0
18.4.1 Tasks replaced by Anonymous Procedures
18.4.2 Link reporters overhauled to be more consistent and flexible
18.4.3 Removal of Applets
18.4.4 Changes to the NetLogo User Interface
18.4.5 Nobody Not Permitted as a Chooser Value
18.4.6 Breeds must have singular and plural names
18.4.7 Removal of “Movie” Prims
18.4.8 Improved Name Collision Detection
18.4.9 Removal of hubnet-set-client-interface
18.4.10 Improved & Updated Extensions API
18.4.11 Add range primitive
18.5 Changes for NetLogo 5.2
18.5.1 hsb primitives
18.5.2 GoGo extension
18.6 Changes for NetLogo 5.0
18.6.1 Plotting
18.6.2 Tick counter
18.6.3 Unicode characters
18.6.4 Info tabs
18.6.5 Model speed
18.6.6 List performance
18.6.7 Extensions API
18.7 Changes for NetLogo 4.1
18.7.1 Combining set and of
18.8 Changes for NetLogo 4.0
18.8.1 Who numbering

18.8.2 Turtle creation: randomized vs. “ordered”
18.8.3 Adding strings and lists
18.8.4 The -at primitives
18.8.5 Links
18.8.6 New “of” syntax
18.8.7 Serial ask
18.8.8 Tick counter
18.8.9 View update modes
18.8.10 Speed slider
18.8.11 Numbers
18.8.12 Agentset building
18.8.13 RGB Colors
18.8.14 Tie
18.9 Changes for NetLogo 3.1
18.9.1 Agentsets
18.9.2 Wrapping
18.9.3 Random turtle coordinates
19. Extension Manager
19.1 Authoring and Sharing
19.2 Interface
20. Shapes
20.1 Getting started
20.1.1 Importing shapes
20.2 Creating and editing turtle shapes
20.2.1 Tools
20.2.2 Previews
20.2.3 Overlapping shapes
20.2.4 Undo
20.2.5 Colors
20.2.6 Other buttons
20.2.7 Shape design
20.2.8 Keeping a shape
20.3 Creating and editing link shapes
20.3.1 Changing link shape properties
20.4 Using shapes in a model
21. Behavior Space
21.1 What is BehaviorSpace?
21.1.1 Why BehaviorSpace?
21.2 How It Works

21.2.1 Managing experiment setups
21.2.2 Creating an experiment setup
21.2.3 Importing and exporting
21.2.4 Special primitives for BehaviorSpace experiments
21.2.5 Running an experiment
21.3 Advanced Usage
21.3.1 Running from the command line
21.3.2 Setting up experiments in XML
21.3.3 Adjusting JVM Parameters
21.3.4 Cluster Computing (HPCs)
21.3.5 Controlling API
22. System Dynamics Modeler
22.1 What is the NetLogo System Dynamics Modeler?
22.1.1 Basic Concepts
22.1.2 Sample Models
22.2 How it Works
22.2.1 Diagram Tab
22.2.2 Code Tab
22.2.3 The System Dynamics Modeler and NetLogo
22.3 Tutorial: Wolf-Sheep Predation
22.3.1 Step 1: Sheep Reproduction
22.3.2 Step 2: NetLogo Integration
22.3.3 Step 3: Wolf Predation
23. Color Picker
23.1 Introduction
23.2 Colors in NetLogo
23.3 Simple Tab
23.4 The Advanced Tab
23.4.1 Introduction
23.4.2 Input Format
23.4.3 Output Format
23.4.4 Output Formats for Different Contexts
23.4.5 Details about RGB Output
23.4.6 Details about the Alpha Bar
24. HubNet
24.1 Understanding HubNet
24.1.1 NetLogo
24.1.2 HubNet Architecture
24.2 Computer HubNet

24.2.1 Activities
24.2.2 Clients
24.2.3 Requirements
24.2.4 Starting an activity
24.2.5 HubNet Control Center
24.2.6 Troubleshooting
24.2.7 Known Limitations
24.3 Teacher workshops
24.4 HubNet Authoring Guide
24.5 Running HubNet in headless mode
24.6 Getting help
25. HubNet Authoring Guide
25.1 Coding HubNet activities
25.1.1 Setup
25.1.2 Receiving messages from clients
25.1.3 Sending messages to clients
25.1.4 Examples
25.2 How to make a client interface
25.3 View updates on the clients
25.4 Clicking in the view on clients
25.5 Customizing the client's view
25.6 Plot updates on the clients
26. Modeling Commons
26.1 Introduction
26.2 Modeling Commons Accounts
26.3 Uploading Models
26.3.1 Upload a New Model
26.3.2 Upload a Child of an Existing Model ("Forking")
26.4 Updating an Existing Model
27. Logging
27.1 Starting Logging
27.1.1 Command Line Switches
27.2 Using Logging
27.2.1 Events
27.2.2 JSON Output
27.2.3 Where Logs are Stored
27.2.4 Managing Log Files
28. Controlling the Model

29. Mathematica Link

29.1 What can I do with it?

29.2 Installation

29.3 Usage

29.4 Known Issues

29.5 Source code

29.6 Credits

30. NetLogo 3D

30.1 Introduction

30.2 Tutorial

30.3 Dictionary

30.3.1 Commands and Reporters

30.3.2 Built-In Variables

30.3.3 Primitives

30.3.4 at-points

30.3.5 distancexyz, distancexyz-nowrap

30.3.6 dz

30.3.7 face, facexyz

30.3.8 left

30.3.9 link-pitch

30.3.10 load-shapes-3d

30.3.11 max-pzcor, min-pzcor

30.3.12 neighbors, neighbors6

30.3.13 orbit-down, orbit-left, orbit-right, orbit-up

30.3.14 __oxcor, __oycor, __ozcor

30.3.15 patch

30.3.16 patch-at

30.3.17 patch-at-heading-pitch-and-distance

30.3.18 pitch

30.3.19 pzcor

30.3.20 random-pzcor

30.3.21 random-zcor

30.3.22 right

30.3.23 roll

30.3.24 roll-left

30.3.25 roll-right

30.3.26 setxyz

30.3.27 tilt-down, tilt-up

30.3.28 towards-pitch, towards-pitch-nowrap

30.3.29 towards-pitch-xyz, towards-pitch-xyz-nowrap

30.3.30 turtles-at, -at
30.3.31 world-depth
30.3.32 zcor
30.3.33 zoom
31. Cluster Computing (HPC)
31.1 Installing NetLogo on a Cluster
31.2 Submitting a Job
31.2.1 A simple example
31.2.2 A more general example
31.3 Advanced Usage
31.3.1 Dividing BehaviorSpace experiments into smaller sub-experiments
31.3.2 Submitting separate jobs for each experiment file
32. Extensions
32.1 Authoring and Sharing
32.2 Using Extensions
32.2.1 Where extensions are located
33. Extension Authoring
33.1 Technical Details
33.2 Sharing Extensions with the Extension Manager
34. Color Picker
34.1 Introduction
34.2 Colors in NetLogo
34.3 Simple Tab
34.4 The Advanced Tab
34.4.1 Introduction
34.4.2 Input Format
34.4.3 Output Format
34.4.4 Output Formats for Different Contexts
34.4.5 Details about RGB Output
34.4.6 Details about the Alpha Bar
35. NetLogo 7 Introduction
35.1 Motivation
35.2 Interface Tab
35.2.1 Overview of Visual Changes to the Interface Tab
35.2.2 Interface Elements (Widgets)
35.2.3 Widget Sizes
35.3 New File Format

35.4 Tools in the NetLogo 7 Interface
35.4.1 Overview: NetLogo Interface Tool Buttons and Dropdown menus
35.4.2 New Interface Tools
35.4.3 New Interface Menu
35.5 The Color Picker
35.6 Notifications
35.7 The Info Tab
35.8 Code Tab
35.8.1 Color Themes and Code Syntax Highlighting
35.8.2 Code Tab Toolbar
35.8.3 Code Tab Preferences
35.8.4 Included Files
35.9 Menu Items
35.9.1 Preferences Dialog
35.9.2 File → Resource Manager dialog
35.10 Tools Menu Changes
35.10.1 Turtle Shapes Editor and Link Shapes Editor
35.10.2 Help Menu
35.11 Collection of anonymous usage statistics
36. NetLogo Preferences
36.1 Overview
36.2 General Preferences
36.2.1 User Interface language
36.2.2 Reload the last used model on startup
36.2.3 Reload model on external changes
36.2.4 Enable remote commands
36.2.5 Bold widget text
36.2.6 Jump sliders to click location (instead of incrementing)
36.2.7 UI Scale
36.2.8 Send anonymous usage statistics
36.3 Code Preferences
36.3.1 Sort order of procedures menu
36.3.2 Always show “Included Files” menu
36.3.3 Auto-focus text or tab with errors when model is compiled
36.3.4 Open code tab in separate window on startup
36.3.5 Auto Indent
36.3.6 Show line numbers
36.3.7 Show auto-complete popup while typing
36.3.8 Code Font

36.4 Logging Preferences

36.4.1 Enable logging

36.4.2 Directory to store logs

36.4.3 Events to log:

36.5 Color Themes

36.5.1 Light

36.5.2 Dark

36.5.3 Classic

36.5.4 Automatic

37. Array

37.1 Using

37.1.1 When to Use

37.1.2 Example use of Array Extension

37.2 Primitives

37.2.1 All Primitives

37.2.2 array:from-list

37.2.3 array:item

37.2.4 array:set

37.2.5 array:length

37.2.6 array:to-list

38. GIS

38.1 Using

38.1.1 How to use

38.1.2 Known Issues

38.1.3 Credits

38.2 Primitives

38.2.1 Coordinate System Primitives

38.2.2 Dataset Primitives

38.2.3 VectorDataset Primitives

38.2.4 RasterDataset Primitives

38.2.5 Drawing Primitives

38.2.6 All Primitives

38.2.7 gis:set-transformation

38.2.8 gis:set-transformation-ds

38.2.9 gis:set-world-envelope

38.2.10 gis:set-world-envelope-ds

38.2.11 gis:world-envelope

38.2.12 gis:envelope-of

38.2.13 gis:envelope-union-of

38.2.14 gis:load-coordinate-system

38.2.15 gis:set-coordinate-system
38.2.16 gis:project-lat-lon
38.2.17 gis:project-lat-lon-from-ellipsoid
38.2.18 gis:load-dataset
38.2.19 gis:store-dataset
38.2.20 gis:type-of
38.2.21 gis:patch-dataset
38.2.22 gis:turtle-dataset
38.2.23 gis:link-dataset
38.2.24 gis:shape-type-of
38.2.25 gis:property-names
38.2.26 gis:feature-list-of
38.2.27 gis:vertex-lists-of
38.2.28 gis:centroid-of
38.2.29 gis:random-point-inside
38.2.30 gis:location-of
38.2.31 gis:set-property-value
38.2.32 gis:property-value
38.2.33 gis:find-features
38.2.34 gis:find-one-feature
38.2.35 gis:find-less-than
38.2.36 gis:find-greater-than
38.2.37 gis:find-range
38.2.38 gis:property-minimum
38.2.39 gis:property-maximum
38.2.40 gis:apply-coverage
38.2.41 gis:create-turtles-from-points
38.2.42 gis:create-turtles-from-points-manual
38.2.43 gis:create-turtles-inside-polygon
38.2.44 gis:create-turtles-inside-polygon-manual
38.2.45 gis:coverage-minimum-threshold
38.2.46 gis:set-coverage-minimum-threshold
38.2.47 gis:coverage-maximum-threshold
38.2.48 gis:set-coverage-maximum-threshold
38.2.49 gis:intersects?
38.2.50 gis:contains?
38.2.51 gis:contained-by?
38.2.52 gis:have-relationship?
38.2.53 gis:relationship-of
38.2.54 gis:intersecting
38.2.55 gis:width-of
38.2.56 gis:height-of

38.2.57 gis:raster-value
38.2.58 gis:set-raster-value
38.2.59 gis:minimum-of
38.2.60 gis:maximum-of
38.2.61 gis:sampling-method-of
38.2.62 gis:set-sampling-method
38.2.63 gis:raster-sample
38.2.64 gis:raster-world-envelope
38.2.65 gis:create-raster
38.2.66 gis:resample
38.2.67 gis:convolve
38.2.68 gis:apply-raster
38.2.69 gis:drawing-color
38.2.70 gis:set-drawing-color
38.2.71 gis:draw
38.2.72 gis:fill
38.2.73 gis:paint
38.2.74 gis:import-wms-drawing
39. Networks
39.1 Usage
39.1.1 Special agentsets vs normal agentsets
39.2 A note regarding floating point calculations
39.3 Performance
39.4 Primitives
39.4.1 Centrality Measures
39.4.2 Clusterer/Community Detection
39.4.3 Clustering Measures
39.4.4 Context Management
39.4.5 Generators
39.4.6 Import and Export
39.4.7 Path and Distance
39.4.8 All Primitives
39.4.9 nw:set-context
39.4.10 nw:get-context
39.4.11 nw:with-context
39.4.12 nw:turtles-in-radius
39.4.13 nw:turtles-in-reverse-radius
39.4.14 nw:distance-to
39.4.15 nw:weighted-distance-to
39.4.16 nw:path-to
39.4.17 nw:turtles-on-path-to

39.4.18 nw:weighted-path-to
39.4.19 nw:turtles-on-weighted-path-to
39.4.20 nw:mean-path-length
39.4.21 nw:mean-weighted-path-length
39.4.22 nw:betweenness-centrality
39.4.23 nw:eigenvector-centrality
39.4.24 nw:page-rank
39.4.25 nw:closeness-centrality
39.4.26 nw:weighted-closeness-centrality
39.4.27 nw:clustering-coefficient
39.4.28 nw:modularity
39.4.29 nw:bicomponent-clusters
39.4.30 nw:weak-component-clusters
39.4.31 nw:louvain-communities
39.4.32 nw:maximal-cliques
39.4.33 nw:biggest-maximal-cliques
39.4.34 nw:generate-preferential-attachment
39.4.35 nw:generate-random
39.4.36 nw:generate-watts-strogatz
39.4.37 nw:generate-small-world
39.4.38 nw:generate-lattice-2d
39.4.39 nw:generate-ring
39.4.40 nw:generate-star
39.4.41 nw:generate-wheel
39.4.42 nw:save-matrix
39.4.43 nw:load-matrix
39.4.44 nw:save-graphml
39.4.45 nw:load-graphml
39.4.46 nw:load
39.4.47 nw:load-from-string
39.4.48 nw:save
39.4.49 nw:save-to-string

40. CSV

40.1 Common use cases and examples

40.1.1 Read a file all at once
40.1.2 Read a file one line at a time
40.1.3 Read a file one line per tick
40.1.4 Write a file

40.2 Primitives

40.2.1 Parsing CSV input to NetLogo data
40.2.2 Formatting NetLogo data as CSV

40.2.3 All Primitives
40.2.4 csv:from-row
40.2.5 csv:from-string
40.2.6 csv:from-file
40.2.7 csv:to-row
40.2.8 csv:to-string
40.2.9 csv:to-file
41. Profiler
41.1 Using the Profiler Extension
41.1.1 How to use
41.1.2 Example
41.2 Primitives
41.2.1 All Primitives
41.2.2 profiler:calls
41.2.3 profiler:exclusive-time
41.2.4 profiler:inclusive-time
41.2.5 profiler:start
41.2.6 profiler:stop
41.2.7 profiler:reset
41.2.8 profiler:report
41.2.9 profiler:data
42. Python
42.1 Using
42.1.1 Error handling
42.2 Configuring
42.3 Primitives
42.3.1 All Primitives
42.3.2 py:setup
42.3.3 py:python
42.3.4 py:python2
42.3.5 py:python3
42.3.6 py:run
42.3.7 py:runresult
42.3.8 py:set
43. Time
43.1 Quickstart
43.2 What is it?
43.3 Installation
43.4 Examples

43.5 Data Types
43.6 Behavior
43.7 Format
43.7.1 Date Format
43.7.2 Supported Format Characters
43.7.3 Date-time Bounds
43.7.4 User Defined Formatting
43.8 Primitives
43.8.1 Date/Time Utilities
43.8.2 Time Series Tool
43.9 back to top
43.9.1 Discrete Event Scheduler
43.10 Building with SBT
43.11 Authors
43.12 Feedback? Bugs? Feature Requests?
43.13 Credits
43.14 Terms of Use
43.15 Primitives
43.15.1 All Primitives
44. Matrix
44.1 Using
44.1.1 When to Use
44.1.2 How to Use
44.1.3 Example
44.2 Primitives
44.2.1 Advanced features
44.2.2 Math operations
44.2.3 Matrix creation and conversion to/from lists
44.2.4 Matrix data retrieval and manipulation
44.2.5 All Primitives
44.2.6 matrix:make-constant
44.2.7 matrix:make-identity
44.2.8 matrix:from-row-list
44.2.9 matrix:from-column-list
44.2.10 matrix:to-row-list
44.2.11 matrix:to-column-list
44.2.12 matrix:copy
44.2.13 matrix:pretty-print-text
44.2.14 matrix:get

44.2.15 matrix:get-row
44.2.16 matrix:get-column
44.2.17 matrix:set
44.2.18 matrix:set-row
44.2.19 matrix:set-column
44.2.20 matrix:swap-rows
44.2.21 matrix:swap-columns
44.2.22 matrix:set-and-report
44.2.23 matrix:dimensions
44.2.24 matrix:submatrix
44.2.25 matrix:map
44.2.26 matrix:times-scalar
44.2.27 matrix:times
44.2.28 matrix:*
44.2.29 matrix:times-element-wise
44.2.30 matrix:plus-scalar
44.2.31 matrix:plus
44.2.32 matrix:+
44.2.33 matrix:minus
44.2.34 matrix:-
44.2.35 matrix:inverse
44.2.36 matrix:transpose
44.2.37 matrix:real-eigenvalues
44.2.38 matrix:imaginary-eigenvalues
44.2.39 matrix:eigenvectors
44.2.40 matrix:det
44.2.41 matrix:rank
44.2.42 matrix:trace
44.2.43 matrix:solve
44.2.44 matrix:forecast-linear-growth
44.2.45 matrix:forecast-compound-growth
44.2.46 matrix:forecast-continuous-growth
44.2.47 matrix:regress

45. GoGo

45.1 NetLogoLab and the GoGo Board Extension for sensors and robotics

45.2 What is NetLogoLab?

45.2.1 The GoGo Board NetLogo extension
45.2.2 GoGo Board: a low-cost robotics and data-logging board
45.2.3 Sensor and actuator toolkits
45.2.4 NetLogo models

45.3 How to get a GoGo Board?

45.4 Installing and testing the GoGo Extension
45.4.1 Windows
45.4.2 Mac OS X
45.4.3 Linux
45.5 Changes
45.6 Primitives
45.6.1 General
45.6.2 Other Outputs
45.6.3 Outputs and Servos
45.6.4 Sensors
45.6.5 Utilities
45.6.6 All Primitives
45.6.7 gogo:primitives
45.6.8 gogo:howmany-gogos
45.6.9 gogo:init
45.6.10 gogo:talk-to-output-ports
45.6.11 gogo:talk-to-servo-ports
45.6.12 gogo:set-output-port-power
45.6.13 gogo:output-port-on
45.6.14 gogo:output-port-off
45.6.15 gogo:output-port-clockwise
45.6.16 gogo:output-port-counterclockwise
45.6.17 gogo:set-servo
45.6.18 gogo:led
45.6.19 gogo:beep
45.6.20 gogo:read-sensors
45.6.21 gogo:read-sensor
45.6.22 gogo:read-all
45.6.23 gogo:send-bytes
45.7 Examples of NetLogoLab models
45.7.1 Controlling a car
45.7.2 A simple sensing project
47. Table
47.1 Using
47.1.1 When to Use
47.1.2 Example
47.1.3 Manipulating Tables
47.1.4 Key Restrictions
47.2 Primitives
47.2.1 All Primitives
47.2.2 table:clear

47.2.3 table:counts
47.2.4 table:group-agents
47.2.5 table:group-items
47.2.6 table:from-list
47.2.7 table:from-json
47.2.8 table:from-json-file
47.2.9 table:get
47.2.10 table:get-or-default
47.2.11 table:has-key?
47.2.12 table:keys
47.2.13 table:length
47.2.14 table:make
47.2.15 table:put
47.2.16 table:remove
47.2.17 table:to-list
47.2.18 table:to-json
47.2.19 table:values
48. Vid
48.1 Concepts
48.1.1 Video Source
48.1.2 Source Lifecycle
48.1.3 Video Recorder
48.1.4 Known Issues
48.2 Primitives
48.2.1 All Primitives
48.2.2 vid:camera-names
48.2.3 vid:camera-open
48.2.4 vid:camera-select
48.2.5 vid:movie-select
48.2.6 vid:movie-open
48.2.7 vid:movie-open-remote
48.2.8 vid:close
48.2.9 vid:start
48.2.10 vid:stop
48.2.11 vid:status
48.2.12 vid:capture-image
48.2.13 vid:set-time
48.2.14 vid:show-player
48.2.15 vid:hide-player
48.2.16 vid:record-view
48.2.17 vid:record-interface

48.2.18 vid:record-source
48.2.19 vid:recorder-status
48.2.20 vid:reset-recorder
48.2.21 vid:start-recorder
48.2.22 vid:save-recording

49. Rnd

49.1 Usage
49.2 A note about performance
49.3 Primitives
49.3.1 AgentSet Primitives
49.3.2 List Primitives
49.3.3 All Primitives
49.3.4 rnd:weighted-one-of
49.3.5 rnd:weighted-n-of
49.3.6 rnd:weighted-n-of-with-repeats
49.3.7 rnd:weighted-one-of-list
49.3.8 rnd:weighted-n-of-list
49.3.9 rnd:weighted-n-of-list-with-repeats

50. LevelSpace

50.1 LevelSpace fundamentals
50.1.1 Headless and Interactive Models
50.1.2 Keeping Track of Models
50.1.3 A general use case: Asking and Reporting
50.1.4 A general use case: Inter-Model Interactions
50.1.5 A general Usecase: Tidying up “Dead” Child Models
50.2 Primitives
50.2.1 Commanding and Reporting
50.2.2 Logic and Control
50.2.3 Opening and Closing Models
50.2.4 All Primitives
50.2.5 ls:create-models
50.2.6 ls:create-interactive-models
50.2.7 ls:close
50.2.8 ls:reset
50.2.9 ls:ask
50.2.10 ls:of
50.2.11 ls:report
50.2.12 ls:with
50.2.13 ls:let
50.2.14 ls:assign

50.2.15 ls:models
50.2.16 ls:show
50.2.17 ls:show-all
50.2.18 ls:hide
50.2.19 ls:hide-all
50.2.20 ls:path-of
50.2.21 ls:name-of
50.2.22 ls:model-exists?
50.2.23 ls:random-seed

51. Arduino

51.1 Using

51.1.1 Notes
51.1.2 Compatibility
51.1.3 Questions

51.2 Primitives

51.2.1 All Primitives
51.2.2 arduino:primitives
51.2.3 arduino:ports
51.2.4 arduino:open
51.2.5 arduino:close
51.2.6 arduino:get
51.2.7 arduino:write-string
51.2.8 arduino:write-int
51.2.9 arduino:write-byte
51.2.10 arduino:is-open?
51.2.11 arduino:debug-to-arduino
51.2.12 arduino:debug-from-arduino

52. Palette

52.1 Using the Palette Extension

52.2 Getting Started

52.3 Background

52.3.1 Review of color representation in NetLogo
52.3.2 More control over the color
52.3.3 Varying an Agent's Transparency or Color
52.3.4 How do I choose a color scheme?
52.3.5 Should I use a continuous color gradient or just a discrete color set?
52.3.6 Example Models
52.3.7 Further Reading

52.4 Primitives

52.4.1 All Primitives

52.4.2 palette:alpha-of
52.4.3 palette:with-alpha
52.4.4 palette:alpha
52.4.5 palette:set-alpha
52.4.6 palette:transparency-of
52.4.7 palette:with-transparency
52.4.8 palette:transparency
52.4.9 palette:set-transparency
52.4.10 palette:hue-of
52.4.11 palette:with-hue
52.4.12 palette:hue
52.4.13 palette:set-hue
52.4.14 palette:saturation-of
52.4.15 palette:with-saturation
52.4.16 palette:saturation
52.4.17 palette:set-saturation
52.4.18 palette:brightness-of
52.4.19 palette:with-brightness
52.4.20 palette:brightness
52.4.21 palette:set-brightness
52.4.22 palette:R-of
52.4.23 palette:with-R
52.4.24 palette:R
52.4.25 palette:set-R
52.4.26 palette:G-of
52.4.27 palette:with-G
52.4.28 palette:G
52.4.29 palette:set-G
52.4.30 palette:B-of
52.4.31 palette:with-B
52.4.32 palette:B
52.4.33 palette:set-B
52.4.34 palette:scale-gradient
52.4.35 palette:scale-gradient-hsb
52.4.36 palette:scheme-colors
52.4.37 palette:scale-scheme
52.4.38 palette:scheme-dialog
52.5 References
53. Bitmap
53.1 Using
53.1.1 What does the Bitmap Extension do?

53.1.2 Getting started
53.2 Primitives
53.2.1 All Primitives
53.2.2 bitmap:average-color
53.2.3 bitmap:channel
53.2.4 bitmap:copy-to-drawing
53.2.5 bitmap:copy-to-pcolors
53.2.6 bitmap:difference-rgb
53.2.7 bitmap:export
53.2.8 bitmap:from-base64
53.2.9 bitmap:to-base64
53.2.10 bitmap:from-view
53.2.11 bitmap:to-grayscale
53.2.12 bitmap:height
53.2.13 bitmap:import
53.2.14 bitmap:scaled
53.2.15 bitmap:width
54. Sound
54.1 Using
54.1.1 How to Use
54.1.2 MIDI support
54.2 Primitives
54.2.1 All Primitives
54.2.2 sound:drums
54.2.3 sound:instruments
54.2.4 sound:play-drum
54.2.5 sound:play-note
54.2.6 sound:play-note-later
54.3 Drum Names
54.4 Instrument Names
55. Resource
55.1 Using
55.1.1 Purpose
55.1.2 Getting Started
55.2 Primitives
55.2.1 All Primitives
55.2.2 resource:get
55.2.3 resource:list
56. Simple R

56.1 Using
56.1.1 Error handling
56.2 Primitives
56.2.1 All Primitives
56.2.2 sr:setup
56.2.3 sr:run
56.2.4 sr:runresult
56.2.5 sr:set
56.2.6 sr:set-agent
56.2.7 sr:set-agent-data-frame
56.2.8 sr:set-data-frame
56.2.9 sr:set-list
56.2.10 sr:set-named-list
56.2.11 sr:set-plot-device
56.2.12 sr:r-home
56.2.13 sr:show-console
56.3 Transitioning from the old R extension
56.3.1 Using the Simple R code converter
56.3.2 Handling a named list
57. Frequently Asked Questions
57.1 General
57.1.1 Why is it called NetLogo?
57.1.2 How do I cite NetLogo or HubNet in a publication?
57.1.3 How do I cite a model from the Models Library in a publication?
57.1.4 Where and when was NetLogo created?
57.1.5 What programming language was NetLogo written in?
57.1.6 What's the relationship between StarLogo and NetLogo?
57.1.7 Under what license is NetLogo released? Is the source code available?
57.1.8 Do you offer any workshops or other training opportunities for NetLogo?
57.1.9 Are there any NetLogo textbooks?
57.1.10 Is NetLogo available in other languages besides English?
57.1.11 Is NetLogo compiled or interpreted?
57.1.12 Has anyone built a model of ?
57.1.13 Are NetLogo models runs scientifically reproducible?
57.1.14 Will NetLogo and NetLogo 3D remain separate?
57.1.15 Can I run NetLogo on my phone or tablet?
57.2 Downloading
57.2.1 Can I have multiple versions of NetLogo installed at the same time?
57.2.2 I'm on a UNIX system and I can't untar the download. Why?
57.2.3 How do I install NetLogo unattended?
57.3 Running

57.3.1 Can I run NetLogo from a CD, a network drive, or a USB drive?
57.3.2 Why is NetLogo so much slower when I unplug my Windows laptop?
57.3.3 Why does NetLogo bundle Java?
57.3.4 How come NetLogo won't start up on my Linux machine?
57.3.5 When I try to install NetLogo on Windows, I see "Windows protected your PC"
57.3.6 When I try to start NetLogo on Windows I get an error "The JVM could not be started". Help!
57.3.7 NetLogo won't start on Mac OS Sierra (or later)
57.3.8 NetLogo won't start on Windows or crashes suddenly on Mac OS Sierra
57.3.9 How can I speed up the launching of NetLogo on a remote server that is behind a proxy?
57.3.10 Can I run NetLogo from the command line, without the GUI?
57.3.11 Does NetLogo take advantage of multiple processors?
57.3.12 Can I distribute NetLogo model runs across a cluster or grid of computers?
57.3.13 Is there any way to recover lost work if NetLogo crashes or freezes?
57.3.14 Why is HubNet Discovery Not Working?

57.4 Usage

57.4.1 When I move the speed slider all the way to the right, why does my model seem to stop?
57.4.2 Can I use the mouse to "paint" in the view?
57.4.3 How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?
57.4.4 Where are the NetLogo Configuration Files?
57.4.5 Can I use GIS data in NetLogo?
57.4.6 My model runs slowly. How can I speed it up?
57.4.7 Can I have more than one model open at a time?
57.4.8 Can I change the choices in a chooser on the fly?
57.4.9 Can I divide the code for my model up into several files?

57.5 Programming

57.5.1 How does the NetLogo language differ from other Logos?
57.5.2 How come my model from an earlier NetLogo doesn't work right?
57.5.3 How do I take the negative of a number?
57.5.4 My turtle moved forward 1, but it's still on the same patch. Why?
57.5.5 How do I keep my turtles on patch centers?
57.5.6 patch-ahead 1 is reporting the same patch my turtle is already standing on. Why?
57.5.7 How do I give my turtles "vision"?
57.5.8 Can agents sense what's in the drawing layer?
57.5.9 I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?
57.5.10 The documentation says that random-float 1 might return 0 but will never return 1. What if I want 1 to be included?
57.5.11 Why is the number value in my monitor widget changing even though nothing is happening in my model?
57.5.12 How can I keep two turtles from occupying the same patch?
57.5.13 How can I find out if a turtle is dead?
57.5.14 Does NetLogo have arrays?
57.5.15 Does NetLogo have hash tables or associative arrays?
57.5.16 How can I use different patch "neighborhoods" (circular, Von Neumann, Moore, etc.)?
57.5.17 How can I convert an agentset to a list of agents, or vice versa?

57.5.18 How do I stop foreach?

57.5.19 I'm trying to make a list. Why do I keep getting the error "Expected a literal value"?

57.6 BehaviorSpace

57.6.1 Why are the rows in my BehaviorSpace table results out of order?

57.6.2 How do I measure runs every n ticks?

57.6.3 I'm varying a global variable I declared in the Code tab, but it doesn't work. Why?

57.7 NetLogo 3D

57.7.1 Does NetLogo work with my stereoscopic device?

57.8 Extensions

57.8.1 I'm writing an extension. Why does the compiler say it can't find org.nlogo.api?

58. NetLogo Dictionary

58.1 Categories

58.1.1 Turtle-related

58.1.2 Patch-related

58.1.3 Link-related

58.1.4 Agentset

58.1.5 Color

58.1.6 Control Flow and Logic

58.1.7 Anonymous Procedures

58.1.8 World

58.1.9 Perspective

58.1.10 HubNet

58.1.11 Input/Output

58.1.12 File

58.1.13 List

58.1.14 String

58.1.15 Mathematical

58.1.16 Plotting

58.1.17 BehaviorSpace

58.1.18 System

58.2 Built-In Variables

58.2.1 Turtle Variable

58.2.2 Link Variable

58.2.3 Patch Variable

58.3 Keywords

58.4 Constants

58.4.1 Mathematical Constants

58.4.2 Boolean Constants

58.4.3 Color Constants

58.5 Primitives and Commands

58.5.1 abs
58.5.2 acos
58.5.3 all?
58.5.4 and
58.5.5 any?
58.5.6 approximate-hsb
58.5.7 approximate-rgb
58.5.8 Arithmetic Operators, +, *, -, /, ^, <, >, =, !=, <=, >=
58.5.9 asin
58.5.10 ask
58.5.11 ask-concurrent
58.5.12 at-points
58.5.13 atan
58.5.14 autoplot?, autoplotx?, autoploty?
58.5.15 auto-plot-off, auto-plot-x-off, auto-plot-y-off, auto-plot-on, auto-plot-x-on, auto-plot-y-on
58.5.16 back, bk
58.5.17 base-colors
58.5.18 beep
58.5.19 behaviorspace-experiment-name
58.5.20 behaviorspace-run-number
58.5.21 both-ends
58.5.22 breed
58.5.23 breed
58.5.24 but-first, butfirst, bf, but-last, butlast, bl
58.5.25 can-move?
58.5.26 carefully
58.5.27 ceiling
58.5.28 clear-all, ca
58.5.29 clear-all-plots
58.5.30 clear-drawing, cd
58.5.31 clear-globals
58.5.32 clear-links
58.5.33 clear-output
58.5.34 clear-patches, cp
58.5.35 clear-plot
58.5.36 clear-ticks
58.5.37 clear-turtles, ct
58.5.38 color
58.5.39 cos
58.5.40 count
58.5.41 create-ordered-turtles, cro

58.5.42 create--to, create--to, create--from, create--from, create--with, create--with, create-link-to, create-links-to, create-link-from, create-links-from, create-link-with, create-links-with
58.5.43 create-turtles, crt
58.5.44 create-temporary-plot-pen
58.5.45 date-and-time
58.5.46 die
58.5.47 diffuse
58.5.48 diffuse4
58.5.49 directed-link-breed
58.5.50 display
58.5.51 distance
58.5.52 distancexy
58.5.53 downhill, downhill4
58.5.54 dx, dy
58.5.55 empty?
58.5.56 end
58.5.57 end1
58.5.58 end2
58.5.59 error
58.5.60 error-message
58.5.61 every
58.5.62 exp
58.5.63 export
58.5.64 export-view, export-interface, export-output, export-plot, export-all-plots, export-world
58.5.65 extensions
58.5.66 extract-hsb
58.5.67 extract-rgb
58.5.68 face
58.5.69 facexy
58.5.70 file-at-end?
58.5.71 file-close
58.5.72 file-close-all
58.5.73 file-delete
58.5.74 file-exists?
58.5.75 file-flush
58.5.76 file-open
58.5.77 file-print
58.5.78 file-read
58.5.79 file-read-characters
58.5.80 file-read-line
58.5.81 file-show
58.5.82 file-type

58.5.83 file-write
58.5.84 filter
58.5.85 first
58.5.86 floor
58.5.87 follow
58.5.88 follow-me
58.5.89 foreach
58.5.90 forward, fd
58.5.91 fput
58.5.92 globals
58.5.93 hatch
58.5.94 heading
58.5.95 hidden?
58.5.96 hide-link
58.5.97 hide-turtle, ht
58.5.98 histogram
58.5.99 home
58.5.100 home-directory
58.5.101 hsb
58.5.102 hubnet-broadcast
58.5.103 hubnet-broadcast-clear-output
58.5.104 hubnet-broadcast-message
58.5.105 hubnet-clear-override, hubnet-clear-overrides
58.5.106 hubnet-clients-list
58.5.107 hubnet-enter-message?
58.5.108 hubnet-exit-message?
58.5.109 hubnet-fetch-message
58.5.110 hubnet-kick-client
58.5.111 hubnet-kick-all-clients
58.5.112 hubnet-message
58.5.113 hubnet-message-source
58.5.114 hubnet-message-tag
58.5.115 hubnet-message-waiting?
58.5.116 hubnet-reset
58.5.117 hubnet-reset-perspective
58.5.118 hubnet-send
58.5.119 hubnet-send-clear-output
58.5.120 hubnet-send-follow
58.5.121 hubnet-send-message
58.5.122 hubnet-send-override
58.5.123 hubnet-send-watch
58.5.124 if

58.5.125 ifelse
58.5.126 ifelse-value
58.5.127 import
58.5.128 import-drawing
58.5.129 import-pcolors
58.5.130 import-pcolors-rgb
58.5.131 import-world
58.5.132 in-cone
58.5.133 in--neighbor?, in-link-neighbor?
58.5.134 in--neighbors, in-link-neighbors
58.5.135 in--from, in-link-from
58.5.136 __includes
58.5.137 in-radius
58.5.138 insert-item
58.5.139 inspect
58.5.140 int
58.5.141 is-agent?, is-agentset?, is-anonymous-command?, is-anonymous-reporter?, is-boolean?, is-directed-link?, is-link?, is-link-set?, is-list?, is-number?, is-patch?, is-patch-set?, is-string?, is-turtle?, is-turtle-set?, is-undirected-link?
58.5.142 item
58.5.143 jump
58.5.144 label
58.5.145 label-color
58.5.146 last
58.5.147 layout-circle
58.5.148 layout-radial
58.5.149 layout-spring
58.5.150 layout-tutte
58.5.151 left, lt
58.5.152 length
58.5.153 let
58.5.154 link
58.5.155 link-heading
58.5.156 link-length
58.5.157 link-set
58.5.158 link-shapes
58.5.159 links
58.5.160 links-own
58.5.161 list
58.5.162 ln
58.5.163 log
58.5.164 loop
58.5.165 lput

58.5.166 map
58.5.167 max
58.5.168 max-n-of
58.5.169 max-one-of
58.5.170 max-pxcor, max-pycor
58.5.171 mean
58.5.172 median
58.5.173 member?
58.5.174 min
58.5.175 min-n-of
58.5.176 min-one-of
58.5.177 min-pxcor, min-pycor
58.5.178 mod
58.5.179 modes
58.5.180 mouse-down?
58.5.181 mouse-inside?
58.5.182 mouse-xcor, mouse-ycor
58.5.183 move-to
58.5.184 my-, my-links
58.5.185 my-in-, my-in-links
58.5.186 my-out-, my-out-links
58.5.187 myself
58.5.188 n-of
58.5.189 n-values
58.5.190 neighbors, neighbors4
58.5.191 -neighbors, link-neighbors
58.5.192 -neighbor?, link-neighbor?
58.5.193 netlogo-version
58.5.194 netlogo-web?
58.5.195 new-seed
58.5.196 no-display
58.5.197 nobody
58.5.198 no-links
58.5.199 no-patches
58.5.200 not
58.5.201 no-turtles
58.5.202 of
58.5.203 one-of
58.5.204 or
58.5.205 other
58.5.206 other-end
58.5.207 out--neighbor?, out-link-neighbor?

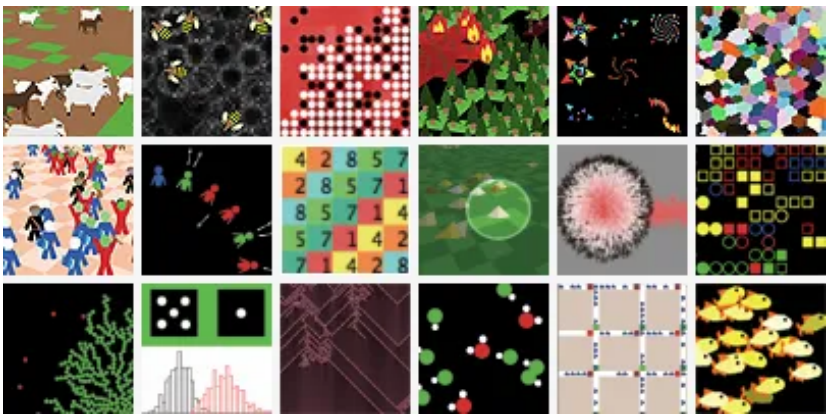
58.5.208 out--neighbors, out-link-neighbors
58.5.209 out--to, out-link-to
58.5.210 output-print, output-show, output-type, output-write
58.5.211 patch
58.5.212 patch-ahead
58.5.213 patch-at
58.5.214 patch-at-heading-and-distance
58.5.215 patch-here
58.5.216 patch-left-and-ahead, patch-right-and-ahead
58.5.217 patch-set
58.5.218 patch-size
58.5.219 patches
58.5.220 patches-own
58.5.221 pcolor
58.5.222 pen-down, pd, pen-erase, pe, pen-up, pu
58.5.223 pen-mode
58.5.224 pen-size
58.5.225 plabel
58.5.226 plabel-color
58.5.227 plot
58.5.228 plot-name
58.5.229 plot-pen-exists?
58.5.230 plot-pen-down, plot-pen-up
58.5.231 plot-pen-reset
58.5.232 plotxy
58.5.233 plot-x-min, plot-x-max, plot-y-min, plot-y-max
58.5.234 position
58.5.235 precision
58.5.236 print
58.5.237 pxcor, pycor
58.5.238 random
58.5.239 random-float
58.5.240 random-exponential, random-gamma, random-normal, random-poisson
58.5.241 random-pxcor, random-pycor
58.5.242 random-seed
58.5.243 random-xcor, random-ycor
58.5.244 range
58.5.245 read-from-string
58.5.246 reduce
58.5.247 remainder
58.5.248 remove
58.5.249 remove-duplicates

58.5.250 remove-item
58.5.251 repeat
58.5.252 replace-item
58.5.253 report
58.5.254 reset-perspective, rp
58.5.255 reset-ticks
58.5.256 reset-timer
58.5.257 resize-world
58.5.258 reverse
58.5.259 rgb
58.5.260 ride
58.5.261 ride-me
58.5.262 right, rt
58.5.263 round
58.5.264 run, runresult
58.5.265 scale-color
58.5.266 self
58.5.267 ; (semicolon)
58.5.268 sentence, se
58.5.269 set
58.5.270 set-current-directory
58.5.271 set-current-plot
58.5.272 set-current-plot-pen
58.5.273 set-default-shape
58.5.274 set-histogram-num-bars
58.5.275 __set-line-thickness
58.5.276 set-patch-size
58.5.277 set-plot-background-color
58.5.278 set-plot-pen-color
58.5.279 set-plot-pen-interval
58.5.280 set-plot-pen-mode
58.5.281 setup-plots
58.5.282 set-plot-x-range, set-plot-y-range
58.5.283 set-topology
58.5.284 setxy
58.5.285 shade-of?
58.5.286 shape
58.5.287 shapes
58.5.288 show
58.5.289 show-turtle, st
58.5.290 show-link
58.5.291 shuffle

58.5.292 sin
58.5.293 size
58.5.294 sort
58.5.295 sort-by
58.5.296 sort-on
58.5.297 sprout
58.5.298 sqrt
58.5.299 stamp
58.5.300 stamp-erase
58.5.301 standard-deviation
58.5.302 startup
58.5.303 stop
58.5.304 stop-inspecting
58.5.305 stop-inspecting-dead-agents
58.5.306 subject
58.5.307 sublist, substring
58.5.308 subtract-headings
58.5.309 sum
58.5.310 tan
58.5.311 thickness
58.5.312 tick
58.5.313 tick-advance
58.5.314 ticks
58.5.315 tie
58.5.316 tie-mode
58.5.317 timer
58.5.318 to
58.5.319 to-report
58.5.320 towards
58.5.321 towardsxy
58.5.322 turtle
58.5.323 turtle-set
58.5.324 turtles
58.5.325 turtles-at
58.5.326 turtles-here
58.5.327 turtles-on
58.5.328 turtles-own
58.5.329 type
58.5.330 undirected-link-breed
58.5.331 untie
58.5.332 up-to-n-of
58.5.333 update-plots

58.5.334 uphill, uphill4
58.5.335 user-directory
58.5.336 user-file
58.5.337 user-new-file
58.5.338 user-input
58.5.339 user-message
58.5.340 user-one-of
58.5.341 user-yes-or-no?
58.5.342 variance
58.5.343 wait
58.5.344 watch
58.5.345 watch-me
58.5.346 while
58.5.347 who
58.5.348 who-are-not
58.5.349 with
58.5.350 -with, link-with
58.5.351 with-max
58.5.352 with-min
58.5.353 with-local-randomness
58.5.354 without-interruption
58.5.355 word
58.5.356 world-width, world-height
58.5.357 wrap-color
58.5.358 write
58.5.359 xcor
58.5.360 xor
58.5.361 ycor
58.5.362 →

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 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 anonymous procedures, 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

What is Agent-Based Modeling?

Agent-based modeling is a powerful technique for understanding complex systems. In an agent-based model, individual entities called “agents” operate according to simple rules. These agents interact with each other and their environment, leading to the emergence of complex behaviors and patterns at the system level. Agent-based modeling is widely used in fields such as social sciences, biology, economics, and ecology to study phenomena like population dynamics, market behavior, and social interactions.

Why is NetLogo Useful for Agent-Based Modeling?

NetLogo is specifically designed for building and simulating agent-based models. Its user-friendly interface and approachable programming language make it accessible to both beginners and experienced modelers. NetLogo provides a rich set of built-in features that facilitate the creation of complex models, including support for mobile agents, networked interactions, and data collection. Additionally, NetLogo’s extensive documentation and active user community provide valuable resources for learning and collaboration in the field of agent-based modeling.

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. 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

The CCL gratefully acknowledges two decades of support for our NetLogo work. The original support came from the National Science Foundation — grant numbers REC-9814682 and REC-0126227. Further support has come from REC-0003285, REC-0115699, DRL-0196044, CCF-ITR-0326542, DRL-REC/ROLE-0440113, SBE-0624318, EEC-0648316, IIS-0713619, DRL-RED-9552950, DRL-REC-9632612, and DRL-DRK12-1020101, IIS-1441552, CNS-1441016, CNS-1441041, CNS-1138461, IIS-1438813, IIS-1147621, DRL-REC-1343873, IIS-1438813, IIS-1441552, CNS-1441041, IIS-1546120, DRL-1546122, DRL-1614745 and DRL-1640201. Additional support came from the Spencer Foundation, Texas Instruments, the Brady Fund, the Murphy fund, and the Northwestern Institute on Complex Systems.

For development of the View 2.5D NetLogo Extension, we acknowledge financial support from Joshua M. Epstein's NIH Director's Pioneer Award (DP1).

NetLogo license

Copyright 1999-2025 by Uri Wilensky.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Commercial licenses

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

NetLogo User Manual license

Copyright 1999-2025 by Uri Wilensky.

The NetLogo User Manual by Uri Wilensky is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

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 library. The license for Scala is as follows:

Copyright (c) 2002-2020 EPFL
Copyright (c) 2011-2020 Lightbend, Inc.

Scala includes software developed at LAMP/EPFL (https://lamp.epfl.ch/) and Lightbend, Inc. (https://www.lightbend.com/).

Licensed under the Apache License, Version 2.0 (the “License”). 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.

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:

- 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 copyright owners, their employers, 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 OWNERS 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.

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:

Copyright 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

Config

NetLogo uses the Typesafe “Config” library. Copyright (C) 2011-2012 Typesafe Inc. http://typesafe.com The Config library is licensed under the Apache 2.0 License. You may obtain a copy of the license at http://www.apache.org/licenses/LICENSE-2.0.

Apache Commons Codec (TM)

The NetLogo compiler uses a digest method from the Apache Commons Codec (TM) library. Apache Commons Codec (TM) is copyright and trademark 2002-2014 the Apache Software Foundation. It is licensed under the Apache 2.0 License. You may obtain a copy of the license at http://www.apache.org/licenses/LICENSE-2.0.

Flexmark

NetLogo uses the Flexmark library (and extensions) for the info tab. The copyright and license are as follows:

Copyright (c) 2015-2016, Atlassian Pty Ltd All rights reserved. Copyright (c) 2016, Vladimir Schneider, 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. 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 HOLDER 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.

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.

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 http://jogamp.org/. Both libraries are distributed under the BSD license:

Copyright 2010 JogAmp Community. 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.

THIS SOFTWARE IS PROVIDED BY JogAmp Community "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 JogAmp Community 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.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of JogAmp Community.

You can address the JogAmp Community via: Web http://jogamp.org/ Forum/Mailinglist http://forum.jogamp.org Chatrooms IRC irc.freenode.net #jogamp Jabber conference.jabber.org room: jogamp (deprecated!) Repository http://jogamp.org/git/ Email mediastream _at_ jogamp _dot_ org

Matrix3D

For 3D matrix operations, NetLogo uses the Matrix3D class. It is distributed under the following license:

Copyright (c) 1994-1996 Sun Microsystems, Inc. All Rights Reserved.
Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, modify and redistribute this software in source and binary code form, provided that i) this copyright notice and license appear on all copies of the software; and ii) Licensee does not utilize the software in a manner which is disparaging to Sun.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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) 2000-2011 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.

PicoContainer

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

Copyright (c) 2004-2011, 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 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.

Parboiled

For reading models, NetLogo uses the Parboiled library. The copyright and license for Parboiled are as follows:

This software is licensed under the Apache 2 license, quoted below. Copyright © 2009-2013 Mathias Doenitz http://parboiled2.org Copyright © 2013 Alexander Myltsev 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.

RSyntaxTextArea

The NetLogo editor uses the RSyntaxTextArea library. The copyright and license are as follows:

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 author 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 ©RIGHT HOLDER& 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.

JCodec

The NetLogo `vid` extension makes use of the JCodec library. The copyright and license for JCodec are as follows:

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. 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.

Java-Objective-C Bridge

NetLogo on Mac OS X makes use of the Java-Objective-C Bridge library. This library was created by Steve Hannah and is distributed under the Apache 2.0 license, available at https://www.apache.org/licenses/LICENSE-2.0.

Webcam-capture

The NetLogo `vid` extension makes use of the Webcam-capture library. The copyright and license for Webcam-capture are as follows:

The MIT License (MIT) Copyright (c) 2012 - 2015 Bartosz Firyn and Contributors Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A

PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Guava

The NetLogo `ls` extension makes use of the Guava library. Guava is released under the Apache License 2.0 (http://www.apache.org/licenses/LICENSE-2.0)

Gephi

The `nw` extension makes use of the Gephi library. Gephi is licensed under the following terms:

Gephi Dual License Header and License Notice
The Gephi Consortium elects to use only the GNU General Public License version 3 (GPL) for any software where a choice of GPL license versions are made available with the language indicating that GPLv3 or any later version may be used, or where a choice of which version of the GPL is applied is unspecified.

For more information on the license please see: the Gephi License FAQs.

License headers are available on http://www.opensource.org/licenses/CDDL-1.0 and http://www.gnu.org/licenses/gpl.

JSSC

The NetLogo arduino extension makes use of the JSSC library. JSSC is released under the LGPL 3.0 license.

Apache Commons CSV

The NetLogo csv extension makes use of the Apache Commons CSV library, which is released under the Apache 2.0 license.

JTS Topology Suite

The NetLogo gis extension makes use of the JTS Topology Suite, which is released under the EPL 2.0 license and the BSD 3-Clause license.

Apache Commons HTTPClient

The NetLogo gis extension makes use of the Apache Commons HTTPClient library, which is released under the Apache 2.0 license.

Apache Commons Logging

The NetLogo gis extension makes use of the Apache Commons Logging library, which is released under the Apache 2.0 license.

ng-units

The NetLogo gis extension makes use of the ng-units library, which is released under the MIT license.

Tinfour

The NetLogo gis extension makes use of the Tinfour library, which is released under the Apache 2.0 license.

json-simple

The NetLogo gis extension makes use of the json-simple library, which is released under the Apache 2.0 license.

hid4java

The NetLogo gogo extension makes use of the hid4java library, which is released under the MIT license.

JAMA

The NetLogo matrix extension makes use of the JAMA library, which does not have a license.

Gson

The NetLogo table extension makes use of the Gson library, which is released under the Apache 2.0 license.

Version History

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

Version 7.1.0-alpha1 (August 2026)

Biggest new feature

- NetLogo modules
  - Modules (`.nlm` files) can now be used to export procedures with the `export` keyword
  - In the current implementation, only procedures may be declared within modules, and they may not refer to anything that was defined in the main Code tab
  - Modules and the main Code tab can import procedures from `my-module.nlm` with the code `import [ that-specific-procedure ] from my-module`
  - More information can be found here
  - The models "Discrete Event Mousetrap" and "Distribution Center Discrete Event Simulator" also provide examples of the system in action

Major new features and changes

- Improved Code Tab
  - The code completion functionality now gives smarter recommendations and appears without needing to invoke a key combination. This functionality can be disabled in the application preferences.
  - Dynamically generated terms (e.g., specific breed names) will now get syntax highlighted
- Improved Info tab
  - Now supports LaTeX, as well as embedded videos and images
- NetLogo can now be installed on macOS simply by dragging the icon into the `Applications` folder in the installer
- There is now an online service for automatically updating old models to the latest version of NetLogo, which can be found here
  - The new online system covers more cases than the in-app auto-converter can
- BehaviorSpace Improvements
  - BehaviorSpace runs in a separate process, preventing memory leaks from sustained use
  - BehaviorSpace displays in a separate window, allowing usage and editing of the model while experiments are running
  - BehaviorSpace can now be given an upper limit of how much memory to allocate to the process
  - The BehaviorSpace "Run" dialog now allows specifying desired behavior for when errors occur: Ignore them, abort the errored run, or abort the entire experiment

NetLogo Web Changes

- The GIS extension now works in NetLogo Web and provides `load-dataset-from-string`, `store-dataset-to-string`, and `store-dataset-to-strings` primitives that provide a NetLogo Web-compatible way of reading and writing GIS data without needing to access the file system.

- The NW extension now works in NetLogo Web and provides `load-from-string` and `save-to-string` primitives that provide a NetLogo Web-compatible way of reading and writing network data without needing to access the file system.
- Ability to set model name for display in NetLogo Web (in View/Model settings)

Minor Features and Changes

- NetLogo once again ships with a copy of the user manual in the form of static HTML files, and will open those when "Quick Help" is used without an internet connection
- Monitor widgets now have a "Copy text" option in their right-click menu
- Command Center input can now consist of more than one line of code with the standard Shift+Enter/Shift+Return key combination
- Command Center input is now saved, per model, across NetLogo sessions
- Improved experience for editing NetLogo files with third-party editors
  - After turning on the "Enable remote commands" preference, NetLogo can receive commands from external sources
  - Code and Info tab contents are always bounded by newlines and `CDATA` tags
- Improved command-line interface for launching NetLogo (now much clearer about errors and available options)
- The Models Library dialog once again has folder and file icons for distinguishing the different types of entries
- `.nls` files are now forbidden to take on names that would be invalid NetLogo variable names
  - The "Distribution Center Discrete Event Simulator" and "Discrete Event Mousetrap" models now demonstrate the usage of the new module system
- `.nls` files can be renamed within NetLogo by right-clicking their tab in the list
- The "internal error" dialog now provides a simple button for directly reporting errors to the NetLogo team
- Updated to Java 21
- While the Scaladocs for the NetLogo API (e.g. for the Controlling API or authoring NetLogo extensions) will still be available on our website, they will no longer be distributed with each installation of NetLogo

Bugs Fixed

- NetLogo language
  - Improved error messages when a string or number is provided as a global variable name
- Widgets
  - "Forever" buttons with "Disable until ticks start" enabled will now stop when `clear-ticks` is called
  - For display text of some particular widths, ellipses were wrongly being shown when creating new widgets
  - Plot widgets now reset the scroll level of plot pen editors after the dialog is closed
  - Plot pen editors now instantly recompile when you click away from them
  - Sliders now handle when widget editor fields are invalid-but-compilable more gracefully
  - Note widgets were showing excessive whitespace in margins when set to render Markdown
  - Output widgets now do a better job of importing long content with `import-world`
  - Output widgets now crop the text on a line-by-line basis
  - Reduced visual artifacts on some rounded borders
- View
  - Drawings could previously be duplicated when combining world wrapping with large pen sizes
- Command Center

    - Fixed a case where the font style was changing when zoomed
    - The “command history” button now matches the look and feel of the rest of the NetLogo 7 UI
- Code tab
    - Error text can now be selected and copied
    - Missing characters (e.g., CJK characters) are now shown in a fallback font when the current font does not define them
- Extensions
    - Fixed a crash with `vid:camera-select` on Windows when no camera was selected
    - Fixed an “in-use” error when opening multiple models in a row that use the `vid` extension
- BehaviorSpace
    - The error messages about invalid world dimensions have been improved
- HubNet
    - After importing a HubNet client interface via “File → Import → Import HubNet Client Interface”, the widgets can now be edited properly
    - Client view widget height now updates correctly when set in the “Edit” dialog
- NetLogo will now suggest using our **external model-converter service**, when opening a model created in a version earlier than 6.0.0
- Auto-converter within NetLogo Desktop
    - Now knows how to remove the old `hubnet-set-client-interface` primitive
    - Will now run `movie-` / `vid` conversions, even if only `movie-set-frame-rate` is used in the model
    - Avoids producing syntactically invalid code when doing conversions
    - Can now relocate declarations that were defined in now-invalid places
    - Can now run on code in monitor widgets, even if that code is not wrapped in parentheses
    - Will not add extra parentheses around code in monitor widgets
    - Can now handle conflicting breed names
    - Can convert from the old `breed \[ xs \]` syntax
    - Can install extensions automatically, when needed
- Miscellaneous
    - The “Reload model on external changes” preference no longer leaves watch services running unnecessarily
    - The “Extension Manager samples” section of the Models Library is now far more organized and navigable
    - Uninstalling an in-use extension will no longer produce internal errors
    - Application icons were not being set properly on Linux
    - Reduced display of redundant announcement banners
    - The “About” dialog’s text can now be selected and copied
    - The “About” dialog no longer shows a text input caret after being clicked
    - Loading an autosave from an old version of NetLogo used to sometimes lead to an unwanted and unnecessary “Save as” dialog
    - “Undo” and “redo” options were presented differently in the Code tab from how they are in the Interface tab
    - Error dialog now points people to the newer URL for downloading the latest NetLogo
    - The error message when loading a model from a version of NetLogo that is simply old no longer claims that the version is unknown
    - The Info tab “Help” links are now valid again
    - The Extension Manager now does a better job of handling registries of extensions from multiple different versions

Models Library Changes

New Sample Models

- CA 1D General

Sample Model Changes

- Distribution Center Discrete Event Simulator, Discrete Event Mousetrap: These models now use the new module system instead of `__includes`
- Bidding Market: Monitors no longer error when there are no agents, and instead simply display "<no agents>"

Version 7.0.4 (April 2026)

New Features

- Users can now select a preferred code font (see the "Code" tab in the preferences)
  - In addition to simply providing more flexibility, this is intended to provide a workaround for users who have reported that Chinese characters were not rendering in code on Windows
  - Note that you can quickly search for a specific font by typing a subset of the font's name after opening the dropdown
- NetLogo will now open the relevant help page in the bundled user manual PDF when an internet connection is not available
  - The static HTML version of the manual will no longer be provided, since it mainly duplicates the content of the PDF
  - NetLogo will still default to opening your browser to the latest docs on docs.netlogo.org when an internet connection is available
- Exported BehaviorSpace experiments now record which version of NetLogo they were exported from
- NetLogo will now ask for confirmation if you are about to discard data from paused BehaviorSpace experiments (e.g. by closing the application or changing models)

Bugs Fixed

- Widgets
  - Mouse cursor icons were sometimes failing to change appropriately when editing widgets
  - Input widgets were getting set to the "String" type after being edited
  - The text input on vertically-oriented sliders was not accepting clicks properly
  - Plot pen code editors could "undo" into unexpected states
  - Plot pen "update" code editors did not provide good scrolling
  - The plot-level pen "update" code editors were not keeping their contents in-sync with their respective pen-level pen "update" code editor
  - Plot code editors were closing on mouse-down (instead of mouse-up), reducing sensitivity to mouse movement and preventing abortive clicks

    - Trying to save a plot with duplicated pen names could lead to multiple error dialogs
    - The “Snap to grid” setting from the previous model was persisting when opening a new, empty model
    - The plot editor’s display panel for runtime errors was displaying incorrectly
    - When panning the 3D view widget, agents could be visible that should not have been, due to the topology
    - The “forever” icon on buttons did not previously respond to zooming
    - Clicking a slider widget would block the use of some other valid keyboard shortcuts
    - Sliders could previously only handle span ranges of up to about 2e9 numbers, but can now span the full range of double-precision floating-point numbers (closer to a span of 3e308 values)
    - Compilation errors were not being displayed within “Edit” dialogs for slider widgets
    - Some widget types were not offering tooltips when title text was truncated by ellipses
    - Button action keys now only work when the interaction tool is selected
    - Making a selection box that intersected the view widget sometimes left erroneous vertical lines in the view
    - Note widgets were not always using the correct font size when zoomed
    - Sliders resized incorrectly when zoomed
    - The arrow icon on chooser widgets failed to resize correctly when zoomed
- Editor
    - The “Comment / Uncomment” action would previously only add `;` to the beginning of the line, but now respects the pre-existing indentation of the text
    - Using the “Comment / Uncomment” action on multiple lines of code was not being treated as a single action, with regards to “Undo”/“Redo”
    - Only the main Code tab editor was allowing use of the keyboard shortcut for toggling the comment state of a line of code. Now it works in widgets too.
    - `__includes` files were always being compiled on-save, unlike the main Code tab
    - `__includes` files would display their compilation errors in the main Code tab
    - `__includes` files would not update the name listed in their tab until hovered by the mouse
    - `__includes` files, when saved on Windows, could produce duplicated tabs
- Extensions
    - GoGo: The daemon process could erroneously continue running if NetLogo closed unexpectedly
    - GoGo: The daemon process previously required a separate, manual installation of Java, but now simply reuses the one that ships with NetLogo
    - Bitmap: Was not always notifying NetLogo of the need to perform a redraw after importing an image
    - LevelSpace: Child models opened in separate code editor tabs could not be closed
    - LevelSpace: The child model UI was not adapting to main application’s selected color theme
- Engine
    - NetLogo was producing an error when the size of an agentset changed while that agentset was being iterated over
    - NetLogo code that calculated an infinite/non-numeric result could result in the wrong error message
    - `resize-world` was not resetting the tick counter
- BehaviorSpace
    - BehaviorSpace was not giving early or informative errors, when the world dimension parameters were clearly invalid
    - BehaviorSpace experiment selection was not being remembered, after editing that experiment
    - BehaviorSpace experiment selection was not being remembered, after importing a new experiment
    - The directory from which the last BehaviorSpace experiment was imported was not being remembered
    - Checkbox selections (i.e. whether or not to update plots or the view) were not being remembered after resuming a paused experiment

    - Tooltips on labels in the experiment editing dialog were hiding too quickly
    - Experiments exported to XML were not being given proper XML headers
- HubNet
    - Some HubNet widgets had unintended differences in sizing and layout from non-HubNet widgets
    - HubNet toolbar actions were sometimes interfering with standard toolbar actions
    - The HubNet "mirroring" option was getting out of sync when opening multiple HubNet models in a row
    - The HubNet activity name was getting out of sync when opening multiple HubNet models in a row
    - "Ghost" monitor widgets were not showing a preview name in the HubNet Client Editor
    - An unnecessary number of blank lines were being printed out to the HubNet Control Center when sending a message
    - HubNet could sometimes display an incorrect list of connected clients, if multiple connections were established at the same time
    - Changes to the drawing layer would sometimes be delayed in appearing to clients
    - Opening the HubNet Client Editor would always cause NetLogo to behave as if the current model had unsaved changes.
- Color themes / Dark mode
    - The default color theme was not syncing with the device's system-wide settings
    - The divider between the widget area and the Command Center could erroneously take on a shade of gray that did not visually distinguish the two sections very well.
    - Apply to System Dynamics Modeler tabs and code editor
    - Apply to "Import from Library" button in shapes editor dialogs
    - Apply to code block text in rendered Info tab content
    - Don't apply to plotting area borders; use value of `set-plot-background-color` instead
- Misc
    - Files were reloading due to external changes, even when the preference for this was disabled
    - Editing a model that was saved in OneDrive (or similar) was causing unnecessary reloads of the model
    - `netlogo-gui.sh` did not properly launch on Linux (although `bin/NetLogo` worked as expected)
    - Errors when parsing invalid `.nlogox` files were sometimes not handled gracefully
    - Multiple "Are you sure you want to overwrite this file?" dialogs were being displayed, when only one was needed
    - `.nlogox` files could be opened by dragging and dropping into the application, but `.nlogo` files could not
    - A dialog was being displayed to warn about unsaved changes when first opening the Models Library window, instead of waiting until a model was actually being opened from the Models Library
    - The analytics/telemetry functionality could lead to a freezes or unnecessary use of resources, when unable to upload the data
    - The announcement banner was made shorter
    - Old autosaves of models were being kept around for far too long
    - Models were not being marked as in need of saving after changing the view update mode
    - Improved the contrast of buttons that are disabled until ticks start
    - The interactable model list in the Models Library dialog had the same background color as the non-interactable sections of that dialog
    - Code editors with compiler errors were sometimes scrolling to the bottom whenever they were made visible
    - Documentation, text labels, and file names were inconsistently using the terms "BehaviorSearch" and "Behaviorsearch". It is now consistently "BehaviorSearch".
    - Attempting to open NetLogo models from Finder with the Mac launcher installed was opening the application but not loading the selected file
    - Error messages were unhelpful when headlessly launching BehaviorSpace experiments in invalid configurations

- When compiling invalid code of the form `set [ 1 2 ] [ 1 2 ]`, the error message was unhelpful
- The 3D view widget's "Model Settings" dialog was not showing the option to mirror the updates to the 2D view
- The 3D view widget would (in rare circumstances) throw an error when being opened or closed

Version 7.0.3 (November 2025)

- When a runtime error was generated, the application was not jumping to the line of code that generated it
- A slider widget could sometimes ignore its initial value
- The "Jump sliders to click location" preference was preventing proper dragging of the "speed" slider knob
- The "Jump sliders to click location" preference was canceling drags of slider widget knobs as soon as the mouse cursor went outside of the slider
- Slider widgets were not correctly changing their dimensions when toggling the "vertical?" setting
- Slider values could get rounded towards the slider's increment, even when manually set to a specific value
- `user-yes-or-no?` was raising an error when "Halt" was selected in the resultant dialog
- Preferences were only toggleable by clicking their respective checkbox, instead of also allowing clicks on the text describing the preference
- The "Auto-detect" option for the interface language preference was defaulting to the previously-selected language, instead of the system default one
- The Extension Manager was not offering to automatically install libraries when it could easily do so
- The GoGo extension was not loading all of its libraries correctly
- The GoGo extension could leave an orphan process running after NetLogo closed
- The block-comment/-uncomment functionality was not properly handling text selections where at least one line was entirely commented out and at least one line wasn't
- Extensions were not properly raising errors when they occurred within a separate thread
- Multiple models had one or more improperly sized widgets, after their conversion to NetLogo 7's new widget format
- Compiling NetLogo code with the system locale set to Turkish could cause compilation errors
- NetLogo 3D was not respecting non-default patch sizes
- NetLogo 3D was failed to report turtle coordinates after setting them in turtle monitors
- "Quick Help" would raise an error if used on a semi-colon (";")
- "Edit" dialogs could sometimes have the wrong focus behavior
- The `.nlogox` file icon was too narrow
- In the Models Library dialog, the file selector's border would glow in response to mouse clicks
- The System Dynamics Modeler dialog was not being closed when Escape was pressed
- The "Print" option was unconventionally appearing below "Quit" in the file menu
- The "headless" version of NetLogo would hang after encountering errors
- The analytics functionality was uploading values entered for the "Directory to store logs" preference. We did not previously realize that this information could be de-anonymizing, so we have now ceased collecting data about what that preference is set to. Only about one user per thousand had data for that preference uploaded to our data collection server, and the data has since been purged.
- On Windows, `__includes` files were always displaying their full path, instead of just the file name
- On Windows, `__includes` files would generate warnings and behave incorrectly if `/` was used as the path separator, instead of `\`

Version 7.0.2 (October 2025)

Major bugs fixed

- Loading a model with a slider's lower bound, upper bound or increment set to anything other than a constant would result in a critical error.

Version 7.0.1 (October 2025)

Bundled software to open multiple instances of NetLogo on Mac

- A *NetLogo Launcher* app is now included on macOS to allow opening multiple instances of NetLogo simultaneously (on other platforms, you can already open multiple instances of NetLogo by default)

Interface changes

- Plots, monitors, and output widgets are now slightly lighter grey in light mode.
- Disabled buttons now have slightly darker text (for increased contrast) in light mode.

Changes to NetLogo Web export

- NetLogo Desktop will now export the latest version of NetLogo Web
  - The most notable change to NetLogo Web is that it now sends anonymized usage data to our Google Analytics
    - The amount of data tracked by this is very minimal, so we ask that users please leave it enabled so that we can better understand how self-hosted NetLogo Web models are being used. However, note that the functionality can be disabled by appending `?disableAnalytics` to the URL
  - Other updates to NetLogo Web are documented here, from version v2.12.1 to v2.12.4

Major bugs fixed

- Primitives for writing text to files were always overwriting the files, rather than appending to them
- Running Grammarly on a Mac could prevent using the 3D View

Minor bugs fixed

- Monitors that were displaying numbers in scientific notation were showing an extra "0" at the end
- Monitors were switching to scientific notation at 10,000,000 instead of at 9,007,199,254,740,993 (2^{53})
- The "Reload model on external changes" functionality was not watching all of the files that it should have been for the full duration that it should have been
- The "Reload model on external changes" functionality could sometimes trigger the display of multiple redundant "Are you sure you want to open a different file?" dialogs
- The "Reload model on external changes" functionality could sometimes raise unnecessary exceptions
- Internal `lastModified.txt` file handles were being inefficiently left open
- NetLogo would sometimes suggest importing an extension immediately after having imported it
- The color theme was defaulting to "light", rather than checking the system's default theme

- Some modal dialogs were being hidden behind other windows
- macOS and Windows were using incorrect file icons for `.nlogo` and `.nlogo3d` files
- macOS was consistently producing more than one "Print" dialog at a time (yes, as in: printing to paper)
- Tabs for `__includes` files were wrongly being foregrounded when reopened
- Tabs for `__includes` files could not be zoomed when popped out into separate windows
- The "Auto-indent" setting was not applying to secondary code editors (e.g. inside of widget "Edit" dialogs)
- `self` and `myself` were reporting incorrect values inside of `carefully` blocks
- Mathematica Link was setting the wrong location to look for NetLogo extensions
- Patches could wrongly persist in stored patchsets after calls to `resize-world`
- The internal optimized form of `any? <breed>-on` was was producing incorrect results
- Printing anonymous procedures to strings (e.g. with the `word` primitive) was over-eagerly eliding parentheses that are actually necessary when calling primitives that don't take a fixed number of arguments
- The Command Center would display illegible gobbledygook on Linux if the output line was wider than 32767 pixels
- The colors of square brackets had poor visibility in dark mode
- Auto-saves of the "File > New" model were never expiring
- "Headless" versions of NetLogo could not open NetLogo 7-formatted BehaviorSpace experiments
- NetLogo would give up on trying to open documentation if it lacked a connection to the internet, instead of falling back to using the copy of the documentation that comes with the NetLogo installation
- Some "help" dialogs were being placed in the wrong location on the screen
- Brackets were being visually matched in editors, even when the containing editor was not foregrounded
- The icon used in the compiler error banner was needlessly low-resolution
- Note widgets were failing to correctly wrap text when rendering in the standard (i.e. non-Markdown) mode
- The Extension Manager could sometimes fail to load the list of extensions properly
- BehaviorSpace was refusing to validate or save any experiments that referenced special variables (e.g. `world-width`, `random-seed`)
- BehaviorSpace could open dialogs at wrong locations and sizes
- BehaviorSpace (pointlessly) allowed pausing after the final run
- NW Extension: Graphs were not always retaining their ordering when they should have been
- Vid Extension: Native library files were not being properly cleaned up when "temporarily" unpacked
- Vid Extension: Would crash when loaded headlessly
- Vid Extension: Could not use webcams on macOS versions before 14
- `set-patch-size` wrongly allowed values of 0 or less
- Opening a new file while focusing a text input could cause an exception
- Extension Manager downloads didn't raise errors when they should have
- Single-line text fields did not respect undo/redo
- When converting models from the old file format to the new format, slider widgets could sometimes lose their default value
- Agent monitors could sometimes render at too small of a size when the model's world was small

Version 7.0.0 (September 2025)

Changes in NetLogo Functionality

This is a major upgrade from NetLogo 6.4. We made many changes to basic features, the NetLogo GUI, the file format and NetLogo library models.

Some of the biggest changes we made in this release are:

- A new modernized GUI with both light and dark modes (and a classic mode) (**see below**)
- Greater customizability, with more preferences and settings (**see below**)
- NetLogo models are now stored in a new format with the extension `.nlogox` (**see below**)
- For users with modern macOS devices that use the Apple silicon architecture, we now provide a separate, higher-performance version of NetLogo to download

Below, we list all the changes since NetLogo 6.4.

Changes to the NetLogo GUI

Major Changes

- The Interface Toolbar has been updated for usability: there are now four different tools for interacting with or editing widgets. These change the way you author and edit models:
  - With the Interaction tool, users can interact with widgets much as they did before NetLogo 7.
  - With the Selection tool, users can select and move widgets by clicking on them.
  - With the Edit tool, users can edit widgets after a single click (no previous selection required).
  - With the Delete tool, users can delete widgets by clicking (no previous selection required).
- There are new tools to align, distribute and stretch widgets. These have a menu in the Interface Toolbar.

Other Changes

- See the **NetLogo 7.0.0 Changes Overview** for more details.
- User interface elements (widgets) have a flat modern look with a new color scheme.
- Users have a choice of color themes: Light, Dark and Classic.
  - The classic color theme is similar to the light theme, except the widgets have the pre-NetLogo 7 colors.
- Widget sizes have been adjusted for better aesthetics and readability.
  - All widgets now open with the new widget *styles*, but pre-NetLogo 7 models will open with the old widget *sizes* by default. However, the new widget styles are not designed to look or function well with the old widget sizes, so model authors are strongly encouraged to update their models to use the new widget sizes.
  - There is a tool to automatically update the layout of a model to better support the new widget sizes.
  - Users can set widgets to have their pre-7.0.0 default sizes if they want.
- Options to delete, copy and paste all selected widgets have been added to the right-click menu.
- A preview is shown when widgets are added.
- Buttons have new icons for observer, turtle, patch and link.
- Sliders now display their values in an editable box.
- For Monitors, users can specify the units represented by numbers.
- The font and background colors for Notes can include transparency and are specified separately for Light and Dark Modes.
- Icons for the Interface, Info and Code Tabs have been updated.
- The icons for moving the command center from the bottom of the Interface Tab to its right side and back are now more intuitive.
- The widget pull-down menu is easier to read.
- The Model Speed slider now has buttons to increase and decrease model speed.
- Code Tabs for included (.nls) files can be closed by clicking on the tab.

- Code Tabs for included files have checkboxes for opening a separate code window.
- Headers in the Info Tab have been redesigned.
- The color themes are color blind friendly.
- The shapes list in the Turtle Shapes Editor and the Link Shapes Editor is now searchable.
- Color Swatches has been replaced by a more powerful Color picker.
- The documentation format has been updated.

Changes to the NetLogo File format

- The extension of model files is now `.nlogox` or `.nlogox3d`.
- Existing `.nlogo` model files will be converted to `.nlogox` files.
- NetLogo 7 only saves models as `.nlogox` files.
- The .nlogox files are XML files.
  - Any third-party scripts that processed `.nlogo` files are not expected to function correctly on `.nlogox` files
- The `.nlogox` format allows the saving of model-specific information.
- The `.nlogox` format allows the saving of user interface element specific information.
  - For example, a user interface element can appear in the new larger size, or the original size.
- The `.nlogox` format is extensible and will allow new types of data to be stored in the future.
  - This makes it easier to add capabilities and keep models backwards compatible.
  - **Resources** (e.g., pictures, data files) needed for a model may be stored in the `.nlogox` file in the future.

Changes in Primitive Functionality

- There was a small bug fixed in the behavior of the `random` primitive. Models that use `random` and `random-seed` may not generate the exact same behavior as in NetLogo 6, though the vast majority of models should show no change in macro-behavior.

Changes for Customizability

- Widgets that have changed size for NetLogo 7 have a checkbox that allows users to switch between the new larger size in NetLogo 7 and the previous size used in NetLogo 6.4.
- The “Auto-indent” and “Show line numbers” global preferences can be set from code section of the NetLogo settings dialog (also available from the code tab).
- There are changes to the Preferences dialog which let you customize various NetLogo settings.
  - External editor support: Users can enable a feature that checks the open model for changes made outside of NetLogo to the model file and to any included files. When a change is detected, the file is updated in NetLogo. This feature can be enabled or disabled using the NetLogo preferences dialog (also available from the code tab). It is disabled by default.
  - There is a new preference option that prevents compilation errors from stealing focus from the current tab. By default, it is not enabled.
  - There is a new preference option that causes a separate code tab to be opened when NetLogo starts. By default, it is not enabled.
  - There is a Linux-only **UI scale** preference that matches NetLogo graphics to the system UI scale.
  - There is a preference that enables users to control whether anonymous data will be collected when they are using NetLogo. It is enabled by default.
- Users can switch between Light, Dark and Classic color themes using an item in the Tools menu.

- A new Color Picker replaces Color Swatches. Users can still select NetLogo colors from a grid of rectangles. The Color Picker has a second tab that allows users to specify colors in most common formats, and that enables easy selection of a more extensive set of colors.

Features and Enhancements

- The NetLogo Graphical User Interface (GUI) has been thoroughly **modernized**.
- The model file format has become **extensible**.
- The NetLogo team can now send announcements which are displayed in a banner on the Interface Tab. Users can click a link for more details.
- Notes on the interface tab can now render markdown/html.
- Users can copy and paste widgets both within and between models.
- There is a new File → Manage Bundled Resources dialog that allows you to view, add, or remove files that are bundled with the current NetLogo model. You can easily embed such files in the Info Tab.
- Plots can now be scaled independently for the x and y axes via checkboxes in the Plot widget.
- The full NetLogo menu bar remains available when you are focused on a separate code tab.
- The Help Menu now has a link to the NetLogo Forum, a website to ask/answer questions and have discussions about NetLogo.
- The Tools→Halt option is disabled if code is not running.
- Plot legends wrap to be multi-line if needed.
- The auto-save functionality will prompt users if there is an auto-save file available (for example if NetLogo crashed).

Language Changes

- The `user-input` primitive has a new optional argument for a default input value.
- The new primitives `auto-plot-x-on`, `auto-plot-y-on`, `auto-plot-x-off`, and `auto-plot-y-off` allow plots to be scaled independently for the x and y axes.
- The new primitive `set-topology` allows you to set the model topology from code (it replaces the "hidden" command `__change-topology`)

Bug Fixes and Minor Changes

- Multi-assign with `let` and `set` can use arbitrarily nested lists to "destructure" variables. For example, `let [[corner1-x corner1-y corner2-x corner2-y] player-name score] get-current-player-info`.
- The primitive `autoplot?` has been restored.
- An optimization speeds up the execution of the constructs `any? turtles-on` and `any? <breeds>-on`.
- An `__includes` statement not followed by brackets no longer causes an exception.
- Reporter procedures without a `report` statement, `run` or `runresult` now throw a compile time error.
- The user receives a specific message when keywords such as `globals`, `turtles-own`, `to` or `end` are used in an incorrect context.
- Unmatched closing parentheses, brackets and braces result in a meaningful error message.
- When a runtime error occurs the user is asked for information that would allow us to reproduce and fix the problem. This is in addition to the stack trace plus system information currently requested.
- "Save as NetLogo Web" adds the contents of any included files to the model code.
- One can now import a world that has an array containing strings with curly braces.

- The Recent Files lists now appropriately contains only 2D or 3D models, according to which NetLogo application is open.
- The “Reload the last model on startup” preference now appropriately opens only 2D or 3D models.
- When the user does a “Save As”, non-NetLogo files are greyed out and cannot be selected on Windows and Linux. It is a MacOS feature that users can still select a greyed-out file. If a user selects a non-NetLogo file on MacOS they are warned that a `.nlogox` or `.nlogox3d` extension will be added.
- The Tools menu now displays the correct Show/Hide Command Center option no matter how the command center is made visible or hidden.
- The process of selecting a log directory has been improved to allow a directory to be created, and to display an error message if the logging directory does not exist when the model is loaded.
- The preferences dialog shows an error dialog and prevents the user from closing it if logging is enabled but no log directory is specified, or if the specified log directory does not exist or if NetLogo does not have the necessary permissions to write to the directory.
- Input widget values of the wrong type, e.g. string values in numeric input widgets, are now rejected.
- Creating a number input box and leaving the value empty no longer causes errors or crashes.
- When the value of an interface element is changed outside of the Edit dialog, the user will be prompted to save the model when closing it.
- An empty link monitor now allows a `who` number to be entered as an end value.
- The value of the “Auto-indent” checkbox now persists between sessions.
- Shortcuts for “Previous word”, “Select previous word”, “Next word”, “Select next word”, “Select word”, “Delete previous word”, and “Delete next word” should work as expected in the code tab and in other text editors.
- The MacOS and Linux convention of specifying a user’s home directory with a tilde at the beginning of a file path is now properly interpreted for included (.nls) files.
- Included files are now recompiled when they are closed without saving the changes.
- Leaving and returning to a “dirty” Code Tab no longer causes compilation.
- There is no exception when reopening an included file that was saved in a separate code tab.
- An included file being edited in a separate window can be saved.
- Opening a separate code tab that contains a compilation error no longer causes an exception.
- The “Save” hot-key works for included files in a separate code tab.
- Doing a text search in a separate code tab that overlaps an interface window will no longer bring the interface window to the front, covering part of the separate code tab. (This was a MacOS-only issue.)
- Changes in focus happen correctly when moving between separate and non-separate tabs.
- Multi-assign with `let` and `set` now works with BehaviorSpace.
- BehaviorSpace works correctly when no measurements have been made. In particular, it no longer produces a runtime error when the following conditions hold: 1) the user specifies a “Run metrics when” reporter, 2) the reporter is never true, and 3) the user has specified both Table and Spreadsheet output.
- BehaviorSpace error messages are improved.
- When a BehaviorSpace experiment is run with one or more specified output files that are open in another program, it now aborts the experiment and shows a useful error message.
- Invalid variable specifications in BehaviorSpace result in useful messages.
- BehaviorSpace output files can now be deleted once an experiment ends. NetLogo does not need to be exited.
- BehaviorSpace experiment edit compilation errors are reported when the dialog is closed rather than at runtime.
- Comments are now supported in BehaviorSpace multiline reporter fields.
- Right clicking the blank space in a new Systems Dynamics Model no longer causes an exception.
- NetLogo no longer hangs moving through a corner with wrapped coordinates and pen-down.
- Number input boxes without a value no longer keep other input boxes or the command center from gaining focus.
- The Page Setup dialog opened when a user wants to Print now closes when its Cancel button is pressed.

- When a procedure in the Code Tab is “folded,” so that only the line with the name is visible, an action that needs to move to inside the code causes it to be “unfolded,” so it is fully visible. This can be a result of a search, find-and-replace or jumping to code with a compilation error.
- If the NetLogo window is maximized in Windows, opening a new model no longer unmaximizes it.
- Plot pens always take account of `plot-pen-up` and `plot-pen-down`.
- The padding and location of internal components of widgets now change proportionally when zooming.
- Models that have not been edited are never considered unsaved.
- Ctrl/Cmd+X when nothing is selected now cuts (copies and deletes) the entire line.
- Double clicking on a model that was saved with compilation errors opens it correctly.
- `mouse-xcor` and `mouse-ycor` no longer degrade performance when called multiple times per iteration of go or in an ask with lots of agents.
- Using a lower case ‘i’ in code with a Turkish locale no longer causes compilations errors
- The resize behavior of edit dialogs has been improved. For example, plot widget “update commands” and “setup commands” fields can now be made taller, and the BehaviorSpace experiment edit dialog can be resized to make controls near the bottom accessible.
- NetLogo now detects autosave files created for a model in case NetLogo crashes. Users are now asked if they want to recover it.
- Mouse coordinates are now correctly reported along the border of a selected view.
- All open files are closed when a runtime error occurs.
- When exporting and then importing a model the last line in an Output area is a newline it will be retained.
- Disabled button state is now preserved by Import world.
- Scientific notation in monitors shows the desired number of decimals.
- Empty 3D models now have more room for widgets.
- Clipboard errors when pasting no longer cause an exception.
- After patch colors have been read from a file the underlying input stream is now closed.
- Some NetLogo display problems are avoided because Microsoft Direct3D graphics are now off by default.
- The default value label tick counter, the names of the panels in the ‘About NetLogo’, and the Preview Commands editor can be internationalized.
- Links to the NetLogo user manual PDF in the documentation installed with NetLogo work correctly.
- Inserting a tab in a Note produces four spaces.
- Adjusting a widget value using arrow keys does not affect other scrollable areas.
- `import-world` correctly handles tables that have a key with a ’:’.
- Better support for worlds with one small and one very large dimension.
- The app icon preview found by searching in the Windows Start Menu, is no longer pixelated.
- Separate windows (such as the pop-out code window) can now be interacted with while other dialog windows are open
- Agent monitor windows will no longer hide after giving a different application focus and then returning to NetLogo
- File choosers will now open to the previously used directory
- Searching for text in the editor will now also center the text’s location in the editor
- The Extension Manager is better able to insert the names of extensions without introducing errors
- Title bars now better match system appearance on Windows and macOS
- The “Preferences” window will offer to restart the application after changing preferences that require a restart
- Code “fold” actions have been reorganized into a submenu of the “Edit” menu
- Code auto-indentation behavior has been improved
- BehaviorSpace will notify about invalid variable values earlier

Extension Updates

- The gogo extension supports the GoGo board 6.
- The dialog extension `user-input` primitive allows an optional default value argument.
- There is a new resource extension that allows you to list and get resources bundled with a model.
- The SimpleR extension is now bundled with NetLogo.
- The R extension is no longer bundled with NetLogo. It is available through the extension manager.
- There is a new bspace extension that allows users to create experiments programmatically.

Documentation Updates

- The Documentation has been updated to describe the many updates that were made for NetLogo 7.0.0. The majority of the images have been updated.
- There are three new pages:
  - NetLogo 7.0.0 Changes Overview
  - Netlogo Preferences
  - Color Picker Guide
- Grammar was improved for NetLogo Tutorial 2.
- The link to the book “An Introduction to Agent-Based Modeling” was updated.
- The BehaviorSpace error message about undefined parameter values was made easier to interpret.
- The BehaviorSpace documentation for the subexperiment syntax is improved and uses examples from the Wolf Sheep Predation Model.
- The effect of topological wrapping on the primitives `neighbors` and `neighbors4` is explained in the documentation.

Models Library Changes

New Sample Models:

- Chladni Figures
- Consumer Choice
- Cyclic CA
- Hockey
- Interstitial Diffusion
- Invasive Plants
- Wolf Sheep Predation - Agent Cognition (folder)
  - Wolf Sheep Predation - Micro-Sims (the main model which runs microsimulations to simulate agent cognition)
  - Wolf Sheep Predation - Micro-Sims Cognitive Model (the submodel for running agent microsimulations)
- The Mathematics section now has a subfolder Conic Sections that contains three models:
  - Emergent Shapes - Ellipse
  - Emergent Shapes - Parabola
  - Emergent Shapes - Hyperbola
- The three models above replace two models from the NetLogo 6.4 mathematics section:
  - Conic Sections 1
  - Conic Sections 2

New 3D Sample Models

- Maze 3D

New Code Examples

- The Ordered Agent Drawing Code Example model demonstrates how to sort turtles for display using `stamp` when precise drawing order is needed.
- The Patch Outlines Example model demonstrates how to outline patches in a reasonably simple way using `stamp`.

Sample Model Changes

- The Moths model was updated with information about new research into why moths circle lights. The idea that this could be modeled with NetLogo 3D was added as a suggestion for further exploration.
- The correct label is used after `hatch`ing in the Plant Speciation model.
- The Discrete Event Mousetrap has additions to the Extending the Model section.
- When the bar is crowded in the El Farol model the patch label is now appropriately yellow.
- An inaccurate comment in Small Worlds model was corrected.
- Multi-line code input widgets were enabled for PNoM 2 Diffusion Sensor Sandbox. Previously single line widgets were incorrectly used.
- The Radical Polymerization model was updated to fix a bug in which Km was used instead of Kr for probability of termination.

Version 6.4.0 (November 2023)

A major change in this release the introduction of new features in BehaviorSpace

BehaviorSpace New Features

- A new syntax that allows parameter combinations to be run separately, rather than being expanded combinatorically.
- Measurements can be conditionally recorded when a boolean reporter is true.
- Commands can be run before the experiment begins.
- Commands can be run after the experiment ends.
- The mean and standard deviation of data from repetitions can be output to a file.
- List data can be output to a file with one list element per cell.
- Experiments can now be paused and resumed.
- Experiments can now be exported to an XML file that can be used when running headlessly.
- Experiments in an XML file can now be imported into a model.
- There is a reduction in memory usage for experiments containing plots when *Update plots and monitors* is unchecked in the *Run options dialog*.

Language Changes

The `let` and `set` primitives now support taking a list of variable names as the first argument and will assign values to those variables from a list of values given as the second argument. This is useful in many situations, but especially when you want to have a reporter procedure calculate and report multiple values to be used elsewhere.

A new reporter primitive has been added, `who-are-not`. It takes an agentset as a lefthand side argument and an agent or another agentset as a righthand side argument and reports all agents in the former who are not in the latter. For example: `let strangers turtles who-are-not link-neighbors`.

Bug Fixes and Changes

- The save dialog will no longer appear when clicking the close button on an already-saved includes file code tab.
- When an includes code file is edited, compiled, and closed without saving, the model will correctly re-compile without the unsaved change.
- When extensions are updated through the extensions manager, existing files and libraries for them will be removed so conflicts between versions cannot occur.
- An error will no longer occur when opening a NetLogo model stored in the user's home directory due to the extension locating code trying to search restricted folders.
- Plotting now uses its own RNG that is cloned from the main RNG and updated when the main RNG changes. This means if plotting is enabled, it will be reproducible from the main RNG, but if plotting is disabled the behavior of the model will not change when using the same main RNG seed.
- A failed compile of a string as a command with `run` could cause the same string to fail to run as a reporter using `runresult` later.
- Patch colors can now be set to RGBA values with an alpha channel, the alpha channel is just ignored as there is nothing "behind" patches. Previously this would cause an error to occur.
- Users can add symbolic links to their models library folders to point to other directories to have those models available for loading in the app. This had worked in a prior version of NetLogo, but was accidentally disabled with other bugfixes.
- The `link-neighbor?` boolean reporter has had some small optimizations done to improve performance.
- Agent labels now support linebreaks.
- Errors that happen when NetLogo is starting but before the full interface appears should now be caught and displayed in a simple dialog. Previously to see such errors NetLogo would have to be started from a command line console.
- The default maximum memory usage for NetLogo is now set to 50% of system memory instead of a static limit of 1 gigabyte. For systems with greater than 2 gigabytes of memory this should give improved performance and reduce the chance of getting out of memory errors during model runs.
- When the extension manager is disabled via the Java environment property, no data will be fetched over the network. Previously the current extension listing would be fetched even though using the list in the manager was disabled.
- The `<breed>-at` primitive now works properly in 3D. Previously it would function as the 2D version, only getting agents with z-coordinate of 0.

BehaviorSpace Bug fixes and changes

- 3D BehaviorSpace runs are now allowed.
- BehaviorSpace no longer exits when there is a plot compilation error.
- Error messages for BehaviorSpace variable errors have been improved.
- BehaviorSpace output files no longer contain trailing spaces.
- Headless console mode now has a `--help` flag.
- The *Experiment dialog* makes use of tooltips and has a cleaner look.
- Changes will not be saved if the close/X button of the *Experiment dialog* is hit.
- The number of threads in the *Run options dialog* is now saved.
- Experiments will not run if the close/X button of the *Run options dialog* is hit.

Extension Updates

- The Bitmap extension now supports reading images from base64-encoded strings and writing images out to base64-encoded strings.

- The GIS extension has a couple of bug fixes: 1) its string-based search of features could include too many items, and 2) the code for creating turtles inside a polygon was executed for existing turtles outside that polygon.

Documentation Updates

- The logical operator precedence for `and` and `or` is clarified in the docs as not exactly matching how other programming languages function.
- Clarify that `mouse-inside?` only works when the mouse is in the world, not only in the view, which can happen with unwrapped worlds and use of the `follow` primitive.
- Removed the `->` variable, a leftover from the switch from the old anonymous task syntax.
- Clarify that an extension must be installed for the extensions keyword to take effect.

Models Library Changes

New Sample Models:

- Food Reward Learning
- Current in a Semiconductor
- Radical Polymerization with Chains
- Food Reward Learning
- Baldwin Effect

Sample Model Changes

- Fixed a minor issue in the Monte Carlo Lennard-Jones model.
- Renamed Prisoner's Dilemma models from "PD" to "Prisoner's Dilemma" and clarified that "confess" means "defect" in the info tab.
- Polymer dynamics model info tab was edited to indicate that this is not strictly speaking a cellular automata.
- Wolf Sheep Predation was updated to include BehaviorSpace experiments that use the new BehaviorSpace features.
- Anisogomy model was updated and improved.

Version 6.3.0 (September 2022)

Bug Fixes and Changes

- The major change in this release is the upgrade of the bundled Java from version 8 to 17. Java 17 is the current long-term-support release of the Java runtime.
- The models library will correctly display when NetLogo is set to a language other than English.
- The log4j library has been removed from NetLogo. Logging is now configured from the NetLogo preferences dialog. The output is in JSON text file format. See the Logging docs for more information.
- BehaviorSpace experiments can now also be run from a terminal or command console using the "NetLogo_Console" app with NetLogo's bundled Java by using the `--headless` flag. See the BehaviorSpace documentation for more information and examples. This avoids the need to install and configure a separate Java runtime to use the `netlogo-headless.sh` script, which remains bundled for those who require it.
- BehaviorSpace output files are now chosen directly during run setup instead of separate prompts at the run start. This simplifies the setup process and resolves a bug where the runs would not occur on macOS.
- The System Dynamics Modeler UI has some small improvements.

- Fixed a bug where slider widgets would allow initial values outside of their range when changing the limits.
- Better handle asymptotic values for tan with a proper error message.
- NetLogo will now correctly handle running `map` or `foreach` when no lists are given.
- When an anonymous task that uses `let` variables is displayed as a string it will now include those variable names.
- Reporters used in concise task syntax no longer give the wrong result when used with fewer arguments than their default.
- The models library location will be correctly set when using GUI scripts.
- Behaviorsearch now sets a flag to alert extensions that they are being run without a UI so that extension code can adjust appropriately.

Extension Updates

- GoGo: Improve newer GoGo board compatibility and improve the stability of the extension.
- LevelSpace: Eliminate crashes when the controlling API is used to run NetLogo without a UI.
- Profiler: The `profiler:data` results are now given in milliseconds instead of nanoseconds, matching the other primitives in the extension.
- Python: Improvements in how the extension searches for the Python binaries when they aren't specified via the configuration menu.
- Vid: Update backing libraries to try to improve stability and usability on macOS and Windows.

Documentation Updates

- Update the system requirements to include Windows 11.
- Fixed a bad link to the extension API version 5 to 6 transition guide.
- Added a FAQ entry on how to save random seeds from runs.

Models Library Changes

New Sample Models:

- Slime Mold Network
- Virus Using Discrete Event Simulator (cross-listed in code examples for time extension)
- Distribution Center Discrete Event Simulator (cross-listed in code examples for time extension)
- CA 1D-Squaring
- Hydrogen Gas Production
- Molecular Dynamics Lennard-Jones

New Curricular Models:

- CT-STEM:
  - Habitat Preference

New Code Examples:

- `time` : Logotimes Example
- `time` : Discrete Event Whack-a-Mole

Sample Model Changes

- Taxi Cabs: Fixed Dongping Zhang's name in the credits.
- Turbulence: Cross-listed under CS/CA.
- Tree Simple: Changed `init-x slider` min/max to -100/100. Before the range was -125/125, which caused an error.
- Traffic Basic: Changed `speed-limit` and `speed-min` from agent variables to be globals.
- Urban Suite - Cells: Renamed the `go * 80` button to `go * 40`, reflecting its actual effect.
- CA 1D Rule 30 - Improved the info tab to explain how Rule 30 has been used as random number generator.

Curricular Models Changes:

- PNoM 4 Virtual Syringe Force Graph: fixed a bug that produced an error message.
- Kidney Reabsorption: Some improvements to the info tab.

Newly Verified Models

- The suite of prisoner's dilemma models have been in the models library for a long time but were previously "unverified." They have been reviewed/verified and moved into the main Social Science folder.

Version 6.2.2 (December 2021)

Fixes and Changes

- When Agent Inspector window opens put focus in Command Line.
- When enter macOS full screen mode move inspector windows to foreground.
- Properly export models to HTML with the **Save As NetLogo Web...** option.
- Fixed a bug where preview images would not generate correctly for models that imported background images on setup. This affected a couple of library models: Robotic Factory and Bug Hunt Camouflage.
- Added an explanation about Extension Sample Models to the models library info screen, and now those models will be listed as unverified.
- Fixed a small typo in the FAQ for the path to the `NetLogo.cfg` file on macOS systems.
- Add a `profiler:data` primitive to the Profiler extension that returns all the profiler data as a list of lists suitable for passing to the CSV extension for output to a file.
- Fixed bug preventing typing into Command Line of first Agent Inspector window opened.
- Fixed bug causing Agent Inspector windows to appear behind NetLogo window when there is an overlap.
- Improved documentation of Fullscreen mode for the 3D view.
- Restored hotkeys for Comment / Uncomment, Shift Left, Shift Right, Jump To Declaration, and Show Usage for separate code tab.
- Added documentation for Jump To Declaration and Show Usage.
- Clarified what the BehaviorSpace Table and Spreadsheet output format options do.

Version 6.2.1 (October 2021)

Features and Changes

- Agent monitor inspection windows can now move across monitors
- Improved error message on BehaviorSpace file access failures

- Use a common error message for `turtle-set`, `patch-set`, and `link-set`
- Use same font for input widgets as choosers/sliders/buttons
- Select the search text on models library dialog open for easy replacement on a new search
- Automatically add the png extension when exporting the world view
- Added a preference to have NetLogo automatically load the last used model on startup
- The “mammoth” shape is available in the shapes library for easy import into models
- Substantial GIS extension improvements and new primitives, see the GIS extension changelog for more information
- Substantial Palette extension improvements and new primitives, see the Palette extension docs for more information

Documentation Updates

- Add a note about transparent colors to the `color` dictionary entry
- Documentation for `inspect` and `stop-inspecting` now includes links
- Make sure https links in standalone dictionary entries work correctly
- Make the BehaviorSpace command-line examples use real models from the library with experiments that can be run without modification
- Link to the new Beginner’s Interactive NetLogo Dictionary site from the help menu (BIND)

Bugfixes

- Shape changes in 3D mode will take effect without a restart
- Runtime exceptions with wrong argument types properly generate stack traces
- When sort-on receives input values that are not all numbers, all strings, or all agents of the same type, a user-friendly error message is generated
- Breeded links are properly checked for equality
- `exp` now errors when it generates an invalid, out-of-range number result
- A proper “file not found” error will be given when a model in the recent files list was deleted or moved and cannot be located
- Bundled extensions updated through the extensions manager can have those updates uninstalled through it, too, just like unbundled extensions
- Properly add the nlogo file extension when saving a model with a filename that contains dots
- When a runtime error occurs, NetLogo will show the location in the code tab, as it did in prior versions
- Detached code tab stays in front when model is saved
- Mathematica-Link: Clear up some issues with the tutorial being out-of-date
- Extension: View2.5d shape change bugfixes
- Extension: LevelSpace bugfixes for running GUI-mode models headlessly

Models Library Changes

All external URLs in models’ INFO tabs were validated resulting in URL updates to 15 models.

New Sample Models:

- Limited Order Book

- River Meanders
- Robotic Factory
- Taxi Cab

New Curricula Models:

- CT-STEM:
  - Calorimetry

New Code Examples:

- `palette`: Color Bubbles
- `palette`: Color Painting
- `palette`: Color Reveal

Sample Models

- Brian's Brain: Rewritten erasing function for NLW compatibility.
- Crystallization Basic: Fixed issue where world size depended on `height` rather than `width`.
- Geometron Top-Down: Promoted from `unverified`. Updated INTERFACE, INFO, and CODE tabs.
- Kaleidoscope: Promoted from `unverified`. Updated INTERFACE, INFO, and CODE tabs.
- Lattice Gas Automaton: Mouse movement moved to a forever button for NLW compatibility.
- Life: Rewritten erasing function for NLW compatibility.
- Life Turtle-Based: Rewritten erasing function for NLW compatibility
- N-Bodies: Rewritten mouse handling routine for NLW compatibility.
- Planarity: Updated for NLW full compatibility
- Small Worlds: Fixed bug with clustering coefficient being undefined for single-link.
- Sunflower Biomorphs: Rewritten mouse handling routine for NLW compatibility
- Surface Walking 2D: Mouse movement moved to forever button for NLW compatibility.
- PANDA BEAR Solo: Mouse movement rewritten for NLW compatibility.

Code Examples

- `gis` – GIS General Examples: Rewritten to take advantage of new primitives released in `gis` v1.3.0.
- `palette` – Palette Example: Rewritten to take advantage of new primitives release in `palette` v2.0.0.

Version 6.2.0 (December 2020)

Features

A premier feature for this release is a pop-out code tab, that allows you to detach the code tab from the main application and into a separate window in order to view the interface and the model code side-by-side.

- The models library now shows code examples that come with extensions installed through the extensions manager.
- We now bundle the Liberica OpenJDK Java runtime instead of the Oracle Java runtime, which has a less restrictive license.
- The `mean` primitive will now work on lists that contain non-numbers as other number processing primitives do.
- The extensions manager can be disabled with a command-line switch to avoid network traffic or to lock down installed extensions.

Bugfixes

- We've made changes to how we bundle NetLogo on macOS in order to satisfy Apple's package notarization process for detecting malware. NetLogo now runs without any special workarounds, as long as your macOS settings allow running software from identified developers.
- When first opening a model and installing a new extension for it through the extensions manager prompt, the model widgets would not be usable until the model's code was recompiled. Now the model will be ready to use right after the extension is installed.
- When using `in-cone` on wrapped worlds, occasionally turtles at the very edge of the wrapped cone would be missing from the results.
- When using `in-radius` on a patch to get nearby turtles, sometimes turtles at the edge of the radius would be missing from the results.
- The performance of a turtle lookup by breed and who-number, like `sheep 10`, could take an unreasonable amount of time.
- Models with a large amount of code (5000+ lines) would take a very long time to check and compile.

Logging Improvements

- Logging can now be enabled and a configuration file chosen through the NetLogo preferences menu. The logging standalone application is no longer necessary, so it has been removed.
- The command line switch to set the logging directory would previously be ignored.
- Users now get a chance to add a comment to the log when they get a compiler error.

NetLogo 3D Bugfixes

- The `in-radius` and `in-cone` primitives could give incorrect results, omitting some agents in rare cases.
- When hatching turtles the z-coordinates would sometimes not be copied from the parent turtle.
- The font size value for the world view would be ignored.
- When using `with` with patches, the results could include only patches with a zero z-coordinate.
- NetLogo 3D should run more reliably on Linux systems.

Extension Changes

- The Time extension is now bundled with NetLogo. This extension lets you work with real-world dates and times in your model, and includes an event scheduler for triggering events at points in time. It also includes an NLS library file you can add to your model to handle time series reading from CSV files. See **the Time extension docs** for more information.
- The View2.5D extension has been significantly upgraded. Turtle stem color can be set by a reporter. Links are now visible in the 2.5D window. They can be visualized either going from turtle to turtle (the default) or in the xy-plane. The choice can be made using radio buttons or through new commands
- LevelSpace now always runs hidden models at full speed and should give better error messages when errors occur when loading a model.
- The Network (NW) extension will not add extra whitespace when using `nw:save-matrix`.
- The Python extension gives better error messages when the paths to the Python executables are incorrect.

Documentation Changes

- Tutorial #1 referenced an older version of the Wolf Sheep Predation model.
- Added a couple more common problems to look out for in the BehaviorSpace guide.
- Clarify the version of NetLogo Web used to export models as HTML files.
- Clarify that the auto scale feature for plots does not scale the X axis when used with `histogram`.
- Clarify the location of the `NetLogo.cfg` files on each operating system.
- Update steps to take for common issues opening on macOS, added an alert about known issues to the download page.
- The docs for `ifelse` had some incorrect argument names.
- Add info about the Pioneer Award for the View 2.5D extension.

Model Changes

New Sample Models:

- Dislocation Motion and Deformation
- Repressilator 1D
- Sex Ratio Equilibrium

New Curricular Models:

- CT-STEM (for more information, visit ct-stem.northwestern.edu):
  - Free Fall
  - Habitat Preference Behavior
  - Hardy Weinberg Equilibrium
  - Kidney Absorption
  - Mendelian Inheritance
  - Natural Selection - Camouflage
  - Pendulum
  - 1D Motion Maps

New Code Examples:

- Extension Examples
  - `time` Extension - Discrete Event Mousetrap
  - `view2.5d` Extension - 2.5d Turtle View Example

Sample Model Updates:

- Anisogamy: INTERFACE re-design, expanded INFO tab, and refactored CODE. Now includes additional BehaviorSpace experiments.
- Autumn: Minor updates to INFO tab.
- Bug Hunt Speeds: Minor updates to INFO tab.
- Fire: Minor updates to INFO tab.
- Fur: Minor updates to INFO tab.
- Grand Canyon: Minor updates to INFO tab.
- Lightning: Minor updates to INFO tab.
- Moths: Minor updates to INFO tab.

- Particle System Basic: Minor updates to INFO tab.
- Paths: Minor updates to INFO tab.
- Shuffle Board: Minor updates to INFO tab.
- Small Worlds: The INTERFACE tab has been overhauled to make the model easier to use. The INFO tab has been rewritten to be more clear and concise. The CODE tab has been restyled and now uses curved link shapes for neighbors' neighbor links.
- Team Assembly: Minor updates to INFO tab.
- Tumor: Additional credits and references added.
- Wealth Distribution: Got rid of one line of CODE that didn't do anything.

Our `Chemistry & Physics` folder now contains enough examples that can be classified as "Materials Science" that they deserve their own folder. This means a number of models have been physically moved into this new sub-folder of Chemistry and Physics:

- Solid Diffusion
- Polymer Dynamics
- Lennard-Jones
- Dislocation Motion and Deformation
- Crystallization Models

Also in this folder are the MaterialSim models. They have been physically moved to the Curricular Models folder, but will be cross-referenced in `Chemistry & Physics/Materials Science`.

Alternative Visualization Updates:

- Heat Diffusion - Alternative Visualization: Minor INFO tab updates.

Curricular Model Updates:

- BEAGLE Evolution
  - Altruism: Minor INFO tab updates.
  - Cooperation: Minor INFO tab updates.
  - Divide the Cake: Promoted from Unverified after updates to CODE and INFO.
  - Guppy Spots HubNet: Promoted from Unverified after updates to CODE and INFO.
- Chemical Equilibrium: Fix plot and monitor bug that incorrectly counted pink molecules.
- EACH (now appears as a sub-directory of the "BEAGLE Evolution" directory)

Note, the above BEAGLE models have now been physically moved into the `Curricular Models/BEAGLE Evolution/` folder. In the Models Library browser, they will appear cross-referenced in the usual places.

Code Examples Updates:

- `view2.5d` Extension - 2.5d Patch View Example: Fixed issue with perspective not being updated on `setup`.
- `gogo` Extension - GoGoMonitor: Minor INFO tab updates.

HubNet Activities Updates:

All HubNet models now include a citation for the HubNet software and their web links have been updated. This resulted in a change to 37 models in total.

IABM Model Updates:

- Arduino Example: Minor INFO tab updates.
- Heroes and Cowards: Minor INFO tab updates.

- Simple Economy: Updated INFO tab and now listed as a Sample Model under the “Economics” folder.

Fixed inconsistent bracket styling across IABM Textbook models. This resulted in changes to 12 models.

- Turtles Circling Simple
- Life Simple
- Wolf Sheep Simple 1
- Wolf Sheep Simple 2
- Wolf Sheep Simple 3
- Wolf Sheep Simple 4
- Wolf Sheep Simple 5
- Agentset Efficiency
- Voting Analysis
- Simple Machine Learning
- Simple Viral Marketing

Version 6.1.1 (September 2019)

Bugfixes

- The `remove-duplicates` primitive will now operate properly on lists of agentsets, considering agentsets duplicates if they contain exactly the same agents.
- Many fixes to the `in-radius` primitive, which could report incorrect values when used with a radius that’s a large percentage of an unwrapped world width or cause runtime errors when world sizes changed or when the radius wrapped around the world back to itself.
- Corrected issues with code tab compile errors causing unexpected behavior or runtime errors in the interface tab, especially when a model had breeded links.
- You can now use the same `let` variable name in different command blocks inside a procedure.
- The `netlogo-headless.sh` scripts for macOS and Linux had issues with running the 3D command switch and properly wrapping the Java executable in quotes in case it contains spaces.

Features

- Runtime speed optimizations when using the `count` primitive on turtle or link agentsets with a comparison operator like `>` or `=`.

Extension Changes

- The Python extension on Windows will no longer give permissions errors when trying to create it’s config file, moved the file to a per-user directory.
- The Python extension will no longer produce a confusing deprecation warning when used with Python 3.7.
- Moved the storage location for the R extension config files to a per-user directory and updated documentation around getting it working. This issue mostly affected Windows users.
- The Mathematica-Link add-on now works correctly with NetLogo 6.1.0 and 6.1.1.

Documentation Changes

- Clarified the file location search order for extensions with the new extension manager.
- Standardized the language used in the conditional primitives (`if`, `ifelse`, `ifelse-value`) and the boolean operators (`and`, `not`, `or`, `xor`).
- Use correct arrow syntax in the `ifelse-value` code example.
- Added notes to `ifelse` and `ifelse-value` that the multi-conditional behavior was newly added in version 6.1.

Model Changes

New Sample Models:

- CRISPR Models
  1. CRISPR Bacterium
  2. CRISPR Bacterium LevelSpace
  3. CRISPR Ecosystem
  4. CRISPR Ecosystem LevelSpace
- Fruit Wars Model

Sample Model Updates:

- Ant Adaptation: Minor Info tab updates.

Curricular Model Updates:

- PNoM Models: Minor Info tab updates.

Version 6.1.0 (May 2019)

Feature Changes

The premier feature for this release is an **Extension Manager** for installing and updating extensions, which is available under the Tools menu. Using extensions with NetLogo could be a painful process: manually finding them, downloading them, and getting them correctly installed in the NetLogo application. The extensions manager addresses these issues by automating the installation of many 3rd-party extensions, and the updating of the bundled extensions between NetLogo releases. All done easily from within the application. In fact, if you open a model that requires an extension you don't have installed that it knows about, the Extension Manager will ask if you'd like to download and install it right on the spot, without any extra effort

We welcome external contributions to the extensions library! If you have developed a NetLogo extension and want to make it available to NetLogo users in the Extension Manager, see these instructions: https://github.com/NetLogo/NetLogo-Libraries#submitting-pull-requests

Other new features include:

- The primitives `ifelse` and `ifelse-value` now take a varying number of clauses, making selecting a single case from a variety of options much simpler.
- Added a new primitive, `up-to-n-of`, which gives as many items from a list as possible up to some limit, without causing an error if it cannot get all of them. Previously, to safely use `n-of` when you were unsure of the number of agents in an agentset would require some verbose code like `ask n-of (ifelse-value (count turtles-here < 10) [count turtles-here] [10]) turtles-here [ fd 10 ]`, but now you can simple write `ask up-to-n-of turtles-here [ fd 10 ]` to get the same effect.

- Added new hotkeys: Command-U for “Show Usage” of a term, Command-E for “Jump to Declaration” of a procedure, and Command-Option-S for “Save All” when using included NetLogo code files. Or Ctrl-U, Ctrl-E, Ctrl-Alt-S for Windows and Linux users.
- Multiline input widgets are now directly editable on the interface tab.
- Runtime optimizations for the `in-radius` and `in-cone` primitives.

Bugfixes

- When a model that has a compile-time error is saved, its sliders will not reset their values to 0 when it is reloaded.
- The procedures search dropdown on the Code tab gives better results with inexact search terms.
- Plot pen names with quotes in them are properly stored and loaded.
- Plot pen names are now case-insensitive, just like other items in NetLogo.
- When a plot is deleted and the action is undone, the plot continues to work afterward.
- 2D NetLogo models saved in NetLogo 3D can be opened back into 3D, or de-converted by re-opening in NetLogo 2D.
- Widgets can no longer be re-sized to be completely outside the interface tab.
- Code comments can be used as normal within an `__includes` keyword file list.
- The error message for getting the `mean` of an empty list should now make more sense.
- Double-clicking an empty experiment list in BehaviorSpace will not cause an error.
- BehaviorSpace experiments are better about warning when there are astronomical numbers of run combinations.
- Anonymous procedures can be given in the conditional reporter blocks of `ifelse-value`.
- Slight changes to how color values are calculated, for smoother interoperability with NetLogo Web.

Extension Changes

- The Python extension, `py`, is now bundled with NetLogo. If you have Python installed on your computer, you can use it from inside a NetLogo model.
- The Control Flow extension, `cf`, has been removed from NetLogo in favor of the new variadic `ifelse` and `ifelse-value` behavior. The `cf` extension is still available from the extensions library if needed for any existing models.
- The CSV extension, `csv`, had some minor documentation updates.
- The GoGo board extension, `gogo`, should work more reliably on Windows 10.
- The Network extension, `nw`, now loads data in a predictable order when imported, and it correctly handles preferential attachment.
- The Table extension, `table`, had some minor documentation updates.
- The View 2.5D extension, `view2.5d`, now does not throw an error when trying to use a turtle-based view.

Documentation Changes

- Added documentation around using primitives like `mean` and `variance` along with agentsets producing slightly different results due to floating point arithmetic.
- Add an extension authoring introduction page with links to the technical documentation.
- Clarify what `clear-ticks` is meant to do.
- Clarify that `clear-globals` doesn’t affect interface globals.
- Fixed various formatting errors and typos.

Model Changes

New Sample Models:

- Ant Adaptation
- Braess' Paradox

New Code Examples:

- `py` Extension
  - Python Basic Example
  - Python Flocking Clusters

New Curricular models:

- PNoM Curriculum
  - PNoM 1 Diffusion Sandbox
  - PNoM 2 Diffusion Sensor Sandbox
  - PNoM 3 Virtual Syringe
  - PNoM 4 Virtual Syringe Force Graph
  - PNoM 5 Virtual Syringe Temperature Graph
  - PNoM 6 Particle Sandbox Gravity
  - PNoM 7 Particle Sandbox Intermolecular

Sample Model Updates:

- Acids & Bases Models: Dissociations now result in random headings for resulting particles.
- Mandelbrot: Exposed `max-iterations` parameter and updated Info Tab.
- Peppered Moths: Increase size of `moth` agents.
- Perceptron: Fix weight visualization bug.
- Raindrops3D: Remove unused procedure.
- Reactor X-Section: Fix typo and extend manual rod depth.
- Rock Paper Scissors: Minor Info tab updates.
- Sandpile: Minor Info tab updates.
- Simple Kinetics 1,2: Dissociations now result in random headings for resulting particles.
- SmoothLife: Allows NLW compatibility by skipping some plot drawing.
- Tumor: Prevent `ask nobody` bug when using
- Wolf Sheep Predation: Fix typos in Code tab.

IABM Model Updates:

- Segregation Simple Models (Simple & Extensions 1 - 3): Reworded comment in Code tab.

Code Example Updates:

- LS Parent Example: Change button display name to avoid confusion with `setup` button.
- Mouse Drag Multiple Example: The model has been rewritten to be NLW compatible.
- Mouse Drag One Example: The model has been updated to be NLW compatible

We've introduced a new subfolder in the `Sample Models/Social Science` folder: "Economics". With this new folder, several models have been moved. The following models are now in the `Social Science/Economics` folder:

- Bidding Market
- Hotelling's Law

- Oil Cartel HubNet
- Sugarscape
- Bank Reserves
- Cash Flow
- Wealth Distribution

With the introduction of the new variadic `ifelse` and `ifelse-value` primitives, the Models Library has been updated to reflect this new functionality. These updates are divided into three major sections:

- Add parens to `ifelse-value` expressions that are arguments of infix operator.
- Remove unnecessary parentheses from `ifelse-value` invocations for models that use `ifelse-value` with a simple infix conditional: `>`, `<`, `<=`, `>=`, and `=`. This results in changes to the following models.
- Remove now unnecessary parens from calls to `ifelse-value` with more complex conditionals.

And here are the models affected by these changes:

- Code Examples:
  - Beatbox
  - HSB and RGB Example
  - NW General Examples
- Curricular Models:
  - BEAGLE: DNA Protein Synthesis, DNA Replication Fork
  - GenEvo 1 Genetic Switch
  - Lattice Land - Explore
  - ModelSim: Bacteria Food Hunt
  - NIELS: Current in a Wire HubNet
  - ProbLab: 4 Blocks, 4 Block Stalagmites, 4 Block Two Stalagmites, Histo Blocks
- HubNet Activities:
  - Minority Game HubNet
  - Predator Prey Game HubNet
  - Prisoners Dilemma HubNet
  - Root Beer Game HubNet
- IABM Textbook:
  - Traffic Basic Adaptive Individuals
- Sample Models:
  - Artificial Neural Net - Multilayer
  - Artificial Neural Net - Perceptron
  - Bidding Market
  - Blood Sugar Regulation
  - DLA Alternate Linear
  - DLA Alternate
  - Echo
  - Hotelling's Law
  - Language Change
  - Minority Game
  - Robby the Robot
  - Signaling Game

- Simple Genetic Algorithm
- Tabonuco Yagrumo
- Tabonuco Yagrumo Hybrid
- Traffic 2 Lanes
- Vision Evolution

Version 6.0.4 (June 2018)

Feature Changes

- NetLogo now supports the Portuguese language.
- BehaviorSpace allows optionally disabling plots and the view when running experiments.

Bugfixes

- Updated the bundled Java 8, which should resolve an issue with NetLogo failing to start on Windows computers.
- Fixed a bug with the nw extension preferential attachment generation and initialization.
- Fixed a numerical error with the `range` primitive.
- Fixed a bug with link lookup when using more than two link breeds.
- Comment and uncomment code hotkeys should work with NetLogo include source files.
- Added a better error message when a web browser couldn't be launched for any reason.

Documentation Changes

- Document that `resize-world` only kills all turtles when coordinates really change.

Model Changes

New Curricular Models

- Mind the Gap Curriculum
  - MTG 1 Equal Opportunities HubNet
  - MTG 2 Random Assignment HubNet
  - MTG 3 Feedback Loop HubNet

Revised Sample Models

- Blood Sugar Regulation: Updated statistical sampling procedures
- Flocking: Changed default `vision` to `5.0` patches
- Rumor Mill: Fixed a minor bug with the spread statistics in early ticks

Revised Code Examples

- Flocking (Perspective Demo): Changed default `vision` to `5.0` patches

Revised Curricular Models

- GenEvo - GenEvo 1: Fixed `save screenshot` bug on Windows machines
- GenEvo - GenEvo 2: Fixed `save screenshot` bug on Windows machines
- GenEvo - GenEvo 3: Fixed `save screenshot` bug on Windows machines

Revised Alternative Visualizations

- Flocking (Alternative Visualizations): Changed default `vision` to `5.0` patches

Version 6.0.3 (March 2018)

Feature Changes

- Improve the look & feel of Agent Monitor windows.
- The procedures menu in the code tab now has a search field and actively filters procedure names as the user types. This should improve navigation speed for large models with lots of procedures. There is also now a hotkey to access this menu: Ctrl-G on Windows and Linux, and Cmd-G on MacOS.
- Increase the speed of the `diffusion` primitive and make the diffusion operation symmetric.
- NetLogo enforces that plot names are unique and that pen names are per-plot unique without taking capitalization into account. This avoids certain cases where `set-current-plot` and `set-current-plot-pen` primitives would set the plot or pen to something different than the user intended because names differed in capitalization only.
- Models with existing plots and pens which have the same name without taking capitalization into account will be automatically converted on first open in 6.0.3.
- We are planning on adding the ability to have multiple conditional branches to `ifelse` and `ifelse-value`. However, as this is a fairly substantial change to the core language, we have implemented it in the cf extension for testing.

Documentation Changes

- Add a table to the Output section of the Programming Guide specifying how the various output primitives differ in whether they print the calling agent, whether they print a newline, and whether they can be read back into NetLogo.

Bugfixes

- Anonymous procedures appearing inside `ask` blocks inside `repeat` blocks now close over unique copies of let-variables introduced in the `repeat` block.
- Fix a longstanding HubNet bug in which HubNet would sometimes crash as a large number of users joined.
- Fix HubNet discovery when the HubNet server is run on a Mac.
- When switching from the code tab to the interface tab, the cursor switches to the command center. This prevents typed commands or copy/paste keyboard shortcuts from affecting the code tab.
- Turtles draw trails when moving backwards (regression in NetLogo 6.0.1).
- Smooth Code Tab scrolling.
- BehaviorSpace now only applies standard CSV string escaping (doubling of double quotes) to string outputs in result files. It used to apply NetLogo style string escaping (backslashes in front of special characters) and then apply CSV escaping on top of that. The new behavior is consistent with the RFC 4180 specification and the current behavior of the `csv` extension.
- The Behaviorspace dialog appears centered in the NetLogo window.
- Due to changes introduced in Java, certain dialogs in NetLogo 6.0.1 would appear under the dialog that created them instead of on top. These dialogs now appear on top of the dialog that created them.

- Non-code changes to the System Dynamics modeler are pulled in on every recompile (prior to this, changes like newly-disconnected flows were sometimes not taken into account on recompiles).
- The system dynamics code tab adjusts itself to match the size of the system dynamics window.
- NetLogo 3D loads and saves system dynamics.
- Fixes an error raised in 3D when the world was cleared following setting a patch-color to an RGB list.
- Miscellaneous small changes to the format of exported worlds. NetLogo 6.0.3 should be able to import worlds exported by older versions of NetLogo and older versions of NetLogo should be able to import worlds exported by NetLogo 6.0.3.
- The "others" execute bit is set for the netlogo-headless.sh script in the Linux installer package.

Extension Changes

- `nw:generate-preferential-attachment` accepts a `min-degree` parameter, specifying the number of links each new turtle preferentially creates to existing turtles.
- The `arduino` extension (and the related sketch and model in the library) have been updated to allow the arduino to send strings back to NetLogo. Two new debug primitives have also been added to help troubleshoot interaction between NetLogo and the arduino.
- The verbose `cf` extension conditional syntax as been replaced with straight-forward generalizations of the `ifelse` and `ifelse-value` primitives. The syntax is much simpler and these primitives run much faster than the old constructs.
- The `ls` extension has much less overhead for calling out to single models, making constructs like `ask turtles [ ls:ask my-model [ run-something ] ]` much faster.
- The `ls` extension now seeds the child model RNG based on the parent model RNG at model creation. Furthermore, and `ls:random-seed` primitive has been introduced, so an entire model system can be seeded simultaneously. This makes it possible to have reproducible runs when using LevelSpace.
- The `ls:let` primitive now handles scoping properly.
- `ls:assign` has been added to LevelSpace to make assigning globals of child models easier.

Documentation Changes

- Clarify documentation around `__includes` by providing examples.
- Fix a broken link in the dictionary.
- Update the Windows unattended installation directions in the FAQ.

Model Changes

New Sample Models:

- Bidding Market
- Blood Sugar Regulation
- Rock Paper Scissors
- Vision Evolution

New Curricular Models

- NIELS - Current in a Wire HubNet

New HubNet Activities:

- Current in a Wire HubNet (same as above)

Revised Code Examples:

- `arduino` Extension: The example model and sketch have been updated to reflect new features in the extension
- `sound` Extension – Sound Workbench: Minor GUI updates
- `sound` Extension – Percussion Workbench: Minor GUI updates
- File Input Example: Fixed a bug in the `save-patch-data` template procedure
- Lottery Example: Corrected *Related Models* from Preferential Attachment to the "Rnd" Example Model in the Info Tab
- Info Tab Example: Corrected reference from `pegdown` to `flexmark` in the Info Tab

Revised Curricular Models:

- BEAGLE Evolution - Fish Tank Genetic Drift: Update duplicate plot-pen names and resize plots
- GenEvo - GenEvo 4: Replaced usage of `cf:when` with nested `ifelse` s
- Lattice Land - Lattice Land Explore: Fixed bug when deleting a segment that didn't exist.
- Lattice Land - Lattice Land Triangles Dissection: Fixed bug when deleting a segment that didn't exist.
- NIELS - Electrostatics: All turtles are now `particles` and each `particle` has a charge; Changed particle coloring; Updated Info Tab and some variable names to reflect later NIELS models
- Urban Suite - Tijuana Bordertowns: Minor Info Tab updates.

Revised Sample Models:

- Artificial Anasazi: Update broken URLs in Info Tab
- Autumn: Update broken URLs in Info Tab
- HIV: The "AIDS" model has been renamed to "HIV"†
- Color Fractions: Removed direct reference to dead web page in Info Tab
- Fairy Circles: Update URLs in Info Tab
- Fur: Update broken URLs in Info Tab
- Hex Cell Aggregation: Update broken URLs in Info Tab
- Ising: Update broken URLs in Info Tab
- Kicked Rotators: Update broken URLs in Info Tab
- Language Change: Update URLs in Info Tab
- Lennard-Jones: Added citation in the Info tab
- Lightning: Update URLs in Info Tab.
- L-System Fractals: Update broken URLs in Info Tab
- Membrane Formation: Update broken URLs in Info Tab
- Moth: Update URLs in Info Tab
- Party: Fix incorrect comment on turtle headings.
- PD Basic: Minor GUI updates
- Segregation: Update broken URLs in Info Tab. Update color scheme for color blindness. Minor GUI updates. Minor Info Tab updates.
- Sugarscape 1 Immediate Growback: Fixed "Wealth distribution" histogram bug (`turtle` with max `sugar` was not displayed)
- Sugarscape 2 Constant Growback: Fixed "Wealth distribution" histogram bug (`turtle` with max `sugar` was not displayed)
- Sugarscape 3 Wealth Distribution: Fixed "Wealth distribution" histogram bug (`turtle` with max `sugar` was not displayed)

Revised IABM Models:

- Arduino Example: The example model and sketch have been updated to reflect new features in the extension (same as above)
- Segregation Simple Extension 1: Update broken URLs in Info Tab
- Segregation Simple Extension 2: Update broken URLs in Info Tab
- Segregation Simple Extension 3: Update broken URLs in Info Tab
- Segregation Simple: Update broken URLs in Info Tab

† - Models Updated to Correctly Reference the HIV Model

- Alternative Visualizations
  - Virus - Alternative Visualization
  - Virus - Circle Visualization
- Curricular Models
  - epiDEM Basic
  - epiDEM Travel and Control
- IABM Models
  - Spread of Disease
- Sample Models:
  - Disease Solo
  - Virus
  - Rumor Mill

Version 6.0.2 (August 2017)

Feature Changes

- The autosuggest functionality introduced in NetLogo 6 will now display extension primitive names.
- A new `insert-item` primitive was added to the language. It works similarly to `replace-item`, but without removing the item at the specified index.
- A new `set-plot-background-color` primitive was added to the language. It can be used in plotting code to set the background color of the plot.
- Behaviorsearch's user interface has been updated. It now uses the JavaFX UI toolkit for a more modern look and feel.

Bugfixes

- The 6.0.1 autoconverter would error when converting tasks with exactly one argument (from 5.3.1 and earlier models only). The conversion would be correct, but it was confusing. This has now been remedied and the autoconverter will correctly convert models from 5.3.1 and earlier without erroring on single-argument tasks.
- Fix a bug where extensions weren't located properly when in a path relative to the model.
- Help browser now opens in 64-bit Windows.
- Fix a speed regression for the `import-world` primitive introduced in 6.0.
- If an unexpected reporter block is passed to `with` when used with `other`, the error will now reflect that the block is unsuitable for `with` instead of `other`.
- When `sort-by` receives an unsuitable reporter block from the user it now shows a useful error instead of a cryptic stack trace.
- String representations of anonymous procedures now show the arguments of those anonymous procedures

- When working in an nls file, switching to a different tab will compile the nls file.
- Display error label on nls file when the code it contains causes an error.
- Some users were unable to open NetLogo 6 and 6.0.1 in Mac OS Sierra. We've changed our signing process in 6.0.2 to attempt to fix this problem. We are continuing to track this issue to determine whether our fix was effective. More information (including a partial workaround for the problem) is available here.

Extension Changes

- Fixed a bug in `gis:patch-dataset` introduced in NetLogo 6.
- The table extension has new `table:group-agents` and `table:group-items` primitives which can be used to build a table of grouped agents and items by supplying an anonymous reporter.

Documentation Changes

- Clarify when various parts of the anonymous procedure syntax are optional.
- Fix a few small documentation bugs around `foreach`.
- Clarify relationship between speed and update modality in the "View Updates" section of the programming guide.

Model Changes

New Sample Models:

- Chaos in a Box
- Fairy Circles
- SmoothLife

New Curricular Models:

- Lattice Land Curriculum
  - Lattice Land - Triangles Explore
  - Lattice Land - Triangles Dissection

Revised Sample Models:

- Anisogamy: New color scheme. Adds BehaviorSpace experiments.
- Ethnocentrism: Now cross-listed as an Evolutionary model.
- Fireworks: Rename `fireworks` to `max-fireworks` and enforce a minimum of 1.
- GenJam - Duple: Minor info tab updates.
- Wolf Sheep Predation: Revised to better clarify the `grass?` construct and code.

Revised Curricular Models:

- Connected Chemistry:
  - Connected Chemistry 1: Fix NetLogo Web incompatibility.
  - Connected Chemistry 3: Update minimum number of particles to 2.
  - Connected Chemistry 8: Minor code updates. Small change to default values.

Revised HubNet Activities:

- Gridlock HubNet: Minor code updates.

Revised Code Examples:

- Info Tab Example: Corrected formatting of un/ordered lists example.
- LS Extension — Model Visualizer and Plotter Example: Updated code dependencies for WSP.
- LS Extension — Model Interactions Example: Updated code dependencies for WSP.

Version 6.0.1 (March 2017)

Feature Changes

- Brackets are required around anonymous procedure reporters only when there are two or more arguments. For instance `[ [x] -> x ]` can now be written `[ x -> x ]`. Note this change makes it possible to create models in NetLogo 6.0.1 that will not run in NetLogo 6.0. If you plan to use your model in NetLogo 6.0, be sure to include brackets around anonymous procedure arguments. For a short period after the release, models which use unbracketed lambda arguments may not work on netlogoweb.org.

Bugfixes

- The NetLogo code editor navigates and indents models much more quickly and efficiently than in NetLogo 6.0.
- The NetLogo 5-to-6 autoconverter now ignores commented-out code.
- The behavior of layout-radial in NetLogo 6 did not match the 5.3.1 behavior. This has been corrected and layout-radial should be identical between NetLogo 6.0.1 and NetLogo 5.3.1
- NetLogo 6 raised a NullPointerException when numbers became too large for NetLogo to handle. This is now properly displayed to the user as a number out of bounds error.
- Improved performance of models which use `let` and anonymous procedures together.
- Reloading a model now clears global variables.
- Clearer warning dialogs when opening an older version of a NetLogo 2D file in NetLogo 3D.
- Using `foreach` improperly in the Command Center will display the same error as it would if used improperly in the code tab.
- Extremely long anonymous procedures no longer cause an exception when compiled.
- `in-radius` is no longer pathologically slow on agentsets created using `with`.

Extension Changes

- A new primitive, `table:values` has been added to the table extension.
- The R extension has been updated to take full advantage of the JavaGD R library using `r:setPlotDevice`.
- Users can now supply a path to the R extension by configuring the value of r.lib.paths in the user.properties file.
- Fixed a LevelSpace bug preventing interactive models from using nls files.
- Fixed a LevelSpace bug causing NetLogo to lock up when trying to load a nonexistent file as an interactive model.

Documentation Changes

- The system dynamics tutorial now instructs the user to set the `dt` to a value which gives stable behavior
- The documentation for `follow`, `ride`, and `watch` has been clarified to indicate that calling one undoes highlights and perspective changes caused by the other
- The documentation for `=` and `!=` indicates that they work with extension objects.

- Several examples have been added to `sort` clarifying the behavior of `sort` on lists featuring different types of objects.
- A new section on User Interface primitives has been added to the programming guide which discusses the behavior of the "Halt" button in the various user interaction dialogs.
- Incorrect example code for `foreach` and `reduce` has been corrected.

Models

All models have been updated to reflect the new optional nature of brackets for zero/one argument anonymous procedures. This resulted in changes to 108 models.

All models have been updated to reflect the availability of the new range primitive. This resulted in changes to 12 models.

New Sample Model

- GenJam - Duple

New Curricular Model

- Lattice Land curriculum:
  - Lattice Land Explore

Revised Sample Models

- PD 2 Person Iterated: info tab updates and extensive fixes for the code.
- Party: fixed bug where a monitor covered a plot.
- Signaling Game: info tab updates.

Revised Curricular Models

- GenEvo curriculum:
  - GenEvo 1 Genetic Switch: info tab updates and new graduated method of displaying lacZ concentration.
  - GenEvo 2 Genetic Drift: info tab updates and interface tweaks. New model preview.
  - GenEvo 3 Genetic Drift and Natural Selection: info tab updates and interface tweaks.
  - GenEvo 4 Competition: info tab updates.
  - Genetic Switch - Synthetic Biology has been renamed and is now Synthetic Biology - Genetic Switch. It also received info tab updates and a few interface changes.

Version 6.0 (December 2016)

Feature Changes

- The NetLogo code editor now offers autocompletion support. Simply press the Control key and the spacebar at the same time while typing a word and you will see a list of similar NetLogo primitives as suggestions.
- The NetLogo code editor offers the option to "fold" procedures to make navigating large models simpler.
- NetLogo supports multi-level agent-based modeling with the LevelSpace extension
- Line numbering can be enabled in the NetLogo code editor by choosing "Show Line Numbers" from the "Preferences" dialog. This dialog can be opened by selecting "NetLogo" > "Preferences" in Mac, or "Tools" > "Preferences" on Linux or Windows.
- The view resizing arrows have been removed and the tick counter has been relocated under the speed slider.
- When editing NetLogo code, users can right-click a variable name or primitive and choose "Show Usage" to see all usages of that name in the file.

- When editing NetLogo code, users can right-click a variable name and choose “Jump to Declaration” to see where in the file that variable is declared.
- The NetLogo interface editor now supports “Undo” for widget addition, deletion, and movement.
- NetLogo can export code to HTML with code-colorization by choosing “Export Code” in the “Export” section of the “File” menu.
- The look and feel of NetLogo on Mac OS X has changed significantly. NetLogo is now using the Oracle-supplied Java look and feel as opposed to a third-party look and feel used in prior versions.
- Plots use a random number generator independent of the main-model random number generator.
- Model Preview Commands can be edited through the GUI by choosing “Preview Commands Editor” in the “Tools” menu. Those commands generate the preview image that appears when you upload your model to the Modeling Commons.
- NetLogo displays a more helpful error message when a program fails due to an “out of memory” error.
- NetLogo and its bundled extensions are now compiled against Java 8 and Scala 2.12.
- NetLogo has upgraded the ASM bytecode library to enable generation of Java 8 bytecode.

Bugfixes

- `*-link-neighbor?` primitives now work the same way for breeded and unbreeded links.
- Resizing the world in NetLogo 3D no longer causes a black view.
- Fixed error caused by right-clicking a widget while dragging.
- Improved error message when a user-defined procedure shadows a breed procedure.
- `neighbors4` and `neighbors` no longer report agentsets containing the same patch more than once.

Language Changes

- Support for plural-only breed names (e.g., `breed [mice]`) has been removed. Write `breed [mice mouse]` instead.
- Tasks have been replaced by anonymous procedures. Tasks made use of `?` variables which were confusing for novices and difficult to read for experts. Additionally, tasks could not refer to the task variables of a containing task. Anonymous procedures may have named arguments which can be accessed by inner anonymous tasks. Upon first opening a NetLogo 5 model in NetLogo 6, tasks like `task [ ?1 + ?2 ]` will be automatically converted to `[ [?1 ?2] -> ?1 + ?2 ]`. See the the transition guide for more information.
- Link reporters have been overhauled to be more consistent and flexible
- The `task` primitive is no longer supported.
- Breed names that conflict with language primitives are now disallowed. For instance, `breed [strings string]` is now disallowed since it makes `is-string?` ambiguous.
- The compiler errors on duplicated breed singular names.
- The compiler detects a greater number of type errors, for instance `not pxcor` now raises a compiler error instead of erroring at runtime.
- `set-plot-pen-color` now accepts RGB lists as arguments.
- The `hubnet-set-client-interface` primitive has been removed.
- The various primitives starting with `movie-` have been removed, as has the movie encoder. They have been replaced with the new `vid` extension. The transition guide provides more details and information.
- The `__change-language` primitive has been removed. You can now change the User Interface Language through the preferences dialog, which can be found by choosing “Preferences...” in the “NetLogo” menu (Mac OS X) or in the “Tools” menu (all other platforms).
- The string representation of anonymous procedures displays the body of the anonymous procedure.

Extension Changes

- NetLogo 6.0 comes with three new, bundled extensions: LevelSpace `ls`, the `vid` extension for video manipulation, and the enhanced visualization extension `view2.5d`.
- `ls` enables multi-level agent-based modeling in NetLogo.
- The Extensions API has been updated from 5.0 to 6.0. This means that all non-bundled extensions will need to be updated to use the new API. Extensions written for NetLogo 5 *will not* work in NetLogo 6. If you're an extension author, see the extension author transition guide for 6.0 for more information. If you regularly use extensions you may want to contact their authors to inform them a new version of NetLogo is on the way and they may want to update their extensions.
- `arduino:get` (in the arduino extension) correctly reports values from Windows 64-bit machines. In prior versions it would only report some values correctly.
- Several new features have been added to the `nw` extension:
  - Added community detection using the Louvain method
  - Added modularity measurement
  - Added Watts-Strogatz small-world network generation
  - Made other network generation algorithms easier to layout
  - Weighted primitives now take symbolic variable names instead of strings. See the transition guide for more information.
- The `qtj` extension is no longer bundled with NetLogo. Users are encouraged to make use of the new `vid` extension.
- The new `vid` extension is now bundled with NetLogo, combining features of the late `qtj` extension and the movie primitives.
- The network extension is no longer bundled with NetLogo. Users are encouraged to use the `nw` extension (nw), which has been bundled with NetLogo for some time.
- The `gogo-serial` extension is no longer bundled with NetLogo. Users are encouraged to transition to the newer `gogo` extension, which uses HID to communicate with the GoGo board.

Operating System Support

- NetLogo will now be used to open `.nlogo` links in PowerPoint and other programs on Microsoft Windows.
- The NetLogo binaries on Mac OS X are installed as runnable by any user.
- NetLogo in Mac OS X will not use "App Nap" while running. This keeps simulations running at full speed when NetLogo is in the background.
- The NetLogo controlling API has changed since NetLogo 5. Programs that rely on the controlling API (such as BehaviorSearch) will not work until they have been changed to match the new API.

Documentation Changes

- New documentation for anonymous procedures
- The NetLogo tutorial screenshots have been updated to correspond to the new Mac OS X Look and Feel.
- The NetLogo dictionary displays the version in which each primitive was introduced next to that primitive.
- A Spanish translation of the NetLogo dictionary is available here.

Internationalization Changes

- A new Spanish translation of the NetLogo dictionary is available as part of the NetLogo manual.

- Language preferences can be changed by using the new “Preferences” menu instead of the `__change-language` primitive.
- A Japanese localization for NetLogo is now available and included with the standard download.
- The Chinese translation for NetLogo has been updated.

Models

New Sample Models:

- Kicked Rotator
- Kicked Rotators
- Mammoths, a legacy StarLogoT model, has been converted to NetLogo.

New Curricular Models:

- GenEvo 1 Genetic Switch
- GenEvo 2 Genetic Drift
- GenEvo 3 Genetic Drift and Natural Selection
- GenEvo 4 Competition

New Code Examples:

- Movie Playing Example (`vid` extension)
- Movie Recording Example (`vid` extension)
- Video Camera Example (`vid` extension)
- Network Extension General Demo (`nw` extension)
- Model Interactions Example (`ls` extension)
- Model Loader Example (`ls` extension)
- Model Visualizer and Plotter Example (`ls` extension)

Promoted Models (improved and no longer “unverified”):

- ProbLab Genetics
- Traffic 2 Lanes

Revised Sample Models:

- Giant Component: added text in the info tab.
- Team Assembly: removed unused switch widget.
- Traffic Basic, Traffic Grid, Traffic Intersection: revised info tab.
- Voting: stopped the model when voting stabilizes.
- Wealth Distribution: fixed typos in info tab, improved code formatting.

Revised HubNet Activities:

- Bug Hunters Competition HubNet, Critter Designers HubNet, Fish Spotters HubNet: removed unnecessary call to `hubnet-set-client-interface`.
- Gridlock HubNet, Gridlock Alternate HubNet: revised info tab.

Revised Curricular Models:

- DNA Replication Fork: fixed a a monitor and runtime error when using a time limit.

Revised IABM models:

- Agentset Efficiency: clarified description of `go-2` in info tab.
- Agentset Ordering: fixed typos in into tab.
- Arduino Example: improved model to demonstrate both directions of communication with the Arduino.
- Heroes and Cowards: removed extra text in info tab.
- Preferential Attachment Simple: removed extra pen in plot and extra "layout" button.
- Random Network: made sure that the number of links is never too big for the number of nodes.
- Segregation Simple Extension 1, 2 and 3: fixed `number-of-ethnicities` slider to avoid runtime errors.
- Traffic Basic Adaptive Individuals, Traffic Basic Utility, Traffic Grid Goal: revised info tab.
- Traffic Basic Adaptive: revised info tab, clarified comment in `adaptive-go` procedure.
- Voting Component Verification: stopped the model when voting stabilizes.
- Voting Sensitivity Analysis: improved code for stopping the model when voting stabilizes.

Revised Code Examples:

- GoGoMonitorSerial, GoGoMonitorSimpleSerial: removed the models, as the `gogo-serial` and `qtj` extensions are no longer bundled.
- Random Network Example: made sure that the number of links is never too big for the number of nodes.
- Since the QuickTime extension (`qtj`) has been replaced by the `vid` extension, the following models have been converted to use the `vid` extension:
  - Movie Example, replaced by Movie Recording Example
  - QuickTime Movie Example, replaced by Movie Playing Example
  - QuickTime Camera Example, replaced by Video Camera Example

Demoted model:

- El Farol Network Congestion, a previously "unverified" model, has been moved to the NetLogo User Community Models

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. NetLogo doesn't run on mobile devices or Chromebooks. For the web, check out NetLogo Web. For mobile devices and Chromebooks, check out Turtle Universe.

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

Application Requirements

Windows

NetLogo runs on Windows 11, 10, 8, 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 17 for NetLogo's private use only. Other programs on your computer are not affected. NetLogo can run using a different Java installed on your system of version 11 or higher.

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 17 runtime for NetLogo's private use only. Other programs on your computer will not be affected. NetLogo can run using a different Java installed on your system of version 11 or higher.

Linux

NetLogo should work on standard Debian-based and Red Hat-based Linux distributions. The NetLogo tarball includes a copy of the Java 17 runtime. NetLogo can run using a different Java installed on your system of version 11 or higher.

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 64-bit version of NetLogo is the simplest way to a working NetLogo installation. Most desktop and laptop computers made since 2005 are 64-bit.

If you have a computer that was made before 2006 you may need to use the 32-bit version of NetLogo instead. 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.

Contact Information

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

Web site

Our website is https://www.netlogo.org/. NetLogo is created at the Center for Connected Learning (CCL). The CCL website, http://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, ask questions on the NetLogo Forum or try this group: http://groups.google.com/d/forum/netlogo-users.

If you have feedback, suggestions, or questions, you may write us at netlogo@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.

You can also report bugs on our forum at: https://forum.netlogo.org/c/bugs/5

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.
- 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 netlogo@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 https://groups.google.com/group/netlogo-devel.

Sample Model

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.

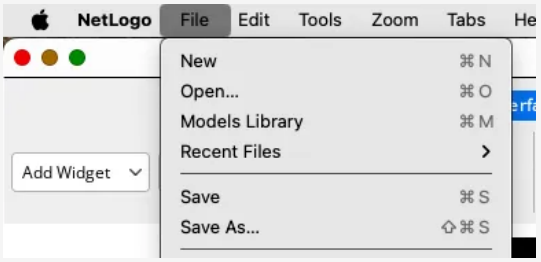
Is there any type of pattern to this kind of grouping?

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.

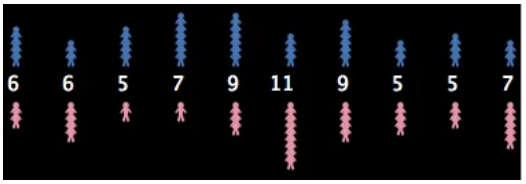
What to do:

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.

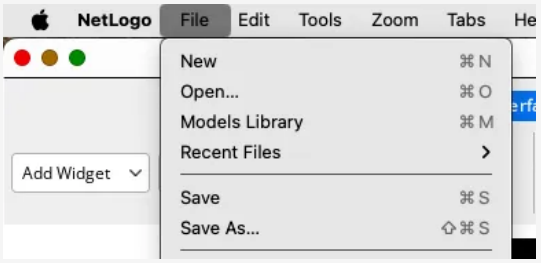
Throughout all of the tutorials, we'll be asking you to make predictions about what the effects of making changes to the models will be. Keep in mind that the effects are often surprising. We think these surprises are exciting and provide excellent opportunities for learning.

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 dark blue; they set up, start, and stop the model. Sliders and switches are light blue; they alter model settings. Monitors and plots are gray; 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.

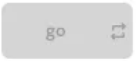
As the model is running, what is happening to the wolf and sheep | populations?

- Press the "go" button to stop the model.

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

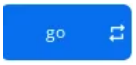
two arrows in the bottom right corners, like this:



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. You may have noticed that the "go" button was gray when you first opened the model. That is because it is inactive until the "setup" button is pressed. Pressing an inactive button will have no effect.



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.

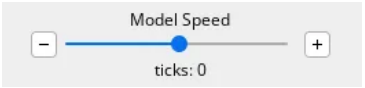
Stopping a forever button is the normal way to pause or stop a model. After pausing you can make it resume by pressing the button again. (You can also stop a model with the "Halt" item on the Tools menu, but you should only do this if the model is stuck for some reason. Using "Halt" may interrupt the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



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.

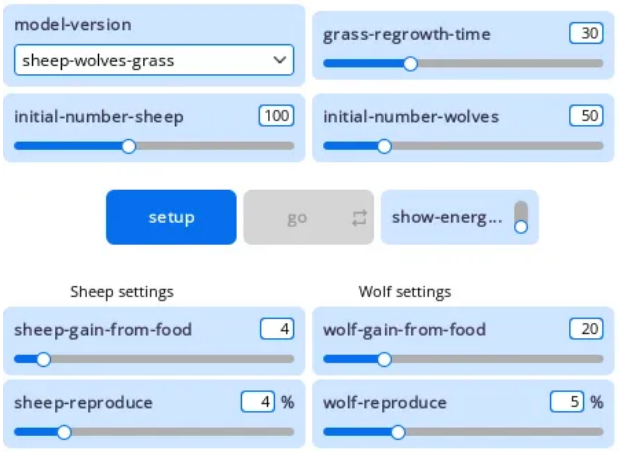
When you move the speed slider to the right the model speeds up. NetLogo will start skipping frames, that is, it won't update the view at the end of every tick, only some ticks. Updating takes time, so fewer view updates means the model progresses faster.

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: Choosers, 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.

Choosers, switches, and sliders give you access to a model's settings. Here are the choosers, 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.

- Change the "model-version" chooser from "sheep-wolves" to "sheep-wolves-grass".
- Press "setup" and "go" and let the model run for a similar amount of time as before.

What did the chooser do? Was the outcome the same as your previous run?

Changing the "model-version" dropdown to "sheep-wolves-grass" affected the outcome of the model. With the chooser set to "sheep-wolves", 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 choosers, a model may also have sliders. While choosers have a short list of preset options, 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?

- Change the "model-version" chooser from "sheep-wolves-grass" to "sheep-wolves".
- 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 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?

One more type of widget is called a switch. A switch has only two values, on and off.

There is a switch in this model called “show-energy?”

- Click “show-energy?” to check the box next to it and then click “setup” and “go”.

How did the appearance of the wolves in the world view change?

- Uncheck the “show-energy?” switch while the model is still running.

What happened to the energy labels you saw before?

When you open a model, all the choosers, 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 choosers, sliders, and switches, some models have input boxes. The Wolf Sheep Predation doesn’t have any of those, 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:

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?***

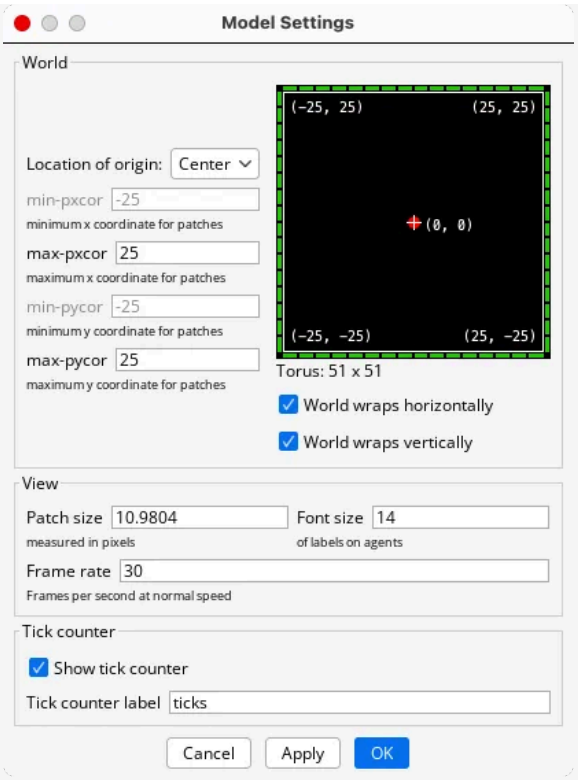
Fast forwarding the model and turning off view updates are useful if you're impatient and want a model to run faster. Fast forwarding (moving the speed slider to the right) drops view updates so the model can run fast, since updating the view takes time that could be used for running the model itself.

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.

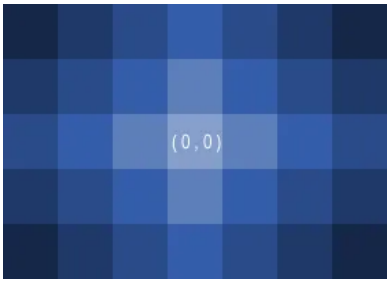
- Drag one of the square black “handles”. The handles are found on the edges and at the corners of the view.
- Unselect the view by clicking anywhere in the white background of the Interface tab.
- Press the “Settings...” button again and look at the settings.

What numbers changed?

What numbers didn’t change?

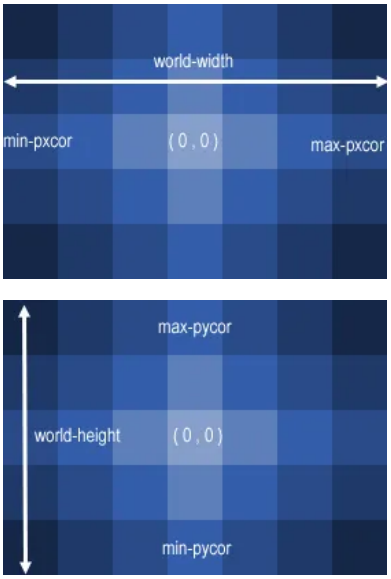
The NetLogo world is a two dimensional grid of “patches”. Patches are the individual squares in the grid. In Wolf Sheep Predation, when the “model-version” is set to “sheep-wolves-grass” the individual patches are easily seen, because some are green, others brown.

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`.

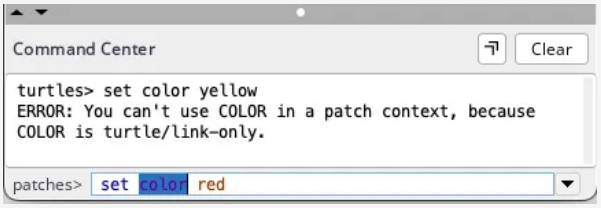
What is the difference between color and pcolor?

- 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:



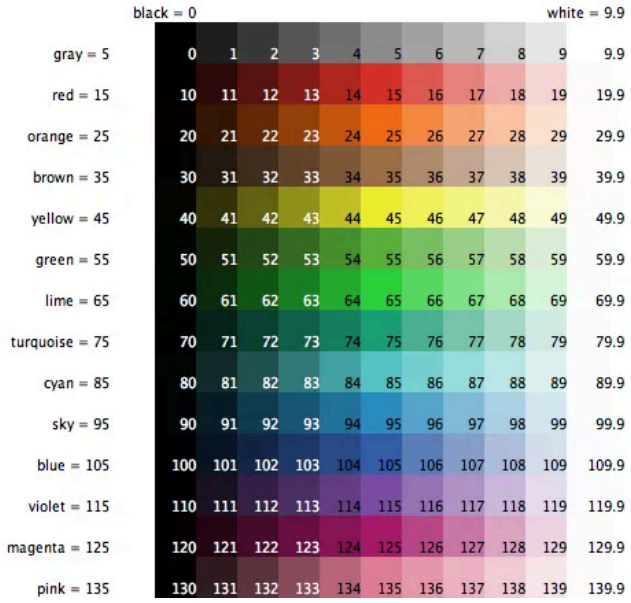
- Type `set pcolor red` instead and press return.

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:



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. NetLogo allows you to use the whole spectrum of For information on the different ways colors are represented in NetLogo see the Colors section of the NetLogo Programming Guide. The Color Picker Guide explains how to choose colors using a dialog.

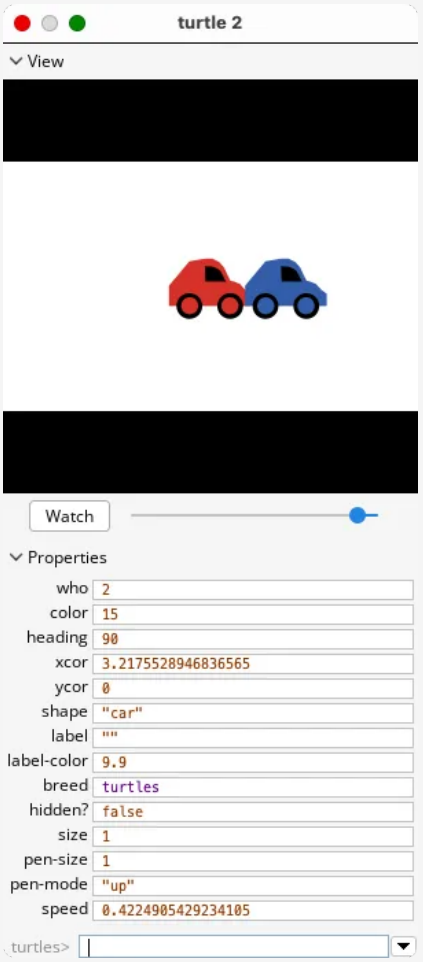
Agent Monitors and Agent Commanders

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:

> ***What is this turtle's who number?***
>
> ***What color is this turtle?***
>
> ***What shape is this turtle?***

This turtle monitor is showing a turtle 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.

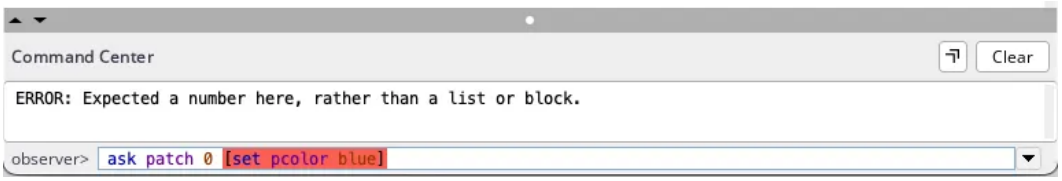
- In the Command Center, select "observer" from the popup menu (or use the tab key).
- Type `ask turtle 0 [set color blue]` and press return.

What happens?

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

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

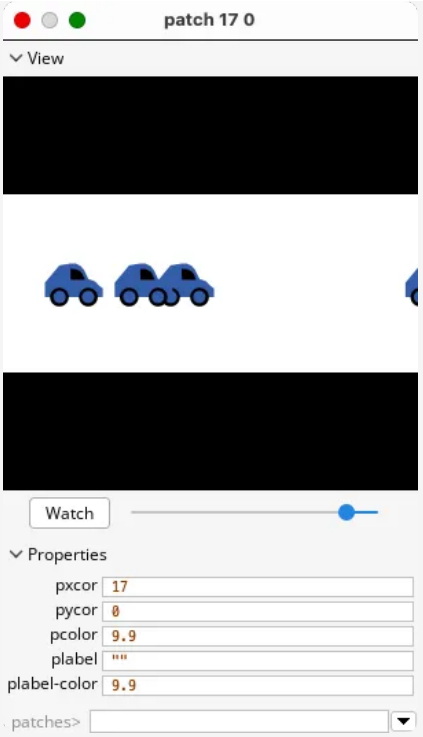
If you try to have the observer `ask patch 0 [set pcolor blue]`, you'll get an error message:



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.

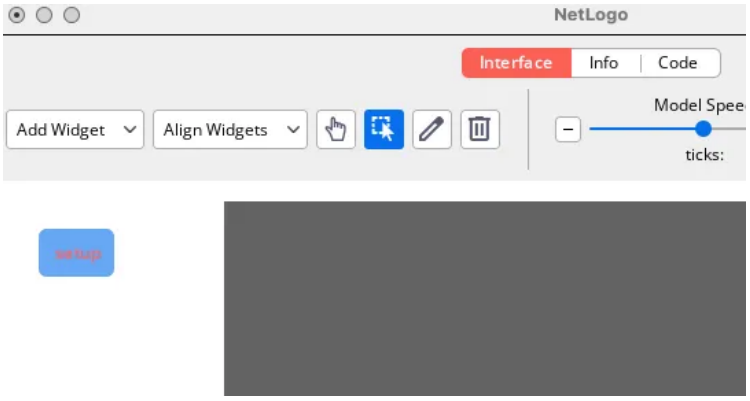
You will now learn to write procedures that make turtles move, eat, reproduce, and die. You will also learn how to make monitors, sliders, and plots. The model we'll build is a simple ecosystem model not unlike Wolf Sheep Predation from Tutorial #1.

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:

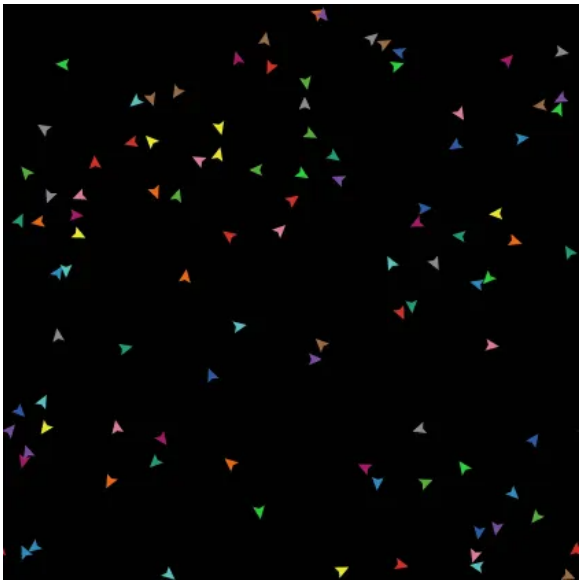
Note that some lines are indented. Most people find it helpful to indent their code. It isn't mandatory, but it makes the code easier to read and change.

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.

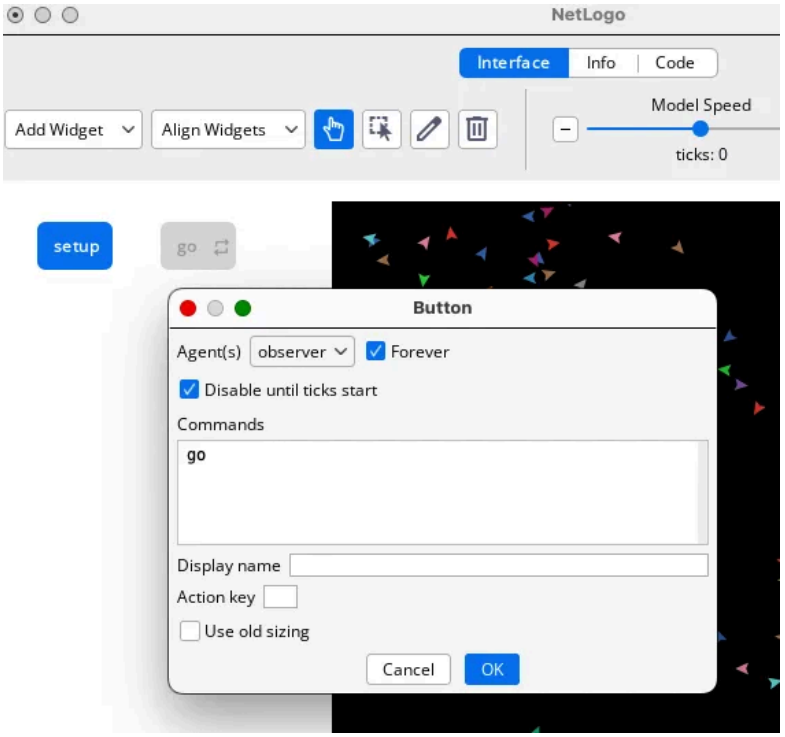
- Find the view updates menu. It's above the view and by default says "continuous".
- Choose "on ticks" instead.

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`.

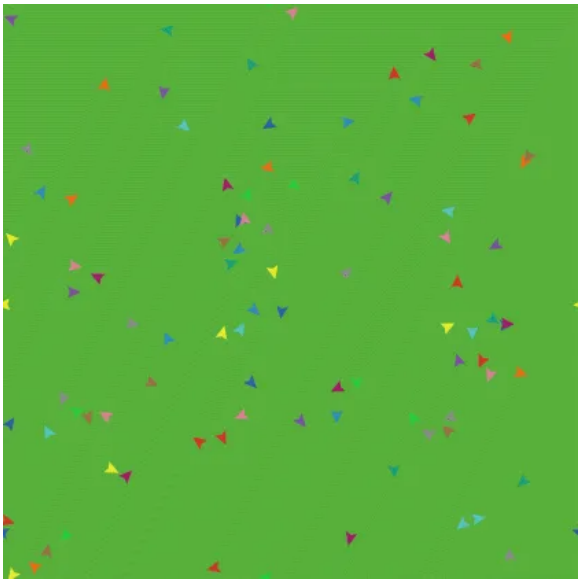
- Add this procedure too:

```
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 `color`. To make a new turtle variable, we add a `turtles-own` 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.

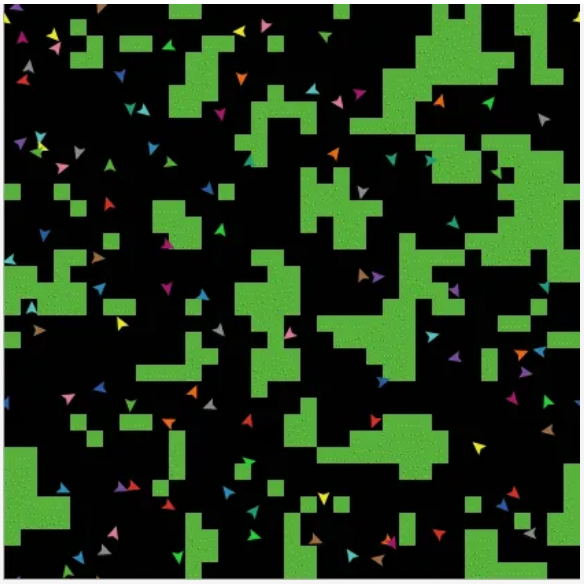
- Rewrite `move-turtles` as follows:

```
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.



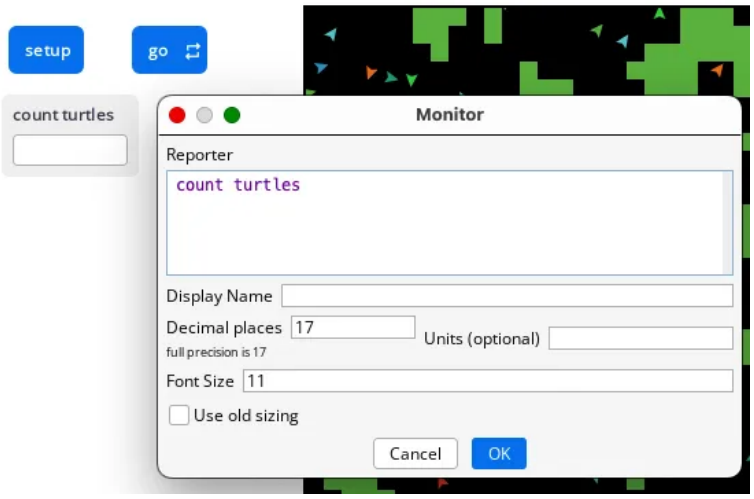
Monitors

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.



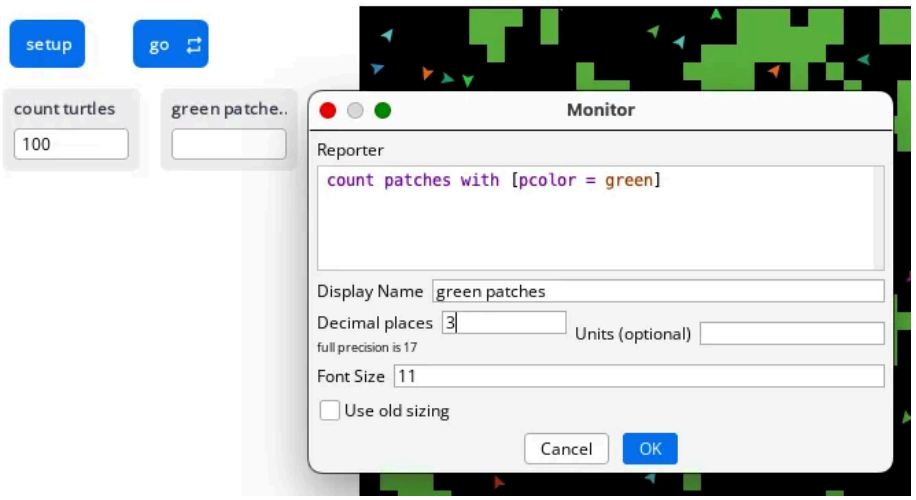
`turtles` is an "agentset", the set of all turtles. `count` tells us how many agents are in that set.

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

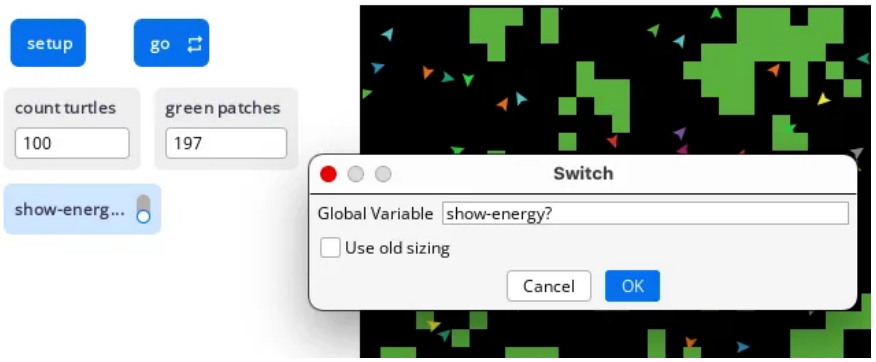
The turtles aren't just turning the patches black. They're also gaining and losing energy. As the model runs, try using a turtle monitor to watch one turtle's energy go up and down.

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:

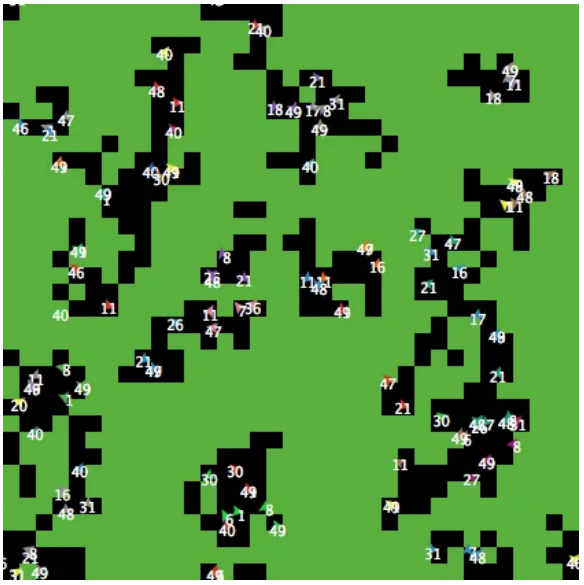
```
to eat-grass
  ask turtles [
    if pcolor = green [
      set pcolor black
      set energy energy + 10
    ]
    ifelse show-energy?
      [ set label energy ]
      [ set label "" ]
  ]
end
```

The `eat-grass` procedure introduces the **`ifelse`** 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.

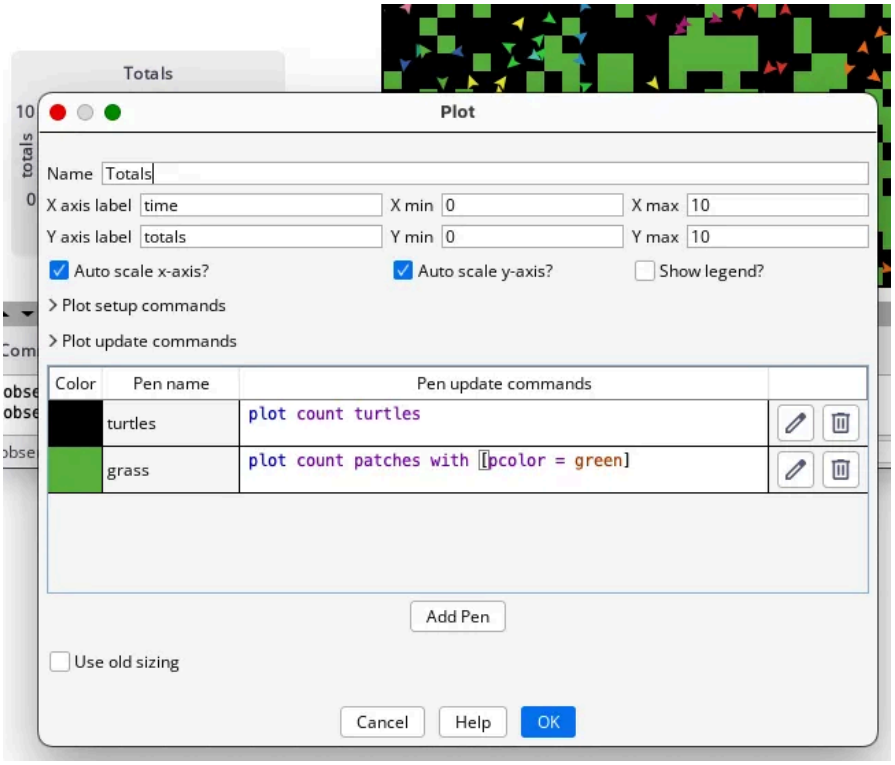
Plotting

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:



- Press OK in the Plot dialog to finish editing.

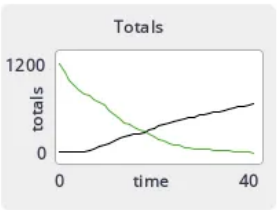
Note that when you create the plot you can also set the minimum and maximum values on the X and Y axes. You'll want to leave the "Auto Scale" checkbox checked, so that if anything you plot exceeds the minimum and maximum values for the axes, the axes will automatically grow so you can see all the data.

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

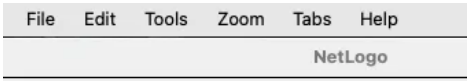
This section of the manual explains the function of each element in NetLogo's user interface.

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

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.
Models Library	A collection of demonstration models.
Recent Files	Re-opens any previously model opened with "File → Open."
Save	Save the current model, or the currently selected source file.
Save As...	Save the current model, or the currently selected source file, using a different name.
Save All	Save the current model and all open source files. This option is only available when one or more source files are open.
Save As NetLogo Web...	Saves your model as an HTML web page that runs using NetLogo Web. Note the version of NetLogo Web used is bundled with NetLogo, so it may not be up to date with the latest changes and features. You can also upload your model to NetLogo Web and export it to HTML from there to get the most recent updates included.
Manage Bundled Resources	Add, edit, and remove resources bundled with the model. See Resource Manager.

File	
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.
Export Code...	Save the model's code to an HTML file, preserving colors.
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.
Print...	Sends the contents of the currently showing tab to your printer.
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.
Comment / Uncomment	Used in the Code tab to add or remove semicolons from code (semicolons are used in NetLogo code to indicate comments).
Shift Left / Shift Right	Used in the Code tab to change the indentation level of code.
Show Usage	Used in the Code tab to show all usages of variable name or primitive that has been right-clicked.
Jump to Declaration	Used in the Code tab to show where in the file a variable name that has been right-clicked is declared.
Format	Used in the Code tab to correct the indentation of the currently selected code.
Snap to Grid	Available only in the Interface Tab. 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.)
Convert from 5.3.1 to 6.0	Available only in an ".nls" code tab. Treats the code in this tab as though it were written in 5.3.1 and converts it to run in NetLogo 6. Note that this will *not* take into account code defined in the main code tab. "Undo" doesn't take this operation into account, so you won't want to save unless you're satisfied with the changes.

Tools	
Preferences...	Opens the [Preferences dialog](netlogopreferences), where you can customize various NetLogo settings. On a Mac, this item is on the NetLogo menu instead.
Extensions...	Opens the Extension Manager, where you can discover and manage Extensions. See the Extension Manager Guide for more information.
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 Monitors for Dead Agents	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.) This option is only available when the Interface Tab is active.
3D View	Opens the 3D view. See the Views section of the Interface Tab Guide for more information.
Color Picker	Opens the Color Picker. 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.
Preview Commands Editor	Allows for easy editing of the commands sequence used to create preview images for models. Gives a way to specify what code will be used (or specify that the image is manually made) and preview the resulting image.
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.
Convert Widgets to New Sizes	Change the size of widgets created before NetLogo 7 to the current size. Optionally automatically rearrange their locations. See the Widget Sizes section of the NetLogo7 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. Keyboard shortcut: Mac Command =, On Windows or Linux, Control =.
Normal Size	Reset the screen size of the model to the normal size. Keyboard shortcut: Mac Command 0, On Windows or Linux, Control 0.
Smaller	Decrease the overall screen size of the model. Keyboard shortcut: Mac Command -, On Windows or Linux, Control -.

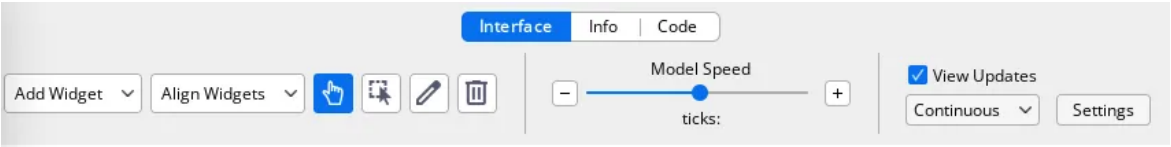
Tabs	
	This menu offers keyboard shortcuts for each of the tabs. On a Mac, it's Command 1 through Command 3. On Windows or Linux, it's Control 1 through Control 3. Additional numbers are used for tabs containing ".nls" files.

Help	
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.
Introduction to Agent-Based Modeling	Opens the MIT Press page for "Introduction to Agent-Based Modeling" (by Uri Wilensky and William Rand) in a

Help	
	web browser.
Donate	Opens the NetLogo donation page in a web browser.
About NetLogo 7.1.0-alpha1...	Information on the current NetLogo version you are running. On a Mac, this menu item is on the NetLogo menu instead.

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.

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, Russian, and Japanese.

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 changing the "User Interface Language" option in the preferences dialog. 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.

Interface Tab

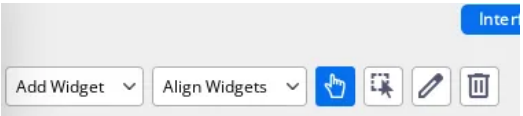
The Interface tab is where you watch your model run. It also has tools you can use to inspect and alter what's going on inside the model.

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.

Overview: Working with interface elements (widgets)

The elements you can use to create the interface for a NetLogo model are: buttons, sliders, switches, choosers, inputs, monitors, plots, the output area, and notes. These interface elements are often referred to as widgets. More details are in the Chart: Interface Widgets.

The Interface toolbar includes four tool buttons which control the meaning of user actions. They are the Interaction Tool, the Selection Tool, the Edit Tool, and the Delete Tool. There are also two pull-down menu tools, Add Widgets Menu, and Align Widgets Menu. For details on the use of these features to create, select, edit, delete, and adjust widgets, click on the tool and menu links above. They are located after the widgets chart.



Tool Buttons

One of the four tools is always active unless the user is currently adding a widget. The active tool is colored in with blue. From left to right on the Interface Toolbar are the Add Widgets and Align Widgets pull-down menus, the Interaction Tool (a finger icon) Selection Tool (a cursor with dotted box icon) the Edit Tool (a pencil icon) and the Delete Tool (a garbage can icon).

Here is a brief introduction to working with widgets.

Adding: To add a widget, select it from the "Add Widget" drop-down menu, or right click on the background and select it from the popup menu. A light image of the widget will appear at the mouse cursor. Move the mouse to the desired location and left click to insert the widget. A dialog will appear so you can set the appropriate values. To cancel the addition of the widget, right click or press the Escape key.

Selecting: With the Selection Tool active you can select a single widget by clicking it, and can select one or more widgets by dragging a rectangle around the widget(s) with your mouse. A outline with square handles will appear around the widget to show that it is selected. Alternatively you can right-click a widget and choose "Select" from the popup menu.

Unselecting: To unselect all interface elements, click the mouse on the white background of the Interface tab. To unselect an individual widget, right-click the widget and choose "Deselect" from the popup menu, or hold Ctrl/Cmd and left click on the widget.

Copying You can copy one or more selected widgets using Ctrl/Cmd + C. Using Ctrl/Cmd + V, these can be pasted into the same model, a different model, or a different instance of NetLogo. The popup menu you get by right-clicking a widget also has options to copy and paste widgets. After pasting the copied widgets, a light image of the widget(s) will appear. This preview behaves the same way as adding a new widget. Copied widgets can be pasted multiple times. Note that when copied into the same model the new widget(s) refer to the same global variable as the original, so you will probably want to edit them. You can copy the View, but pasting it into a model which already has a View will cause the old view to be replaced. The same logic applies to the Output widget.

Editing: With the Editing Tool active you can click on a widget to bring up its edit dialog. Alternatively you can right-click the widget and choose "Edit" from the popup menu.

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

Resizing: Select the widget, then drag the handles in the selection border.

Deleting: With the Delete Tool active you can click on a widget to delete it. Clicking the Delete button will delete any selected widgets. You may also delete a widget by right-clicking it and choosing "Delete" from the popup menu.

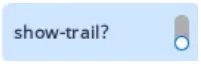
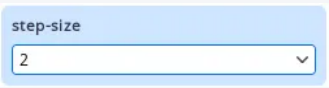
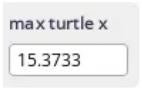
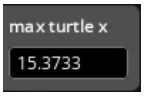
Adjusting: After selecting multiple widgets you can use the Align Widget menu to align, stretch or distribute them.

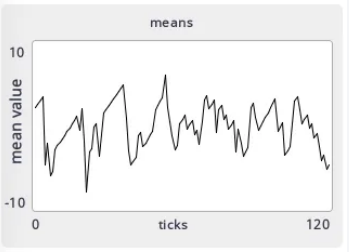
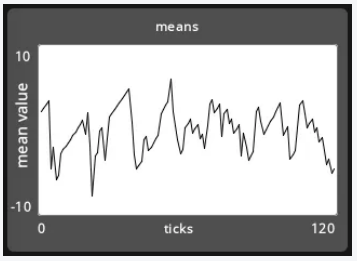
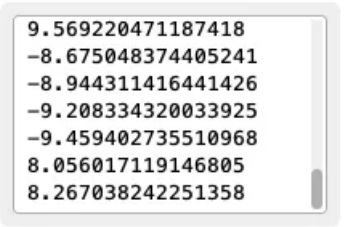
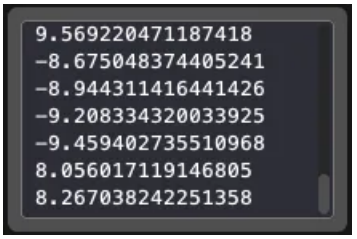
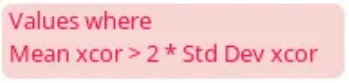
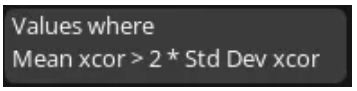
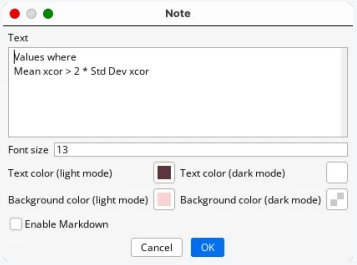
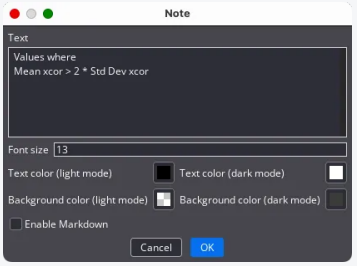
To learn more about the different kinds of interface elements, refer to the chart below.

Charts: Interface Widgets

Buttons perform actions when clicked. The input widgets set values for global variables. The input widgets are: sliders, switches, choosers and inputs. Monitors and plots are two output widgets that display a global variable value. There is also the output area which can be written to by the model as it runs. Notes are used to add labels, instructions and other information to the interface. Buttons are blue. Input widgets are a shades of blue (lighter for the Light Theme, darker for the Dark Theme). Output widgets are shades of grey. The background and font colors for notes are user settable.

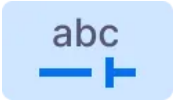
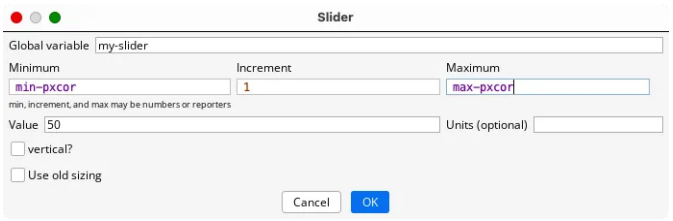
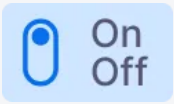
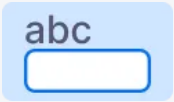
Widget Appearance: Light and Dark Themes

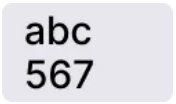
Description	Widget Light Color Theme	Widget Dark Color Theme
Buttons		
Forever Turtle Button		
Input Widgets		
Slider		
Switch		
Chooser		
Input (Color)		
Output Widgets		
Monitor		

Description	Widget Light Color Theme	Widget Dark Color Theme
Plot		
Output		
Notes		
Note		
Editing a Note		

Widget Icons and Functionality

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. By default the instructions are executed by the *observer*. However there is an agents pull-down menu that lets you specify that *turtles*, *patches*, or *links* will execute the instructions. This will be indicated by an icon on the left side of the button. 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 the Command Center,

		or otherwise not in the main Interface Tab area, 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 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. The value can also be set by typing a number in the slider's text box, or by clicking on the background of the slider and using the left and right or up and down arrow keys. Most of the fields in the edit dialog that open after you place the slider 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. 
	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 Box	Input Boxes represent global variables that contain numbers, strings or colors. The model author chooses what types of values you can enter. Number input boxes read any type of constant number reporter which allows more expressivity than a slider. When clicked, color input boxes open a NetLogo color picker. Currently colors are NetLogo colors, which are represented by numbers, but in future releases they may be RGB or RGBA colors. Therefore the global variable should only be used with Color commands or variables. If the global variable type is String(reporter), it can be evaluated using the `runresult` primitive. If the type is String(commands), it can be executed using the `run` primitive.
	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.
	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.
	Note	Notes let you add informative text labels to the Interface tab. The contents of notes do not change as the model runs.

Other Interface Toolbar controls

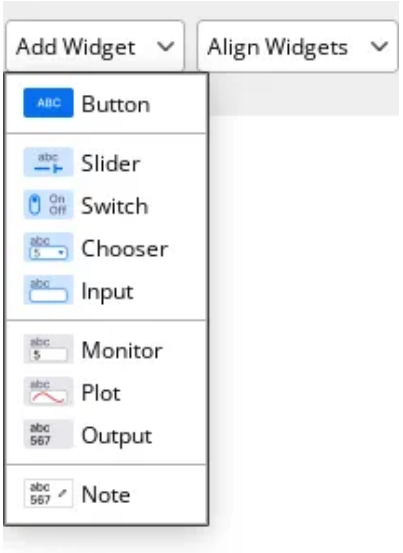
The other controls in the Interface toolbar allow you to control the view updates and various other model properties.

The speed slider is located in the Interface Toolbar. You can make the model run faster by moving the slider to the right, or by clicking one or more times on the "+" button, and slower by moving the slider to the left, or by clicking one or more times on the "-" button. Twenty clicks on the "-" or "+" button will move the slider thumb from the center to its minimum or maximum rate. The increase or decrease in speed is not linear with the amount the slider knob is moved.

- The speed slider lets you control how fast the model runs by changing the frequency of view updates. You can fast-forward the model by moving the slider to the right or by clicking one or more times on the "+" button, or slow down the the model by moving the slider to the left or by clicking one or more times on the "-" button. Slower can be valuable since some models run so fast they're hard to follow. Twenty clicks on the "-" or "+" button will move the slider thumb from the center to its minimum or maximum rate. The increase or decrease in speed is not linear with the amount the slider knob is moved.
- 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. "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 "Settings" button allows you to change model settings.

Interface Menus

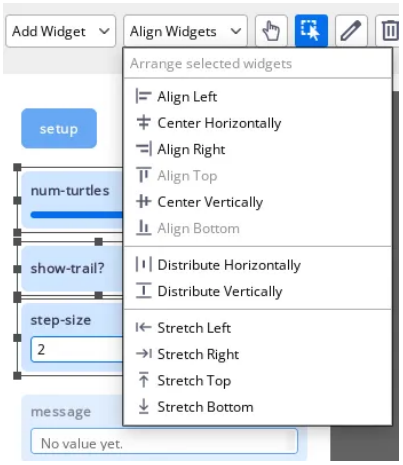
Add Widgets Menu



This menu allows users to add widgets. This tool is entered by clicking the “Add Widget” dropdown menu on the Interface toolbar and making a widget selection. The movement of the mouse is accompanied by a light image of the widget and there is no cursor. Clicking places the widget in its current location. Right-clicking allows you to change the widget type. You can also add a widget by right-clicking anywhere in the white background. This brings up a menu, allowing you to select the widget type and place the widget.

The Interaction Tool will be activated once you close the widget’s Edit dialog by clicking “OK” to insert the widget or “Cancel” to abort the process.

Align Widgets Menu



The Align Widgets is a popup menu of options to align, distribute, and stretch widgets. These options require at least two widgets to be selected. If an action is not possible it will be greyed out. This can happen if not enough widgets are selected, or if the action would result in undesireable overlap of widgets. See **Widget Overlap** for more details. The items remain selected after the action is completed. The Align Widgets menu can also be accessed by right clicking on a widget when multiple widgets are selected.

Aligning

With Align Left the left-most widget remains in place. The other widgets move to align to left-most widget edge. There are similar alignment options for the right, top and bottom. Widgets can also centered vertically or horizontally.

Distributing

Distribute Vertically keeps the top and bottom widgets fixed, and moves the other widgets so the vertical spacing between the widgets is the same. Distribute Horizontally works similarly.

Stretching

If Stretch Left is selected the widget with the left-most widget remains in place. All the other widgets stretch so they have the same left-most edge. There are similar alignment options for the right, top and bottom.

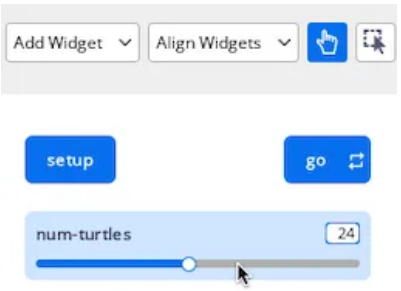
Widget Overlap

The actions of aligning, distributing or stretching will not be completed if they would create a new overlap of widgets. Not completing an action may result in unexpected behavior, but protects users from other unexpected behavior, such as one widget becoming completely hidden behind another.

Interface Tools

Interaction Tool

The Interaction Tool is for making direct use of widgets to run commands and change global variables.



The effect of clicking is widget specific, as described in the widget chart.

Right-clicking

Users can Select, Edit and Delete widgets in interaction mode by right-clicking and bringing up a menu. With Plots one can also Copy Image, Export, and Clear. With the Output widget one can Copy Selected Text, Clear, and Export. The View cannot be deleted. The available options are Select, Edit, Copy View, Export View, Switch to 3D View, Inspect Globals, Inspect a patch, and Inspect, Watch or Follow a turtle.

Ctrl-clicking or Cmd-clicking

Selection of a widget can be toggled on and off using Ctrl-click (Windows and Linux) or Cmd-click (MacOS). Performing this action when using the Interaction Tool will activate the Selection Tool.

Drag Select

One option that is not available with the Interaction Tool is using Drag-Select to select one or more widgets. To Drag-Select first press the Selection Tool. However Ctrl+Drag does work with the Interaction tool.

Exiting

Pressing the Selection, Editing or Deletion icons exits the Interaction Tool. One can also activate the Selection Tool by using Ctrl-Click (Windows or Linux) or Cmd-Click (macOS) on a widget, or by using the Select option with the right-click pull-down menu. Using the Edit or Delete option with the right-click pull-down menu does not exit the Interaction Tool because they are “transient” actions.

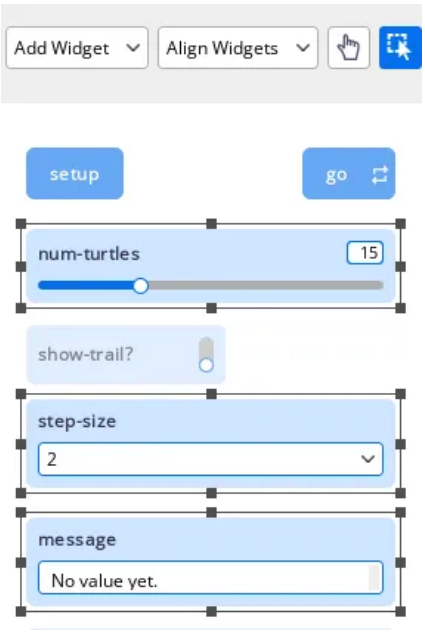
Selection, Editing and Deletion Tools

These explicit tools are new for NetLogo 7, although they incorporate behaviors associated with certain NetLogo 6.4 icons.

Each of these tools can be entered by pressing the appropriate icon. When one of these tools is active widgets will be faintly dimmed to indicate that they are not currently interactive, and all “forever” buttons will halt. Each can be exited by selecting a different tool icon. The Esc key will return you to the Interaction Tool. Moving the focus to another context such as the Command Center, the Info Tab, or the Code Tab will activate the Interaction Tool.

Selection Tool

This tool allows users to select and subsequently edit or move a widget or change its size. Enter this tool by pressing the “Selection Tool” icon on the Interface toolbar (cursor with dashed rectangle).



Selecting

Press the “Selection Tool” Icon. Widgets will dim. Clicking on a widget selects it. A line border with square handles will appear around the widget to show it is selected. This will be black for the Light Color Theme, and grey for the Dark Color Theme. A widget can also be selected by dragging a rectangle around it with your mouse or by right-clicking a widget and choosing Select from the menu. You can toggle selection and deselection by using Ctrl+Click (Windows/Linux) or Cmd+Click(MacOS) on a widget. Any method of selecting one or more widgets will activate the Selection Tool. Clicking the background area will deselect any selected widgets. Dragging one widget will move all selected widget(s). You can also click and drag an individual widget before the selection markers appear.

Selecting multiple items

You can select multiple interface elements at the same time by including them in the rectangle you drag. Additional elements can also be added by using the alternative methods.

Deselecting

An individual selected widget can be selected by pressing Ctrl+Click (Windows/Linux) or Cmd+Click (MacOS) or by right-clicking the widget and choosing “Deselect” from the popup menu. To deselect all selected interface elements, click the mouse on the white background of the Interface tab. The Selection Tool will remain active. Any method of choosing a different tool will exit Selection tool.

The following actions require that at least one widget is selected:

Moving

Select one or more widgets, then drag with your mouse to the new location. You can use the arrow keys to move the widgets a small increment right, left, up, or down. Using shift plus an arrow key increases the increment. If you hold down the shift key while dragging, the elements will move only straight up and down or straight left and right. The items remain selected.

Resizing

Select the widget, then drag the square handles in the selection border. If multiple elements are selected the handles can be dragged one at a time, and the items remain selected.

Editing Tool

This tool allows users to change the characteristics of an interface widget Enter this tool by pressing the “Editing Tool” Icon on the Interface toolbar (a pencil).

Editing: Press the “Editing Tool” Icon and click on a widget to edit it. When the user finishes editing a widget, the Editing Tool remains active. Clicking another widget will edit it.

If a widget is already selected you can press the editing icon to activate the Editing Tool and open the widget for editing. Clicking the Edit icon when multiple items have already been selected deselects the items, and enters the Edit tool. You can also begin editing by double-clicking a selected widget, or right-clicking an unselected widget and choosing “Edit” from the popup menu.

Deletion Tool

This tool allows users to delete interface elements. Enter this tool by pressing the “Deletion Tool” icon on the Interface toolbar (trashcan). After a widget is deleted the user will remain in the Deletion Tool and ready to delete more items. The view cannot be deleted.

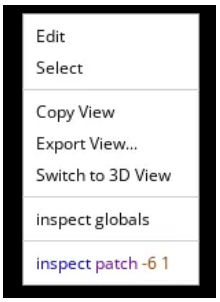
You can also select the widget(s) you want to delete, then press the “Delete” button to delete them all at once.

Deleting

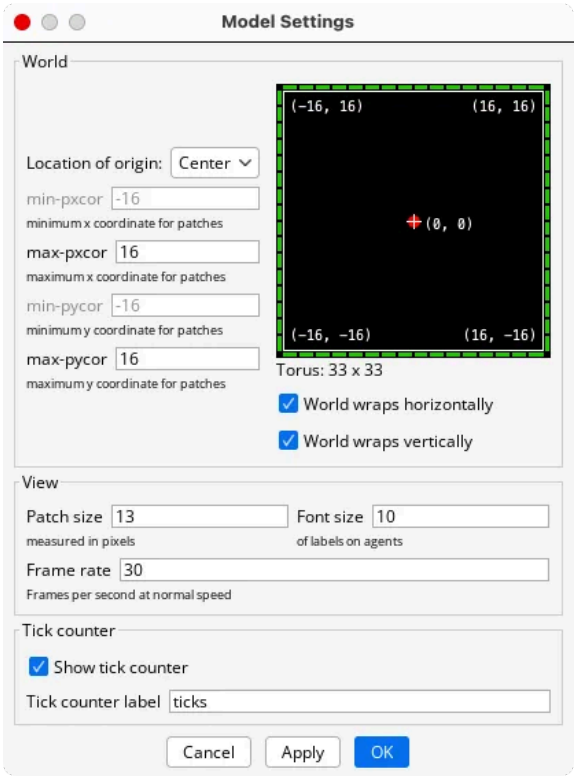
Press the widget. You may also delete a widget by right-clicking it and choosing “Delete” from the popup menu. If you use this latter method, it is not necessary to select the widget first.

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 right clicking (ctrl-clicking on Mac) on the View and choosing "Switch to 3D View" (this option is also available in the "Tools" menu).



There are a number of 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 tick 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.

World Settings

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 patch at (0, 0) and 10 patches to the right of the patch at (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 the patch at (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 what happens to a turtle when it reaches the edge 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.

View Settings

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

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

Interacting with Agents

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 3D View window there are buttons to move the observer, or change the perspective from which you are looking at the world.

While you are adjusting one of these settings, a blue cross will appear at the current focus point of the 3D observer. 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 “Orbit” 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

Fullscreen mode makes the 3D view of NetLogo Desktop fit on the entire screen. It is often useful for exhibits in museums and other locations.

To enter fullscreen mode, press the “Full Screen” button, and to exit fullscreen mode, press the Esc key.

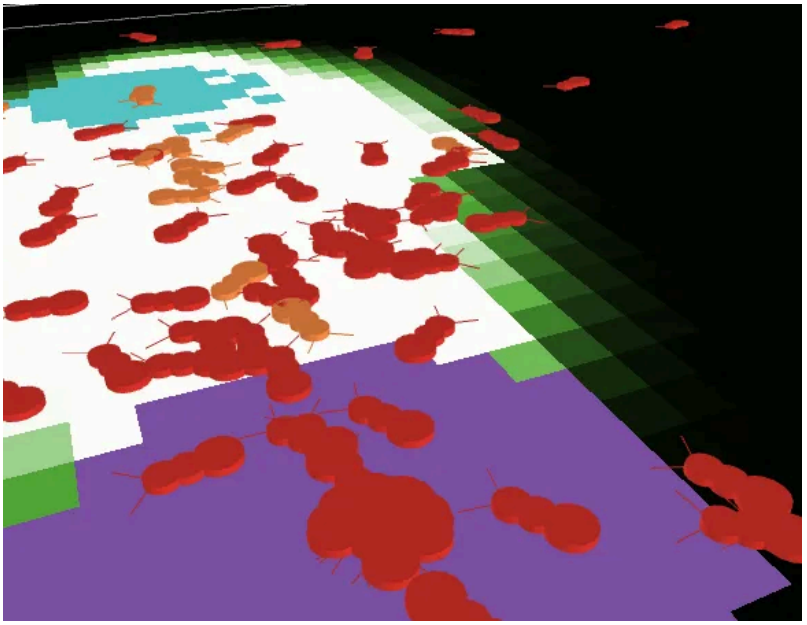
Note: Fullscreen mode doesn’t work on every computer. It depends on your graphics card. See the System Requirements for details.

3D Shapes

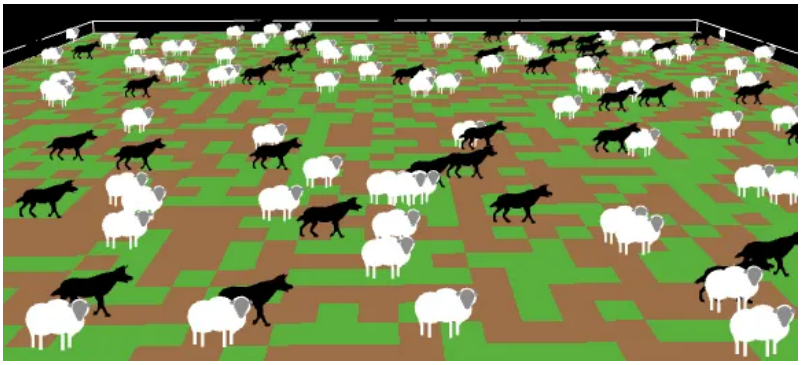
Some shapes are automatically mapped to true 3D counterparts in the 3D view. For example, the 2D circle shape becomes a sphere in the 3D view.

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. You can use the up arrow to access the commands you previously typed, and the down arrow to navigate back through the list.

To the left of where you type is a label that initially says "observer>". Clicking on the label will open a popup menu where 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 `show` 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 select a section of the text in this box and use Ctrl+C (Windows/Linux) or Cmd+C (Mac) or select Copy from the Edit menu 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 command history, choose “Clear History” on the history popup menu (accessed using the small downward pointing triangle to the right of where you type commands).

Arranging

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

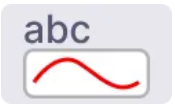
To resize the command center, drag the dark gray bar that separates it from the model interface. Or, click one of the little arrows on the left end of the bar to make the command center either fill the interface or be hidden altogether.

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.

Points added to a plot using `plot` or `plotxy` will not be visible if they are outside the extent of the axes. The x and y axes can be automatically (and independently) readjusted by checking the Auto scale x-axis? and Auto scale y-axis? boxes respectively. Note that if you use `histogram` Auto scale y-axis? will work but Auto scale x-axis? will have no effect. Note that these checkboxes can be overridden by Plotting commands such as `auto-plot-x-on`, `auto-plot-y-on`, `auto-plot-x-off`, `auto-plot-y-off`, `set-plot-x-range` and `set-plot-y-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.

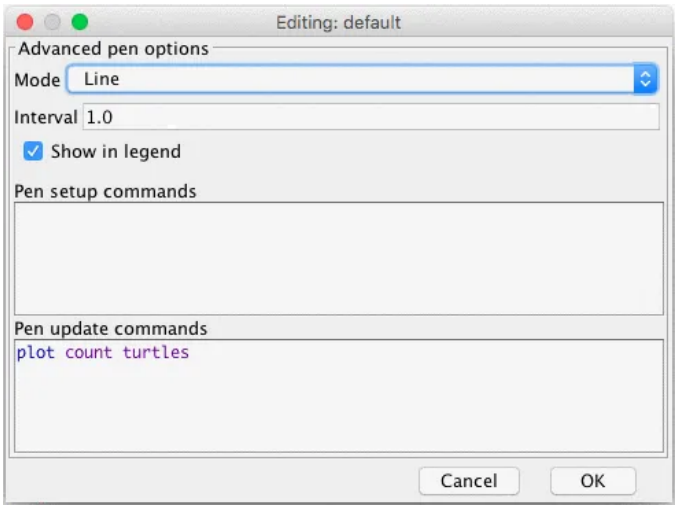
To edit the pen's name, double click the name.

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:



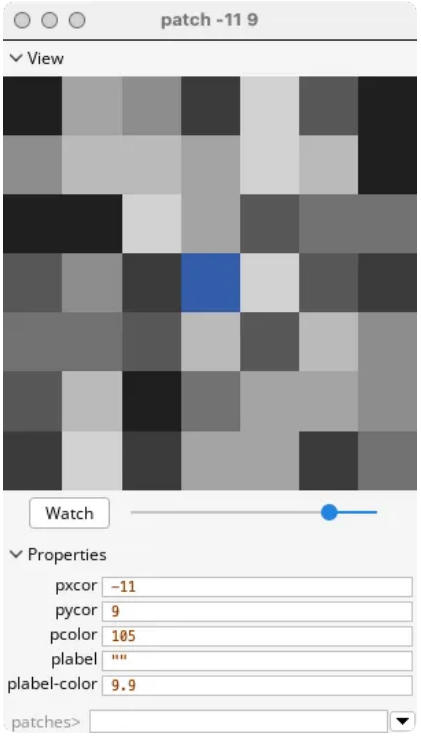
- Mode allows you to change the appearance of the plot pen: line, bar (for a bar chart), or point (a scatter plot of individual points).
- Interval is the amount by which x advances every time you use the `plot` command.
- If the "Show in legend" checkbox is checked the selected pen will be a part of the legend below 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.

Agent Monitors

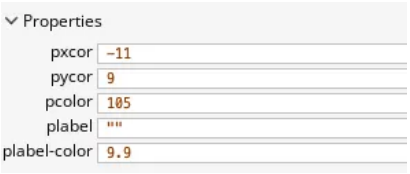
Agent monitors display both the values of all the variables for a particular agent and a mini-view that shows the agent and a small area around it.

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" button.

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

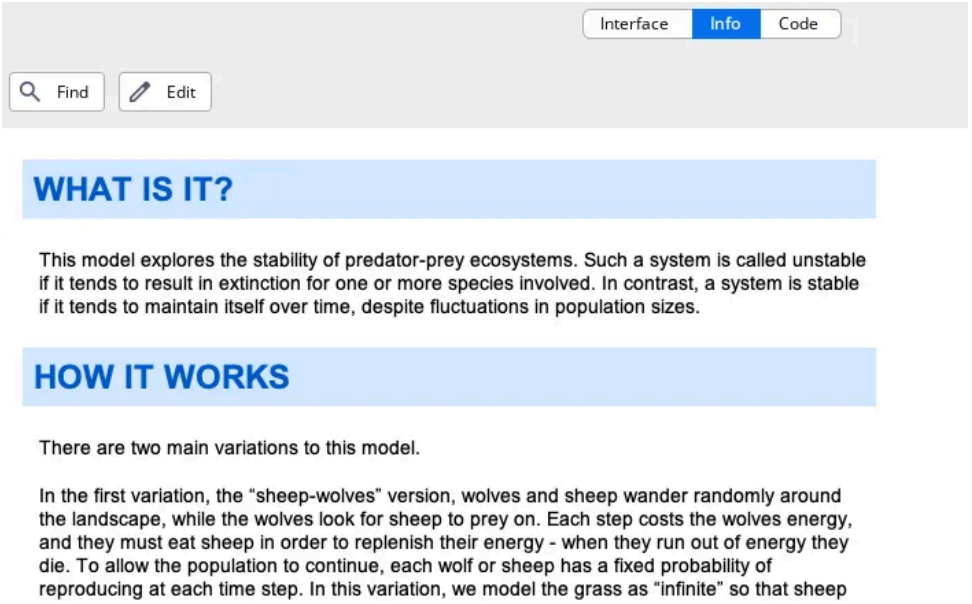


Below the agent variable area there is a miniature command center. Rather than running code as the observer, or talking to all of the turtles, patches, or links, the code entered in this command center is run only by the agent shown in the window.

You can close an agent monitor by clicking the window's close button, or by pressing the Esc key. You can close all the agent monitors by using the "Close All Monitors" option in the Tools Menu or close just those for dead agents by choosing "Close Monitors for Dead Agents."

Info Tab Guide

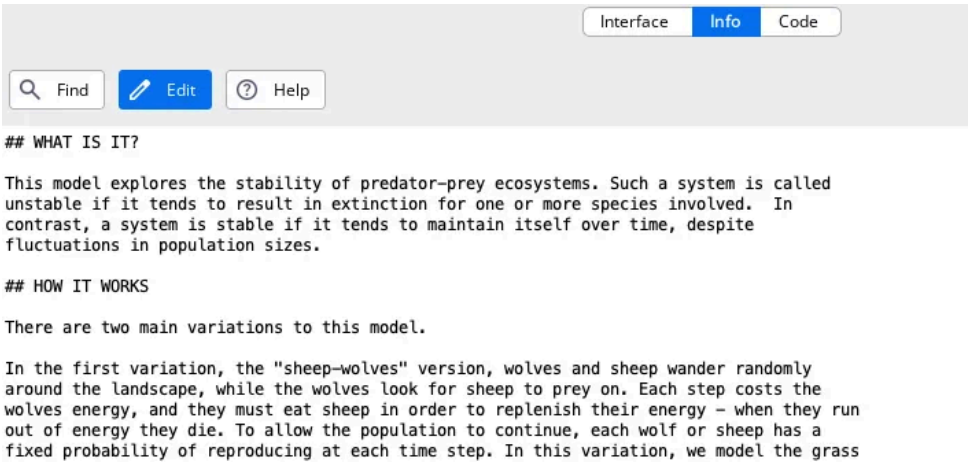
The Info tab provides an introduction to a model. It explains what system is being modeled, how the model was created, and and how to use it. It may also suggest things to explore and ways to extend the model, or call your attention to particular NetLogo features the model uses.



You may wish to read the Info tab of an existing model before interacting with its widgets or its code.

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

A heading begins with one or more hash marks (`#`). The number of hash marks used controls the size and prominence of the header, with one hash mark indicating the largest header.

Input

```
# First-level heading

## Second-level heading

### Third-level heading

#### Fourth-level heading
```

Paragraphs

Example

```
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 one line with three sentences.

Single line breaks in the input
do not make line breaks in the output,
as demonstrated here.
```

Formatted

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 one line with three sentences.

Single line breaks in the input do not make line breaks in the output, as demonstrated here.

Italicized and bold text

Example

```
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_**
```

Formatted

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

Example

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

  1. Ordered lists are indented 2 spaces.
    a) 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. a) 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

Example

Bundled images

Images that are bundled with the model (See **Resource Manager**) can also be displayed in the Info tab. To display a bundled image, use the same syntax as for local files above, but omit the `file:` specifier. For example, if there is a bundled image called `wolf-sheep`, you can include it in the Info tab as follows:

```
![Wolf Sheep](wolf-sheep)
```

To control various properties of an image, such as its size, you can use the equivalent HTML tag instead of Markdown syntax, as follows:

```
<img src="wolf-sheep" alt="Wolf Sheep" width="100" />
```

In either case, if the provided name refers to a resource that is not an image, a dummy image will be displayed in its place.

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.
```

Formatted

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. Another way is to surround it with a three backticks line before and after the block. (If you don't want your code to be colored as NetLogo code, add `text` after the first three backticks.)

Example

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

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

or:

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

```
; 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 put the code block:

```
; 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>0

2x<sup>4</sup> + x<sup>2</sup> + 42

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.

Other features

Markdown has additional features that we have not shown here.

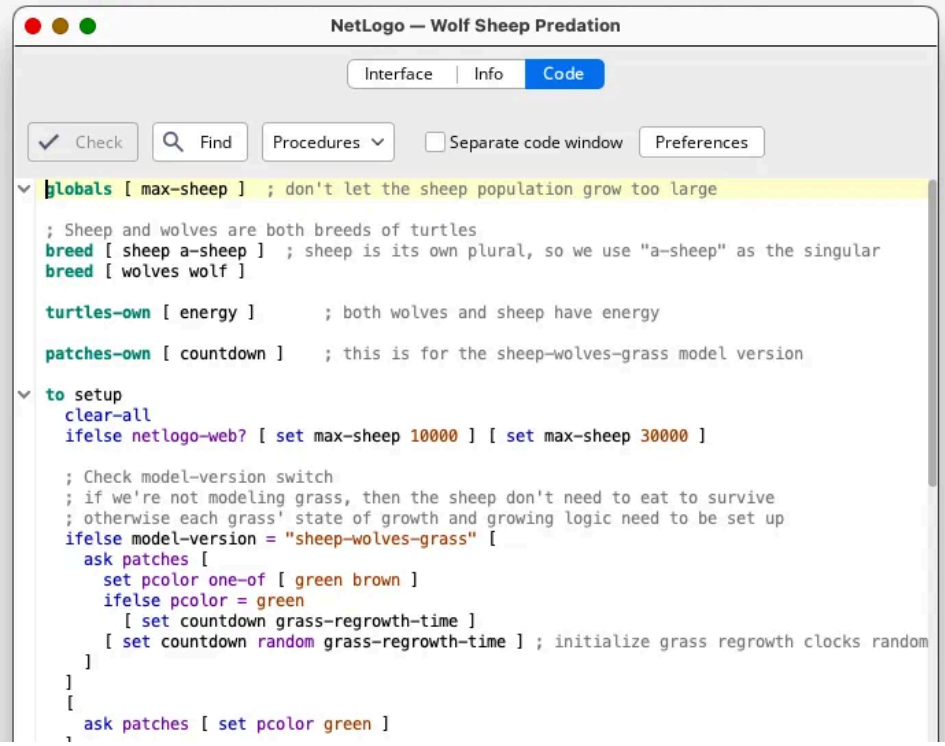
We have tested the features shown above on a variety of systems. If you use other Markdown features, you may find that they work on your computer, or not. Even a feature that works on your computer might work differently, or not work at all, for someone with a different operating system or Java virtual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

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

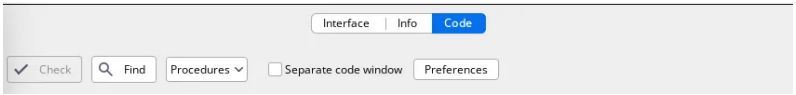
More information about Markdown is at https://daringfireball.net/projects/markdown/. For rendering Markdown, NetLogo uses the Flexmark-java library.

Code Tab

The Code tab is where the code for the model is stored. Commands you only want to use once can be typed in the Command Center, but collections of commands that you want to save for later use can be stored in the Code tab.



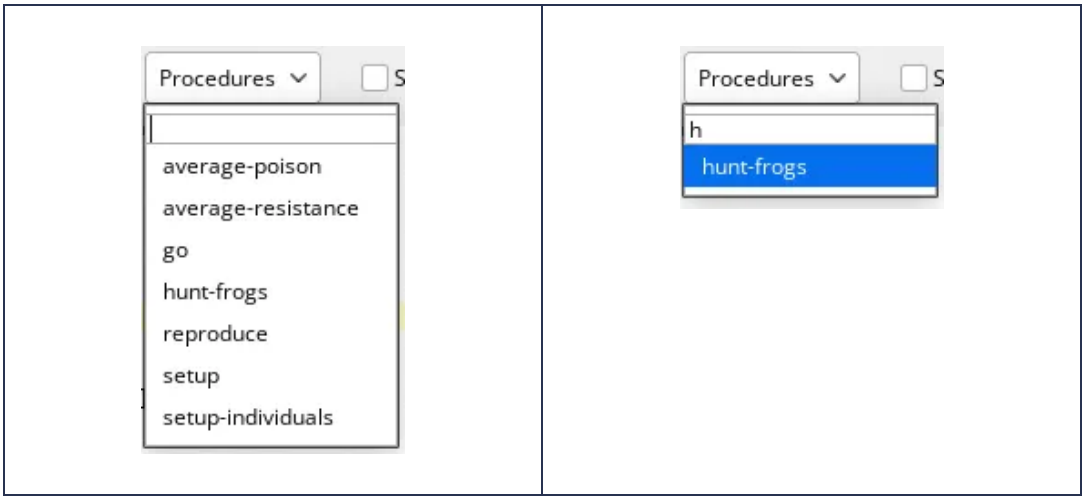
Code Tab Toolbar



- Check
- Find
- Procedures
- Separate code window
- Preferences (also described in relevant sections of this document)

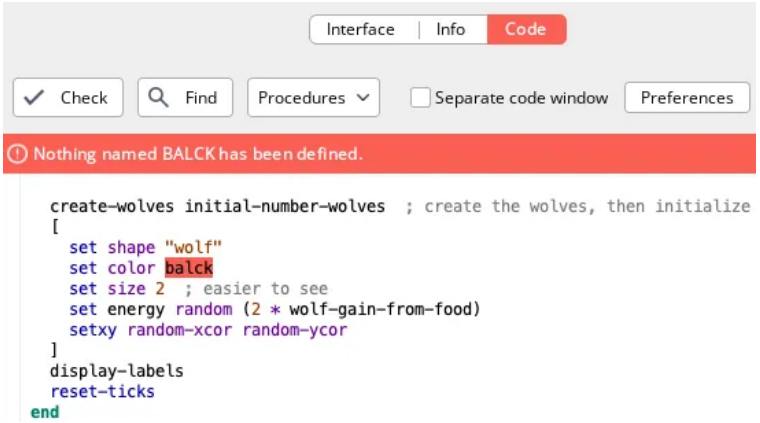
Procedures menu

This menu includes a list of all the procedures in the Code tab. Clicking on a procedure moves the cursor to that procedure. Typing in the text box at the top will filter the list, only showing procedures whose name contains the typed text. By default procedures are listed alphabetically, but this can be changed to their order of appearance by using the "Sort order of procedures menu" code preference.



Checking for Errors

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 orange, the code that contains the error will be highlighted, the cursor will move to the error location, and an error message will appear in an orange banner at the top of the code area. If you don’t want the cursor to move when there is a compilation error (for example when there is more code than fits on the screen) uncheck the “Auto-focus text or tab with errors when model is compiled” preference. The error will not be highlighted. If the error occurs in the Interface tab, for example if any button or plot commands result in a compilation error, the Interface tab will turn orange instead.



Switching tabs also causes the code to be checked, so if you just switch tabs, pressing the Check button first isn’t necessary.

Separate Code tab

The Code tab can be opened in a separate window from the main NetLogo window. It can then be moved and resized.

- The “Code Tab in separate window” checkbox can be used to switch the Code tab location between the main NetLogo window and a separate window.
- A Control-click with the mouse on the tab bar of the Code tab will also switch its location.
- To separate a Code tab from the main NetLogo window you can use Command + Shift + W on Mac, Control + Shift + W on other operating systems.
- To return a separated Code tab to the main NetLogo window you can use Command + W on Mac, Control + W on other operating systems or click the window’s close button.
- There is a Code preference to “Open code tab in separate window on startup”.

Find & Replace

To find a fragment of code in the Code tab, 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.

Automatic Indentation

When the “Auto indent” Code preference 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.

```
to setup
  create-turtles 5
  [
    set color red
    set shape "circle"
  ]
end
```

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.

Line Numbers

By default line numbers are shown on the left edge of the code area. They can be removed by unselecting the “Show line numbers” Code preference.

Code Completion

The Code tab features an inline code completion popup, which appears as you type and displays a list of suggested keywords, primitives, variables, and procedures, based on what you’ve typed so far in the current word. You can use the up and down arrow keys to navigate through the list, and the Enter/Return key to accept a suggestion. Note that the automatic opening of the popup as you type can be disabled in Code Preferences. At any time, whether or not this preference is enabled, you can also manually open the popup by pressing Control + Space.

More Editing Options

To find a particular procedure definition in your code, use the “Procedures” popup menu in the Code tab. The menu lists all procedures in the order they appear in the file. To search for a particular procedure, use the search field at the top of the popup. It will filter the list of procedures using fuzzy matching as you type. Fuzzy match will include procedures whose names contain the characters that you type in the order that they appear in the search field, but the characters do not have to be right next to each other. For instance, “fnn” will match“**f**ind-**n**earest-**n**eighbors” and “wol**f**-down-**n**eighbor” but not “nearest-wolf-neighbor”. You may use the arrow keys or mouse to select a particular procedure and enter or click to jump to it. This can be a very convenient way to navigate your file. To facilitate this, you can use the shortcut `ctrl-g` (`cmd+g` on Mac OS) to access the procedures menu.

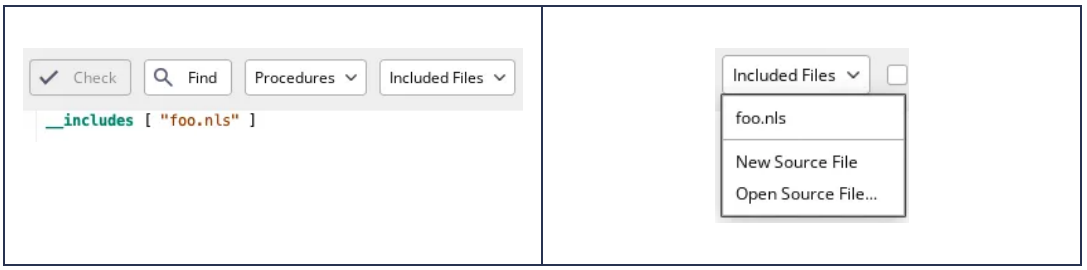
The “Shift Left”, “Shift Right”, “Comment”, and “Uncomment” items on the Edit menu are used in the Code tab to change the indentation level of your code or add and remove semicolons, which mark comments, from sections of code.

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

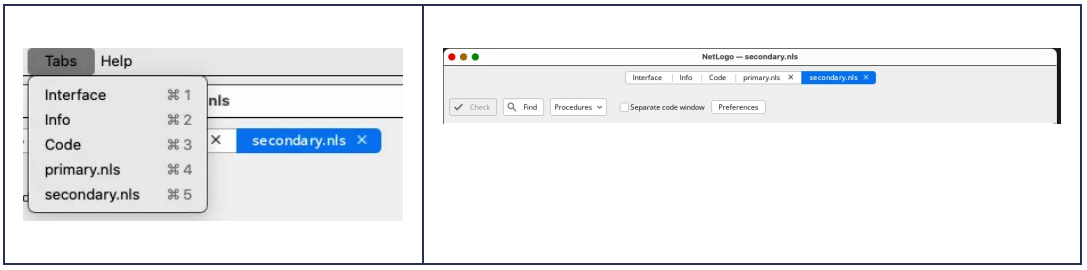
Included Files

Caution: The includes facility is still experimental.

If you have a file with the extension .nls all of its breeds, variables, and procedure definitions are available in the model if you add it by using the __includes keyword to a model. When that keyword is used the “Included Files” appears menu to the right of the procedures menu appears. This menu lists all the NetLogo source files (.nls) included in this model. 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 the “New Source File” or “Open Source File” items .You can make this menu always visible using the “Always show “Included Files” menu” Code preference.



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). If the main Code tab is in a separate window, the tabs for included files will appear there too. When an included (.nls) file tab is selected it becomes blue. An included file tab can be closed by clicking on the “x” on the right side of the tab. Each included file tab has a toolbar indentical to the main Code Tab’s toolbar; the options in this toolbar apply to all Code tabs, not just the one that is currently visible.



Note that you can create or open an .nls file before including it in a model with an `__includes` statement. In this case, the contents of the file will not be compilable, which is indicated by a warning label below the toolbar and by the Check button being disabled. This does not affect the rest of the model or cause any errors, but the contents of the file will not be accessible from any other code tabs in the model.

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 and Agentsets

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.

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 **`breed`** 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 **`turtle`** 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 **`create-<breeds>`**. Others have the singular name of the breed in them, such as **`<breed>`**.

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

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 of built in variables`.)

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
```

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 link designations: undirected and directed. A directed link is *out* of, or *from*, one node and *into*, 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 designation from the previous links.)

In general, link primitive names indicate what kind of links they deal with:

- Primitives that have "out" in their name utilize outgoing and undirected links. You can think of these as "the links I can use to get *from* the current node *to* other nodes." In general, these are probably the primitives you want to use.
- Primitives that have "in" in their name utilize incoming and undirected links. You can think of these as "the links I can use to get *to* the current node *from* other nodes."
- Primtives that do not specify "in" or "out", or have "with" in their name utilize *all* links, both undirected and directed, incoming and outgoing.

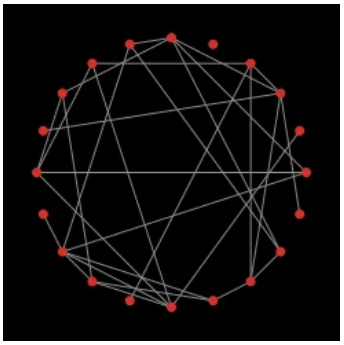
A link's `end1` and `end2` variables contain the two turtles the link connects. If the link is directed, it goes from end1 to end2. If the link is undirected, end1 is always the older of the two turtles, that is, the turtle with the smaller who number.

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.

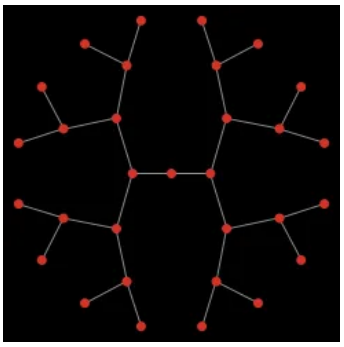
Layouts

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.

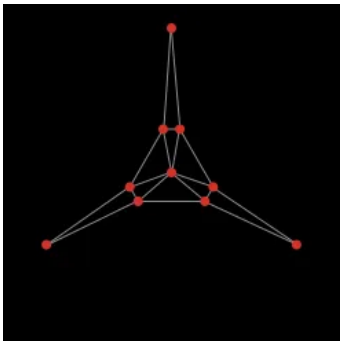


`layout-radial` is a good layout if you have something like a tree structure, though even if there are some cycles in the tree it will still work, though as there are more and more cycles it will probably not look as good. `layout-radial` takes a root agent to be the central node places it at (0,0) and 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

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

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.

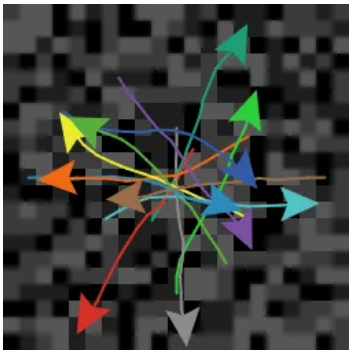
You can see the drawing, but the turtles (and patches) can't. They can't sense the drawing or react to it. The drawing is just for people to look at.

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.

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.

Programming

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
```

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.

Because agentset members are always read in a random order, when `ask` is used with an agentset each agent will take its turn in a random order. See Agentsets for more information.

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 `setup` procedure above, the turtle with who number 0 is asked to get the patch east (and no patches north) of itself like this: `ask turtle 0 [ ask patch-at 1 0 [ set pcolor red ] ]`.

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.

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.

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`, `end`, `import`, and `export`, 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.

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 `to` or `to-report`, the procedure name, and an optional bracketed list of input names. Every procedure definition ends with `end`. 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 `list` 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:

- with, `with-min`, `with-max`, `at-points`, `in-radius`, `in-cone`, `who-are-not`
- of
- All other primitives and user-defined procedures, including `not`
- `^`
- `*`, `/`, `mod`
- `+`, `-`
- `<`, `>`, `<=`, `>=`
- `=`, `!=`
- and, `or`, `xor`
- ifelse-value

When an expression contains multiple operators with the same precedence, they are read in code order left-to-right. Example: `true or false and false` is read as `(true or false) and false` and so results in `(true) and false`, which is `false`. This is different than many other programming languages where `or` has a lower precedence than `and`, so the above statement would be read as `true or (false and false)`, `true or (false)`, so `true`. You can surround expressions in parentheses to ensure you get your desired order of operations.

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	NetLogo

`to square :x`
`output :x *`
`:x`
`end`
	`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 anonymous procedures 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 anonymous procedures, but you must use the `->`, `run`, and `runresult` primitives for that, you cannot make them implicit.)
- Anonymous procedures (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.

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.

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.

Anonymous procedures

Anonymous procedures let you store code to be run later. Just like regular NetLogo procedures, an anonymous procedures can be either a command (anonymous command) or a reporter (anonymous reporter).

Anonymous procedures are values, which means they may be passed as input, reported as a result, or stored in a variable.

An anonymous procedure might be run once, multiple times, or not at all.

In other programming languages anonymous procedures are known as first-class functions, closures, or lambda.

Anonymous procedure primitives

Primitives specific to anonymous procedures are `->`, `is-anonymous-command?`, and `is-anonymous-reporter?`.

The `->` creates an anonymous procedure. The anonymous procedure it reports might be a command or a reporter, depending on what kind of block you pass it. For example `[ -> fd 1 ]` reports an anonymous command, because `fd` is a command, while `[ -> count turtles ]` reports an anonymous reporter, because `count` is a reporter.

These primitives require anonymous procedures as input: `foreach`, `map`, `reduce`, `filter`, `n-values`, `sort-by`. When calling these primitives, using an `->` is optional if your anonymous procedure contains a single primitive which has requires no more inputs than are are provided by the primitive. For example one may write simply `foreach mylist print` instead of `foreach mylist [ [x] -> print x ]`, though the latter is also accepted. Depending on the anonymous procedure, various parts of the anonymous procedure syntax can be omitted. For a summary of optional syntax, see the table below.

The `run` command accepts anonymous commands as well as strings.

The `runresult` reporter accepts anonymous reporters as well as strings.

`run` and `runresult` allow passing inputs to an anonymous procedure. As with all primitives accepting varying number of inputs, the whole call must be surrounded with parentheses, so for example `(run my-anonymous-command 5)` or `(runresult my-anonymous-reporter "foo" 2)`. When not passing input, no parentheses are required.

Anonymous procedure inputs

An anonymous procedure may take zero or more inputs. The inputs are referenced the variables declared before the arrow. For instance, in the anonymous reporter `[ [a b] -> a + b ]`, `a` and `b` are inputs.

Anonymous procedures and strings

Creating and running anonymous procedures is fast. To use `run` or `runresult` on a new string for the first time is about 100x slower than running an anonymous procedure. Modelers should normally use anonymous procedures 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
;; prints 1 through 4
```

In older NetLogo versions (4 and earlier), 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 ? ]
;; prints 1 through 4
```

Anonymous procedures as closures

Anonymous procedures 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 anonymous procedure. (This is backward-compatible with older NetLogo versions.)

Anonymous procedures and extensions

The extensions API supports writing primitives that accept anonymous procedures 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 anonymous procedures.
- Anonymous procedures can't be variadic (accept a varying number of inputs).
- Anonymous reporters 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 anonymous reporter, passing it any needed inputs.
- Anonymous procedures are not interchangeable with command blocks and reporter blocks. Only the primitives listed above accept anonymous procedures as input. Control primitives such as `ifelse` and `while` and agent primitives such as `of` and `with` don't accept anonymous procedures. So for example if I have an anonymous reporter `let r [ -> if random 2 == 0 ]` and two anonymous commands `let c1 [ -> tick ]` and `let c2 [ -> stop ]`, I can't write `ifelse r c1 c2`, I must write `ifelse runresult r [ run c1 ] [ run c2 ]`.
- The concise syntax where `->` 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 an anonymous procedure as input, it must be called as e.g. `p [ -> ... ]` not `p [ ... ]`.

What is Optional?

There are several different ways of writing anonymous procedures which allow users to omit part or all of the anonymous procedure syntax. These are summarized in the table below.

What is the anonymous procedure like?	What can be left out?	Examples
The anonymous procedure is a single primitive	• input names • arrow • block brackets	• `foreach mylist stamp ; no inputs` • `foreach mylist print ; single input` • `(foreach xs ys setxy) ;`

What is the anonymous procedure like?	What can be left out?	Examples
		`multiple inputs` • `map round [1.3 2.4 3.5] ; reporter, single input` • `(map + [1 2 3] [4 5 6]) ; reporter, multiple inputs`
The anonymous procedure takes no inputs	• input names • arrow	• `foreach mylist [ print "abc" ]` • `map [ 4 ] mylist`
The anonymous procedure has zero or one input(s)	• brackets around input names	• `foreach mylist [ -> stamp ] ; no inputs` • `foreach mylist [ x -> print x ] ; single input` • `foreach mylist [ x -> rt x fd x ] ; multiple primitives, single input` • `map [ -> world-width ] mylist ; reporter, no inputs` • `map [ x -> x ^ 2 ] mylist ; reporter, single input`
Anonymous procedure takes more than one input	• nothing	• `(foreach xs ys [ [ x y ] -> setx x + y ])` • `(map [ [ x y ] -> x mod round y ] xs ys)`

Note: brackets around input names were **always required** in NetLogo 6.0.0. If you copy and paste code into NetLogo 6.0.0 using anonymous procedures with unbracketed input names, the code will not compile until you add the brackets.

Code example

Code Example: State Machine Example

World

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.

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. 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.

Changing the update mode also affects model speed. Updating the view takes time; often enforcing a single update per tick (by using tick-based updates) will make your model faster. On the other hand, continuous updates will be faster when running a single tick is faster than drawing a frame of the model. Most models run faster under tick-based updates, but for an example of a model which is faster with continuous updates see the "Heroes and Cowards" library 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.

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.

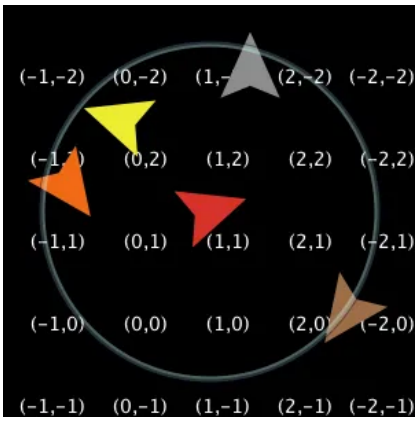
A box does not wrap in either direction. The world is bounded so turtles that try to move off the edge of the world cannot. Note that the patches around edge of the world have fewer than eight neighbors; the corners have three and the rest have five.

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.

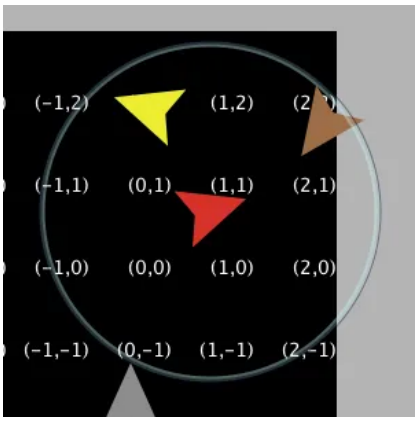
> **Code Example:** Neighbors Example

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:



> **Code Example:** Termites Perspective Demo (torus), Ants Perspective Demo (box)

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 left corner (min-pxcor, min-pycor) and the upper right 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 `can-move?` 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 `can-move?`. If you need to do something smarter that simply turning around it may be useful to use `patch-at` with `dx` and `dy`.

```
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.

Observer 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. `watch-me` is also a patch or link command. 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 (or patch or link) 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. Use the `reset-perspective` command to stop the observer from watching, following, or riding. In the 3D view, the observer also returns to its default position (above the origin, looking straight down).

> **Code Example:** Perspective Example

Input/Output

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.

How Output Primitives Differ

This information is a quick reference for more advanced users.

The `print`, `show`, `type`, and `write` primitives differ on the following facets:

- What types of values does the primitive accept?
- Does the primitive output a newline at the end?
- Are strings output with quotes surrounding them?
- Does the primitive output the agent which printed it?

The following table summarizes the behavior of each primitive.

Primitive	Acceptable values	Adds newline?	Strings quoted?	Outputs `self`?
`print`	any NetLogo value	yes	no	no
`show`	any NetLogo value	yes	yes	yes

Primitive	Acceptable values	Adds newline?	Strings quoted?	Outputs `self`?
`type`	any NetLogo value	no	no	no
`write`	boolean, number, string, lists containing only these types	no	yes	no

User Interaction Primitives

NetLogo features several primitives which allow a model to interact with the user. These primitives include user-directory, user-file, user-new-file, user-input, user-message, user-one-of, and user-yes-or-no?.

These primitives differ in precisely what interaction they take with the user. `user-directory`, `user-file`, and `user-new-file` are all reporters which prompt the user to select an item from the file system and report the path of the selected item to NetLogo. `user-yes-or-no?`, `user-one-of`, and `user-input` all prompt the user to provide input in the form of text or a selection. `user-message` simply presents a message to the user.

Note that all active forever buttons will pause when one of these primitives is used and will resume only when the user completes the interaction with the button.

What does "Halt" mean?

The primitives which prompt the user for input, as well as `user-message` all provide a "Halt" button. The effect of this button is the same for all of these primitives - it halts the model. When the model is halted all running code is stopped, including buttons and the command center. Since halting stops code in the middle of whatever it happened to be doing at the time it was halted, you may see strange results if you continue to run the model after a halt without setting it up again.

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.

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]
  [ n -> crt n
    show (word "created " n " turtles") ]
=> created 2 turtles
=> created 4 turtles
=> created 6 turtles
```

In the block, the variable `n` holds the current value from the input list.

Here are some more examples of **`foreach`**:

```
foreach [1 2 3] [ steps -> ask turtles [ fd steps ] ]
;; turtles move forward 6 patches
foreach [true false true true] [ should-move? -> ask turtles [ if should-move? [ 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 the variable named in the anonymous procedure (`x` in the examples below) to refer to the current item in the list.

Here are a couple more examples of **`map`**:

```
show map [ x -> x < 0 ] [1 -1 3 4 -2 -10]
;; prints [false true false false true true]
show map [ x -> x * x ] [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 **anonymous procedures**. Anonymous procedures are explained in more detail in Anonymous procedures, 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 [ [a b] -> [size] of a < [size] of b ] 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 [ the-turtle ->
  ask the-turtle [
    ...
  ]
]
```

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.

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

Mathematics

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.

> **Code Example:** Random Seed Example

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.

Saving a Random Seed for a Run

If you want to know what the random seed used by a "run" of a model was, you can add some simple code to track it. Then if an interesting behavior emerges or an intermittent error condition occurs, you have a way to reproduce the run by grabbing the seed and re-using it.

Add a `starting-seed` global variable, then use `new-seed` to give it a value after you use `clear-all` in your `setup` procedure. Then give that value to the `random-seed` command so it will be used for the rest of the run.

```
globals [starting-seed]

to setup
  clear-all
  set starting-seed new-seed
  random-seed starting-seed
  ; ... rest of normal setup code
end

to go
  ; ... the rest of normal model code will use the random seed set in `setup`
end
```

You can then add a monitor for the `starting-seed` global or simply output it to the command center whenever you want to know its value. If you are using BehaviorSpace to run experiments, you can also include it in the output to be able to later reproduce runs that it generates.

Plotting

NetLogo's plotting features let you create plots to help you understand what's going on in your model.

Most plots can be created by creating and editing plot widgets in the Interface tab, as described in its Plots Section. However it is possible to control plots from procedures, although you need to create them in the Interface tab first.

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-background-color`, `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

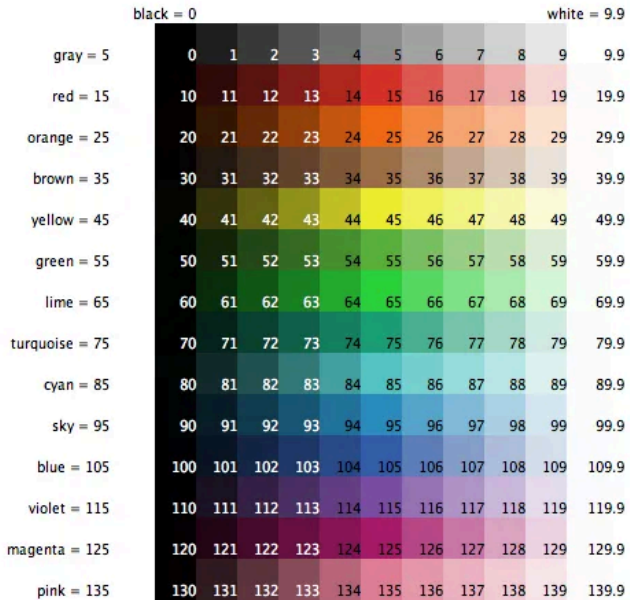
Not every aspect of NetLogo's plotting system has been explained here. See the Plotting section of the NetLogo Dictionary for information on additional commands and reporters related to plotting.

Many of the Sample Models in the Models Library illustrate various advanced plotting techniques. Also check out the following code examples:

> Code Examples: Plot Axis Example, Plot Smoothing Example, Rolling Plot Example

Color

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 NetLogo colors.



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.

See the color section of the NetLogo Dictionary for a full list of color commands.

> **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. 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. Patches only use the alpha value of an RGBA `pcolors` in NetLogo 3D, it's ignored in 2D NetLogo. 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]
```

Note that decimal values will be rounded towards 0 before the range is checked. An error will be thrown if a value is outside the range 0-255 after this rounding.

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:

```
let my-color approximate-rgb 0 0 255 ;; my-color is now 104.7
show extract-rgb my-color ;; shows [48 88 161] which is pretty far from [0 0 255], the color we
started with
```

This is an approximation, but it is still near NetLogo blue which is 105.

Using RGBA colors, the user has more options when it comes to an agent's color.

Example: you can change any turtle from its existing NetLogo color to a sixty percent transparent version of that color using:

```
ask one-of turtles [set color lput 102 extract-rgb color]
```

Note: because 255 is fully opaque and sixty percent transparent is equal to forty percent opaque, the correct alpha value is 255 * .4 = 102. See the palette extension section below for an alternate implementation.

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

Palette Extension

The palette extension offers primitives that give the user more control over colors. It allows for direct manipulation of RGBA and HSB components without the requiring list manipulations.

Example: changing the transparency of any turtle to sixty percent transparent with the palette extension:

```
extensions [palette]
ask one-of turtles [palette:set-transparency 60]
```

The extension offers similar commands for Red, Green, Blue, Alpha, Hue, Saturation, and Brightness.

See the documentation of the palette extension for more information.

> **Code Examples:** Palette Example, Color Bubbles, Color Painting, Color Reveal

Color Picker dialog

The Color Picker dialog helps you experiment with and choose colors. Open it by choosing Color Picker on the Tools Menu.

Movies

This section describes how to capture an ".mp4" movie of a NetLogo model.

First, use the `vid:start-recorder` command to start the video recorder.

To add a frame to your movie, use either `vid:record-view` or `vid:record-interface`, depending on whether you want the movie to show just the current view, or the entire Interface tab. In a single movie, the resolution will be one of the following:

- The resolution specified in the call to `vid:start-recorder` *`width height`* if you specified the resolution. These are optional parameters.
- The resolution of the view if you did not specify a resolution in the call to `vid:start-recorder` and call `vid:record-view` before calling `vid:record-interface`
- The resolution of the interface if you did not specify a resolution in the call to `vid:start-recorder` and call `vid:record-interface` before calling `vid:record-view`

Note that if the resolution of a recorded image doesn't match the resolution of the recording it will be scaled to fit which can result in images which look blurry or out-of-focus.

When you're done adding frames, use `vid:save-recording`. The filename you provide should end with `.mp4`, the extension for MP4-encoded movies (playable in QuickTime and other programs).

```
;; export a 30 frame movie of the view
extensions [vid]

;...

setup
vid:start-recorder
vid:record-view ;; show the initial state
repeat 30
[ go
  vid:record-view ]
vid:save-recording "out.mp4"
```

A movie will play back at 25 frames per second. To make the movie playback faster or slower, consider using a video postprocessing tool.

To check whether or not you are recording, call `vid:recorder-status`, which reports a string that describes the state of the current recorder.

To throw away the movie currently being recorded, call `vid:reset-recorder`.

> **Code Example:** Movie Example

Movies generated when running headless, or by background runs in a parallel BehaviorSpace experiment may use only `vid:record-view` primitive. Movies generated in NetLogo GUI may also use `vid:record-interface` and `vid:record-source`.

NetLogo movies are exported as H.264-encoded MP4 files. To play an MP4 movie, you can use the VLC Player, a free download from the VideoLAN organization.

Movies 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.

NetLogo 7 Introduction

The bolded items in the table of contents are the most important changes in NetLogo 7.0.0

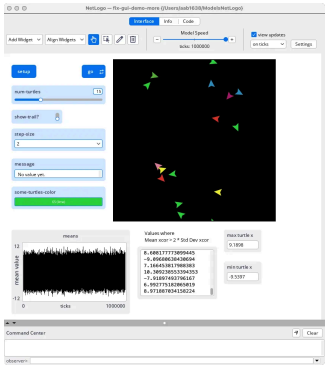
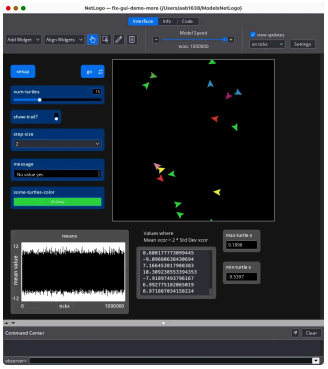
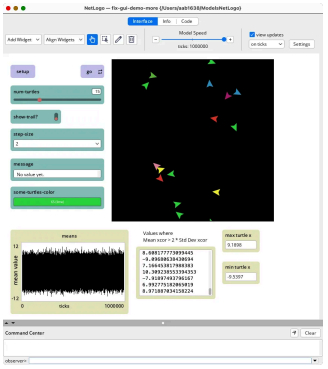
Motivation

We interviewed many NetLogo users and one of the most desired updates was a more modern User Interface (UI). In response to user feedback we have made major changes to the UI for NetLogo 7.0.0. We plan to continue modernizing NetLogo in future releases. Your feedback via the NetLogo Forum will help guide UI improvements.

Interface Tab

Overview of Visual Changes to the Interface Tab

The most obvious change is that interface elements (widgets) have changed in appearance to a modern flat look. They are larger and more legible. A new feature is color themes which affect the appearance of the entire visual interface. They have been designed with visual coherence in mind. Navigate to the Color Themes tab in the Preferences window to select Light, Dark or Classic themes. For the Classic Color Theme the pre-7.0.0 widget colors are used. In other regards it is the same as the Light Theme. The Color Theme also affects the appearance of the Info Tab and the Code Tab, as well as any other windows that open in the app.

Light Theme	Dark Theme	Classic Theme
		

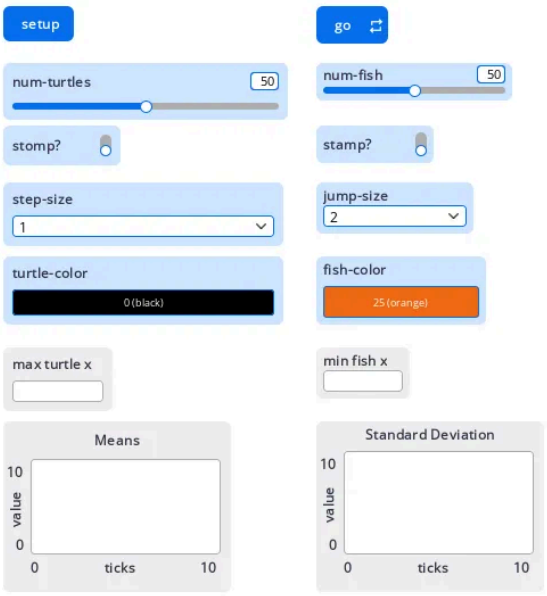
Interface Elements (Widgets)

Buttons perform actions when clicked. The input widgets are: sliders, switches, choosers and inputs. The output widgets are: monitors, plots and the output area. Notes are used to add labels, instructions and other information to the interface. Buttons and input widgets are colored shades of blue. Output widgets are shades of grey. Widgets by and large perform as they did in previous releases of NetLogo. A new feature of the slider is a text box in which values can be added directly. A new feature of the Notes is that the user can set both the text color and the background color for both light and dark mode.

A new feature is copying widgets.

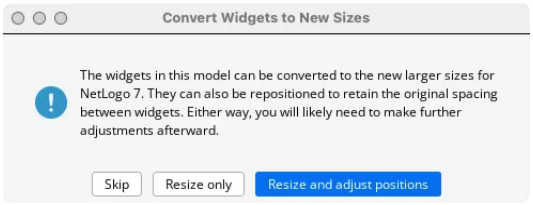
Widget Sizes

In NetLogo 7 some widget sizes have been adjusted for greater readability. The default height of sliders, switches, choosers, inputs, monitors and plots has been increased. The image below shows some widgets in their new sizes on the left, and the original/pre-NetLogo 7 sizes on the right.

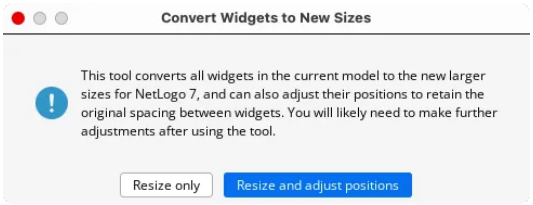


When you create a new model, widgets will appear by default in their new (generally larger) sizes. Existing models will open with new style widgets that retain their original size. Some widgets in the Models Library have been converted to the new sizes.

Note that opening an old model in NetLogo 7 will automatically prompt you to perform an automatic widget conversion, as shown in the dialog below.



Click “Resize and adjust positions” to resize the widgets and change their placements. Click “Resize only” to resize the widgets without changing their location. Since redesigning interfaces is an art, more work may be required to make the design meet your personal standards. The Edit → Undo menu will let you undo all the widget conversions at once if you are not satisfied with the result of the automatic conversion. If you choose not to make the full conversion, you can later use the tools menu item *Tools → Convert Widgets to New Sizes*.



You can also switch the size of an individual widget by clicking the *Use old sizing* checkbox. This only holds for widgets that have undergone a size change for NetLogo 7. The size of the output area and notes have remained the same.

New File Format

Models in NetLogo 7 have a new format which is extensible and makes it easier to add new features to NetLogo. The new file format follows the XML 1.0 standard, and is compatible with any external tools for working with XML. Models saved with this format will have the extension .nlogox or .nlogox3d, rather than .nlogo or .nlogo3d. All models going forward will use the new file format. Existing model files, with the extension .nlogo or .nlogo3d, can be opened in NetLogo 7 and will be automatically converted without loss of information. However .nlogox and .nlogox3d files cannot be opened with previous versions of NetLogo. Any tools users have created that rely on the format of .nlogo and .nlogo3d files will need to be re-written.

Tools in the NetLogo 7 Interface

Overview: NetLogo Interface Tool Buttons and Dropdown menus

New Interface Tools

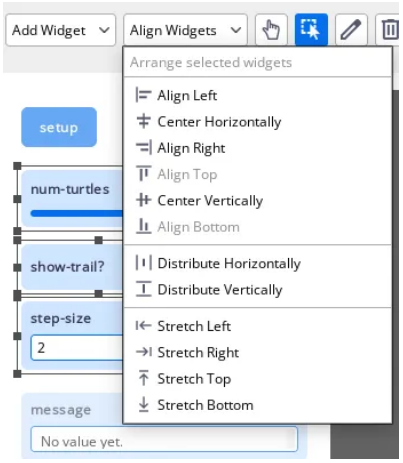
The key motivations for introducing these tools are

1. to make user actions more discoverable
2. to separate actions needed to use an existing model from those used when creating or modifying a model.

The Interface toolbar includes four new tool buttons which control the meaning of user actions. They are the Interaction Tool, the Selection Tool, the Edit Tool, and the Delete Tool. For a brief explanation of the use of these tools see this section of the Interface Tab Documentation. For details on the use of these features click the links above.

New Interface Menu

There is a new Align Widgets Menu. This menu is only active if more than one element is selected. Only one of a set of related options will be explained. Use the link above for more details.



With Align Left the left-most widget remains in place. The other widgets move to align to left-most widget edge.

Distribute Vertically keeps the top and bottom widgets fixed, and moves the other widgets so the vertical spacing between the widgets is the same. Distribute horizontally works similarly.

With Stretch Left the left-most widget remains in place. All the other widgets stretch so they have the same left-most edge.

The Color Picker

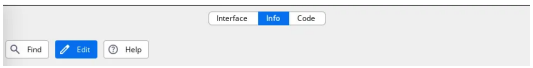
Color Swatches has been replaced by the Color Picker, a new, more powerful way of selecting colors.

Notifications

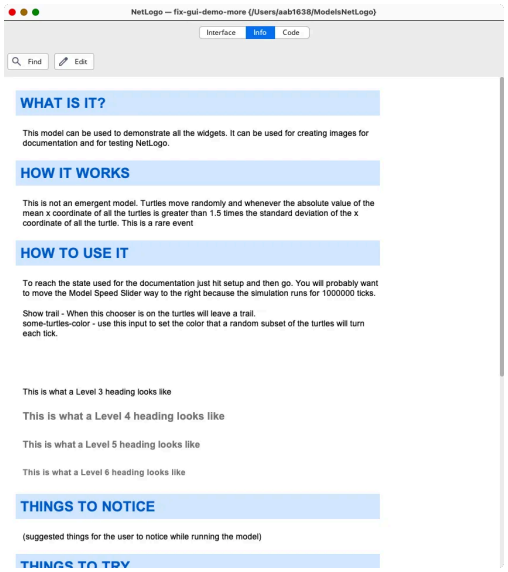
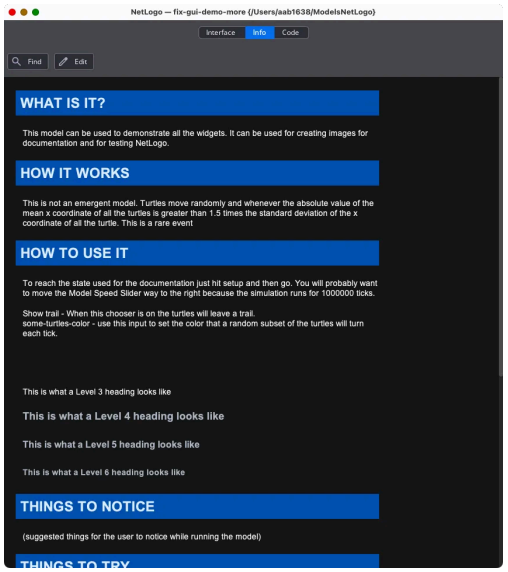
When the NetLogo team sends out a new notification message a banner will appear below the Interface Toolbar with a brief message. If you click on the message you will be taken to a web page that gives the full information of all messages to date in reverse chronological order. You can also click the X button on the right to dismiss the message.

The Info Tab

The changes to the Info Tab are purely visual. The Find and Edit icons and the section headers have been updated, as has the Help Icon that is visible when editing the Info Tab.



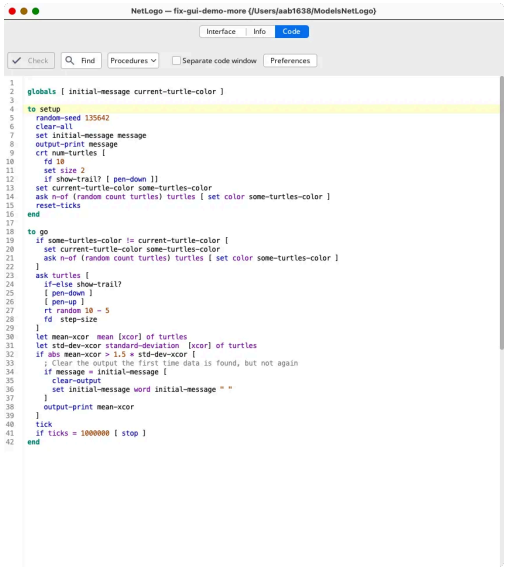
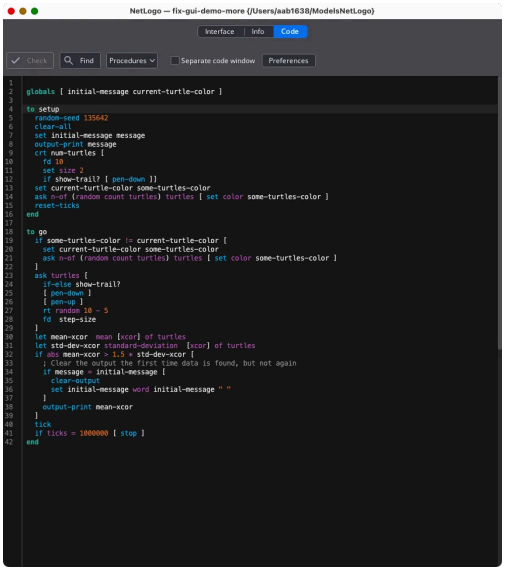
These are the new Info Tab color themes.

Light Theme	Dark Theme
	

Code Tab

Color Themes and Code Syntax Highlighting

The code syntax highlighting colors have changed slightly for the Light Color Theme, and there is a separate set of code colors for the Dark Color Theme.

Light Theme	Dark Theme
	

Code Tab Toolbar

The Code Tab toolbar features new icons. See Code Tab Toolbar for more details.

Code Tab Preferences

Code related preferences are now set in the Code tab of the Preferences dialog. See Code Preferences for details.

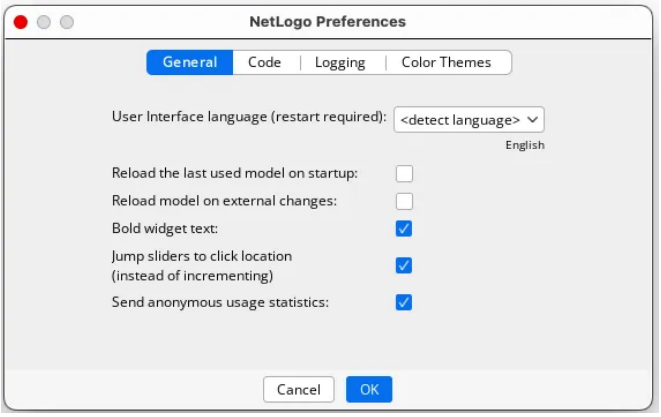
Included Files

An included (.nls) file tab now has separate code window checkbox, and a Preferences button. Note that these buttons and preferences apply to all code tabs, not just the one from which they were clicked/set. Included file tabs can be closed by clicking on the “x” on the right side of the tab. More details are in the Included Files documentation.

Menu Items

Preferences Dialog

This dialog is now organized into four tabs. You use it customize various NetLogo settings. On Windows and Linux, it is opened with the Tools menu “Preferences” item. For Macs, it’s under the “NetLogo” menu and reads “Settings”.



Here is a brief description of the new preferences. See the Preferences Documentation for more details.

Reload model on external changes: If you use an editor outside of NetLogo to change the model or any included .nls files the model will be reloaded.

Bold widget names: By default, widget names are not in bold, use this option to make them bold.

UI Scale: This option is for Linux only. It must be set to the same value as your system UI scale or NetLogo will fail to resize its visual components correctly.

Auto-focus text or tab with errors when model is compiled: If checked, NetLogo will bring you to the location of an error if one occurs during compilation.

Open code tab in separate window on startup: When checked the Code tab will appear in a separate window when NetLogo is opened.

Jump sliders to click location (instead of incrementing): When checked, clicking anywhere in the slider bar of a slider widget will move the thumb to that location. When unchecked, the pre-NetLogo 7 behavior will be applied, and clicking in the slider bar will increase or decrease the value of the slider by one increment, depending on which side of the bar is clicked.

File → Resource Manager dialog

The Resource Manager dialog allows you to view, add, or remove files that are bundled with the current NetLogo model. These files can be read during a model run using the Resource Extension.

Tools Menu Changes

The new menu items “Color Themes”, “Color Picker” and “Convert Widgets to New Sizes” have been described in earlier sections. The menu item “Color Swatches” has been removed.

The Turtle Shapes Editor and the Link Shapes Editor have been updated, as described below.

Turtle Shapes Editor and Link Shapes Editor

These Shapes Editors now have buttons at the top, for easier use. A search field has been added so users can quickly search through the model’s collection of shapes.

Help Menu

The Help Menu now has a link to the NetLogo Forum, a website to ask/answer questions and have discussions about NetLogo.

Collection of anonymous usage statistics

When you first open NetLogo after installing it, a dialog titled “Data Consent” will appear. If you select “Yes”, data will be sent to the NetLogo team. We analyze the data in order to understand how often certain NetLogo features are being used. This informs development priorities. No personal or identifying information will be collected. You can always change your preference by using the **Send anonymous usage statistics** option of the **NetLogo Preferences Dialog** to enable or disable data collection. If you have any questions or concerns about the collection and storage of this data, feel free to contact the NetLogo team at **netlogo@northwestern.edu**.

NetLogo Preferences

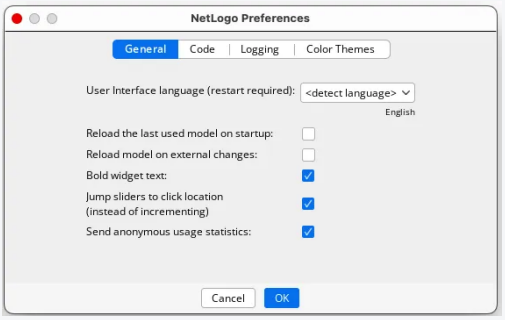
Overview

This dialog lets you customize various NetLogo settings. On Windows and Linux, it is opened with the Tools menu "Preferences" item. For Macs, it's under the "NetLogo" menu and reads "Settings".

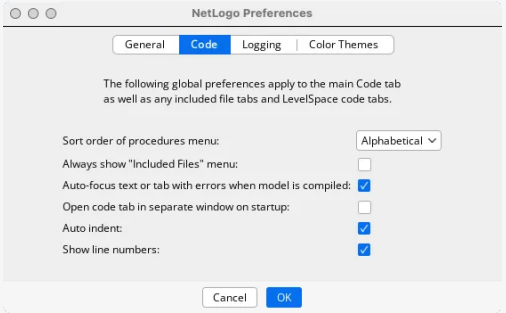
Items are organized into four tabs:

- General Preferences
- Code Preferences
- Logging Preferences
- Color Themes

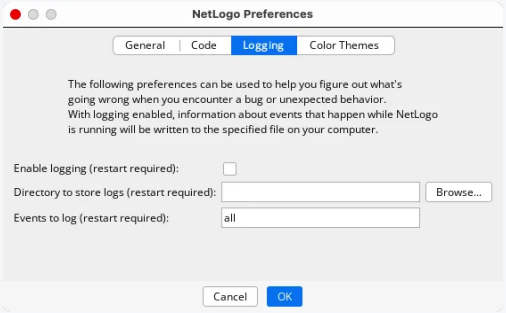
General Preferences



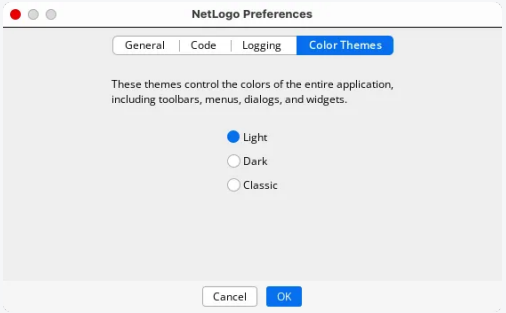
Code Preferences



Logging Preferences



Color Themes



Use the check boxes, pull-down menus and text fields to change the default preference values. When you are done, click the OK button to save your changes, or the Cancel button to revert the preferences to the state they were in before you opened the dialog. Most preferences take effect immediately. You must exit and re-open NetLogo for the preferences that indicate "(restart required)" to take effect.

General Preferences

User Interface language

Choose a language for user interface items and error messages. NetLogo must be restarted for this preference to take effect. The default option is “Detect Language”, which will set the interface language to your system language, or English if your system language is not supported by NetLogo. The other available choices are English, Spanish, Chinese, Portuguese, Russian, and Japanese.

Reload the last used model on startup

NetLogo opens with an empty model by default. When this option is selected the most recently opened model will be loaded when NetLogo starts instead of an empty model.

Reload model on external changes

If you use an editor outside of NetLogo to change the model or any included .nls files, the changed file is updated in NetLogo.

Enable remote commands

When this option is enabled, NetLogo listens for commands to execute on a Unix Domain Socket. Received commands are executed in the observer context as if they were entered in the Command Center. This can be used by external software such as editor plugins for better integration with NetLogo.

If the environment variable `XDG_RUNTIME_DIR` is set, the listening socket is created under `XDG_RUNTIME_DIR`. Otherwise, it is created under the NetLogo directory:

- On Mac OS X: `Library/Application Support/NetLogo/7.1`
- On Windows: `AppData\NetLogo\7.1`
- On Linux: `.netlogo/7.1`

The listening socket is named using the following pattern `netlogo-<PID>` where `<PID>` is the process ID of the associated NetLogo process.

To send a command to NetLogo, encode the command in a JSON object as follows, replacing `<COMMAND>` with the command. The resulting string should then be written to the listening socket.

```
{
    "type": "nl-run-code",
    "code": "<COMMAND>"
}
```

Once the command is received and processed, NetLogo will respond with the following JSON object. If the command is successfully parsed and executed, `<STATUS>` will be `ok`. Otherwise, it will be a message describing the error that occurred. Note that this will only report errors that occur during the processing of the request. Runtime errors that occur as a result of executing the provided command are not reported here.

```
{
    "type": "nl-status",
    "status": "<STATUS>"
}
```

Bold widget text

By default, widget text is not in bold, use this option to bold it.

Jump sliders to click location (instead of incrementing)

By default, clicking a slider location will cause the value of the slider to be set to the value at the click location and the slider widget thumb will jump to that location. Deselecting this option will cause the value to change by one increment towards the mouse click, also moving the widget thumb. This option applies to slider widgets and the Speed Slider.

UI Scale

This option is for Linux only. It must be set to the same value as your system UI scale or NetLogo will fail to resize its visual components correctly. This value must be a floating point number, or NetLogo will produce an error message.

Send anonymous usage statistics

By default NetLogo will now collect anonymous usage statistics from users while the application is running. Uncheck this option to disable data collection. When this option is enabled, NetLogo will collect information about your usage of tools and features of the NetLogo application and the NetLogo language. This information is analyzed by the NetLogo team in order to understand how often certain NetLogo features are being used, so as to inform development priorities. No personal or identifying information will be collected. If you have any questions or concerns about the collection and storage of this data, feel free to contact the NetLogo team at netlogo@northwestern.edu.

Code Preferences

Sort order of procedures menu

Choices:

- Alphabetical (default)
- Order of appearance

Always show “Included Files” menu

By default the “Included Files” menu is only shown when the Code Tab has an “includes” statement. With this option the “Included Files” menu will always be in the Code Tab toolbar. That menu allows user to select and open an included file or choose the **New Source File** or **Open Source File** options.

Auto-focus text or tab with errors when model is compiled

If checked, NetLogo will bring you to the location of an error if one occurs during compilation.

Open code tab in separate window on startup

When checked the Code tab will appear in a separate window when NetLogo is opened.

Auto Indent

When selected, indentation in the Code Tab will be determined by the compiler, instead of by manually pressing the Tab key to change the indentation of a line.

Show line numbers

When selected line numbers will be shown on the left edge of the Code Tab.

Show auto-complete popup while typing

When selected, a code completion popup will automatically appear as you type in the Code tab. Whether or not this preference is enabled, the popup can always be opened by pressing Control + Space.

Code Font

This preference allows you to set the font to use wherever NetLogo would normally display a monospaced font (usually for code). The list of font options is provided by your operating system, so you can add third-party fonts to the list by installing them to your operating system.

The places where the selected font will be used include:

- Code tab editor
- Command Center input
- Command Center output
- Button dialog editor
- Slider dialog editors
- Chooser dialog editor
- Input widget inputs (when Number, String (Reporter), or String (Command))
- Monitor dialog editor
- Plot dialog editors
- Output widget output
- Info tab editor
- Info tab rendered content (e.g. the rendering of `my favorite code`)
- All editors in agent inspection windows

Logging Preferences

Enable logging

Create a log file containing user actions. This option can be useful for debugging and research. NetLogo must be restarted for this preference to take effect. If logging is enabled you must also specify a log directory, or NetLogo will produce an error message.

Directory to store logs

Specifies the directory to which log files will be written. You can type in the directory name, or select a directory using the Browse... dialog. NetLogo must be restarted for this preference to take effect.

If logging is enabled one of the following may happen.

- If you type in a non-existent directory you will see a dialog that says “The specified log directory does not exist, would you like to create it?” with options “Yes” and “No.”
- You must have write permission for the log directory, otherwise you will get the message “NetLogo does not have permission to write to the specified log directory.”

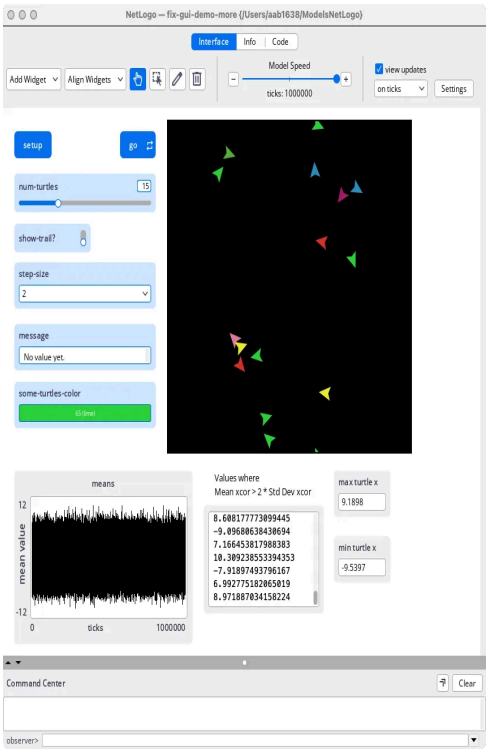
Events to log:

Enter a comma-separate list of the event names or “shortcut” names. If you leave this value blank with logging enabled a set of default events will be logged. For detailed options see the Events section of the Logging documentation. The “all” option causes all the events be logged, including button, chooser, comment, compile, command-center, global, input-box, link, model-open, slider, speed-slider, start, switch, stop, turtle, widget-edit, and tick events.

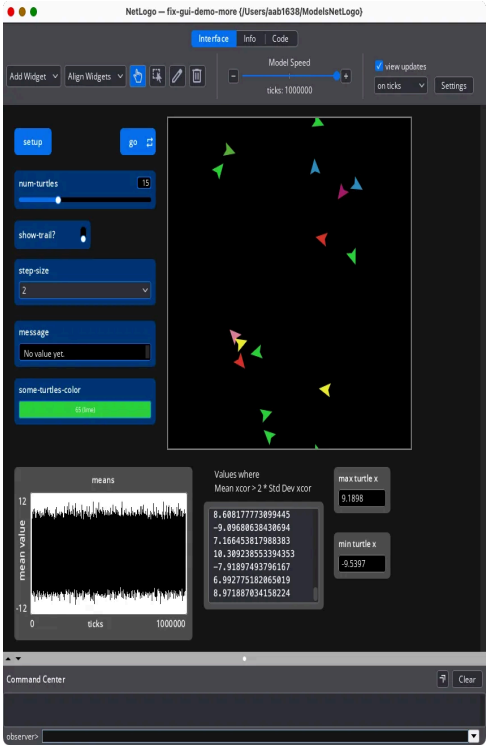
Color Themes

Use the radio buttons to change the color theme. See the images below for an example of each theme.

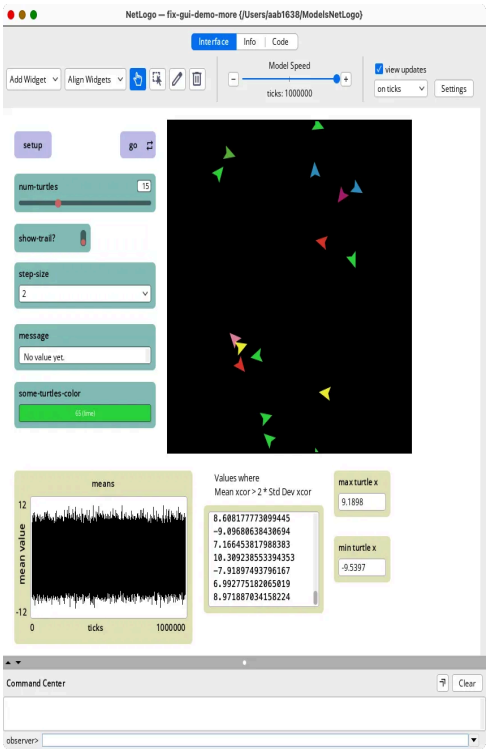
Light



Dark



Classic



Automatic

If this option is selected, NetLogo will use either the Light theme or the Dark theme, based on the system setting.

Transition Guide

Many models created in earlier versions of NetLogo also work in NetLogo 7.0. 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 7.0.0

See the following page for a list of important changes since NetLogo 6.4: NetLogo 7.0.0 Changes Overview.

Changes directly affecting models are the new file format and widget size conversion.

Changes for NetLogo 6.1.0

CF Extension Removal

The CF extension is no longer bundled with NetLogo. The multi-conditional `cf:ifelse` and `cf:ifelse-value` primitives are now now supported directly in NetLogo without an extension by `ifelse` and `ifelse-value`. Simply remove the `cf:` from the primitives in your models and they should work as they did before with CF.

`ifelse-value` Precedence Change with Infix Operators

In previous versions, `ifelse-value` expressions would be parsed before infix expressions they were a part of, like `+` or `-`. Now `ifelse-value` has a lower precedence to let it work with a variable number of clauses.

This used to work, but will now complain about finding a block or list where it wanted a number, since the `+` is parsed before the `ifelse-value`:

```
ifelse-value (true) [10] [3] + ifelse-value (false) [5] [7]
```

Now you just need to wrap your `ifelse-value` in parenthesis to tell NetLogo that you want to add the separate expressions:

```
(ifelse-value (true) [10] [3]) + (ifelse-value (false) [5] [7])
```

Changes for NetLogo 6.0.3

Arduino Extension Changes

The `arduino` extension has substantially changed the way it receives values from Arduino boards. Please consult the arduino example sketch (included within the "models" folder under Chapter 8 of the IABM textbook) for an updated sketch compatible with the Arduino example model. If you have an existing arduino sketch, you will need to adjust the format used to send values to NetLogo. Old sketches will have code which sends back messages like ";A,2.5;". For the new version of the `arduino` extension to receive the same message, sketches should send ";A,D,2.5;" instead. The added 'D,' informs Netlogo that the value being sent is a number and not a string.

CF Extension Changes

All primitives have been removed from the `cf` extension and replaced by the `cf:ifelse` and `cf:ifelse-value` multi-branch primitives. We encourage existing `cf` users to adjust their code to use these new primitives. Over time, we hope to use these primitives as an example on which to remodel NetLogo's existing `ifelse` and `ifelse-value` primitives. Because this would be a relatively large language change, we would love to hear any **feedback address** you may have from using these `cf` extension primitives.

Changes for NetLogo 6.0

Tasks replaced by Anonymous Procedures

In NetLogo 6.0, tasks have been replaced by anonymous procedures. This means that `task` is no longer a primitive, it's been replaced by the new **arrow syntax** for creating anonymous procedures. Similarly, question mark variables like `?`, `?1`, and `?2` are now just ordinary names in NetLogo and can be used to name procedure variables, let variables, or anonymous procedure variables. Finally, `is-reporter-task?` and `is-command-task?` have been replaced by `is-anonymous-reporter?` and `is-anonymous-command?`.

To make this transition easier, we've added an automatic conversion step which should allow most models saved in NetLogo 5 to be converted to use the new syntax automatically. The autoconverter has been a substantial piece of effort and we've tested it on all the models in the models library. To use it, ensure the model compiles and run properly in NetLogo 5 or later, then save it from NetLogo 5 or later. Then, simply open the model in NetLogo 6. If all goes well, you'll see the converter has changed code like `task [?1 + ?2]` to `[ [?1 ?2] -> ?1 + ?2 ]`. The question marks are meant to serve as temporary placeholders for conversion. They enable your model to run, but you can (and should) replace these variables with meaningful names. If you open a model with tasks and it has *not* been converted, the autoconverter wasn't able to convert your model. Rather than attempt to autoconvert your model and break something, the model will open, you will be shown the appropriate errors and given a chance to edit your model.

While we have tested the autoconverter thoroughly, we expect there to be some cases it doesn't cover.

- If you make use of extensions that aren't yet compiled for NetLogo 6, the autoconverter will not work until those extensions have been updated.
- If your code uses `run` or `runresult` to evaluate strings containing tasks, the autoconverter will not change those strings to be anonymous procedures. To make `run` and `runresult` work as expected, look at strings in your model and change any which rely on `task` or `?`-variables to instead rely on anonymous procedures

If your model doesn't fall into the above categories and doesn't convert or converts incorrectly, please email our **feedback address** and we'll be happy to offer whatever assistance we can.

Link reporters overhauled to be more consistent and flexible

In previous versions of NetLogo, link reporters have had a number of inconsistencies regarding directed and undirected links. For example, `my-links` would report *all* links connected to a turtle, whereas `link-neighbors` would only report neighbors connected by *undirected* links. Furthermore, it was quite difficult to work with models where the links could either be directed or undirected.

To alleviate these issues, the link primitives have been overhauled in 6.0. These changes only affect existing models that use both directed and undirected links while also using the unbreeded link primitives.

The changes are as follows:

- Link reporters that contain the word “out” now utilize both directed, outgoing links and undirected links. That is, they now specify links that can be used to get from the current node to other nodes. For example, `out-link-neighbor?` will report `true` if the current turtle is connected to the given turtle by either an outgoing directed link or an undirected link. If you only want directed, outgoing links, you can use a breed-specific reporter or `my-out-links with [ is-directed? self ]`. Generally, when working with models that have both directed and undirected links, you will probably want to use the “out” primitives for most things now.
- Link reporters that contain the word “in” now utilize both directed, incoming links and undirected links. That is, they now specify links that can be used to get to the current node from other nodes. For example, `in-link-neighbor?` will report `true` if the current turtle is connected to the given turtle by either an incoming directed link or an undirected link. If you only want directed, incoming links, you can use a breed-specific reporter or `my-in-links with [ is-directed? self ]`.
- Link reporters that do not specify “out” or “in” utilize *all* links.

Furthermore, there are no longer restrictions regarding which reporters can be used with which breeds. For unbreeded links, this makes it possible to use the same primitives regardless of whether your network ends up being directed or undirected.

The new behavior (including all changed primitives) is summarized by the following table, where “un” refers to undirected links, “out” refers to directed, outgoing links, and “in” refers to directed, incoming links.

New link reporter behavior

	un	out	in	un-breeded	out-breeded	in-breeded
my-links	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
link-neighbors	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
link-neighbor?	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
link-with	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
my-out-links	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
out-link-neighbors	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
out-link-neighbor?	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
out-link-to	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
my-in-links	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
in-link-neighbors	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
in-link-neighbor?	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
in-link-from	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
my-<breeds>	n/a	n/a	n/a	TRUE	TRUE	TRUE
<breed>-neighbors	n/a	n/a	n/a	TRUE	TRUE	TRUE
<breed>-neighbor?	n/a	n/a	n/a	TRUE	TRUE	TRUE
<breed>-with	n/a	n/a	n/a	TRUE	TRUE	TRUE
my-out-<breeds>	n/a	n/a	n/a	TRUE	TRUE	FALSE
out-<breed>-neighbors	n/a	n/a	n/a	TRUE	TRUE	FALSE
out-<breed>-neighbor?	n/a	n/a	n/a	TRUE	TRUE	FALSE
out-<breed>-to	n/a	n/a	n/a	TRUE	TRUE	FALSE
my-in-<breeds>	n/a	n/a	n/a	TRUE	FALSE	TRUE
in-<breed>-neighbors	n/a	n/a	n/a	TRUE	FALSE	TRUE
in-<breed>-neighbor?	n/a	n/a	n/a	TRUE	FALSE	TRUE
in-<breed>-from	n/a	n/a	n/a	TRUE	FALSE	TRUE

Old link reporter behavior

	un	out	in	un-breeded	out-breeded	in-breeded
my-links	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
link-neighbors	TRUE	error	error	TRUE	FALSE	FALSE
link-neighbor?	TRUE	error	error	TRUE	FALSE	FALSE
link-with	TRUE	error	error	TRUE	TRUE	TRUE
my-out-links	error	TRUE	FALSE	TRUE	TRUE	FALSE
out-link-neighbors	error	TRUE	FALSE	FALSE	TRUE	FALSE
out-link-neighbor?	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
out-link-to	error	TRUE	FALSE	TRUE	TRUE	FALSE
my-in-links	error	FALSE	TRUE	FALSE	FALSE	TRUE
in-link-neighbors	error	FALSE	TRUE	FALSE	FALSE	TRUE
in-link-neighbor?	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE
in-link-from	error	FALSE	TRUE	FALSE	FALSE	TRUE
my-<breeds>	n/a	n/a	n/a	TRUE	error	error
<breed>-neighbors	n/a	n/a	n/a	TRUE	error	error
<breed>-neighbor?	n/a	n/a	n/a	TRUE	error	error
<breed>-with	n/a	n/a	n/a	TRUE	error	error
my-out-<breeds>	n/a	n/a	n/a	error	TRUE	FALSE
out-<breed>-neighbors	n/a	n/a	n/a	error	TRUE	FALSE
out-<breed>-neighbor?	n/a	n/a	n/a	error	TRUE	FALSE
out-<breed>-to	n/a	n/a	n/a	error	TRUE	FALSE
my-in-<breeds>	n/a	n/a	n/a	error	FALSE	TRUE
in-<breed>-neighbors	n/a	n/a	n/a	error	FALSE	TRUE
in-<breed>-neighbor?	n/a	n/a	n/a	error	FALSE	TRUE
in-<breed>-from	n/a	n/a	n/a	error	FALSE	TRUE

Removal of Applets

Oracle, the company behind Java, has announced that Java applets are deprecated (see this blog post for more information). This comes as the major browsers have removed support for plug-ins (like java applets) or announced that they plan to do so.

Instead of using applets to distribute your model, NetLogo offers the option to export to NetLogo Web. While NetLogo Web doesn't yet offer the full functionality of desktop NetLogo (in particular, extensions aren't supported), it is now capable of running most of the models in the NetLogo models library and we hope that most model distributors will find that it meets their needs. To export to NetLogo Web, choose the "Save As NetLogo Web" option from the "File" menu.

Changes to the NetLogo User Interface

Users will notice several tweaks to the NetLogo User interface when opening NetLogo 6 for the first time. We've removed the bar border above the view. To open the 3D View in 6.0, you can right click on the view and choose "Switch to 3D View", or choose the same option from the "Tools" menu. Ticks are now displayed in the interface tab toolbar beneath the speed slider. To adjust the label used for "ticks" and other view properties, you can choose the "Settings" button at the far right of the interface tab toolbar or right-click on the view and choose "Edit..." from the context menu that appears.

Nobody Not Permitted as a Chooser Value

In NetLogo 6.0, `nobody` is no longer a valid chooser value. Just as you can't put `turtle 0` or `turtles`, `nobody` refers to a non-literal value which isn't supported in choosers. As part of this transition, choosers containing `nobody` (or `nobody` within a nested list) will have all uses of `nobody` changed to `"nobody"` when opened in NetLogo 6.0.

Breeds must have singular and plural names

In NetLogo 6.0, you must specify both plural and singular breed names. In prior versions, declarations like `breed [mice]` were legal, but this support has been removed in 6.0. If you have models which use only plural breed names, it is recommended that you convert them to specify both names before opening in 6.0 since doing so will permit the NetLogo converter to work most effectively on any other code in your model which needs conversion.

Removal of “Movie” Prims

The NetLogo movie prims hadn’t been updated in quite some time and generated invalid quicktime movie files. They have been replaced by prims in the new `vid` extension. The full documentation for the `vid` extension is available in the Vid Extension section of the manual. As with all extensions, users will need to include `vid` in the `extensions` section of their NetLogo model.

Many of the movie primitives have direct parallels in the `vid` extension which can be found in the following table:

movie prim	vid prim
`movie-cancel`	`vid:reset-recorder`
`movie-close`	`vid:save-recording *file-name*`
`movie-grab-view`	`vid:record-view`
`movie-grab-interface`	`vid:record-interface`
`movie-start *file-name*`	`vid:start-recorder *optional-width* *optional-height*`
`movie-status`	`vid:recorder-status`

When you first open a file in NetLogo 6.0, your file will be automatically converted to use the new primitives. This will include adding a new global variable - `_recording-save-file-name` to track the name of the active recording, as well as adding the `vid` extension to the model. You should verify that the conversion took place correctly. There is no replacement for `movie-set-frame-rate`. The vid extension records frames at 25 per second, slightly more than the default 15 frames-per-second of the movie prims. If your recording is sensitive to framerate, consider recording each existing frame twice (2/25 is fairly close to 1/15) or consider using a postprocessing tool (like gstreamer or ffmpeg) to adjust the video playback speed.

Improved Name Collision Detection

In NetLogo 6.0, expanded error-checking in the NetLogo compiler causes models which define `undirected-link-breed [ undirected-links undirected-link ]` and/or `directed-link-breed [ directed-links directed-link ]` to error for redefining a primitive reporter (either `is-directed-link?` or `is-undirected-link?`). If your model doesn’t use `is-directed-link?` or `is-undirected-link?` at all, simply changing the breed names should resolve the error.

If you used either of the `is-<directedness>-link?` prim, there are several ways you might modify your model to account for this change. If your model has no other breeded links, consider removing the link breed and using the built-in link primitives. If your model has other breeded links, but only of different directedness, simply changing the breed name (and all related primitive names) should resolve the problem. Note that in this case `is-directed-link?` and/or `is-undirected-link?` continue to behave the same as before. If your model has other breeded links of the same directedness, the change will vary depending on your model. The breed name(s) must be changed, but you must decide whether you used `is-directed-link?` / `is-undirected-link?` to check link directedness or to check that link breed membership. If you used it to check link directedness leaving it as-is should keep the current behavior of the model. Otherwise, it can simply be replaced by `is-<breed>?`.

Removal of `hubnet-set-client-interface`

The `hubnet-set-client-interface` primitive was rendered obsolete by the introduction of the HubNet client editor and end of support for calculator HubNet. We have found it used in very few models and have decided to remove it from the language. On opening an existing model in 6.0, the autoconverter should remove all uses of `hubnet-set-client-interface` from your code.

Improved & Updated Extensions API

One of our goals in NetLogo 6.0 has been to make it easier to develop extensions and easy to develop more *powerful* extensions. To that end, we've bumped the extension API from 5.0 to 6.0. Existing extensions will need to recompile changing the "NetLogo-Extension-API-Version" in their jar's MANIFEST.MF from 5.0 to 6.0.

Some of the changes we've made to the extensions API include:

- `org.nlogo.api.Context` now allows access to the current world and workspace objects without requiring a cast to an `org.nlogo.nvm.ExtensionContext`.
- `org.nlogo.api.Workspace` has been introduced as a stable API for extensions to depend on.
- A NetLogo jar is now available from BinTray.

For a full list of changes between 5.0 and 6.0, please visit our Extension Transition Guide on GitHub.

In service of making it easier to build extensions, we've expanded and improved the NetLogo Extension Plugin for `sbt`, the Scala Build Tool. Sbt is a powerful tool for building JVM projects and can be used in projects that use Scala, Java, or a combination of the two. We're now using the Extension Plugin to build all of the bundled extensions and we strongly recommend extension authors take advantage of the plugin as it makes configuring a NetLogo extension build extremely straightforward. The plugin handles fetching the NetLogo jar which extensions compile against as well as generation of a jar for the extension containing the appropriate metadata.

Add `range` primitive

A new `range` primitive was added in NetLogo 6. As "range" may appear in existing models as a procedure or variable name, we have added an autoconversion step which will ensure that these models continue to operate immediately upon opening in NetLogo 6. Existing uses of `range` in models authored before NetLogo 6 will be converted to `_range` upon first opening in NetLogo 6. Once the model opens, you can rename `_range` to suit your model.

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.

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` and `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
```

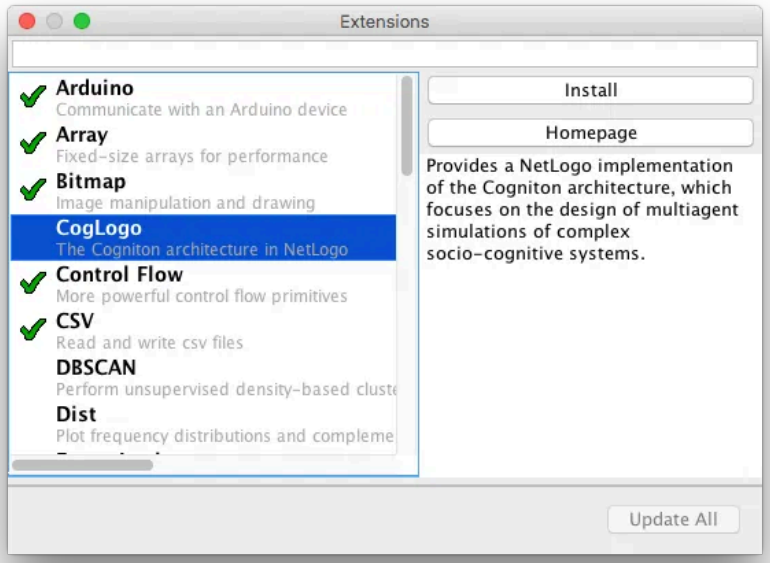
Extension Manager

Authoring and Sharing

Interested in adding some new functionality to NetLogo by creating your own extension? Have an extension you made and want to add it to the Extension Manager? See the extension authoring page to get started.

Interface

The Extension Manager can be shown by choosing “Extensions...” from the Tools menu.



On the left side of the Extension Manager window, you can see a list of all available extensions. Extensions that are already installed and up-to-date have a green checkmark next to them. Extensions that are installed but have a newer version available are marked with an orange arrow. Extensions that are not installed have no indicator. Extensions have an orange exclamation point next to them if any installation of that extension through the Extension Manager (past or future) would be overridden by a version of the extension that is present in the NetLogo installation’s `extensions` directory or in the current model’s directory.

When an extension is selected, the text on the right side provides a detailed description of the extension. Above the description are buttons for various context-sensitive operations, such as installing, updating, and uninstalling. The “Add to Code Tab” button will add the extension to the list of your model’s extensions (for example, changing the line of code `extensions [array table]` to `extensions [array table web]` when used for the Web extension). The “Homepage” button will open documentation in your web browser, which will provide you with information about the extension, a list of its primitives, and how to use them.

At the top of the window, there is a text box that can be used to filter down the list of extensions based on what you type. Along the bottom of the Extension Manager window is a button that will allow you to instantly update all of your extensions to their latest versions (if any updates are available). If NetLogo detects any extensions that have been installed outside of the Extension Manager, a “View Conflicting Libraries” button will appear at the bottom of the window, allowing you to display all of the other extensions and their locations on your computer.

Note that you can also select multiple extensions in the list with your operating system’s standard keys for performing multi-item selections (e.g. Ctrl, Shift, Command). This feature can be used to, for example, install multiple extensions all at once.

Shapes

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.

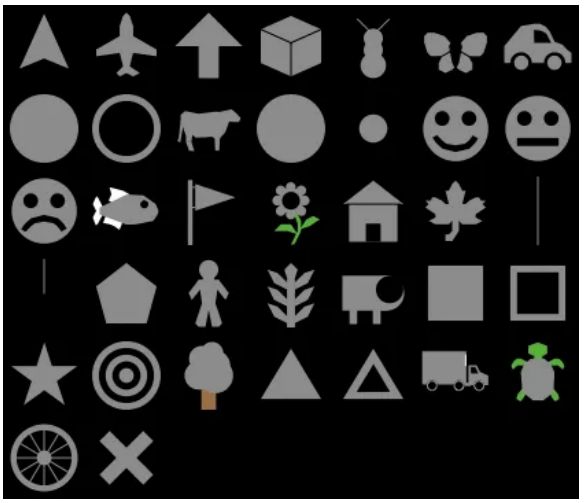
Importing shapes

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:

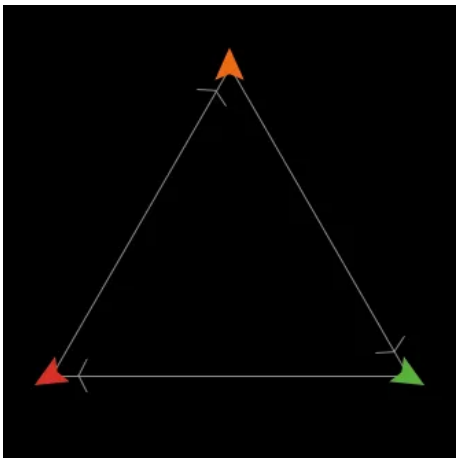


First row: default, airplane, arrow, box, bug, butterfly, car Second row: circle, circle 2, cow, cylinder, dot, face happy, face neutral Third row: face sad, fish, flag, flower, house, leaf, line Fourth row: line half, pentagon, person, plant, sheep, square, square 2 Fifth row: star, target, tree, triangle, 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

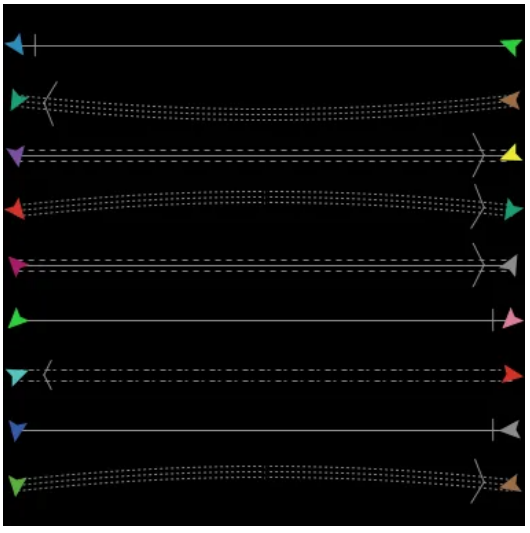
Managing link shapes is very similar to managing turtle shapes. So, you can create a new shape by pressing the **New** button or you can edit existing shapes. When you are done editing a shape press **Done** if you want to keep it.

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.

Behavior Space

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, from your own Java code, or on an HPC.

A number of new features were introduced in NetLogo 6.4:

- **Subexperiment syntax**: A syntax for allowing parameter combinations to be run separately, rather than being expanded combinatorically.
- **Run metrics when**: A reporter can be used to conditionally record measurements.
- **Pre experiment commands**: Commands can be run before the experiment begins.
- **Post experiment commands**: Commands can be run after the experiment ends.
- **Statistics output**: The mean and standard deviation of data from repetitions can be saved in an output file.
- **Lists output**: List data can be output in a file with one list element per cell.
- **Importing and exporting**: Experiments can now be exported to an XML file that can be used when running headlessly. Experiments can also be imported into a model.
- **Paused experiments**: Experiments can now be paused and resumed.
- **Run options: update plots and monitors**: Reduction in memory usage when box is unchecked.

More about changes to BehaviorSpace in NetLogo 6.4:

- Experiments created in versions prior to 6.4 can still be opened, but experiments created using new features are not backwards compatible.
- For information on format changes to output files see **Output File Changes**.
- The Experiment dialog now uses tooltips rather than text below each input element see **Creating an experiment setup**.
- Experiments using the new features can be found in the model Sample Models⇒Biology⇒Wolf Sheep Predation, with with additional notes in the Info tab.
- Hovering over the edit dialog entry titles will show a helpful 'tooltip'.
- Additional minor changes can be found by searching this page for *(Since 6.4)*.

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, you can specify how many model runs will happen in parallel. Using multiple runs in parallel will increase the memory used by BehaviorSpace.

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 flashes-to-reset — that have approximately 2000, 100, 10 and 3 possible values, respectively. That means there are 2000 * 100 * 10 * 3 = 6,000,000 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 by sampling the model's parameter space — not exhaustively, but enough so that you will be able to see relationships form between different slider values and the behavior of the system. One way to do this is to specify a subset of values from the ranges of each slider. See **Combinatorial syntaxes**. BehaviorSpace will run the model with each possible combination of those values and, during each model run, record the results. Since NetLogo 6.4 it has been possible to specify non-combinatorial sets of slider values. See **Subexperiment syntax**. 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, import, export, and run experiment setups. Experiments are listed by name and total number of model runs.

Experiment setups are considered part of a NetLogo model and are saved as part of the model, but can be also be exported as individual files. See **Importing and exporting** (*Since 6.4*).

To create a new experiment setup, press the "New" button. To edit an existing experiment setup, press the "Edit" button. The same dialog is used in both cases, what differs is the information that is already filled in. The dialog is non-blocking, which is useful if you want to copy something from the Code Tab, or view sliders in the Interface Tab (*Since 6.4*).

Creating an experiment setup

The information that can be included in the Experiment dialog is detailed below. Note that it is not necessary to specify everything; some parts can be left blank or with their default values. Hovering over the edit dialog entry titles will show a helpful 'tooltip', so you don't have to revisit this page as often *(Since 6.4)*.

Experiment name: Experiments in the same model must have different names. If you open a model that contains experiments with duplicate names, the conflicting names will be altered to ensure that all experiment names remain unique *(Since 6.4)*.

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. You may notice that the view and plots do not correspond to the values in the widgets once the experiment is completed. This is because the globals that are controlled by widgets are reset to their initial state at the end of the experiment. To synchronize your view and plots with the widgets, run your procedure that initializes the view.

Variables can also include `max-pxcor`, `min-pxcor`, `max-pycor` and `min-pycor`, `world-width`, `world-height` and `random-seed`. These are not variables, strictly speaking, 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. Setting `world-width` and `world-height` is valid under two circumstances. 1) If the origin is centered in width or height BehaviorSpace will keep it centered in that dimension. This requires that the corresponding value(s) `world-width` and/or `world-height` must be odd. 2) If one of the bounds is zero it will remain 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 remain zero and `max-pxcor` will be set to 11, 12, and 13 for successive runs. If neither of these conditions is true you cannot you cannot vary `world-height` or `world-width` directly but must 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 Programming Guide.

Combinatorial syntaxes

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.

All combinations of the specified values will be run. For example, if you have two values for a variable `a` and three values of a variable `b` six runs will result.

```
["a" 1 2]
```

```
["b" 2 4 10]
```

This would create six runs, organized as follows:

a	b
1	2
1	4
1	10
2	2

a	b
2	4
2	10

Run combinations in sequential order checkbox This box is checked by default, and causes variables specified later to vary more quickly than those specified earlier. When the box is unchecked, non-sequential order results, with variables specified earlier varying more quickly than those specified later. The non-sequential order for the variable specification above is:

a	b
1	2
2	2
1	4
2	4
1	10
2	10

Subexperiment syntax

(Since 6.4)

For more advanced users, there is a third available syntax for varying parameters, the subexperiment syntax. For example, in the Wolf Sheep Predation model, if you wanted to try two values for the `grass-regrowth-time` variable, each with its own values for the `initial-number-sheep` and `initial-number-wolves` variables, you could write it as follows:

```
[["grass-regrowth-time" 30]["initial-number-sheep" 116]["initial-number-wolves" 107]]
```

```
[["grass-regrowth-time" 40]["initial-number-sheep" 100]["initial-number-wolves" 77]]
```

Note the use of doubly nested square brackets to separate each variable within a subexperiment. This would create two runs, organized as follows:

grass-regrowth-time	initial-number-sheep	initial-number-wolves
30	116	107
40	100	77

The subexperiment syntax also allows you to define constants using the standard syntax, which will be applied to each subexperiment where they are not overwritten. To add to the example above, if you wanted to try those combinations of `initial-number-sheep` and `initial-number-wolves` but with the same value for `grass-regrowth-time`, you could write it as follows:

```
["grass-regrowth-time" 30]
```

```
[["initial-number-sheep" 116]["initial-number-wolves" 107]]
```

```
[["initial-number-sheep" 100]["initial-number-wolves" 77]]
```

This would set `grass-regrowth-time` to 30 for both subexperiments, resulting in the following runs:

grass-regrowth-time	initial-number-sheep	initial-number-wolves
30	116	107
30	100	77

You can also override a constant in a subexperiment, as in the following example:

```
["grass-regrowth-time" 30]
```

```
[["initial-number-sheep" 116]["initial-number-wolves" 107]]
```

```
[["grass-regrowth-time" 40]["initial-number-sheep" 100]["initial-number-wolves" 77]]
```

```
[["initial-number-sheep" 80]["initial-number-wolves" 153]]
```

This would produce three runs with the following combinations:

grass-regrowth-time	initial-number-sheep	initial-number-wolves
30	116	107
40	100	77
30	80	153

Note that all constants must be defined before any subexperiments.

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.

The order in which you list the variables determines the run order. 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. With sequential ordering repetitions occur in sequential runs:

a	b
1	2
1	2

a	b
2	2
2	2
1	6
1	6
2	6
2	6
1	10
1	10
2	10
2	10

With non-sequential ordering repetitions occur as a second group of runs:

a	b
1	2
2	2
1	6
2	6
1	10
2	10
1	2
2	2
1	6
2	6
1	10
2	10

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 `export-world` command. You can use reporters you have defined in the Code tab. Reporters appear as column headers. If you prefer compact headers you could replace `count patches with [ pcolor = red ]` with a reporter `red-patches` defined in the Code tab.

Run metrics 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 want to either only measure model runs at the end of the run or if you want to specify certain conditions when measurements should be taken.

Run metrics when

(Since 6.4)

This reporter will be used to determine when measurements should be recorded if they are not being recorded at every step. Measurements will be always be taken at the end of each model run, even if this text box is empty as was previously the case when the *Measure runs at every step* (now *Run metrics every step*) was unchecked. For example *ticks mod 10 = 0* would record every tenth tick, as well as the last tick. Multiple reporters can be combined using *and* and *or*.

Pre experiment commands

These commands will be run once, before the experiment begins.

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. If you want the same results each time you run an experiment, you could use something like `random-seed 473 setup` or to have different results for repetitions `random-seed (474 + behaviorspace-run-number) setup`

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.

Post run commands: These are any extra commands that you want run when each 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.

Post experiment commands

These are any commands that you want to run at the end of the experiment, after all runs have completed.

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.

Note on pre and post experiment commands: These commands execute outside of the actual runs, so they cannot affect the global variables or agents in the model (use the Setup commands and Post run commands for that). If you use primitives like file-open or the CSV extension in the pre or post experiment commands, their internal state will not carry over into the runs and errors will likely occur if used there.

Importing and exporting

(Since 6.4)

Although experiments are tied to a model and are usually saved along with a model, they can also be imported and exported individually to xml files. This allows you to easily transfer experiments between models, and also prepares experiments to be run headlessly. After an experiment is exported to an xml file, it can be edited by hand or by another script, not just within NetLogo.

The **Import** button allows you to import experiments from an xml file. The selected files may contain any number of experiments, but any experiments that are formatted incorrectly will not be loaded. If you load an experiment that has the same name as an existing experiment, the name of the loaded experiment will be slightly altered to ensure that experiment names remain unique.

The **Export** button allows you to export experiments to an xml file. Any number of experiments may be selected for export at once, but they will all be combined into a single output file.

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 choose to create data output files in two primary formats, **Table output** and **Spreadsheet output**. If one or both of these formats is selected, you can also select the supplementary **Lists output** and **Statistics output**. Each file path can be entered in its corresponding text box, or using the **Browse...* button to select a file path using the system file dialog. If you don't wish to use a particular format, you can clear the file text or click the **Disable** button and it'll be cleared for you. The frequency of data collection is determined by the settings of the **Run metrics every step** and **Run metrics when** options. For **Table output** and **Spreadsheet output** formats, the initial state of the system is recorded, after the **setup** commands run but before the **go** commands run for the first time.

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.

All four output formats will include a header section that has rows for 1) the NetLogo app version used, 2) the name of the NetLogo model file used, 3) the name of the BehaviorSpace experiment used, 4) the date and time at the start, and 5) the dimensions of the world used at the start.

Table output

This format lists each measurement step from each run in its own row, with each metric in a separate column. The measurement rows will appear in the order they happen in real time. With the **parallel runs** option the measurements may appear in a mixed order as multiple runs can happen simultaneously. To help identify which run a row belongs to, there is a column titled `[run number]` along with a column titled `[step]` that indicates the measurement number for that run. Each row also includes the values used for the variables during initial setup of the run; these values are the same for each of that run's measurement rows. **Table output** data is written to the file as each run completes.

The **Table output** format is good to use when you want to further process the data using another tool, such as importing into a database, a statistics package, or a spreadsheet application for analysis.

In the **Table output** sample image below, the header section is in red, the run number and step columns are in blue, the initial values of the variables for the run are in green, and the measurement metric data is in purple.

	A	B	C	D	E	F	G	H	
1	BehaviorSpace results (NetLogo 6.4.0)	Table version 2.0							
2	Red Queen.nlogo								
3	Beta2								
4	09/11/2023 13:01:50:312 -0400								
5	min-pxcor	max-pxcor	min-pycor	max-pycor					
6	-17	17	-17	17					
7	[run number]	initial-resistance-mean	initial-poison-me	initial-number-fro	initial-number-sn	[step]	count frogs	count snakes	aver
8	3	25	25	150	150	0	150	150	25.0
9	5	25	30	150	150	0	150	150	29.9
10	6	25	30	150	150	0	150	150	29.9
11	4	25	30	150	150	0	150	150	29.9
12	8	25	35	150	150	0	150	150	34.9
13	2	25	25	150	150	0	150	150	25.1
14	7	25	35	150	150	0	150	150	34.9
15	5	25	30	150	150	1	156	138	29.9

Spreadsheet output

This format lists the step numbers as well as each metric for each run in a separate column, with each row corresponding to a measurement step that applies to all runs. If one run finishes before another due to a **stop condition**, then its step numbers after that point will be blank. At the top of the file there is a `[run number]` row that will have the run number repeated for each metric at the top of the file. After that are rows for the initial values given to each variable for the run. The spreadsheet output also calculates the min, mean, max, and final values for the step numbers as well as each metric and lists those after the initial values before the individual run measurements. Then it lists the actual number of steps a run went through in a `[total steps]` row.

The **Spreadsheet output** data makes it easier to quickly compare runs against each other, as they will be aligned vertically when imported into a spreadsheet application. It can also make generating comparative graphs of results with initial variable changes across runs easy for the same reason.

It is important to note that **Spreadsheet output** data is not written to the results file until the experiment finishes. Since the data is stored in memory until the experiment is done, very large experiments could run out of memory. You should disable **Spreadsheet output** unless you really want it. Also, if anything interrupts the experiment no spreadsheet results will be written. Possible sources of interruptions would be runtime errors in the model, running out of memory, system crashes, or power outages. For long experiments you may want to also enable **Table output** format as a precaution so that if something happens and you get no **Spreadsheet output** you'll at least get partial data output.

In the **Spreadsheet output** sample image below, the header section is in red, the run number row is in blue, the initial values of the variables for the run are in green, the extra calculated metrics are in orange, and the measurement metric data is in purple.

	A	B	C	D	E	F	I
1	BehaviorSpace results (NetLogo 6.4.0)	Spreadsheet version 2.0					
2	Red Queen.nlogo						
3	Beta2						
4	09/11/2023 13:01:50:754 -0400						
5	min-pxcor	max-pxcor	min-pycor	max-pycor			
6	-17	17	-17	17			
7	[run number]	1	1	1	1	1	2
8	initial-resistance-mean	25					25
9	initial-poison-mean	25					25
10	initial-number-frogs	150					150
11	initial-number-snakes	150					150
12	[reporter]	[step]	count frogs	count snakes	average-poison	average-resistar	[step]
13	[final]	100	236	177	28.96516147141	30.08098402487	100
14	[min]	0	150	110	24.97936785367	24.90075680463	0
15	[max]	100	250	179	28.96516147141	30.08098402487	100
16	[mean]	50	217.3267326732	142.7722772277	26.69288690085	27.26011743633	50
17	[total steps]	100	100	100	100	100	100
18							
19	[all run data]	[step]	count frogs	count snakes	average-poison	average-resistar	[step]
20		0	150	150	24.97936785367	24.90075680463	0
21		1	154	147	25.02601750319	24.91820123228	1
22		2	156	144	25.03685627510	24.96490239985	2
23		3	162	143	25.07749519751	25.02321164654	3

Statistics output

(Since 6.4)

You can create a file with the mean and standard deviation of each numeric metric by enabling the **Statistics output**. Metrics are the reporters specified in the "Measure runs using these reporters as metrics:" box. You must also use the **Table output** and/or the **Spreadsheet output** because one of them will used to calculate the mean and standard deviation of each numeric metric across repetitions for each step. These calculations are done at the end of the experiment. The statistics are then saved in the specified file. It is important to note that metrics that produce non-numeric values such as strings are not included in the statistics. If the metrics produce lists, the statistics are calculated across elements with the same index for each list. Statistics are not calculated for metrics that produce lists that contain non-numeric elements. Also, the type of the measurement must remain constant. For example, statistics are not calculated if a metric produces measurements that can be both lists and numbers. Furthermore, the standard deviation may be "N/A", which means that there were two or fewer collected measurements for that step and parameter combination, producing a result that is not well-defined. The experiment must use at least 3 repetitions to produce standard deviation values.

In the **Statistics output** sample image below (creating using the "Red Queen" library model), the header section is in red, the parameter combinations are in green, the steps are in blue, and the statistics are in purple. Starting with column F, the statistics are the mean of "count frogs", the standard deviation of "count frogs", followed by the same statistics for "count snakes", average-poison, and average-resistance. Column E shows the step, and columns A-D show the parameter combination that was used. The statistics are calculated across all the repetitions.

	A	B	C	D	E	F	G	H	I
1	BehaviorSpace results (NetLogo 6.4.0)	Stats version 2.0							
2	Red Queen.nlogo								
3	Beta2								
4	09/11/2023 13:01:50:809 -0400								
5	min-pxcor	max-pxcor	min-pycor	max-pycor					
6	-17	17	-17	17					
7	initial-resistance-mean	initial-poison-mean	initial-number-fro	initial-number-sn	[step]	(mean) count fro	(std) count frogs	(mean) count sn	(std) count
8	25	25	150	150	0	150	0	150	0
9	25	25	150	150	1	150.6666666666	3.39934634239	149	1.4142135
10	25	25	150	150	2	152.3333333333	2.867441755680	145.6666666666	3.0912061
11	25	25	150	150	3	154.3333333333	5.792715732327	146	4.2426406
12	25	25	150	150	4	156.6666666666	6.128258770283	149	6.4807406
13	25	25	150	150	5	157.6666666666	4.189935029992	148	9.2014491
14	25	25	150	150	6	160	4.320493798938	149.3333333333	8.2596744
15	25	25	150	150	7	165.3333333333	7.586537784494	149.6666666666	9.5684667

Lists output

(Since 6.4)

This format is a supplement to the other two primary formats, as opposed to a complete data collection format. If you have any reporters that return a list, you can use the **Lists output** format to get properly formatted output for those

reporters. Data is produced for all list reporters in the "metrics" box. In both **Spreadsheet output** and **Table output** formats, lists returned by reporters will be condensed into a single cell, rather than their elements being spread out with one value per cell. If you need the list values in individual cells, select the **List output** format in addition to one or both of the other formats.

Note that the **Lists output** format, like the **Spreadsheet output** format, will not contain any data until the experiment is complete. Also note that the **Lists output** must be used in tandem with another format, it cannot be used on its own.

In the **Lists output** sample image below, the header section is in red, the reporter name is in orange, the run number and step columns are in blue, the initial values of the variables for the run are in green, and the expanded list data is in purple. The list data starts in row 7, with the headers (the index into the list, starting from 0) starting in column E.

	A	B	C	D	E	F	G	H	I
1	BehaviorSpace results (NetLogo 6.4.0)	Lists version 2.0							
2	list-example.nlogo								
3	experiment								
4	09/11/2023 13:13:58:505 -0400								
5	min-pxcor	max-pxcor	min-pycor	max-pycor					
6	-17	17	-17	17					
7	[reporter]	[run number]	num-turtles	[step]	[0]	[1]	[2]	[3]	[4]
8	[xcor] of turtles	1	3	0	-4	14	-5		
9	[xcor] of turtles	1	3	1	-4	14	-5		
10	[ycor] of turtles	1	3	0	3	15	-14		
11	[ycor] of turtles	1	3	1	3	15	-14		
12	[xcor] of turtles	2	5	0	8	-12	12	14	0
13	[xcor] of turtles	2	5	1	8	-12	12	14	0
14	[ycor] of turtles	2	5	0	6	-2	13	-1	0
15	[ycor] of turtles	2	5	1	6	-2	13	-1	0

Output File Changes

(Since 6.4)

- Output files have version numbers.
- Spreadsheet output always includes step information.
- Spreadsheet output column header `[initial & final values]` was changed to the more accurate `[final value]`.
- **Statistics output** was added.
- **Lists output** was added.

Run options: update plots and monitors

The "Run Options" dialog lets you choose whether to update plots and monitors or not. Performance is better when the box is unchecked. Note that *(Since 6.4)* if you begin the experiment with the box unchecked, you will not be able to toggle between enabling and disabling the update plots checkbox in the "Running Experiments" dialog. This gives you an even greater performance improvement than was obtained before NetLogo 6.4, when such toggling was possible. Check the box if you you need to export plot data using primitives such as `export-interface`, `export-plot`, `export-all-plots`, and `export-world`.

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. The default and recommended maximum number of parallel runs are shown below the text box *(Since 6.4)*. Your choice of number of parallel runs is remembered from experiment to experiment *(Since 6.4)*. Because each additional run increases memory usage we recommend starting with the default number of runs, and then trying more parallel runs if you want. The recommended maximum number of parallel runs is just an estimate, and some users have found that performance is significantly decreased when using the maximum.

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 or slow to complete runs as contention will build for memory between the runs themselves. If your model uses a large amount of memory, you may find that reducing the number of runs will enable the runs to complete in less time overall since work will be done by the system keeping the memory for each run available. A good rule of thumb might be to start with the default value shown in the "Run Options", and bump up or down from there to see where your "sweet spot" is for least time to complete all runs.

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**). By default NetLogo will not exceed 50% of your system's memory.

Sixth, each parallel run will get its own world for the model to run in. This world is *not* cleared automatically by BehaviorSpace if a parallel run gets re-used for another repetition, which happens quite frequently. This means, for example, if you do `ask patches [ set pcolor red ]` in one run and do not use `clear-all` or `clear-patches` in the setup commands of the next run, then the patches will all still be red. In general using `clear-all` before each run would be a best practice, but there are times when you might not want to, such as loading data from a file that doesn't change run-to-run. Just be careful with whatever data is not cleared out.

Seventh, there is a very, very small chance that at startup multiple parallel runs could wind up with the same random number generator state if they startup at the exact same moment in time. This means the runs would produce identical output for all random operations and likely the same results. This would have a chance to happen when running on very fast processors and with lots of parallel runs at once. If you need to make sure this doesn't impact your results, you can add `random-seed new-seed` to your setup commands to re-generate a new unique random seed for each run. In fact, storing the `new-seed` as a global variable so you can output it with the rest of your results would let you re-run a run later on by manually using that value to set the `random-seed`.

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 "Run metrics 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.) If you don't need to see the plots update, then use the checkboxes in the "Running Experiment" dialog to turn the updating off. This will make the experiment go faster. However, if you already disabled updating plots and monitors in the "Run Options" dialog, this checkbox will be disabled *(Since 6.4)*.

If you want to stop your experiment before it's finished, you have two options. To stop the experiment after the current runs have completed and save your progress for later, press the "Pause" button *(Since 6.4)*. To stop the experiment immediately without waiting for the current runs to complete, press the "Abort" button. Any output generated so far will still be saved, but pressing "Abort" can lead to fragmented data, so aborted experiments cannot be resumed.

When all the runs have finished, the experiment is complete. Spreadsheet, Lists and Stats output are created at this point.

Paused experiments

(Since 6.4)

Paused experiments will appear in the BehaviorSpace window marked with "In Progress". To resume an experiment where you paused it, select it and press the "Run" button. To reset a paused experiment to its initial state, select it and press the "Abort" button.

If you are using Spreadsheet output a file containing the data up until the experiment is paused will be written. This data will be used as part of the creation of a complete Spreadsheet file. Note that if your experiment is writing to its own external file you may need to make some changes in order for pausing to work correctly. For example you should use `file-flush` or `file-close` at the end of each run to ensure all the data is written to the file, and should do `file-open`

before doing any writing during a run. When opening a file in writing mode, all new data will be appended to the end of the original file, which is probably the behavior you want.

Note that moving or deleting output files before resuming a paused experiment will cause an error. Outputting new experiment data to a file associated with an existing paused experiment may also ause an error when that experiment is resumed.

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 NetLogo using the `NetLogo_Console` app with the `--headless` command line argument. This is found in the root directory of your NetLogo installation. The `NetLogo_Console` script supports the following arguments:

- `--headless`: Enable headless mode to run a BehaviorSpace experiment (required, will open the graphical interface otherwise).
- `--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)
- `--lists <path>`: pathname to send lists output to (or - for standard output), cannot be used without `--table` or `--spreadsheet`
- `--stats <path>`: pathname to send statistics output to (or - for standard output) cannot be used without `--table` or `--spreadsheet`
- `--threads <number>`: use this many threads to do model runs in parallel, or 1 to disable parallel runs. defaults to `floor(0.75 * <number of processors>)`.
- `--update-plots`: enable plot updates. Include this if you want to export plot data, or exclude it for better performance.
- `--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.

Note that prior to NetLogo 6.3.0 the directions were to use `netlogo-headless.sh` (or `netlogo-headless.bat` on Windows) along with a separate installation of Java of the system to run BehaviorSpace experiments. The `netlogo-headless.sh` script is still included with NetLogo and can still be used as before, which might be preferrable in server environments where the installed Java version is strictly controlled. But the recommended method for on a personal computer is to use the `NetLogo_Console --headless` app. Because `NetLogo_Console` uses the Java that comes bundled with NetLogo it requires no extra software installation or configuration.

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, run from the NetLogo 7.1.0-alpha1 installation folder so the paths to the `NetLogo_Console` app and `Wolf Sheep Simple 5.nlogo` model are correct.

The below commands should work as-is in a terminal on macOS and Linux. On Windows in the Command Prompt you can use `^` instead of `\` to break the command across multiple lines, or just put the command on a single line.

```
./NetLogo_Console --headless \
  --model "models/IABM Textbook/chapter 4/Wolf Sheep Simple 5.nlogo" \
  --experiment "Wolf Sheep Simple model analysis" \
  --table -
```

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 NetLogo headless, 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.

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:

```
./NetLogo_Console --headless \
  --model "models/IABM Textbook/chapter 4/Wolf Sheep Simple 5.nlogo" \
  --experiment "Wolf Sheep Simple model analysis" \
  --max-pxcor 5 \
  --min-pxcor -5 \
  --max-pycor 5 \
  --min-pycor -5
```

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:

```
./NetLogo_Console --headless \
  --model "models/IABM Textbook/chapter 4/Wolf Sheep Simple 5.nlogo" \
  --experiment "Wolf Sheep Simple model analysis" \
  --table wsp5-table-output.csv \
  --spreadsheet wsp5-spreadsheet-output.csv \
  --lists wsp5-lists-output.csv \
  --stats wsp5-stats-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 specifies 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.

The optional `--lists <filename>` argument specifies that lists 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. Lists data is not written out until all runs in the experiment are finished.

The optional `--stats <filename>` argument specifies that stats 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. Stats 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. If you use `--lists` or `--stats` at least one of the `--table` or `--spreadsheet` options must be used.

Here is an example that shows how to run an experiment setup which is stored in a separate XML file, instead of in the model file (see below for more information on the XML file format). This assumes you've created a `my-wsp-setups.xml` file with a `My WSP Exploration" experiment and placed it in your home directory. The most straight-forward way to create a setup file is to create an experiment using BehaviorSpace in the NetLogo GUI and use the **Export ** option.

```
./NetLogo_Console --headless \
  --model "models/IABM Textbook/chapter 4/Wolf Sheep Simple 5.nlogo" \
  --setup-file ~/my-wsp-setups.xml \
  --experiment "My WSP Exploration"
```

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 a NetLogo 3D experiment, run headless with the `--3D` argument, for example:

```
./NetLogo_Console --headless \
  --3D \
  --model "models/3D/Sample Models/GasLab/GasLab Free Gas 3D.nlogo3d" \
  --experiment "100 runs" \
  --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, export them *(Since 6.4)* and then examine the resulting .xml file(s) in a text editor. The experiment setups can also be found towards the end of the .nlogo file, in a section that begins and ends with a `experiments` tag. Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE experiments SYSTEM "behaviorspace.dtd">
<experiments>
  <experiment name="My WSP Exploration" repetitions="5" runMetricsEveryStep="false">
    <setup>setup</setup>
    <go>go</go>
    <timeLimit steps="2000"/>
    <metric>count wolves</metric>
    <metric>count sheep</metric>
    <metric>sum [grass-amount] of patches</metric>
    <enumeratedValueSet variable="energy-gain-from-grass">
      <value value="2"/>
    </enumeratedValueSet>
    <steppedValueSet variable="number-of-wolves" first="5" step="1" last="15"/>
    <enumeratedValueSet variable="movement-cost">
      <value value="0.5"/>
    </enumeratedValueSet>
    <enumeratedValueSet variable="energy-gain-from-sheep">
      <value value="5"/>
    </enumeratedValueSet>
    <enumeratedValueSet variable="number-of-sheep">
      <value value="500"/>
    </enumeratedValueSet>
    <enumeratedValueSet variable="grass-regrowth-rate">
      <value value="0.3"/>
    </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.

If you want to create a setup file for NetLogo 6.3.0 and earlier versions for which **Export** is not available you need to know that in a model file the XML for experiment setups does not begin with any XML headers, because the not whole file is XML, only part of it. Therefore if you manually create a separate file for experiment setups, 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 `ISO-8859-1`.

Adjusting JVM Parameters

Opening the NetLogo Headless launcher script will show the options used to launch java when running NetLogo Headless. You can adjust various JVM parameters in this script. You may also pass in Java properties starting with `-D` to the launcher.

NetLogo allocates a maximum of half your total system memory for running your model as it is needed. If you want to set a maximum amount of memory for BehaviorSpace to use you can use the `-Xmx` setting to specify a particular heap size.

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.

Cluster Computing (HPCs)

BehaviorSpace experiments are often run on large-scale computing clusters. More on that topic can be found here.

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 Modeler

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.

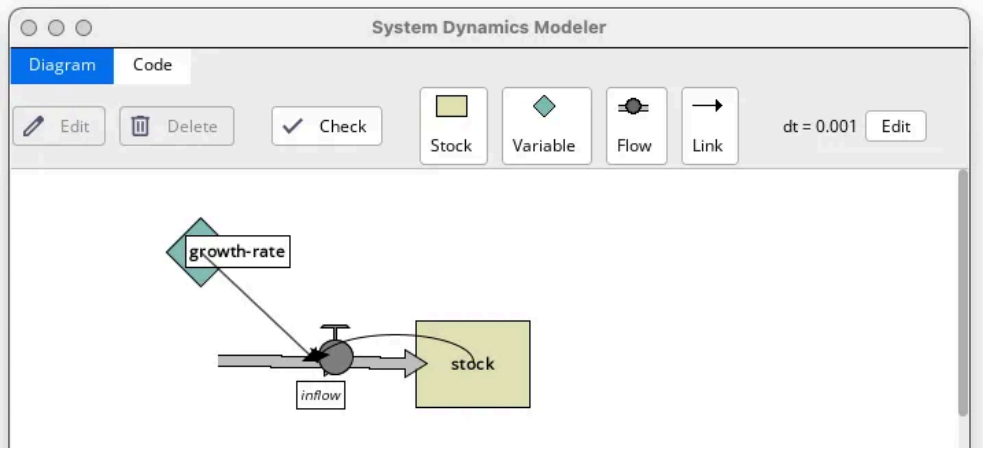
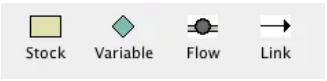


Diagram Tab

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 or use the right mouse button to 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, press the Link button, or using the right mouse button, 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

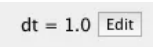
When you create a Stock, Variable, or Flow, you see a red question-mark on the element. The question-mark indicates that the element doesn't have a name yet. The red color indicates that the Stock is incomplete: it's missing one or more values required to generate a System Dynamics model. When a diagram 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.

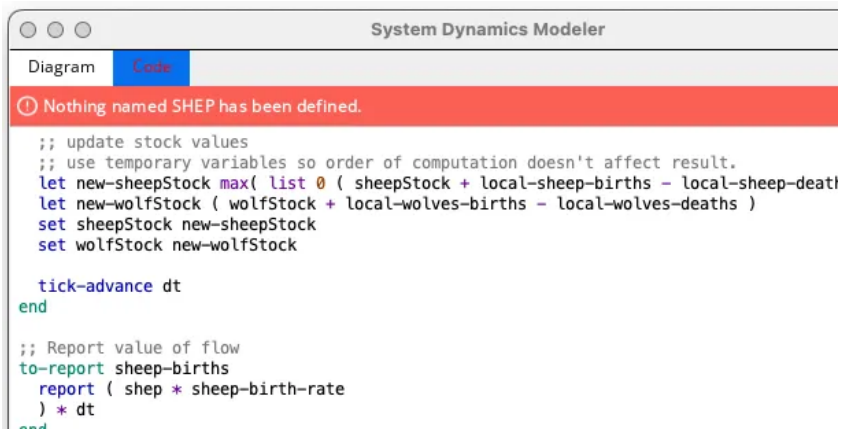
Editing dt



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

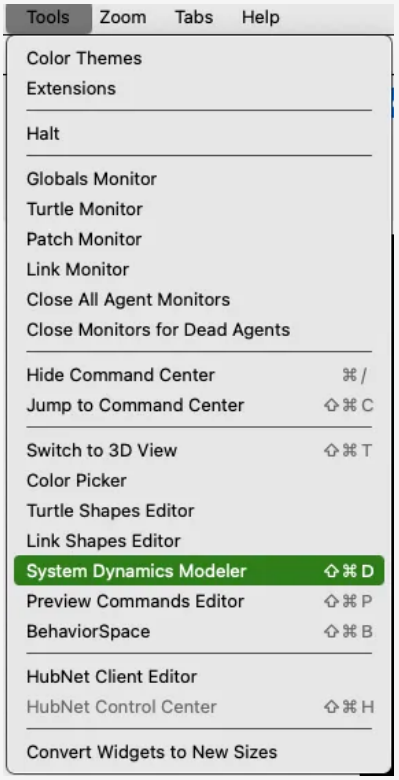
The diagram you create with the System Dynamics Modeler, and the procedures generated from your diagram, are part of your NetLogo model. When you a save the NetLogo model, your diagram is saved with it, in the same file.

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.

Diagram Code

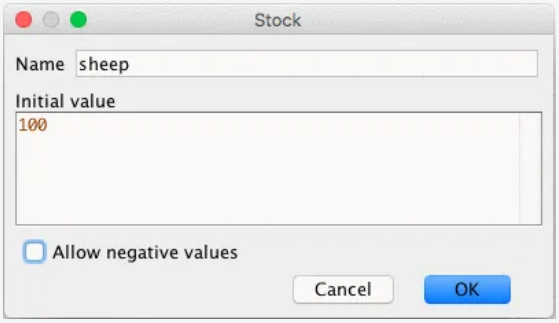
```
;; System dynamics model globals
globals [
  ;; constants
  sheep-birth-rate
  ;; stock values
  sheep
  ;; size of each step, see SYSTEM-DYNAMICS-GO
  dt
]

;; Initializes the system dynamics model.
;; Call this in your model's SETUP procedure.
to system-dynamics-setup
  reset-ticks
  set dt 1.0
  ;; 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
```

- 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.

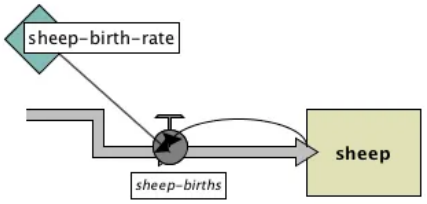
- Draw a Link from the `sheep` Stock to the `sheep-births` Flow.

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`.

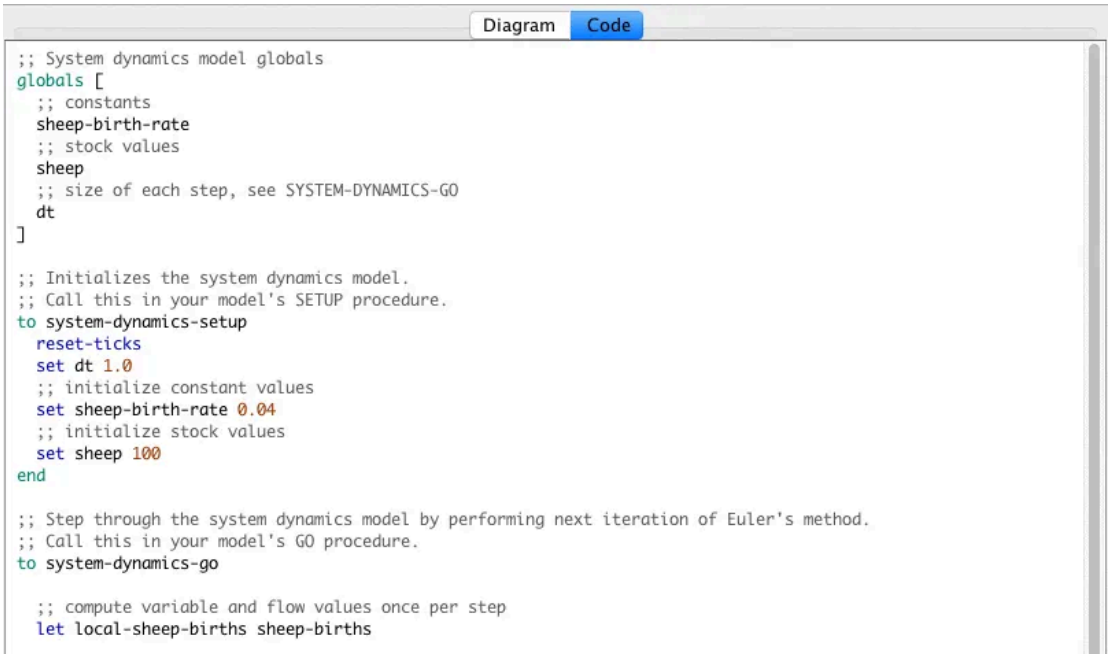
Your diagram should look something like this:



Our diagram has the correct structure but we aren't yet finished because it the amount of sheep flowing into the stock doesn't depend upon the number of sheep and 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:



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
    set-current-plot "populations"
    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.

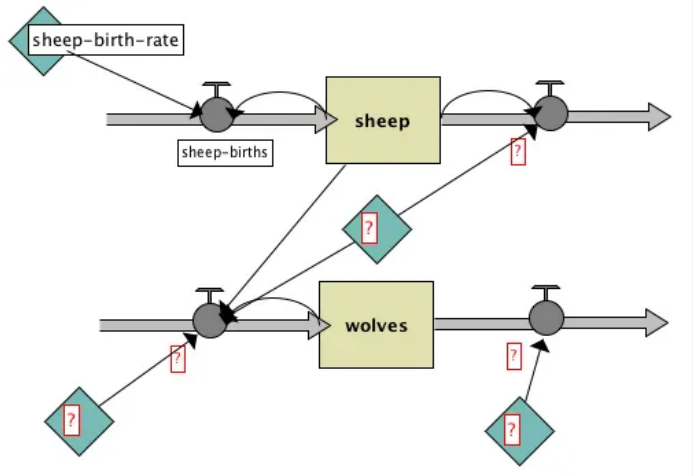
- Press the setup button.
- Don't press the "go" button yet. Instead, type `go` four or five times into the Command Center.

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.

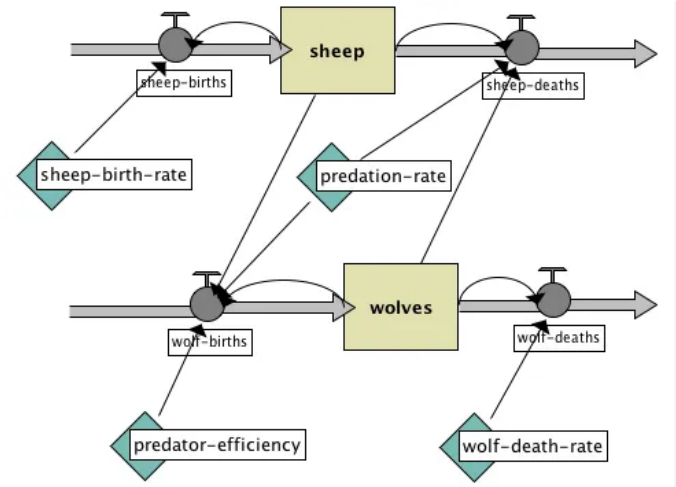
To fix that, let's finish our diagram by introducing a population of wolves which eat sheep.

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:

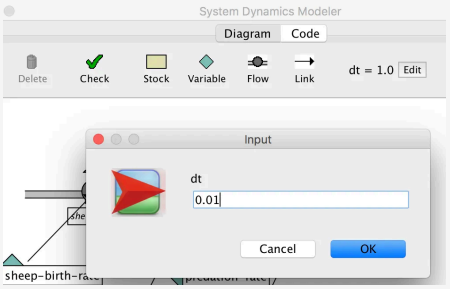


- Add one more Flow from the `wolves` Stock to the Flow that goes out of the Sheep stock.
- Fill in the names of the diagram elements so it looks like this:



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`.

- Adjust the `dt` of the system dynamics model by selecting "Edit" next to `dt` in the toolbar of the system dynamics modeler. In the dialog that appears, enter `0.01`.



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:

Color Picker

Introduction

NetLogo 7 introduces a new, more powerful way of selecting colors, which replaces Color Swatches. As was the case previously there are four different contexts in which colors can be selected: from the Tools menu, from a Color Input widget, when selecting a plot pen color and when editing a Note. The ways in which the versions differ involve whether a selected color can be copied or "picked," whether transparency is allowed, and what forms of input and output specifications are allowed, as will be described later.

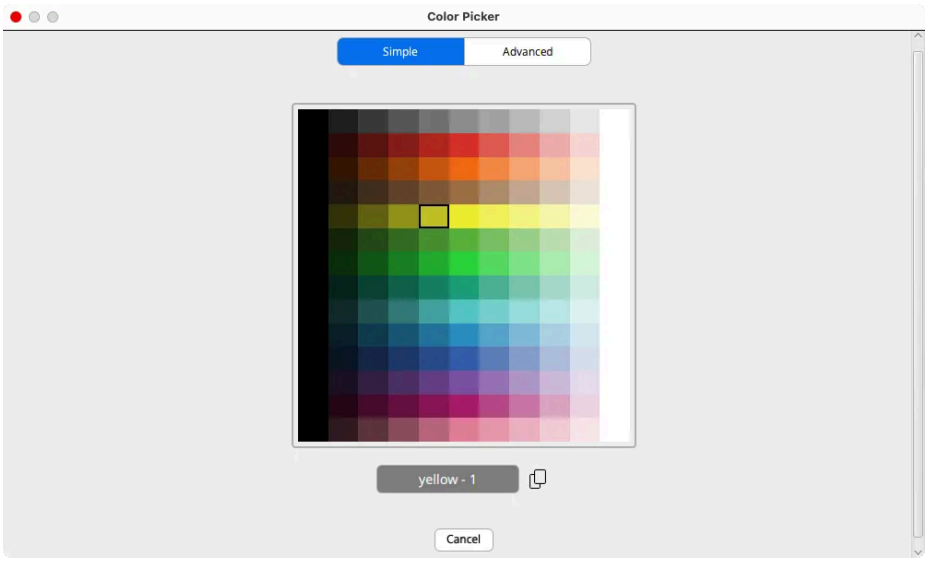
Colors in NetLogo

For information on the different ways colors are represented in NetLogo see the Color section of the NetLogo Programming Guide. The NetLogo primitives for working with colors are 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. The Palette Extension offers additional ways to work with colors.

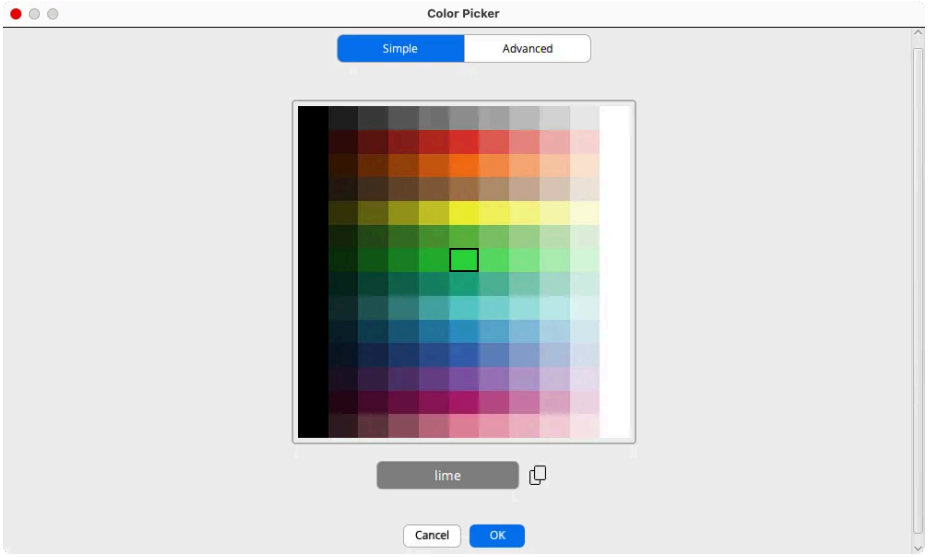
All versions of the Color Picker have two tabs, Simple, and Advanced. The Simple tab allows users to pick a color from a rectangular grid. This is analogous to the Color Swatches dialog, although it is more visually streamlined.

Simple Tab

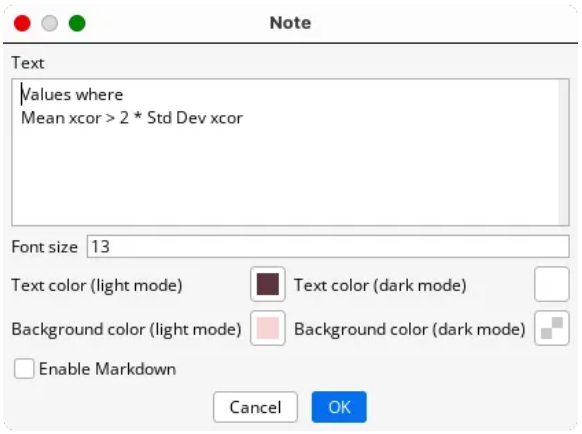
The Simple tab of the Color Picker in the Tools menu has buttons to allow the user to copy the color (icon to the right of the grey rectangle), or exit the dialog (cancel).



As shown below Simple tab for the Color Input adds a blue 'OK' button to the right of 'Cancel.' Choosing 'OK' changes the Input to the selected color, exiting the dialog. 'Copy' puts the NetLogo Color number corresponding to the selected color in the clipboard. It can then be copied to the Code Tab, or any other text area. The dialog remains open. Cancel closes the dialog without changing the color input widget.



When editing a Note, there are four buttons that can be clicked to open a Color Picker.

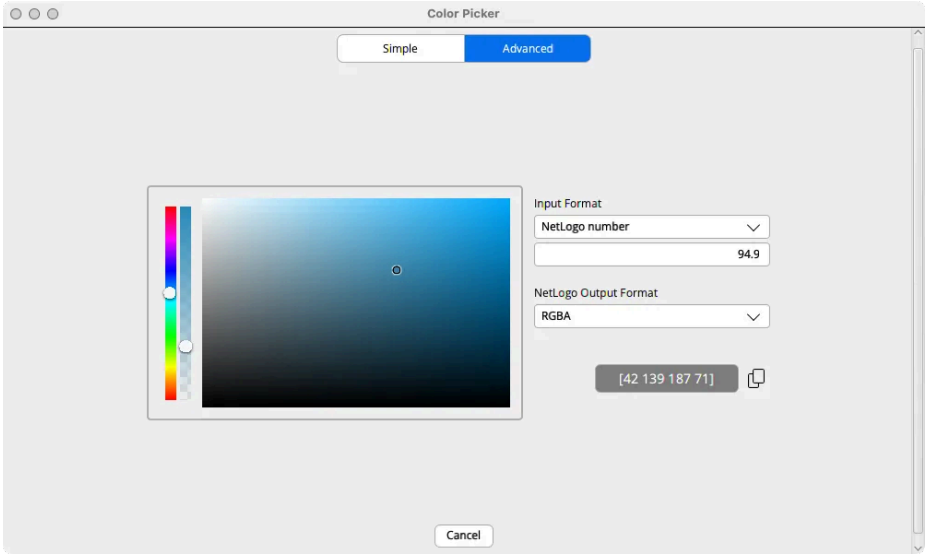


The 'Simple' tab of the color picker is the same, regardless of context. The 'Advanced' tab allows different output formats, depending on the context.

The Advanced Tab

Introduction

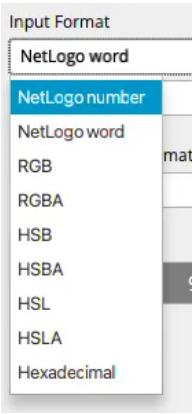
The Advanced Tab allows a much more nuanced way to choose a color.



The rainbow slider on the left allows users to select a hue. The slider to its right allows the user to select the transparency of the color. By default, the color is fully opaque, and the knob is at the top of the slider, moving it down makes the color more transparent. You can also drag the small circle in the rectangle to the right of the sliders to quickly explore variants of the selected hue. Moving from it from left to right (horizontally) increases the saturation. Moving it bottom to top (vertically) increases the lightness.

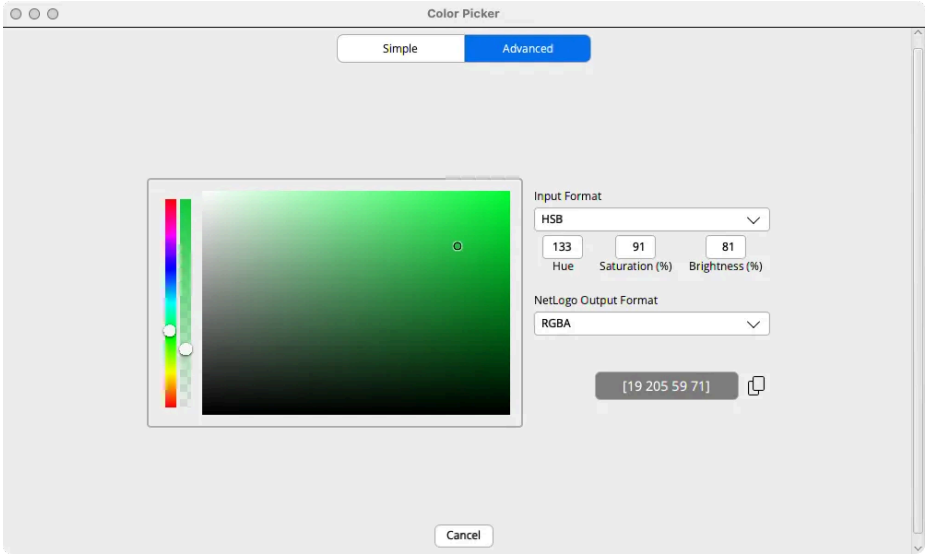
Input Format

A powerful feature of the Advanced tab is that you can import colors using the following common formats RGB, RGBA, HSB, HSBA, HSL, HSLA and hexadecimal as well as NetLogo Number or NetLogo word. Use the Input Format pulldown menu to select the format and you will be able to set the appropriate value(s). The pulldown menu below shows the output options, which have the acronyms above spelled out. There follow two examples of specifying an input. For a NetLogo word there is a text field. For HSB (Hue, Saturation, Brightness) there are three numeric fields. The color selected using the Input format can then be modified by using the hue slider, transparency slider or the palette rectangle.



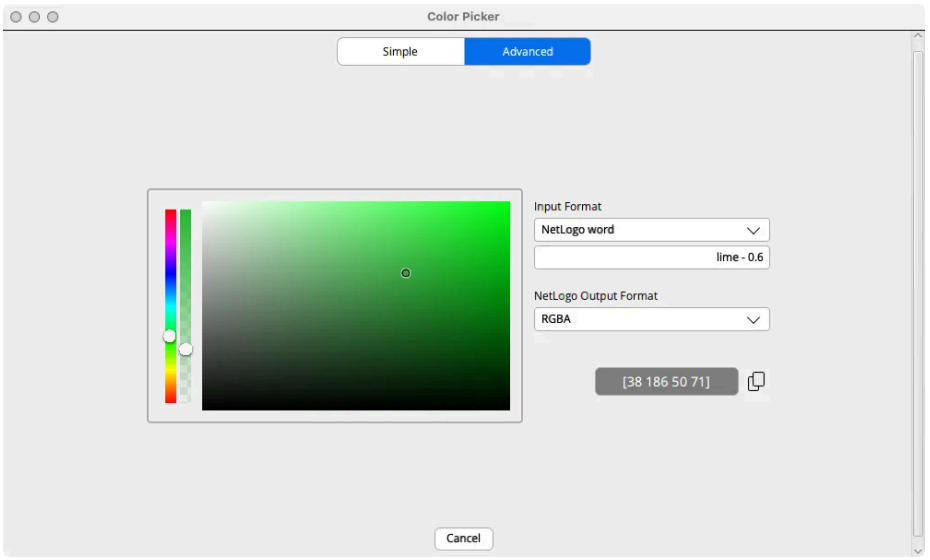
In this example the input is a NetLogo word.

In this example the input is HSB.



Output Format

The color that results after any adjustments can be output in the selected Output Format. The full list of output formats is NetLogo Number, NetLogo word and RGBA (Red, Green, Blue, Alpha). However in some contexts only a NetLogo Number can be output. As the user changes a color the value of the Input value is updated. In the image below it was initially lime - 0.3, but changed to lime - 0.6



Output Formats for Different Contexts

For the Color Input and plot pen colors the output format is a NetLogo number. Transparency is not supported for the Color Input in this release, but may be available in the future.

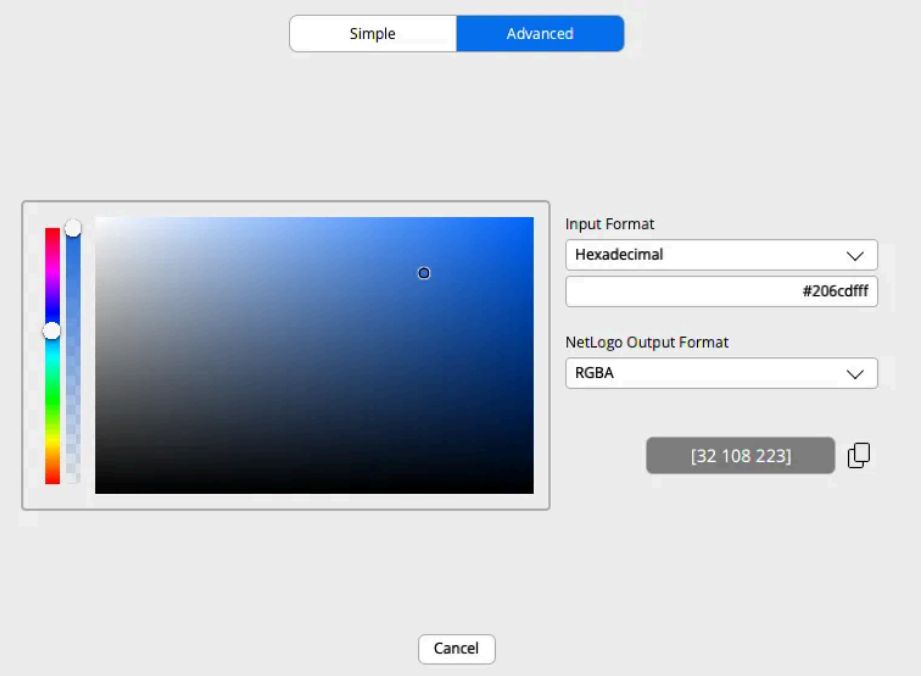
For Notes the output format is RGBA.

For the Tools menu Color Picker the output formats are NetLogo number, NetLogo word and RGBA.

Details about RGB Output

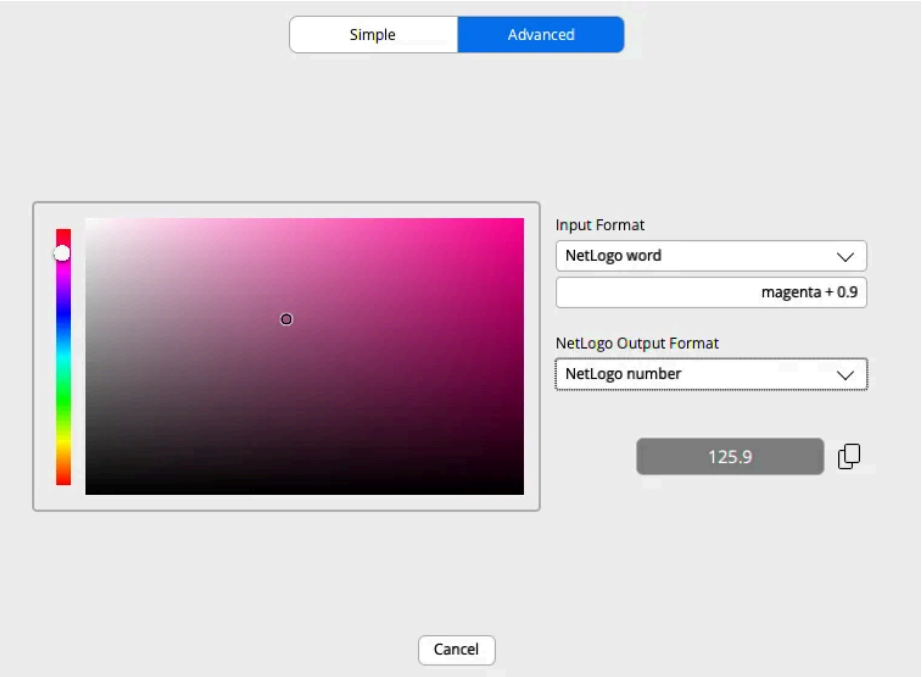
There is no explicit RGB output format. However when you select “RGBA”, and your color is opaque (maximum alpha), the result is RGB format - a 3-item list (no alpha). Otherwise “RGBA” output produces a 4-item list (with alpha). In the

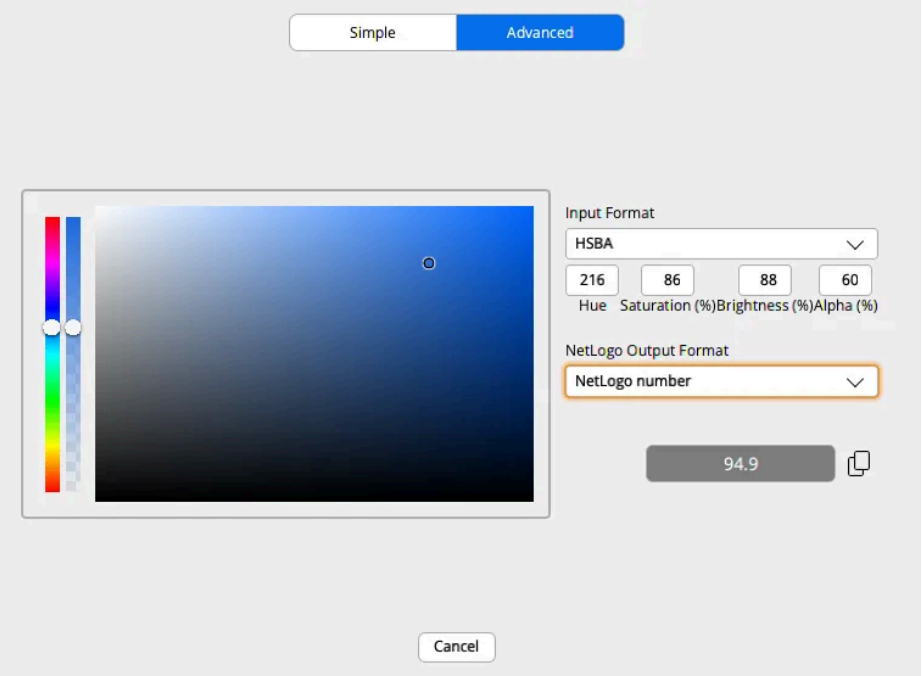
example below you can see that the output is a 3-item list, because there is no transparency and RGBA output is selected.



Details about the Alpha Bar

Some users may want to skip this section on first reading. The “alpha” bar of the Advanced tab is used to adjust the transparency of a color. If neither the input format nor the output format allows transparency the alpha bar will be hidden. The formats that do not allow transparency are: NetLogo number, NetLogo word RGB, HSB, and HSL. If the input supports alpha, but the output does not, the output dropdown will be highlighted and will have a tooltip that explains that the output color will be opaque. The first example shows the case where there is no alpha bar. The second example show the highlighted output because there is transparency, but an opaque output format.




Simple
Advanced
Input Format
HSBA
216
86
88
60
Hue
Saturation (%)
Brightness (%)
Alpha (%)
NetLogo Output Format
NetLogo number
94.9
Cancel

HubNet

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.

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.

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.

While HubNet is only supported via the Java Desktop clients at the moment, we hope to add support for other types of clients such as tablets and phones in the future.

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 class-room settings we also suggest an attached projector for the leader to project the entire simulation to the participants.

Starting an activity

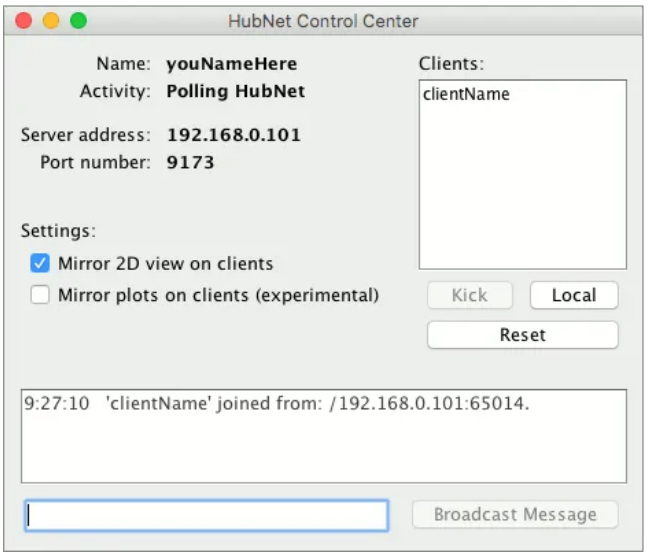
You'll find the HubNet activities in NetLogo's Models Library, in the HubNet Activities folder. We suggest doing a few practice runs of an activity before trying it in front of a class.

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. You may also see a broadcast network selection dropdown for which more extensive documentation is available in the FAQ. For now, just 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.

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

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]`

data type	`hubnet-broadcast` example	`hubnet-send` example
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

Introduction

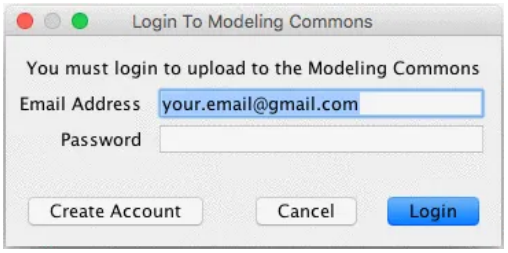
The Modeling Commons is a Web-based collaboration system for NetLogo modelers. Users of the Modeling Commons can share, download, modify, create variations of, comment on, and run NetLogo models — both those that are a part of the NetLogo models library, and also those that have been uploaded by other 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 `.nlogox` 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 register an account and log in. Unregistered users can view and download models, but cannot upload, edit, or comment on them.

The first time that you use "Upload to Modeling Commons" in a NetLogo session, 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.

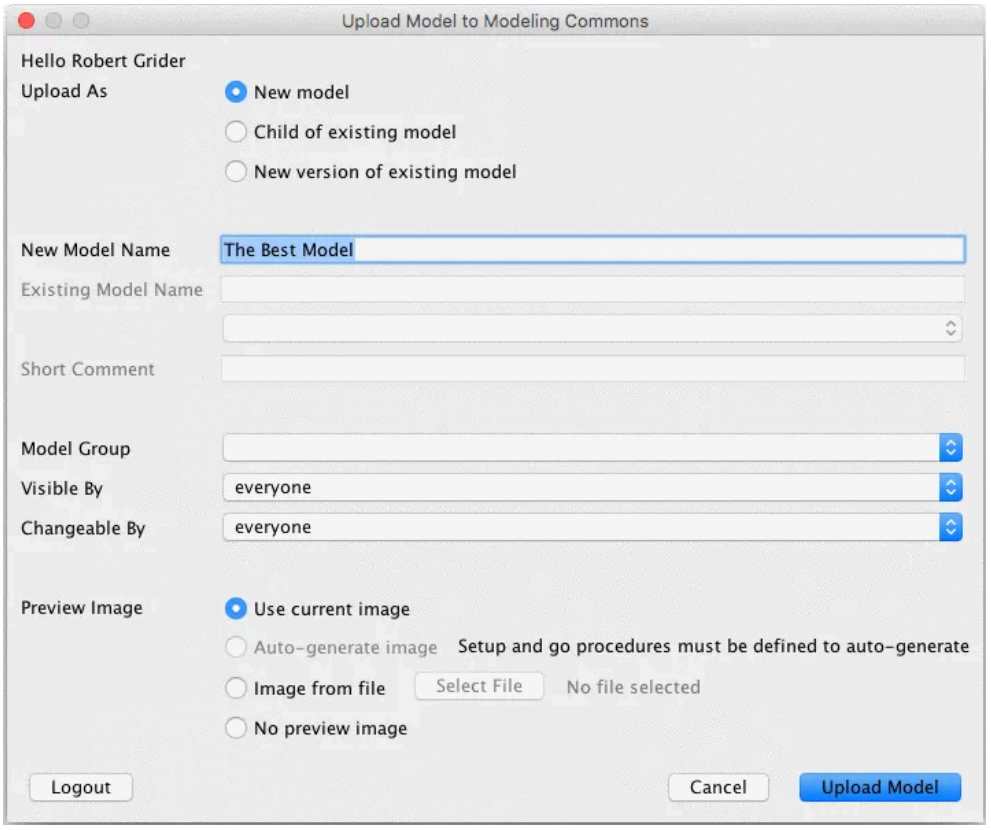
If you don't yet have an account with the Modeling Commons, then you will need to create one. Click on the "Create Account" button, and you will be redirected to the registration form in your browser.

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.

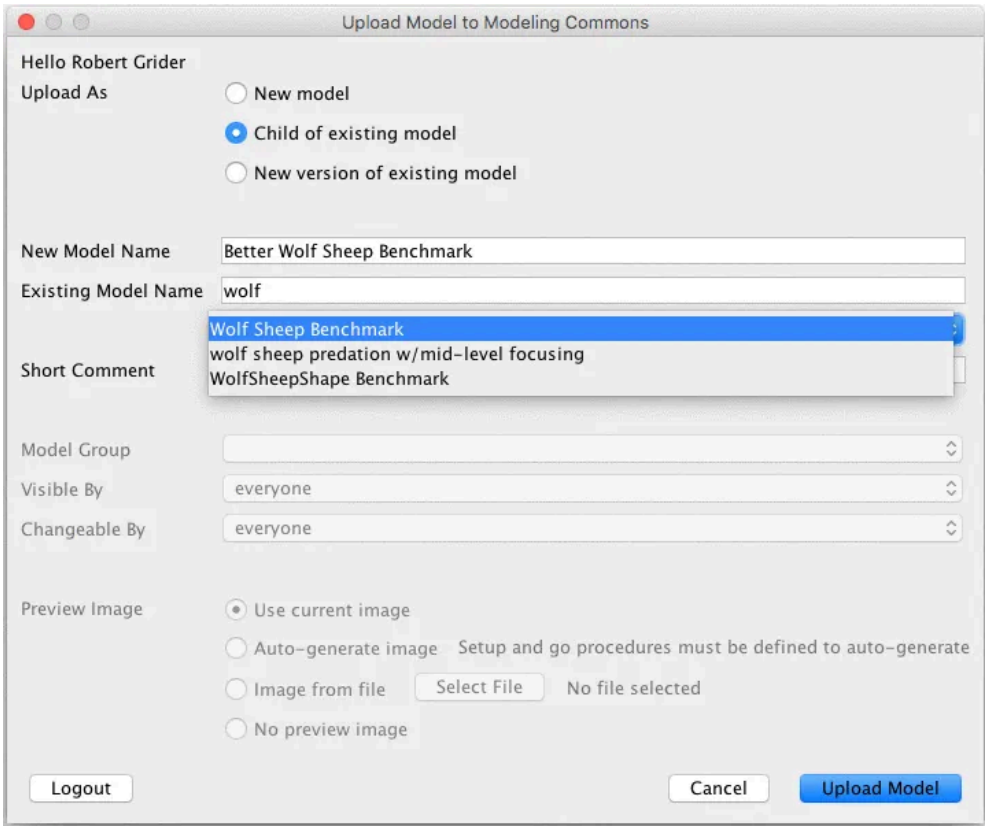
By default, anyone can view, fork, and update your model. You can restrict the ability to view and fork your model by changing the visibility permission. You can restrict the ability to update your model by changing the changeability permission. In order to set permissions for multiple people, assign 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 ]`. Note that this option will not work unless 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 is invisible to the student once initiated. The researcher can choose the type of events logged through the NetLogo preferences or by using a custom command line switch to start NetLogo.

In NetLogo version 6.2.2 and earlier, log4j was the library used and the logging output was in XML format. See the documentation for that prior version if you're not on the latest NetLogo release.

Starting Logging

There is a NetLogo preference to enable logging:

- Open NetLogo, then open **Preferences** from the **Tools** menu.
- Check the box for **Enable Logging:** then click **OK**.
- Restart NetLogo and logging will be enabled.

The setting will remain in effect until it is unchecked and NetLogo is restarted again.

You can also specify the **Directory to store logs:** in the NetLogo **Preferences**. If you leave this value blank with logging enabled the default location of the user's home directory will be used. See below for more information on the default location.

You can also specify the **Events to log:** in the NetLogo **Preferences**. This is a comma-separate list of the event names or "shortcut" names. If you leave this value blank with logging enabled a set of default events will be logged. See below for more information on the event names and what the defaults are.

Command Line Switches

You can also enable logging and set the log directory and log events with command line switches. If you specify either the `--log-directory` or `--log-events` switch, then logging will be enabled and the switch values will override any preferences set in the NetLogo UI.

Let's assume you are running on Linux and you have NetLogo installed in a `NetLogo 6.2.3` directory in your home folder. To enable logging, invoke the `NetLogo` executable like so:

```
~/NetLogo\ 6.2.3/NetLogo --log-directory ~/NetLogoLogs/ --log-events "default, ticks, globals"
```

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.

Events

Below are all of the events that can be logged with the logging app. By default all of these events are logged *except* "global", "link", "tick", and "turtle" since they can cause a large number of log entries to be created.

- button - button widget presses
- chooser - chooser widget changes

- comment - any user comments, currently only generated for compiler errors
- compile - when the model code is recompiled
- command-center - when code is run through the command center
- global - when a global variable value changes
- input-box - when an input box widget changes
- link - when a link agent is created or dies
- model-open - when a model is opened
- slider - when a slider widget changes
- speed-slider - when the speed slider changes
- start - always the first entry of a log, contains system info
- switch - when a switch widget changes
- stop - always the last entry of a log
- turtle - when a turtle agent is created or dies
- widget-edit - when a widget is added to or removed from the model
- tick - when the tick counter is advanced

There are also some shortcut names you can use in the preferences to refer to collections of events:

- all - button, chooser, comment, compile, command-center, global, input-box, link, model-open, slider, speed-slider, start, switch, stop, turtle, widget-edit, tick
- agents - link, turtle
- defaults - everything but global, link, tick, and turtle
- greens - chooser, input-box, slider, switch - the "global variable" widgets

Example logging strings you could put in the **Events to log:** setting in preferences dialog or use via the `--log-events` command line switch.

- "button, model-open, widget-edit" - log only button widget presses, model open events, and widget add/remove
- "defaults, global" - add global variable change events to the defaults list
- "button, greens, model-open" - log widget use and model open events.

JSON Output

The output is in JSON text format. Below is an example log file with some small modifications to content to fit better in this document.

Each line has an "event" identifying its type along with a "timeStamp" of when it occurred. Then there is an "eventInfo" entry that contains further information specific to the event. For example, "model-open" event contains the model "name", and the "button" events contain the "buttonName" and whether the click stopped a forever button, "wasStopped".


```
[
  {"event":"start","timeStamp":"2022-06-17 11:29:56.737","eventInfo":
{"studentName":"Larry","modelName":"Slime.nlogo","version":"NetLogo
6.#.#","ipAddress":"192.168.1.11","loginName":"cpu_user","events":["input-box","switch",...,"widget-
edit"]}}
, {"event":"model-open","timeStamp":"2022-06-17 11:29:56.738","eventInfo":{"name":"Slime.nlogo"}}
, {"event":"compile","timeStamp":"2022-06-17 11:29:57.201","eventInfo":{"code":"patches-own
[chemical]\n...","success":true}}
, {"event":"button","timeStamp":"2022-06-17 11:29:58.617","eventInfo":
{"buttonName":"setup","wasStopped":false}}
, {"event":"tick","timeStamp":"2022-06-17 11:29:58.675","eventInfo":{"ticks":0.0}}
, {"event":"button","timeStamp":"2022-06-17 11:29:58.676","eventInfo":
{"buttonName":"setup","wasStopped":true}}
, {"event":"button","timeStamp":"2022-06-17 11:29:58.961","eventInfo":
{"buttonName":"go","wasStopped":false}}
, {"event":"tick","timeStamp":"2022-06-17 11:29:58.993","eventInfo":{"ticks":1.0}}
, {"event":"tick","timeStamp":"2022-06-17 11:29:59.030","eventInfo":{"ticks":2.0}}
, {"event":"tick","timeStamp":"2022-06-17 11:29:59.076","eventInfo":{"ticks":3.0}}
```

```
, {"event":"tick","timeStamp":"2022-06-17 11:29:59.123","eventInfo":{"ticks":4.0}}
, {"event":"button","timeStamp":"2022-06-17 11:29:59.921","eventInfo":
{"buttonName":"go","wasStopped":true}}
, {"event":"slider","timeStamp":"2022-06-17 11:30:01.681","eventInfo":
{"valueChanged":true,"min":1.0,"globalName":"population","max":1500.0,"inc":1.0,"buttonReleased":false,"n
ewValue":399.0}}
, {"event":"slider","timeStamp":"2022-06-17 11:30:01.698","eventInfo":
{"valueChanged":true,"min":1.0,"globalName":"population","max":1500.0,"inc":1.0,"buttonReleased":false,"n
ewValue":391.0}}
, {"event":"slider","timeStamp":"2022-06-17 11:30:02.317","eventInfo":
{"valueChanged":true,"min":1.0,"globalName":"population","max":1500.0,"inc":1.0,"buttonReleased":false,"n
ewValue":179.0}}
, {"event":"button","timeStamp":"2022-06-17 11:30:02.910","eventInfo":
{"buttonName":"setup","wasStopped":false}}
, {"event":"tick","timeStamp":"2022-06-17 11:30:02.919","eventInfo":{"ticks":-1.0}}
, {"event":"tick","timeStamp":"2022-06-17 11:30:02.922","eventInfo":{"ticks":0.0}}
, {"event":"button","timeStamp":"2022-06-17 11:30:02.922","eventInfo":
{"buttonName":"setup","wasStopped":true}}
, {"event":"button","timeStamp":"2022-06-17 11:30:03.254","eventInfo":
{"buttonName":"go","wasStopped":false}}
, {"event":"tick","timeStamp":"2022-06-17 11:30:03.258","eventInfo":{"ticks":1.0}}
, {"event":"tick","timeStamp":"2022-06-17 11:30:03.293","eventInfo":{"ticks":2.0}}
, {"event":"button","timeStamp":"2022-06-17 11:30:03.763","eventInfo":
{"buttonName":"go","wasStopped":true}}
, {"event":"command-center","timeStamp":"2022-06-17 11:30:08.489","eventInfo":{"owner":"Command
Center","code":"ask turtles [ set color black ]","agentType":"O","success":true}}
, {"event":"stop","timeStamp":"2022-06-17 11:30:09.703"}
]
```


JSON is a very common data format so there are numerous ways to use it. Most scripting and programming languages have easy ways to read JSON data in so that it can be processed, fed to another system (such as a database), or re-formatted somehow (perhaps to a plot using a plotting library).

Modern versions of Microsoft Excel can import JSON directly. Google Sheets does not yet support JSON import natively, but there are add-on scripts that can be used. There are also numerous online converters that can turn JSON to a "flattened" CSV or just convert it to XML as well.

Where Logs are Stored

By default logs are stored in the operating system specific user's home directory. On most Unix-like systems that is `/home/<username>/`. On Windows the logs can be found in `C:\Users\<username>\`, where `<username>` is the logged in user. On macOS the usual location is `/Users/<username>/`.

Managing Log Files

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`. Both commands stop the current log file (so it can be zipped or deleted) and a new one is started afterwards.

Controlling the Model

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

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:

1. Go to the menu bar in Mathematica
2. Click on File and select Install...
3. In the Install Mathematica Item dialog
4. Select Package for Type of item to install
5. Click Source, and select From file...
6. In the file browser, go to the location of your NetLogo installation,
7. click on the Mathematica Link subfolder, and select NetLogo.m.
8. 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 7.1.0-alpha1/"

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 7.1.0-alpha1/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

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
```

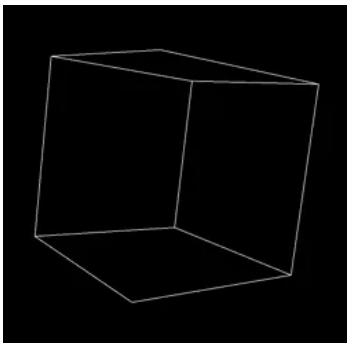
A file declaring just a two dimensional, patch-sized, square in the xy-plane centered at the origin would look like this:

```
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.

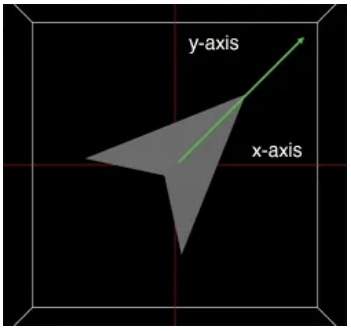
Step 2: Turtle Movement

- 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

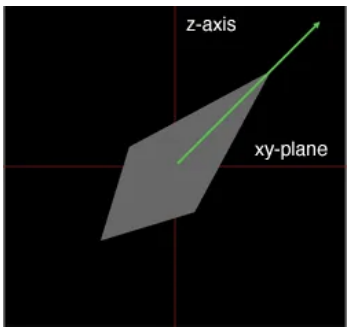
Take a moment to look for the controls and monitors. In the bottom left you'll notice a group of monitors that describe the location and orientation of the turtle, though until you press the setup button they'll all say "N/A".

- Press the "setup" button

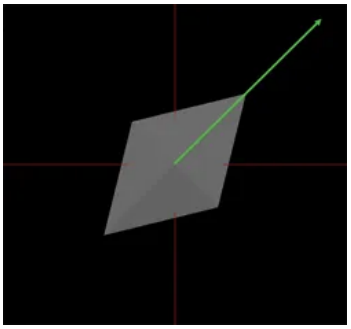
Heading, pitch, and roll are turtle variables that represent the orientation of the turtle. Heading is absolute in relation to the x/y plane; it is the rotation of the turtle around the z-axis.



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

`at-points` · `distancexyz` (`distancexyz-nowrap`) · `distancexyz` (`distancexyz-nowrap`) · `dz` · `face` (`facexyz`) · `left` · `link-pitch` · `load-shapes-3d` · `max-pzcor` (`min-pzcor`) · `neighbors` (`neighbors6`) · `orbit-down` (`orbit-left`, `orbit-right`, `orbit-up`) · `__oxcor` (`__oycor`, `__ozcor`) · `patch` · `patch-at` · `patch-at` · `patch-at-heading-pitch-and-distance` · `patch-at-heading-pitch-and-distance` · `random-pzcor` · `random-zcor` · `right` · `roll-left` · `roll-right` · `setxyz` · `setxyz` · `tilt-down` (`tilt-up`) · `towards-pitch` (`towards-pitch-nowrap`) · `towards-pitch-xyz` (`towards-pitch-xyz-nowrap`) · `turtles-at` (`<breeds>-at`) · `turtles-at` (`<breeds>-at`) · `world-depth` · `zoom`

Built-In Variables

`pitch` · `pzcor` · `roll` · `zcor`

Primitives

at-points

Since 4.1

***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

Since 4.1

distancexyz-nowrap

Since 4.1

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

Since 4.1

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

Since 4.1

`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

Since 4.1

`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

Since 4.1.2

`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

Since 4.1

`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

Since 4.1

min-pzcor

Since 4.1

`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

Since 4.1

neighbors6

Since 4.1

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

Since 4.1

orbit-left

Since 4.1

orbit-right

Since 4.1

orbit-up

Since 4.1

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

Since 4.1

patch *x y z*

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

Since 4.1

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

Since 4.1

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

Since 4.1

`random-pzcor`

Reports a random integer ranging from min-pzcor to max-pzcor 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

Since 4.1

`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

Since 4.1

`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

Since 4.1

`roll-left` *`number`*

The wingtip of the turtle rotates to the left *number* degrees with respect to the current heading and pitch.

roll-right

Since 4.1

`roll-right` *`number`*

The wingtip of the turtle rotates to the right *number*degrees with respect to the current heading and pitch.

setxyz

Since 4.1

`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

Since 4.1

tilt-up

Since 4.1

`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

Since 4.1

towards-pitch-nowrap

Since 4.1

`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

Since 4.1

towards-pitch-xyz-nowrap

Since 4.1

`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

Since 4.1

<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

Since 4.1

`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

Since 4.1

`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.

Cluster Computing (HPC)

This document is an advanced guide that provides examples of how to run NetLogo BehaviorSpace experiments on a High-Performance Computing (HPC) cluster. The scripts provided in this guide are specifically for **Slurm**, a popular open-source workload manager and job scheduler, but they can be adapted for use with other schedulers.

Installing NetLogo on a Cluster

To run BehaviorSpace experiments, NetLogo must be installed on your HPC in a location accessible to the user.

- The easiest method is to download the desired NetLogo version (for most clusters this is the 64-bit Linux installation, available at https://www.netlogo.org/downloads/linux/) , unzip and move the entire folder to a directory on your HPC.
- Alternatively, you can install NetLogo from the command line if your HPC allows it. For example, you can download and unzip the 64-bit Linux version of NetLogo 7.0.0 with:

```
curl -O https://ccl.northwestern.edu/netlogo/7.0.0/NetLogo-7.0.0-64.tgz
tar zxf NetLogo-7.0.0-64.tgz
```

Note: Although NetLogo comes with a bundled Java installation, some HPC clusters require users to run only approved Java Runtime Environments (JREs). If that is the case, you must ensure that your HPC has a compatible JRE (Java 17 or higher). Please consult your HPC documentation or system administrator if unsure.

Submitting a Job

This section provides two examples of how to run a BehaviorSpace experiment on an HPC using Slurm.

1. A **simple script**, which must be customized for each experiment and model.
2. A **general script**, which takes arguments to automatically generate and submit a Slurm job script.

The simple script is easier to debug, while the general script is more convenient for repeated use. Both approaches are valid, and the scripts can also be found here on GitHub.

A simple example

The script below is a simple Slurm job submission template for NetLogo. Lines beginning with `###` provide explanations.

Save this script as `simple_standalone_job.sh` . After updating it to match your HPC configuration, model, and experiment, you can submit it with:

`sbatch simple_standalone_job.sh`

Script:

```
#!/bin/sh

### ---
### simple_standalone_job.sh
### ---

### Template standalone Slurm job submission script for NetLogo BehaviorSpace experiments

#SBATCH --account=PROJECTID       ### replace PROJECTID with your allocation/project ID number.
#SBATCH --partition=short         ### replace short with desired parition: medium, long, gengpu, etc.
#SBATCH --time=1:30:00            ### time format is hh:mm:ss
```

```
#SBATCH --nodes=1                ### leave the --nodes as 1 because NetLogo can run into problems
when trying to use multiple nodes
#SBATCH --ntasks-per-node=16     ### replace 16 with the number of CPU cores you will request
#SBATCH --mem=4G                 ### replace 4G with the amount of memory you will request; do not
forget to put specify the units.
#SBATCH --job-name=EXAMPLE_JOB   ### replace EXAMPLE_JOB with the name of your job (can be anything)
#SBATCH --output=%x-%j.out       ### automatically generates an output file. Slurm replaces %x with
job name you provided in the previous instruction and %j with the unique job id.
#SBATCH --error=%x-%j.err        ### automatically generates an error file.
#SBATCH --mail-type=ALL          ### replace ALL with BEGIN, END, or FAIL if you'd like to.
#SBATCH --mail-user=U@SCHOOL.EDU  ### Replace U@SCHOOL.edu with your own email address so that Slurm
can send you emails when your job is queried, started, completed, or terminated due to an error.

module purge all
module load java/jdk-17.0.2+8    ### You may need to update this line if your HPC doesn't have the
jdk-17.0.2+8 module available.

JVM_OPTS=(-Xmx1024m -server -XX:+UseParallelGC -Dfile.encoding=UTF-8 -
Dnetlogo.extensions.dir="${BASE_DIR}/extensions")

### Make sure to update the NetLogo installation path, experiment name, model filename, experiment
name and output filename variables in the following command.
java ${JVM_OPTS[@]} -classpath PATH/TO/NETLOGO/lib/app/netlogo-7.0.0.jar org.nlogo.headless.Main --
model "MODELFILE.nlogo" --experiment "EXPERIMENT_NAME" --threads 16 --table
PATH/TO/OUT/FOLDER/OUTPUTFILE.csv
```

A more general example

Editing a submission file for each experiment can be tedious. The following scripts automate the process by generating a temporary job script with your chosen arguments and then submitting it.

Save the following as `config.sh` and `generate_and_submit.sh`. Update `config.sh` with your account details, and adjust `generate_and_submit.sh` as needed for your HPC.

Run the script with:

```
bash generate_and_submit.sh <model-file> <experiment-name> <category> <maxtime> <threads> <ram>
```

Where the command-line arguments are:

- `model-file` (including the file extension)
- `experiment-name`
- `category` (which partition to use: short, normal, long, gpu, etc.)
- `maxtime` (in *hh:mm:ss* format)
- `threads` (number of CPU cores to request)
- `ram` (in megabytes)

This will automatically generate an intermediary bash script and then use it to submit the job. It will also delete the intermediary file after submission to prevent folder pollution.

Example:

```
bash generate_and_submit.sh fire.nlogo densityexp short 1:00:00 32 4096
```

Scripts:

```
#!/bin/bash

### ---
### config.sh
### ---

# Update this script with your Slurm account information

# your email address to receive notifications
email=youremail@yourinstitution.edu

# The following variable should match your Slurm Allocation/Project ID
```

```
project=p00000

# the experiment directory (absolute path)
path=/home/username/projectfolder
```

```
#!/bin/bash

### ---
### generate_and_submit.sh
### ---

### A script to automate the slurm job submission process for NetLogo BehaviorSpace experiments

model=$1
experiment=$2
category=$4
maxtime=$5
threads=$6
ram=$7

# NetLogo struggles to divide experiments between multiple physical processors
machines=1

# load the config file to pull the shared variables
source config.sh

### AUTO GENERATE A TEMPORARY SCRIPT TO RUN THE EXPERIMENT
### This approach minimizes the repetitive editing and potential errors
### that may arise from trying to manually edit long scripts

totalram=$((ram*threads+4096))
modelpath=${path}/${model}

runfile=run_${experiment}.sh

# if a file with the same name exists, just delete it
if test -f $runfile ; then
    rm $runfile
fi

# create the slurm job definition within the run file
echo "#!/bin/sh" >> $runfile
echo "#SBATCH --account="${project} >> $runfile
echo "#SBATCH --partition="${category} >> $runfile
echo "#SBATCH --time="${maxtime} >> $runfile
echo "#SBATCH --nodes="${machines} >> $runfile
echo "#SBATCH --ntasks-per-node="${threads} >> $runfile
echo "#SBATCH --mem="$((totalram/1024))G >> $runfile
echo "#SBATCH --job-name="${experiment} >> $runfile
echo "#SBATCH --output=output_%x-%j.out" >> $runfile
echo "#SBATCH --mail-type="ALL >> $runfile
echo "#SBATCH --mail-user="${email} >> $runfile

echo "module purge all" >> $runfile
echo "module load java/jdk-17.0.2+8" >> $runfile

echo "JVM_OPTS=(-Xmx${totalram}m -server -XX:+UseParallelGC -Dfile.encoding=UTF-8 -
Dnetlogo.extensions.dir="${BASE_DIR}/extensions")" >> $runfile

echo "java ${JVM_OPTS[@]} -classpath ${path}/netlogo/lib/app/netlogo-7.0.0.jar org.nlogo.headless.Main
--model ${modelpath} --setup-file ${path}/xml/${experiment}.xml --threads $threads --table
${path}/csv/${experiment}.csv" >> $runfile

echo "Job ID: "

# submit the job and print the job number
sbatch --parsable $runfile

# remove the temporary run file to keep things clean
rm $runfile
```

Advanced Usage

NetLogo includes built-in parallelization, but this often does not integrate well with multi-node HPC architectures. Large BehaviorSpace experiments with many parameter combinations and repetitions that require hundreds or thousands of runs may crash if distributed directly across multiple nodes.

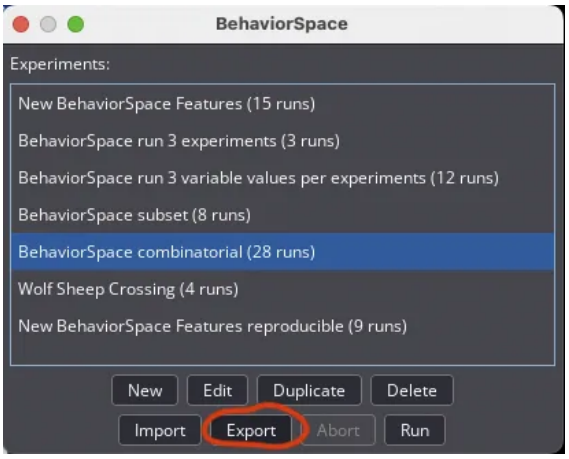
A better strategy is to split the experiment into many small jobs (one per parameter combination and repetition).

To do so, first we need to subdivide our NetLogo BehaviorSpace experiment into separate BehaviorSpace experiment files (xml files) for each parameter combination.

Dividing BehaviorSpace experiments into smaller sub-experiments

This step can be performed on your local machine, not necessarily on an HPC.

1. Export your BehaviorSpace experiment as an XML file. In BehaviorSpace, select your experiment and click *Export*.



- For maximum parallelization, set repetitions to **1** before exporting. This allows submitting a separate job for each repetition.

2. Use the exported XML file to generate individual experiment files. Instructions and a Jupyter notebook can be found here.

Alternatively, you can run the Python script below (`split_bspace_exps.py`) if you have Python installed on your machine. It is the same script linked above which generates one XML experiment file per parameter combination.

Run as:

```
python split_bspace_exps.py <input_file_name> <output_folder_path>
```

Example:

```
python split_bspace_exps.py /path/to/example-experiment.xml /path/to/output/folder/
```

Script:

```
"""split_bspace_exps.py

This script splits a NetLogo BehaviorSpace experiment into multiple experiments, each with a different
set of parameters.
The inputs are a BehaviorSpace experiment xml file and the output folder path.
The output is a folder with the new experiments, one for each combination of parameters.

Usage: python split_bspace_exps.py <input_file_name> <output_folder_path>
Example: python split_bspace_exps.py /path/to/example-experiment.xml /path/to/output/folder/
"""

import sys
import xml.etree.ElementTree as ET
from itertools import product
from pathlib import Path

def dict_product(d):
    """
    Given a dictionary of parameters and their values, return a generator that yields a dictionary for
each combination of parameters.
    """
    keys = d.keys()
    for element in product(*d.values()):
```

```
        yield dict(zip(keys, element))

def construct_pdict(file_name):
    """
    Given a BehaviorSpace experiment xml file, return a dictionary of parameters and their values.
    """
    tree = ET.parse(file_name)
    root = tree.getroot()
    pdict = {}
    for node in root.findall('.//enumeratedValueSet'):
        plist = []
        for child in node:
            plist.append(child.attrib.get('value'))
        if len(plist) > 1:
            pdict[node.get('variable')] = plist
    for node in root.findall('.//steppedValueSet'):
        rstart = node.attrib.get('first')
        rend = node.attrib.get('last')
        rstep = node.attrib.get('step')
        plist = range(int(rstart), int(rend)+int(rstep), int(rstep))
        pdict[node.get('variable')] = [str(num) for num in plist]
    print(pdict)
    return pdict

def process_xml(file, dict, new_folder):
    """
    Given a BehaviorSpace experiment xml file, a dictionary of parameters and their values, and a new
folder,
    create a new experiment xml file for each combination of parameters and save it to the new folder.
    """
    tree = ET.parse(file)
    root = tree.getroot()
    counter = 0
    name_list = []
    Path(new_folder).mkdir(parents=True, exist_ok=True)
    for cd in dict_product(dict):
        exp_name_str = 'exp' + str(counter)
        name_list.append([exp_name_str,list(cd.values())])
        root[0].set('name', exp_name_str)

        for k,v in cd.items():
            # first delete all elements matching the tag
            elements_to_remove = root.findall('.//*[@variable="'+k+'"]')
            for element in elements_to_remove:
                # Iterate through the entire tree to find the parent of the current element
                for parent in root.iter():
                    if element in parent:
                        parent.remove(element)
                        break
            # Now insert the appropriate elements
            target_item = root.find('.//constants')
            new_element_str = f'<enumeratedValueSet variable="{k}"><value value="{v}"></value>
</enumeratedValueSet>'
            new_element = ET.fromstring(new_element_str)
            target_item.append(new_element)

        with open(new_folder+exp_name_str+'.xml', 'wb') as f:
            ET.indent(root, space="  ")
            tree.write(f, 'utf-8', xml_declaration=True)
        counter += 1
    print(f'{counter} experiments created')
    print('\nexp_name', list(cd.keys()))
    for i in name_list:
        print(i)

def split_exps(file_name,new_folder):
    """
    Given a BehaviorSpace experiment xml file and a new folder,
    construct a dictionary of parameters and their values,
    and create a new experiment xml file for each combination of parameters and save it to the new
folder.
    """
    pdict = construct_pdict(file_name)
    process_xml(file_name, pdict, new_folder)

def check_args(args):
    """
    Given a list of arguments, check if the arguments are valid.
    """
    if len(args) != 3:
        print('Error: Invalid number of arguments')
        print('Usage: python split_bspace_exps.py <input_file_name> <output_folder_path>')
        print('Example: python split_bspace_exps.py /path/to/example-experiment.xml
/path/to/output/folder/')
        sys.exit(1)
    if not Path(args[1]).exists():
        print('Error: input_file_name does not exist')
        sys.exit(1)


if __name__ == '__main__':
```

```
    check_args(sys.argv)
    split_exps(sys.argv[1], sys.argv[2])
    print('Done')
```

Submitting separate jobs for each experiment file

Once you have a directory of XML files, you can use the script below (`deployment.bash`) to submit each as a separate job. It uses the `generate_and_submit.sh` script described earlier.

Run as:

```
bash deployment.bash <experiment-directory> <netlogo-model> <num-repetitions> <category> <maxtime>
<threads> <ram>
```

Where the arguments are:

- `experiment-directory` (path to directory that contains the xml files)
- `netlogo-model`
- `num-repetitions` (how many repetitions to run for each experiment. note that this should be 1 if you did not edit the number of repetitions in your experiment before exporting)
- `category` (which partition to use: short, normal, long, gpu, etc.)
- `maxtime` (in *hh:mm:ss* format)
- `threads` (number of CPU cores to request)
- `ram` (in megabytes)

Example:

```
bash deployment.bash /directory/of/xmls/ fire.nlogo 1 short 1:00:00 1 4096
```

Script:

```
### ---
### deployment.bash
### ---

if [ $# -ne 7 ]
then
  echo "Usage 'bash deployment.bash /directory/of/xmls/ /path/to/model.nlogo number-of-reps partition
max-time threads memory'"
  exit 1
fi

#XML_DIR= directory of BehaviorSpace exoeriment files
XML_DIR=$1
#MODEL= path to NetLogo model
MODEL=$2
#REPS= number of repetitions for each experiment
REPS=$3
#CATEGORY= partition
CATEGORY=$4
#MAXTIME= time limit
MAXTIME=$5
#THREADS= number of cpus (should be 1 in general)
THREADS=$6
#RAM= max ram for each cpu in mb
RAM=$7

for iter in $(seq 1 $REPS); do
  for SETUPFILE in $(ls ${XML_DIR}/*.xml); do
    EXPR="$(basename ${SETUPFILE%.*})"
    #echo $SETUPFILE
    #echo $EXPR
    bash generate_and_submit.sh $MODEL $SETUPFILE $CATEGORY $MAXTIME $THREADS $RAM;
    sleep 2 # Avoid overloading the scheduler
  done
done
```

Note: This script inserts a 2-second pause after each submission to avoid overwhelming the scheduler. For very large experiments, consider running it in the background with `screen`. For example:

```
screen -dmS session_name bash deployment.sh /directory/of/xmls/ fire.nlogo short 1:00:00 1 4096
```

The most up-to-date versions of these scripts, along with additional utilities and scripts, can be found on BehaviorSpace HPC GitHub Repository.

Extensions

NetLogo extensions allow users to write new commands and reporters in Java and other languages for use in NetLogo models. This section of the User Manual introduces extensions and shows how to use an extension in your model once you have obtained or made one.

Note that, in addition to the numerous extensions bundled with NetLogo, there are many other extensions that can be installed through the Extension Manager.

Authoring and Sharing

Interested in adding some new functionality to NetLogo by creating your own extension? Have an extension you made and want to add it to the Extension Manager? See the extension authoring page to get started.

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.

`extensions` is followed by a pair of square brackets containing a list of extension names. For example:

```
extensions [sound speech]
```

Using `extensions` instructs NetLogo to make the specified extensions' commands and reporters available in the current model, just as if they were built-in NetLogo primitives. An extension must be installed for the `extensions` keyword to take effect.

Where extensions are located

NetLogo will look for extensions in several places:

1. In the folder of the current model
2. The `extensions` folder within your installation of NetLogo. For typical NetLogo installations:
   - On Mac OS X: `/Applications/NetLogo 7.1.0-alpha1/extensions`
   - On 64-bit Windows with 64-bit NetLogo or 32-bit Windows with 32-bit NetLogo: `C:\Program Files\NetLogo 7.1.0-alpha1\extensions`
   - On 64-bit Windows with 32-bit NetLogo: `C:\Program Files (x86)\NetLogo 7.1.0-alpha1\extensions`
   - On Linux: the `bin/extensions` subdirectory of the NetLogo directory extracted from the installation `.tgz`
3. This subfolder (relative to your home directory):
   - On Mac OS X: `Library/Application Support/NetLogo`
   - On Windows: `AppData\NetLogo`
   - On Linux: `.netlogo`
4. The `.bundled` subfolder of the `extensions` folder mentioned in Item 2 (for example, `/Applications/NetLogo 7.1.0-alpha1/extensions/.bundled` on Mac OS X).

The easiest way to install new extensions is to use the Extension Manager. If an extension you want to use is not available through the Extension Manager, you will have to manually download it and place it into a location described by Item

1 or Item 2 above. Manually modifying the contents of Item 3 and Item 4 above is not supported.

The order listed above is the **priority** the Extension Manager will use. This means that if the Extension Manager finds a requested extension for a model manually installed in the `extensions` folder, it will not check the extensions library to see if there are any updated versions to install. As such, you're locked in to the manually installed version until you decide to remove it. Also, any extensions you install through the Extension Manager will override the bundled extensions that come with NetLogo. This means you can update the bundled extensions if fixes for them come out after the NetLogo release with which they were bundled, you don't have to wait for a full NetLogo release to get updates.

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`.

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.

Extension Authoring

Technical Details

For more technical information on creating your own extensions, including tutorials and samples, please see this page on the NetLogo repository wiki. It also includes ways to get help or to discuss NetLogo extension development.

Sharing Extensions with the Extension Manager

To share an extension you've made with the built-in Extension Manager, you'll need to have it added to our NetLogo-Libraries repository. See the instructions on the repository website for more information.

Color Picker

Introduction

NetLogo 7 introduces a new, more powerful way of selecting colors, which replaces Color Swatches. As was the case previously there are four different contexts in which colors can be selected: from the Tools menu, from a Color Input widget, when selecting a plot pen color and when editing a Note. The ways in which the versions differ involve whether a selected color can be copied or "picked," whether transparency is allowed, and what forms of input and output specifications are allowed, as will be described later.

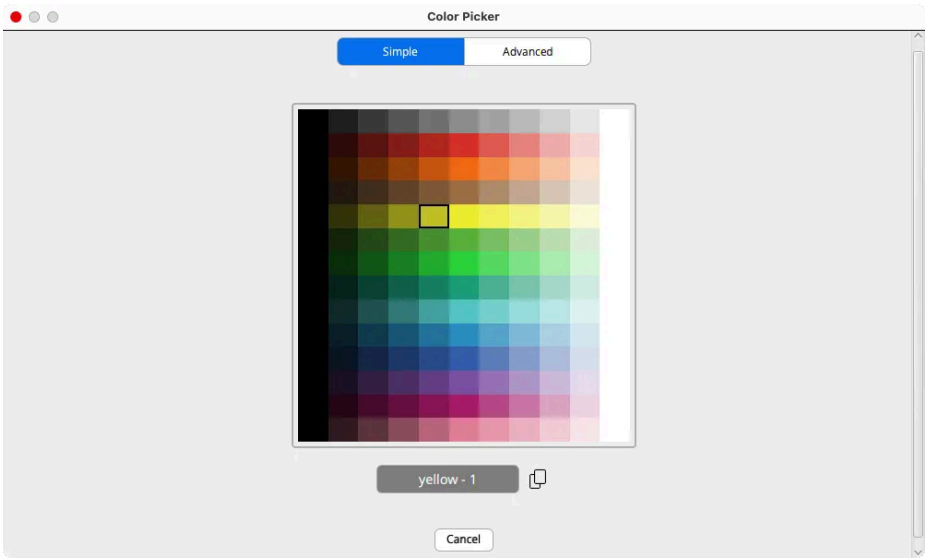
Colors in NetLogo

For information on the different ways colors are represented in NetLogo see the Color section of the NetLogo Programming Guide. The NetLogo primitives for working with colors are 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. The Palette Extension offers additional ways to work with colors.

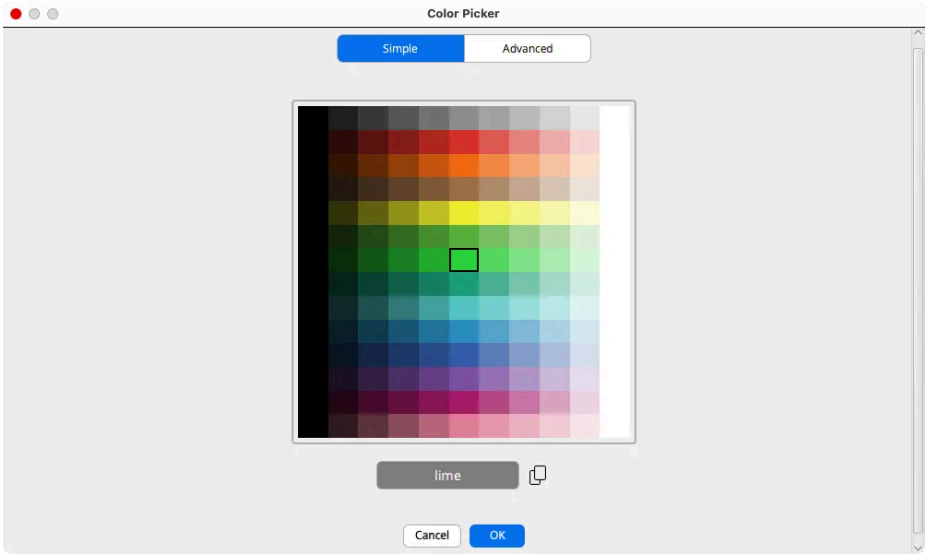
All versions of the Color Picker have two tabs, Simple, and Advanced. The Simple tab allows users to pick a color from a rectangular grid. This is analogous to the Color Swatches dialog, although it is more visually streamlined.

Simple Tab

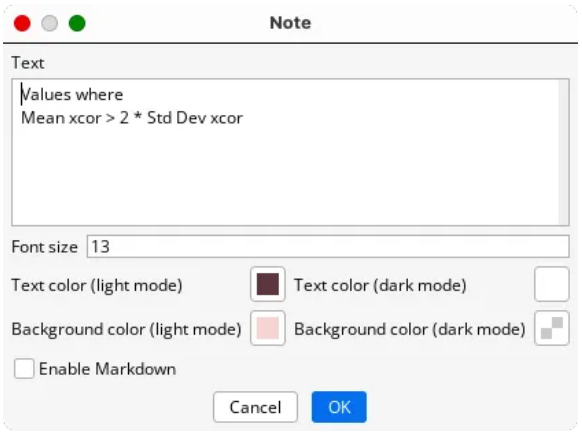
The Simple tab of the Color Picker in the Tools menu has buttons to allow the user to copy the color (icon to the right of the grey rectangle), or exit the dialog (cancel).



As shown below Simple tab for the Color Input adds a blue 'OK' button to the right of 'Cancel.' Choosing 'OK' changes the Input to the selected color, exiting the dialog. 'Copy' puts the NetLogo Color number corresponding to the selected color in the clipboard. It can then be copied to the Code Tab, or any other text area. The dialog remains open. Cancel closes the dialog without changing the color input widget.



When editing a Note, there are four buttons that can be clicked to open a Color Picker.

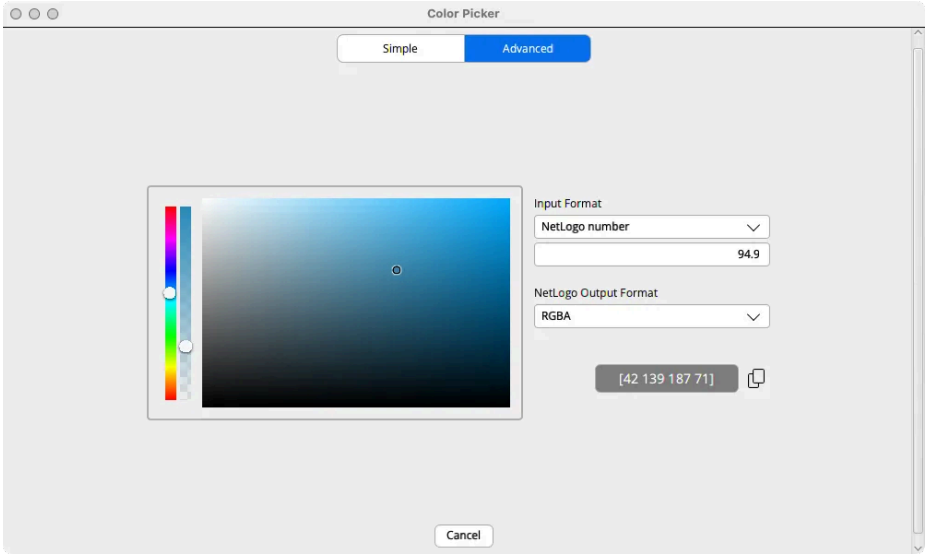


The ‘Simple’ tab of the color picker is the same, regardless of context. The ‘Advanced’ tab allows different output formats, depending on the context.

The Advanced Tab

Introduction

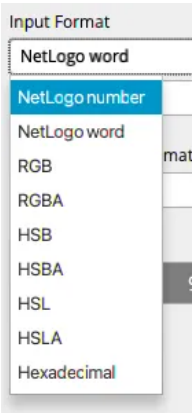
The Advanced Tab allows a much more nuanced way to choose a color.



The rainbow slider on the left allows users to select a hue. The slider to its right allows the user to select the transparency of the color. By default, the color is fully opaque, and the knob is at the top of the slider, moving it down makes the color more transparent. You can also drag the small circle in the rectangle to the right of the sliders to quickly explore variants of the selected hue. Moving from it from left to right (horizontally) increases the saturation. Moving it bottom to top (vertically) increases the lightness.

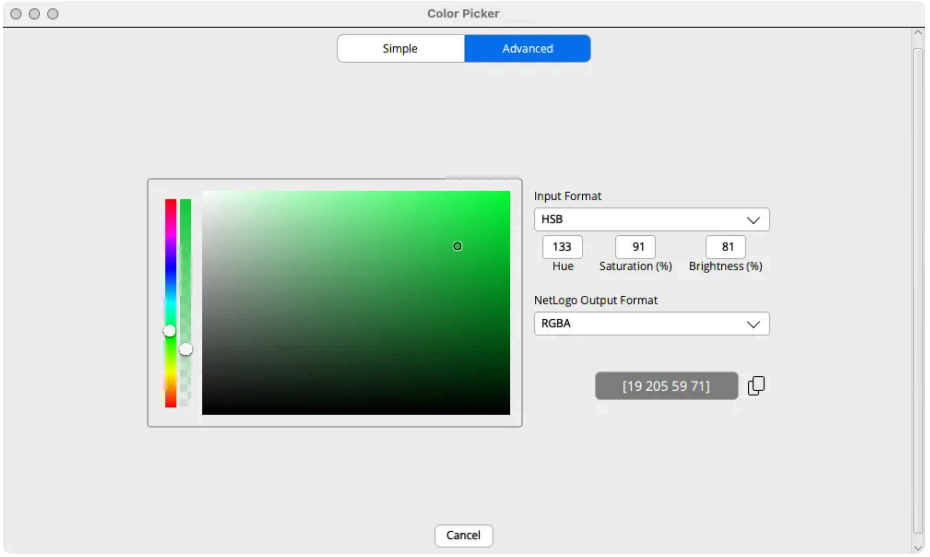
Input Format

A powerful feature of the Advanced tab is that you can import colors using the following common formats RGB, RGBA, HSB, HSBA, HSL, HSLA and hexadecimal as well as NetLogo Number or NetLogo word. Use the Input Format pulldown menu to select the format and you will be able to set the appropriate value(s). The pulldown menu below shows the output options, which have the acronyms above spelled out. There follow two examples of specifying an input. For a NetLogo word there is a text field. For HSB (Hue, Saturation, Brightness) there are three numeric fields. The color selected using the Input format can then be modified by using the hue slider, transparency slider or the palette rectangle.



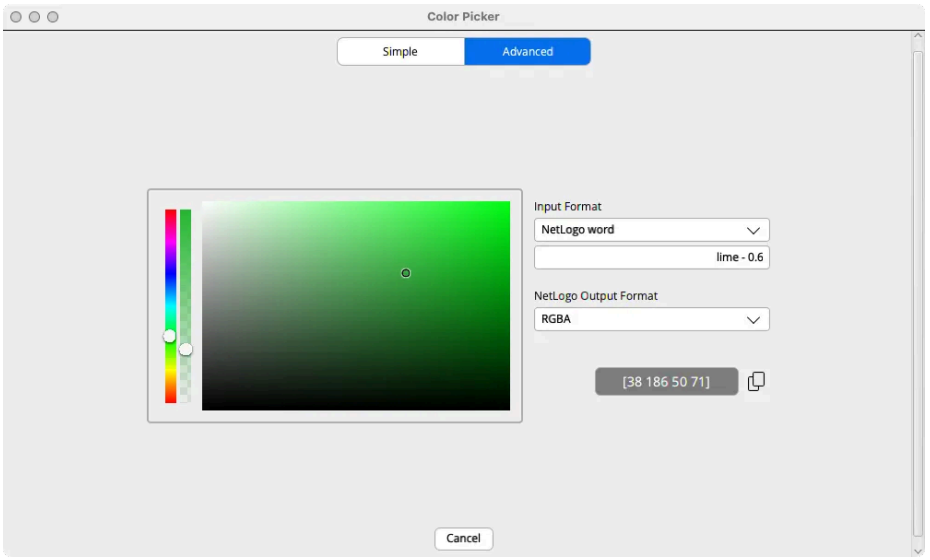
In this example the input is a NetLogo word.

In this example the input is HSB.



Output Format

The color that results after any adjustments can be output in the selected Output Format. The full list of output formats is NetLogo Number, NetLogo word and RGBA (Red, Green, Blue, Alpha). However in some contexts only a NetLogo Number can be output. As the user changes a color the value of the Input value is updated. In the image below it was initially lime - 0.3, but changed to lime - 0.6



Output Formats for Different Contexts

For the Color Input and plot pen colors the output format is a NetLogo number. Transparency is not supported for the Color Input in this release, but may be available in the future.

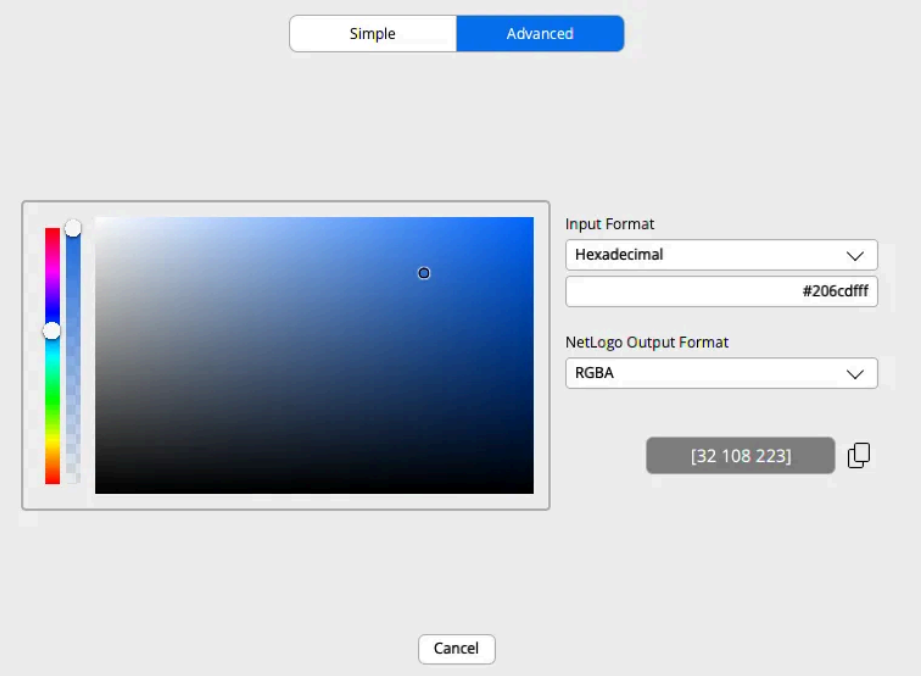
For Notes the output format is RGBA.

For the Tools menu Color Picker the output formats are NetLogo number, NetLogo word and RGBA.

Details about RGB Output

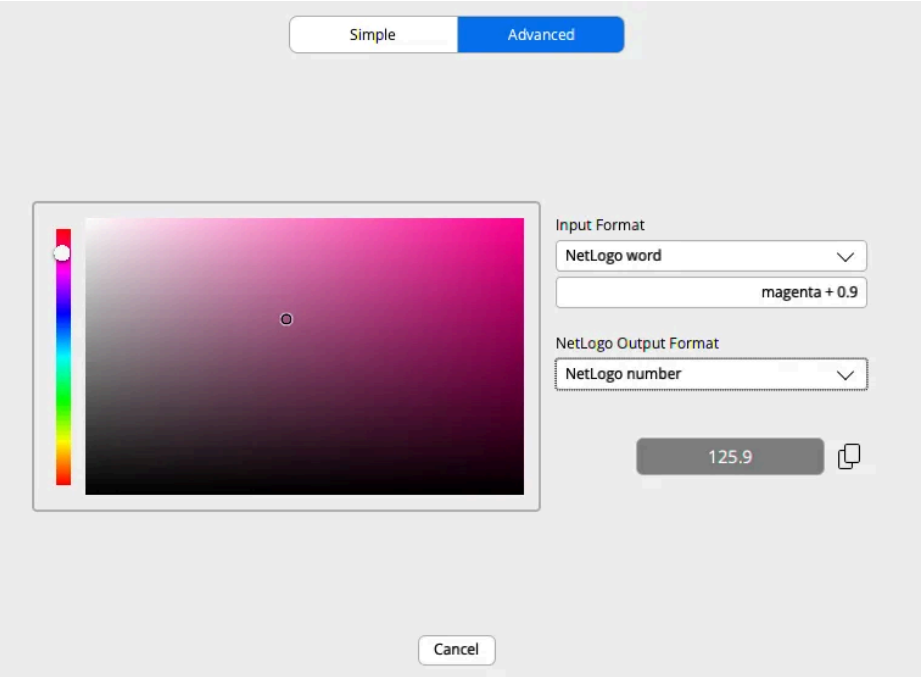
There is no explicit RGB output format. However when you select “RGBA”, and your color is opaque (maximum alpha), the result is RGB format - a 3-item list (no alpha). Otherwise “RGBA” output produces a 4-item list (with alpha). In the

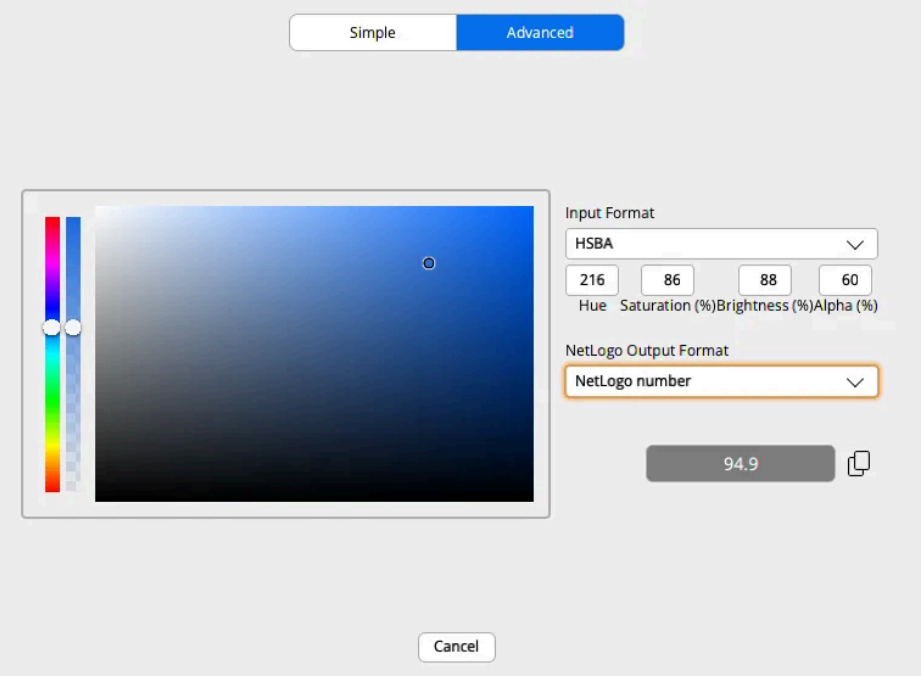
example below you can see that the output is a 3-item list, because there is no transparency and RGBA output is selected.



Details about the Alpha Bar

Some users may want to skip this section on first reading. The “alpha” bar of the Advanced tab is used to adjust the transparency of a color. If neither the input format nor the output format allows transparency the alpha bar will be hidden. The formats that do not allow transparency are: NetLogo number, NetLogo word RGB, HSB, and HSL. If the input supports alpha, but the output does not, the output dropdown will be highlighted and will have a tooltip that explains that the output color will be opaque. The first example shows the case where there is no alpha bar. The second example show the highlighted output because there is transparency, but an opaque output format.




Simple
Advanced
Input Format
HSBA
216
86
88
60
Hue
Saturation (%)
Brightness (%)
Alpha (%)
NetLogo Output Format
NetLogo number
94.9
Cancel

NetLogo 7 Introduction

The bolded items in the table of contents are the most important changes in NetLogo 7.0.0

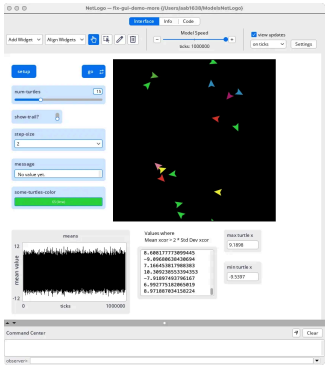
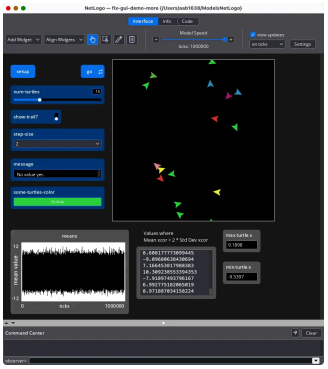
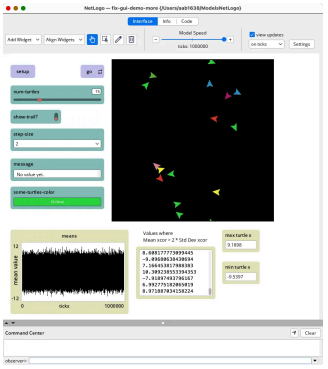
Motivation

We interviewed many NetLogo users and one of the most desired updates was a more modern User Interface (UI). In response to user feedback we have made major changes to the UI for NetLogo 7.0.0. We plan to continue modernizing NetLogo in future releases. Your feedback via the NetLogo Forum will help guide UI improvements.

Interface Tab

Overview of Visual Changes to the Interface Tab

The most obvious change is that interface elements (widgets) have changed in appearance to a modern flat look. They are larger and more legible. A new feature is color themes which affect the appearance of the entire visual interface. They have been designed with visual coherence in mind. Navigate to the Color Themes tab in the Preferences window to select Light, Dark or Classic themes. For the Classic Color Theme the pre-7.0.0 widget colors are used. In other regards it is the same as the Light Theme. The Color Theme also affects the appearance of the Info Tab and the Code Tab, as well as any other windows that open in the app.

Light Theme	Dark Theme	Classic Theme
		

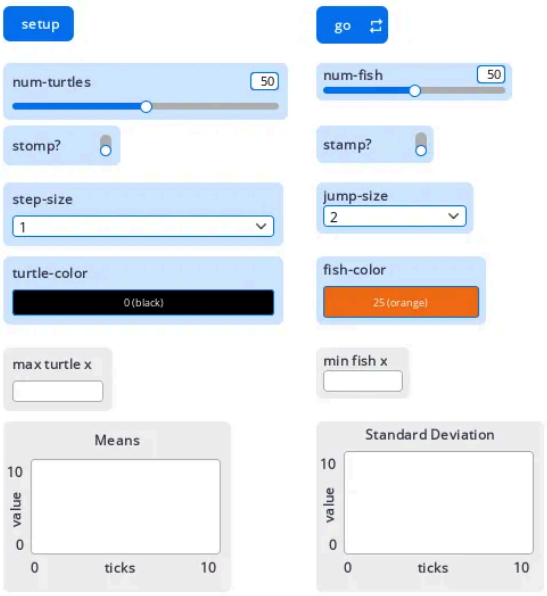
Interface Elements (Widgets)

Buttons perform actions when clicked. The input widgets are: sliders, switches, choosers and inputs. The output widgets are: monitors, plots and the output area. Notes are used to add labels, instructions and other information to the interface. Buttons and input widgets are colored shades of blue. Output widgets are shades of grey. Widgets by and large perform as they did in previous releases of NetLogo. A new feature of the slider is a text box in which values can be added directly. A new feature of the Notes is that the user can set both the text color and the background color for both light and dark mode.

A new feature is copying widgets.

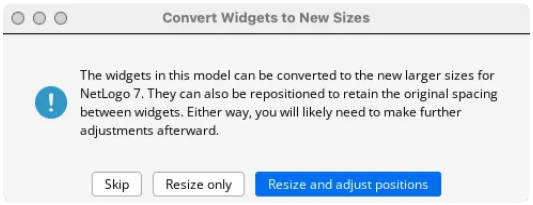
Widget Sizes

In NetLogo 7 some widget sizes have been adjusted for greater readability. The default height of sliders, switches, choosers, inputs, monitors and plots has been increased. The image below shows some widgets in their new sizes on the left, and the original/pre-NetLogo 7 sizes on the right.

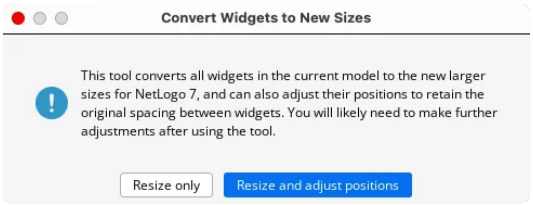


When you create a new model, widgets will appear by default in their new (generally larger) sizes. Existing models will open with new style widgets that retain their original size. Some widgets in the Models Library have been converted to the new sizes.

Note that opening an old model in NetLogo 7 will automatically prompt you to perform an automatic widget conversion, as shown in the dialog below.



Click “Resize and adjust positions” to resize the widgets and change their placements. Click “Resize only” to resize the widgets without changing their location. Since redesigning interfaces is an art, more work may be required to make the design meet your personal standards. The Edit → Undo menu will let you undo all the widget conversions at once if you are not satisfied with the result of the automatic conversion. If you choose not to make the full conversion, you can later use the tools menu item *Tools → Convert Widgets to New Sizes*.



You can also switch the size of an individual widget by clicking the *Use old sizing* checkbox. This only holds for widgets that have undergone a size change for NetLogo 7. The size of the output area and notes have remained the same.

New File Format

Models in NetLogo 7 have a new format which is extensible and makes it easier to add new features to NetLogo. The new file format follows the XML 1.0 standard, and is compatible with any external tools for working with XML. Models saved with this format will have the extension .nlogox or .nlogox3d, rather than .nlogo or .nlogo3d. All models going forward will use the new file format. Existing model files, with the extension .nlogo or .nlogo3d, can be opened in NetLogo 7 and will be automatically converted without loss of information. However .nlogox and .nlogox3d files cannot be opened with previous versions of NetLogo. Any tools users have created that rely on the format of .nlogo and .nlogo3d files will need to be re-written.

Tools in the NetLogo 7 Interface

Overview: NetLogo Interface Tool Buttons and Dropdown menus

New Interface Tools

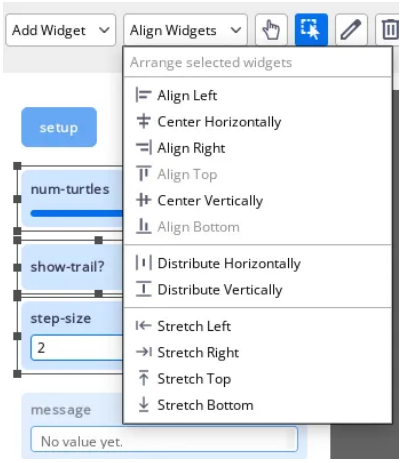
The key motivations for introducing these tools are

1. to make user actions more discoverable
2. to separate actions needed to use an existing model from those used when creating or modifying a model.

The Interface toolbar includes four new tool buttons which control the meaning of user actions. They are the Interaction Tool, the Selection Tool, the Edit Tool, and the Delete Tool. For a brief explanation of the use of these tools see this section of the Interface Tab Documentation. For details on the use of these features click the links above.

New Interface Menu

There is a new Align Widgets Menu. This menu is only active if more than one element is selected. Only one of a set of related options will be explained. Use the link above for more details.



With Align Left the left-most widget remains in place. The other widgets move to align to left-most widget edge.

Distribute Vertically keeps the top and bottom widgets fixed, and moves the other widgets so the vertical spacing between the widgets is the same. Distribute horizontally works similarly.

With Stretch Left the left-most widget remains in place. All the other widgets stretch so they have the same left-most edge.

The Color Picker

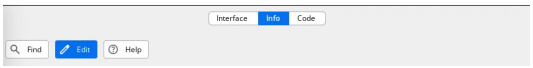
Color Swatches has been replaced by the Color Picker, a new, more powerful way of selecting colors.

Notifications

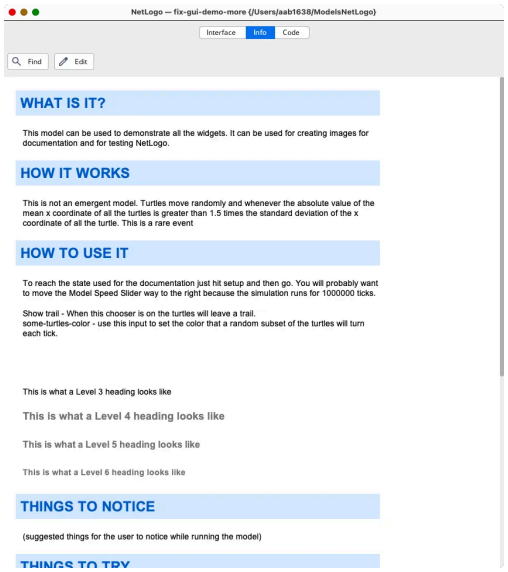
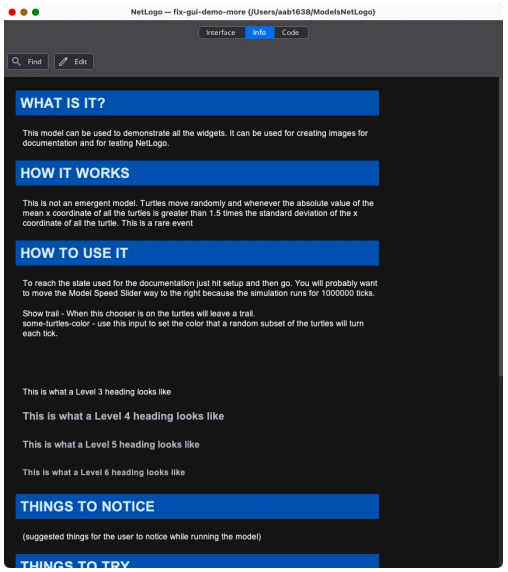
When the NetLogo team sends out a new notification message a banner will appear below the Interface Toolbar with a brief message. If you click on the message you will be taken to a web page that gives the full information of all messages to date in reverse chronological order. You can also click the X button on the right to dismiss the message.

The Info Tab

The changes to the Info Tab are purely visual. The Find and Edit icons and the section headers have been updated, as has the Help Icon that is visible when editing the Info Tab.



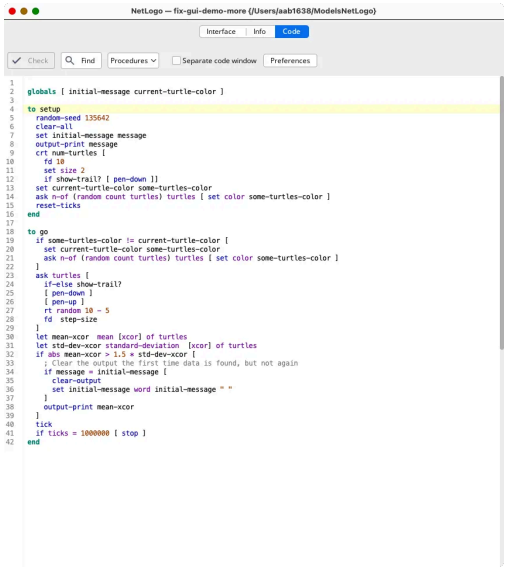
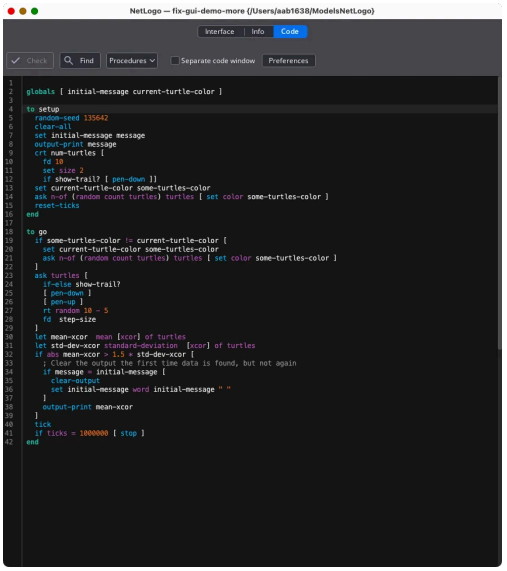
These are the new Info Tab color themes.

Light Theme	Dark Theme
	

Code Tab

Color Themes and Code Syntax Highlighting

The code syntax highlighting colors have changed slightly for the Light Color Theme, and there is a separate set of code colors for the Dark Color Theme.

Light Theme	Dark Theme
	

Code Tab Toolbar

The Code Tab toolbar features new icons. See Code Tab Toolbar for more details.

Code Tab Preferences

Code related preferences are now set in the Code tab of the Preferences dialog. See Code Preferences for details.

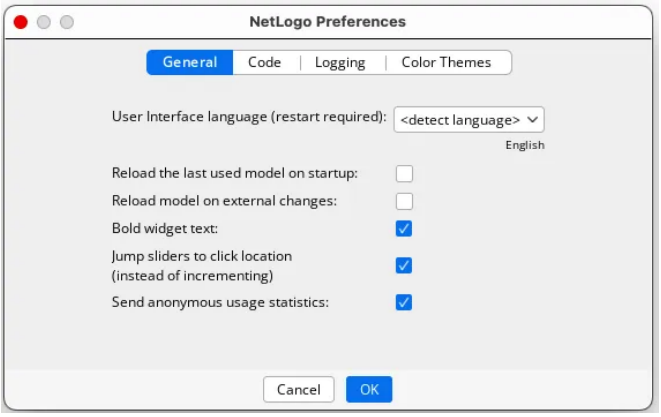
Included Files

An included (.nls) file tab now has separate code window checkbox, and a Preferences button. Note that these buttons and preferences apply to all code tabs, not just the one from which they were clicked/set. Included file tabs can be closed by clicking on the “x” on the right side of the tab. More details are in the Included Files documentation.

Menu Items

Preferences Dialog

This dialog is now organized into four tabs. You use it customize various NetLogo settings. On Windows and Linux, it is opened with the Tools menu “Preferences” item. For Macs, it’s under the “NetLogo” menu and reads “Settings”.



Here is a brief description of the new preferences. See the Preferences Documentation for more details.

Reload model on external changes: If you use an editor outside of NetLogo to change the model or any included .nls files the model will be reloaded.

Bold widget names: By default, widget names are not in bold, use this option to make them bold.

UI Scale: This option is for Linux only. It must be set to the same value as your system UI scale or NetLogo will fail to resize its visual components correctly.

Auto-focus text or tab with errors when model is compiled: If checked, NetLogo will bring you to the location of an error if one occurs during compilation.

Open code tab in separate window on startup: When checked the Code tab will appear in a separate window when NetLogo is opened.

Jump sliders to click location (instead of incrementing): When checked, clicking anywhere in the slider bar of a slider widget will move the thumb to that location. When unchecked, the pre-NetLogo 7 behavior will be applied, and clicking in the slider bar will increase or decrease the value of the slider by one increment, depending on which side of the bar is clicked.

File → Resource Manager dialog

The Resource Manager dialog allows you to view, add, or remove files that are bundled with the current NetLogo model. These files can be read during a model run using the Resource Extension.

Tools Menu Changes

The new menu items “Color Themes”, “Color Picker” and “Convert Widgets to New Sizes” have been described in earlier sections. The menu item “Color Swatches” has been removed.

The Turtle Shapes Editor and the Link Shapes Editor have been updated, as described below.

Turtle Shapes Editor and Link Shapes Editor

These Shapes Editors now have buttons at the top, for easier use. A search field has been added so users can quickly search through the model’s collection of shapes.

Help Menu

The Help Menu now has a link to the NetLogo Forum, a website to ask/answer questions and have discussions about NetLogo.

Collection of anonymous usage statistics

When you first open NetLogo after installing it, a dialog titled “Data Consent” will appear. If you select “Yes”, data will be sent to the NetLogo team. We analyze the data in order to understand how often certain NetLogo features are being used. This informs development priorities. No personal or identifying information will be collected. You can always change your preference by using the **Send anonymous usage statistics** option of the **NetLogo Preferences Dialog** to enable or disable data collection. If you have any questions or concerns about the collection and storage of this data, feel free to contact the NetLogo team at **netlogo@northwestern.edu**.

NetLogo Preferences

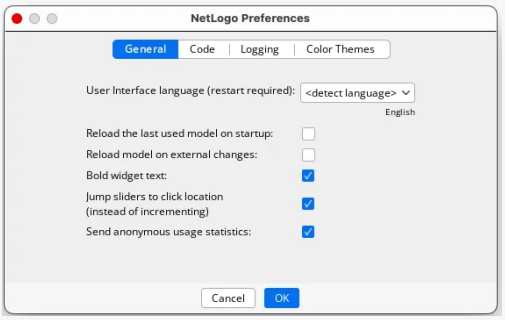
Overview

This dialog lets you customize various NetLogo settings. On Windows and Linux, it is opened with the Tools menu "Preferences" item. For Macs, it's under the "NetLogo" menu and reads "Settings".

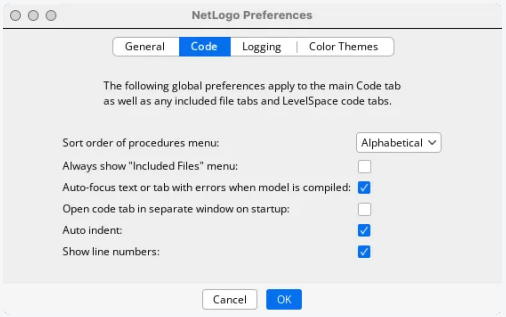
Items are organized into four tabs:

- General Preferences
- Code Preferences
- Logging Preferences
- Color Themes

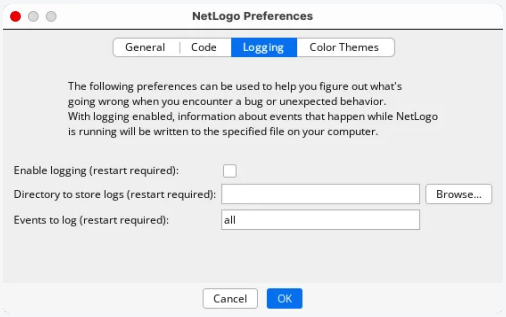
General Preferences



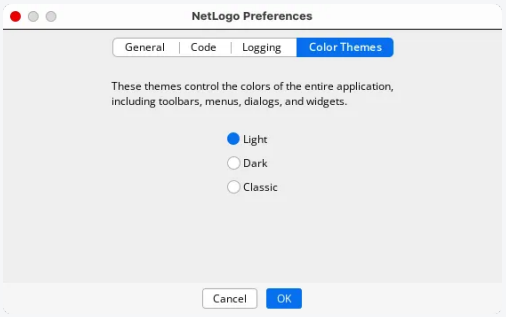
Code Preferences



Logging Preferences



Color Themes



Use the check boxes, pull-down menus and text fields to change the default preference values. When you are done, click the OK button to save your changes, or the Cancel button to revert the preferences to the state they were in before you opened the dialog. Most preferences take effect immediately. You must exit and re-open NetLogo for the preferences that indicate "(restart required)" to take effect.

General Preferences

User Interface language

Choose a language for user interface items and error messages. NetLogo must be restarted for this preference to take effect. The default option is “Detect Language”, which will set the interface language to your system language, or English if your system language is not supported by NetLogo. The other available choices are English, Spanish, Chinese, Portuguese, Russian, and Japanese.

Reload the last used model on startup

NetLogo opens with an empty model by default. When this option is selected the most recently opened model will be loaded when NetLogo starts instead of an empty model.

Reload model on external changes

If you use an editor outside of NetLogo to change the model or any included .nls files, the changed file is updated in NetLogo.

Enable remote commands

When this option is enabled, NetLogo listens for commands to execute on a Unix Domain Socket. Received commands are executed in the observer context as if they were entered in the Command Center. This can be used by external software such as editor plugins for better integration with NetLogo.

If the environment variable `XDG_RUNTIME_DIR` is set, the listening socket is created under `XDG_RUNTIME_DIR`. Otherwise, it is created under the NetLogo directory:

- On Mac OS X: `Library/Application Support/NetLogo/7.1`
- On Windows: `AppData\NetLogo\7.1`
- On Linux: `.netlogo/7.1`

The listening socket is named using the following pattern `netlogo-<PID>` where `<PID>` is the process ID of the associated NetLogo process.

To send a command to NetLogo, encode the command in a JSON object as follows, replacing `<COMMAND>` with the command. The resulting string should then be written to the listening socket.

```
{
    "type": "nl-run-code",
    "code": "<COMMAND>"
}
```

Once the command is received and processed, NetLogo will respond with the following JSON object. If the command is successfully parsed and executed, `<STATUS>` will be `ok`. Otherwise, it will be a message describing the error that occurred. Note that this will only report errors that occur during the processing of the request. Runtime errors that occur as a result of executing the provided command are not reported here.

```
{
    "type": "nl-status",
    "status": "<STATUS>"
}
```

Bold widget text

By default, widget text is not in bold, use this option to bold it.

Jump sliders to click location (instead of incrementing)

By default, clicking a slider location will cause the value of the slider to be set to the value at the click location and the slider widget thumb will jump to that location. Deselecting this option will cause the value to change by one increment towards the mouse click, also moving the widget thumb. This option applies to slider widgets and the Speed Slider.

UI Scale

This option is for Linux only. It must be set to the same value as your system UI scale or NetLogo will fail to resize its visual components correctly. This value must be a floating point number, or NetLogo will produce an error message.

Send anonymous usage statistics

By default NetLogo will now collect anonymous usage statistics from users while the application is running. Uncheck this option to disable data collection. When this option is enabled, NetLogo will collect information about your usage of tools and features of the NetLogo application and the NetLogo language. This information is analyzed by the NetLogo team in order to understand how often certain NetLogo features are being used, so as to inform development priorities. No personal or identifying information will be collected. If you have any questions or concerns about the collection and storage of this data, feel free to contact the NetLogo team at netlogo@northwestern.edu.

Code Preferences

Sort order of procedures menu

Choices:

- Alphabetical (default)
- Order of appearance

Always show “Included Files” menu

By default the “Included Files” menu is only shown when the Code Tab has an “includes” statement. With this option the “Included Files” menu will always be in the Code Tab toolbar. That menu allows user to select and open an included file or choose the **New Source File** or **Open Source File** options.

Auto-focus text or tab with errors when model is compiled

If checked, NetLogo will bring you to the location of an error if one occurs during compilation.

Open code tab in separate window on startup

When checked the Code tab will appear in a separate window when NetLogo is opened.

Auto Indent

When selected, indentation in the Code Tab will be determined by the compiler, instead of by manually pressing the Tab key to change the indentation of a line.

Show line numbers

When selected line numbers will be shown on the left edge of the Code Tab.

Show auto-complete popup while typing

When selected, a code completion popup will automatically appear as you type in the Code tab. Whether or not this preference is enabled, the popup can always be opened by pressing Control + Space.

Code Font

This preference allows you to set the font to use wherever NetLogo would normally display a monospaced font (usually for code). The list of font options is provided by your operating system, so you can add third-party fonts to the list by installing them to your operating system.

The places where the selected font will be used include:

- Code tab editor
- Command Center input
- Command Center output
- Button dialog editor
- Slider dialog editors
- Chooser dialog editor
- Input widget inputs (when Number, String (Reporter), or String (Command))
- Monitor dialog editor
- Plot dialog editors
- Output widget output
- Info tab editor
- Info tab rendered content (e.g. the rendering of `my favorite code`)
- All editors in agent inspection windows

Logging Preferences

Enable logging

Create a log file containing user actions. This option can be useful for debugging and research. NetLogo must be restarted for this preference to take effect. If logging is enabled you must also specify a log directory, or NetLogo will produce an error message.

Directory to store logs

Specifies the directory to which log files will be written. You can type in the directory name, or select a directory using the Browse... dialog. NetLogo must be restarted for this preference to take effect.

If logging is enabled one of the following may happen.

- If you type in a non-existent directory you will see a dialog that says “The specified log directory does not exist, would you like to create it?” with options “Yes” and “No.”
- You must have write permission for the log directory, otherwise you will get the message “NetLogo does not have permission to write to the specified log directory.”

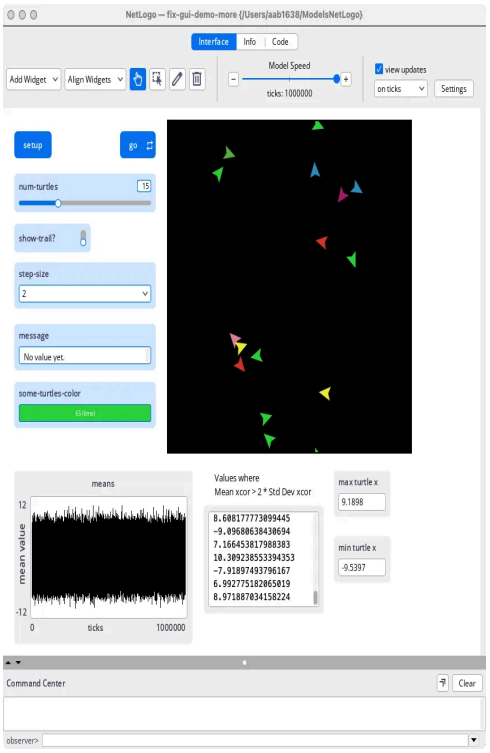
Events to log:

Enter a comma-separate list of the event names or “shortcut” names. If you leave this value blank with logging enabled a set of default events will be logged. For detailed options see the Events section of the Logging documentation. The “all” option causes all the events be logged, including button, chooser, comment, compile, command-center, global, input-box, link, model-open, slider, speed-slider, start, switch, stop, turtle, widget-edit, and tick events.

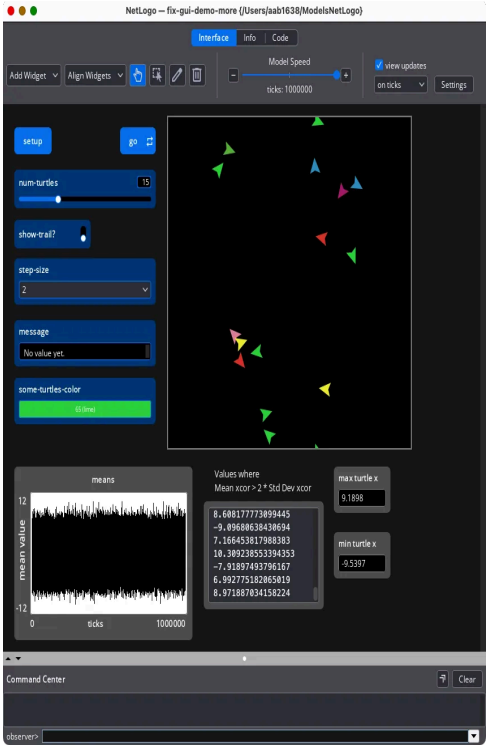
Color Themes

Use the radio buttons to change the color theme. See the images below for an example of each theme.

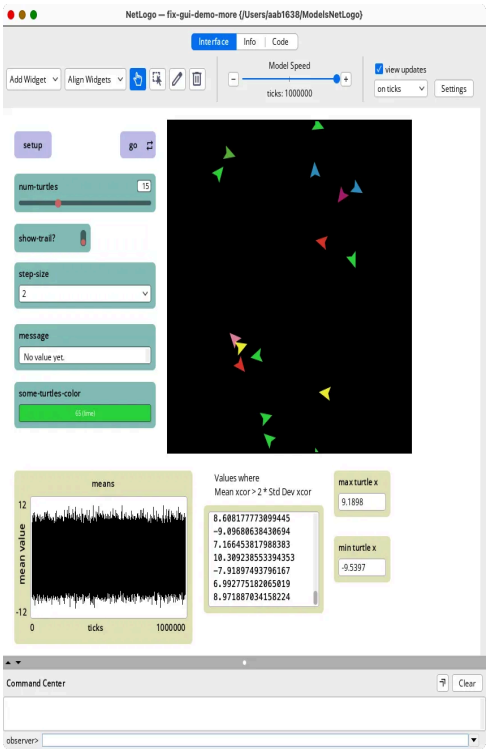
Light



Dark



Classic



Automatic

If this option is selected, NetLogo will use either the Light theme or the Dark theme, based on the system setting.

Array

Using

The array extension is pre-installed in NetLogo.

To use the array extension in your model, add a line to the top of your Code tab:

```
extensions [array]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `array` to the list.

For more information on using NetLogo extensions, see the Extensions Guide

When to Use

In general, anything you can do with an array in NetLogo, you could also just use a list for. But you may want to consider using an array instead for speed reasons. Lists and arrays 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.

Unlike NetLogo's lists and strings, arrays are "mutable". That means that you can actually modify them directly, rather than constructing an altered copy as with lists. If the array 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.

Example use of Array Extension

```
let a array:from-list n-values 5 [0]
print a
=> {{array: 0 0 0 0 0}}
print array:length a
=> 5
foreach n-values 5 [ i -> i ] [ i -> array:set a i i * i ]
print a
=> {{array: 0 1 4 9 16}}
print array:item a 0
=> 0
print array:item a 3
=> 9
array:set a 3 50
print a
=> {{array: 0 1 4 50 16}}
```

Primitives

Looking for the primitive reference for the Array extension? You can find the full reference here.

All Primitives

array:from-list

`array:from-list` *`netlogo-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.

GIS

Using

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 and GeoJSON files. The shapefile (.shp) and GeoJSON (.geojson) formats are 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

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.

For the common use case of converting a VectorDataset of points into a corresponding set of turtles with the same attributes, the gis:create-turtles-from-points primitive should be used.

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.

For the common use case of spawning a number of turtles inside the bounds of a Polygon VectorFeature, the gis:create-turtles-inside-polygon primitive should be used. Though the gis:random-point-inside primitive can also be used if you don't want the spawned turtles to take on the attributes of the Polygon.

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.

Significant updates, features, and fixes were added by James Hovet in 2020 and 2021.

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

Primitives

Looking for the primitive reference for the GIS extension? You can find the full reference here.

Coordinate System Primitives

`gis:set-transformation` · `gis:set-transformation-ds` · `gis:set-world-envelope` · `gis:set-world-envelope-ds` · `gis:world-envelope` · `gis:envelope-of` · `gis:envelope-union-of` · `gis:load-coordinate-system` · `gis:set-coordinate-system` · `gis:project-lat-lon` · `gis:project-lat-lon-from-ellipsoid`

Dataset Primitives

`gis:load-dataset` · `gis:store-dataset` · `gis:type-of` · `gis:patch-dataset` · `gis:turtle-dataset` · `gis:link-dataset`

VectorDataset Primitives

`gis:shape-type-of` · `gis:property-names` · `gis:feature-list-of` · `gis:vertex-lists-of` · `gis:centroid-of` · `gis:random-point-inside` · `gis:location-of` · `gis:set-property-value` · `gis:property-value` · `gis:find-features` · `gis:find-one-feature` · `gis:find-less-than` · `gis:find-greater-than` · `gis:find-range` · `gis:property-minimum` · `gis:property-maximum` · `gis:apply-coverage` · `gis:create-turtles-from-points` · `gis:create-turtles-from-points-`

`manual` · `gis:create-turtles-inside-polygon` · `gis:create-turtles-inside-polygon-manual` · `gis:coverage-minimum-threshold` · `gis:set-coverage-minimum-threshold` · `gis:coverage-maximum-threshold` · `gis:set-coverage-maximum-threshold` · `gis:intersects?` · `gis:contains?` · `gis:contained-by?` · `gis:have-relationship?` · `gis:relationship-of` · `gis:intersecting`

RasterDataset Primitives

`gis:width-of` · `gis:height-of` · `gis:raster-value` · `gis:set-raster-value` · `gis:minimum-of` · `gis:maximum-of` · `gis:sampling-method-of` · `gis:set-sampling-method` · `gis:raster-sample` · `gis:raster-world-envelope` · `gis:create-raster` · `gis:resample` · `gis:convolve` · `gis:apply-raster`

Drawing Primitives

`gis:drawing-color` · `gis:set-drawing-color` · `gis:draw` · `gis:fill` · `gis:paint` · `gis:import-wms-drawing`

All 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)
```

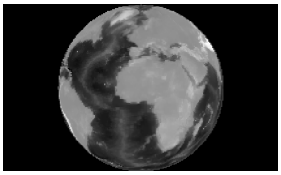
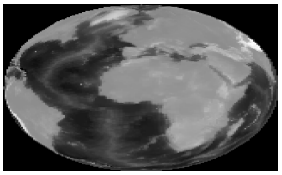
However, if you're setting the envelope of the NetLogo world, you should probably be using `gis:set-world-envelope`.

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 cover the entire NetLogo View, which in this case distorts the sha 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

gis:project-lat-lon

`gis:project-lat-lon` *`latitude longitude`*

Report the position, in NetLogo space, of the given latitude and longitude pair according to the current map projection and transformation.

Like the `gis:location-of` primitive, the reported xcor and ycor values are reported in a two-item list of `[xcor ycor]` and an empty list if the specified point is outside of the bounds of the netlogo world. For instance:

```
let location-of-abbey-road-studios gis:project-lat-lon 51.5320787 -0.1802646
let abbey-road-xcor item 0 location-of-abbey-road-studios
let abbey-road-ycor item 1 location-of-abbey-road-studios
```

Note that this primitive assumes that the given lat/lon pair are relative to the WGS84 datum/ellipsoid. If your data is based on GPS observations or GeoJson files, then your data is already relative to WGS84. If you are unsure about what datum your data is, then you should probably just assume it is WGS84 and use this primitive. However, if you do know that your data is relative to a different datum and that extra degree of precision is important to you (if you are, say, comparing values from location-of and project-lat-lon) then you should use `gis:project-lat-lon-from-ellipsoid` and specify the desired datum's ellipsoid.

gis:project-lat-lon-from-ellipsoid

`gis:project-lat-lon-from-ellipsoid` *`latitude longitude ellipsoid-radius ellipsoid-inverse-flattening`*

Report the position, in NetLogo space, of the given latitude and longitude pair according to the current map projection and transformation and the given ellipsoid parameters.

Like the `gis:location-of` primitive, the reported xcor and ycor values are reported in a two-item list of `[xcor ycor]` and an empty list if the specified point is outside of the bounds of the netlogo world.

The two defining parameters of a ellipsoid for the purposes of this primitive are the radius and the inverse flattening metric. These parameters can be easily found by examining either the WKT definition of a given projection/datum pair or the .prj file for the desired datum. For example, if you open the .prj file for a shapefile exported with the WGS66 datum in a text editor, you will see, somewhere in the file, this bit of text:
`DATUM["D_WGS_1966",SPHEROID["NWL_9D",6378145,298.25]]`. If you look at the `SPHEROID` section of that text, the first number is the radius of that ellipoid and the second is the inverse flattening.

Once we have these numbers, we can project data that is relative to WGS66 like so:

```
let location gis:project-lat-lon my-lat my-lon 6378145 298.25
```

For more on earth ellipoids, see: https://en.wikipedia.org/wiki/Earth_ellipsoid

gis:load-dataset

`gis:load-dataset` *`file`*

Loads the given data file, re-projecting the data as necessary.

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.

For ESRI shapefiles and ESRI grid files, if there is a ".prj" file associated with the file, then `gis:load-dataset` will consult that file and re-project to the current global projection if needed. If no ".prj" file is found, then the data is assumed to use the same projection as the current global coordinate system.

For GeoJSON files, as per the most-recent specification (RFC 7946), the coordinate system for GeoJSON files is always WGS84 and will be imported accordingly.

Currently, three 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, `gis: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, `gis:load-dataset` reports a RasterDataset.
- "**.geojson**" or "**.json**" (GeoJSON): contains vector data similar to shapefiles and similarly reports a VectorDataset.

Note that not all aspects of the GeoJSON standard are supported. In particular, to be properly imported, a GeoJSON file must satisfy the following:

- It only contain numeric or string data within the properties. all other json data will be stringified.
- All "Features" within a "FeatureCollection" must be of the same shape type ("Point", "LineString", etc.) Additionally, if not all the "Features" in the "FeatureCollection" have the same set of property names, default values will be supplied where there are missing entries (0 for numbers and "" for strings.)
- It must not use "GeometryCollection", which is not supported Elevation/Z data is partially supported. For both .geojson and .shp files, single points with Z data will have their Z coordinate moved to a new "_Z" property that can be accessed with `gis:property-value` like any other property. Any Z information for other shape types will be discarded upon import.

gis:store-dataset

`gis:store-dataset` *`dataset file`*

Exports the given dataset to the given file.

For VectorDatasets, two file formats are supported, ESRI shapefiles and GeoJSON. If the given file name ends in ".geojson" or ".json", then the file will be exported as a GeoJSON file. If the file name ends in any other extension or no extension at all, the dataset will be exported as a shapefile and the associated file extensions will be supplied (".shp", ".prj", etc.)

For RasterDatasets, only ESRI ASCII grid files are supported and the associated file extensions will be automatically supplied.

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.

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 `gis:apply-raster`. `gis: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.

gis:shape-type-of

`gis:shape-type-of` *`vector-dataset`*

Reports the shape type of the given dataset. The possible output values are "POINT", "LINE", and "POLYGON".

gis:property-names

`gis:property-names` *`vector-dataset`*

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` *`vector-dataset`*

Reports a list of all VectorFeatures in the given dataset.

gis:vertex-lists-of

`gis:vertex-lists-of` *`vector-feature`*

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` *`vector-feature`*

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:random-point-inside

`gis:random-point-inside` *`vector-feature`*

Reports a single randomly-generated Vertex that lies within the given feature polygon. Generated points are uniformly distributed within the polygon and both polygon holes and multi-polygon features are supported.

```
; create 100 turtles randomly distributed throughout a VectorFeature [[vf]]
crt 100 [
  let loc gis:location-of gis:random-point-inside vf
  set xcor item 0 loc
  set ycor item 1 loc
]
```

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:set-property-value

`gis:set-property-value` *`vector-feature property-name value`*

Sets the value of the given property of the given VectorFeature. The type of the given value (string or number) must match the property type of the VectorFeature. This command may be used in conjunction with store-dataset to make changes to VectorFeatures and export them back as GIS datasets.

gis:property-value

`gis:property-value` *`vector-feature 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` *`vector-dataset property-name specified-value`*

Reports a list of all VectorFeatures in the given dataset whose value for the property *property-name* matches *specified-value* (a string or number).

For strings, value comparison is not case sensitive, and the wildcard character "*" will match any number of occurrences (including zero) of any character.

For numbers, beware that there are some numbers that are too big to be represented as an integer in NetLogo, so integer data imported into NetLogo may start to lose precision if they are larger than around 10^14.

gis:find-one-feature

`gis:find-one-feature` *`vector-dataset property-name specified-value`*

Reports the first VectorFeature in the dataset whose value for the property *property-name* matches the given string or number. Reports `nobody` if no matching VectorFeature is found.

For strings, 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.

For numbers, beware that there are some numbers that are too big to be represented as an integer in NetLogo, so if you want to be able to identify features based on a unique ID, keep the IDs you use on the relatively small side to play it safe. Any number that can fit into a 32 bit unsigned integer will work, but any larger than 10^14 and you could run into issues. (Helpfully, the NetLogo app will warn you if you try to type one of these too-large numbers into the editor if you want to check.)

gis:find-less-than

`gis:find-less-than` *`vector-dataset 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` *`vector-dataset 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` *`vector-dataset 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` *`vector-dataset 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` *`vector-dataset 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` *`vector-dataset 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:create-turtles-from-points

`gis:create-turtles-from-points` *`vector-dataset breed commands`*

For each point in a VectorDataset of points, create a turtle of the specified breed at the point's location. For each agent variable (as defined in `<breeds>-own`), if there is a property with the same name in the dataset, set that variable's value to be the value of that property. Finally, execute any commands in the optional command block. To use generic turtles as the chosen breed, simply supply `turtles` as the breed argument.

Property names and variable names are compared case-insensitively. Keep in mind that when importing shapefiles, property names may be modified for backwards compatibility reasons. The names given by **`gis:property-names`** can always be trusted as authoritative. For manually specifying a mapping between property names and variable names, see the **`create-turtles-from-points-manual`** primitive.

For multi-point datasets, a turtle is created at each point of multi-point feature, each with the same set of variable values.

Built-in variables such as "label" and "heading" are supported. NetLogo color numeric representations are supported for setting "color" and "label-color", as well as the 15 default color string representations ("red", "blue", "black", etc.).

As an example: say you wanted to create a turtle of breed "cities/city" for each city in a dataset of cities like the one included in the "GIS General Examples" model from the models library. The cities dataset has four properties, "NAME", "COUNTRY", "POPULATION", and "CAPITAL". To map them all to NetLogo turtle variables and set their shapes to circles, you could do this:

```
extensions [gis]
breed [cities city]
cities-own [name country population capital]
globals [cities-dataset]
```

```
to setup
  set cities-dataset gis:load-dataset "cities.shp"
  gis:create-turtles-from-points cities-dataset cities [
    set shape "circle"
  ]
end
```

gis:create-turtles-from-points-manual

`gis:create-turtles-from-points-manual` *`vector-dataset breed property-mapping commands`*

Like `gis:create-turtles-from-points`, creates a turtle for each point in a VectorDataset of points and populates their variables with the values of corresponding gis properties.

This primitive can be used to specify additional mappings between gis property names and NetLogo variable names. These mappings are specified as a list of lists of strings like so: `[["property-name" "turtle-variable-name"] ["property-name" "turtle-variable-name"] (etc.)]`

These manual mappings modify the automatic mapping process that takes place in the `gis:create-turtles-from-points` primitive, so you only need to specify the changes you want to make to the default mappings, and the rest of the mappings will be untouched.

To return to the cities example from the create-turtles-from-points entry, the variable name "capital" is not very descriptive. something like "is-capital?" fits the NetLogo style much better. To make that change, you would modify the example like so.

```
extensions [gis]
breed [cities city]
cities-own [name country population is-capital?]
globals [cities-dataset]

to setup
  set cities-dataset gis:load-dataset "cities.shp"
  ;; Since we only want to change how the "CAPITAL" property is mapped, we only need to specify that
one change.
  gis:create-turtles-from-points-manual cities-dataset cities [["CAPITAL" "is-capital?"]] [
    set shape "circle"
  ]
  ;; Each city turtle still has a name, country, and population set just like the non-manual version.
end
```

gis:create-turtles-inside-polygon

`gis:create-turtles-inside-polygon` *`vector-feature breed n commands`*

Randomly create "n" turtles of the given breed within the given VectorFeature and for each agent variable (as defined in `<breeds>-own`), if there is a property with the same name in the dataset, set that variable's value to be the value of that property. Finally, execute any commands in the optional command block. To use generic turtles as the chosen breed, simply supply `turtles` as the breed argument.

Property names and variable names are compared case-insensitively. Keep in mind that when importing shapefiles, property names may be modified for backwards compatibility reasons. The names given by `gis:property-names` can always be trusted as authoritative. For manually specifying a mapping between property names and variable names, see the `gis:create-turtles-inside-polygon-manual` primitive.

Built-in variables such as "label" and "heading" are supported. NetLogo color numeric representations are supported for setting "color" and "label-color", as well as the 15 default color string representations ("red", "blue", "black", etc.).

As an example: say you had a VectorDataset of polygons representing different zip codes within a state and you want to create 100 different turtles within each zip code and have each turtle know which zip code it originated in. The

VectorDataset has a field named “zip”, so you should add a variable named “zip” to the turtles with `turtles-own`. Then, loop through each VectorFeature in the VectorDataset and use the `gis:create-turtles-inside-polygon` primitive to create 100 new turtles.

```
extensions [gis]
globals [dataset]
turtles-own [zip]

to setup
  set dataset gis:load-dataset "dataset.shp"
  gis:set-world-envelope envelope-of dataset
  gis:set-drawing-color red
  gis:draw dataset 1

  foreach gis:feature-list-of dataset [ this-vector-feature ->
    gis:create-turtles-inside this-vector-feature turtles 100 [
      set shape "person"
    ]
  ]
end
```

gis:create-turtles-inside-polygon-manual

`gis:create-turtles-inside-polygon-manual` *vector-feature breed n property-mapping commands*

Like `gis:create-turtles-inside-polygon`, creates “n” different turtles within the given VectorFeature and populates their agent variables with values corresponding to the property values of the VectorFeature.

This primitive can be used to specify additional mappings between gis property names and NetLogo variable names. These mappings are specified as a list of lists of strings like so: `[["property-name" "turtle-variable-name"] ["property-name" "turtle-variable-name"] (etc.)]`

These manual mappings modify the automatic mapping process that takes place in the `gis:create-turtles-inside-polygon` primitive, so you only need to specify the changes you want to make to the default mappings, and the rest of the mappings will be untouched.

See the `gis:create-turtles-from-points-manual` entry for an example of how to override default mappings with manual ones.

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`*

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
y	Interior	T	*	*
	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.

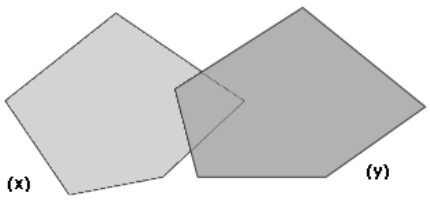
gis:relationship-of

`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:



have the following DE-9IM matrix:

		x		
		Interior	Boundary	Exterior
y	Interior	2	1	2
	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.

gis:width-of

gis:width-of *raster-dataset*

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 *raster-dataset*

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 *raster-dataset 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 *raster-dataset 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` ***`raster-dataset`***

Reports the highest value in the given raster dataset.

gis:maximum-of

`gis:maximum-of` ***`raster-dataset`***

Reports the lowest value in the given raster dataset.

gis:sampling-method-of

`gis:sampling-method-of` ***`raster-dataset`***

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 `gis:raster-sample`, `gis:resample`, `gis:convolve`, and `gis: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` ***`raster-dataset 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 `gis:raster-sample`, `gis:resample`, `gis:convolve`, and `gis:apply-raster`. The sampling method must be one of the following:

- `"NEAREST_NEIGHBOR"`
- `"BILINEAR"`
- `"BICUBIC"`
- `"BICUBIC_2"`

See `gis:sampling-method-of` above for a more specific description of each sampling method.

gis:raster-sample

`gis:raster-sample` *`raster-dataset 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 `gis: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 `gis: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` *`raster-dataset 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` *`raster-dataset 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 `gis: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` *`raster-dataset 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` *`raster-dataset 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 `gis:resample` rather than `gis: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 `gis:raster-sample` reports values for locations outside the raster.

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` *`raster-dataset 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).

Networks

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 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.

Primitives

Looking for the primitive reference for the Networks extension? You can find the full reference here.

Centrality Measures

`nw:betweenness-centrality` · `nw:eigenvector-centrality` · `nw:page-rank` · `nw:closeness-centrality` · `nw:weighted-closeness-centrality`

Clusterer/Community Detection

`nw:bicomponent-clusters` · `nw:weak-component-clusters` · `nw:louvain-communities` · `nw:maximal-cliques` · `nw:biggest-maximal-cliques`

Clustering Measures

`nw:clustering-coefficient` · `nw:modularity`

Context Management

`nw:set-context` · `nw:get-context` · `nw:with-context`

Generators

`nw:generate-preferential-attachment` · `nw:generate-random` · `nw:generate-watts-strogatz` · `nw:generate-small-world` · `nw:generate-lattice-2d` · `nw:generate-ring` · `nw:generate-star` · `nw:generate-wheel`

Import and Export

`nw:save-matrix` · `nw:load-matrix` · `nw:save-graphml` · `nw:load-graphml` · `nw:load` · `nw:load-from-string` · `nw:save` · `nw:save-to-string`

Path and Distance

`nw:turtles-in-radius` · `nw:turtles-in-reverse-radius` · `nw:distance-to` · `nw:weighted-distance-to` · `nw:path-to` · `nw:turtles-on-path-to` · `nw:weighted-path-to` · `nw:turtles-on-weighted-path-to` · `nw:mean-path-length` · `nw:mean-weighted-path-length`

All Primitives

nw: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 are exceptions to this rule.)

See the usage section for a much more detailed explanation of `nw:set-context`.

nw: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 we 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)]]
```

nw: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.

nw:turtles-in-radius

`nw:turtles-in-radius` *`radius`*

Returns the set of turtles within the given distance (number of links followed) of the calling turtle in the current context, including the calling turtle.

`nw:turtles-in-radius` form will follow both undirected links and directed **out** links. You can think of `turtles-in-radius` as "turtles **who I can get to** 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.

nw:turtles-in-reverse-radius

`nw:turtles-in-reverse-radius` *`radius`*

Like `nw:turtles-in-radius`, but follows in-links instead of out-links. Also follow undirected links. You can think of `turtles-in-reverse-radius` as "turtles **who can get to me** in *radius* steps".

nw:distance-to

`nw:distance-to` *`target-turtle`*

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. Each link counts for a distance of one.

Example:

```
to go
  clear-all
  create-turtles 5
  ask turtle 0 [ create-link-with turtle 1 ]
  ask turtle 1 [ create-link-with turtle 2 ]
  ask turtle 0 [ create-link-with turtle 3 ]
  ask turtle 3 [ create-link-with turtle 4 ]
  ask turtle 4 [ create-link-with turtle 2 ]
  ask turtle 0 [ show nw:distance-to turtle 2 ]
end
```

Will output:

```
(turtle 0): 2
```

nw:weighted-distance-to

`nw:weighted-distance-to` *`target-turtle weight-variable`*

Like `nw:distance-to`, but takes link weight into account. 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:weighted-distance-to turtle 2 weight ]
end
```

Will output:

```
(turtle 0): 1.5
```

nw:path-to

`nw:path-to` *`target-turtle`*

Finds the shortest path to the target turtle and reports the actual path between the source and the target turtle. The path is reported as the list of links that constitute the path.

If no path exist between the source and the target turtles, `false` will be reported instead.

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 ]
  ask turtle 1 [ create-link-with turtle 2 ]
  ask turtle 0 [ create-link-with turtle 3 ]
  ask turtle 3 [ create-link-with turtle 4 ]
  ask turtle 4 [ create-link-with turtle 2 ]
  ask turtle 0 [ show nw:path-to turtle 2 ]
end
```

Will output:

```
(turtle 0): [(link 0 1) (link 1 2)]
```

nw:turtles-on-path-to

`nw:turtles-on-path-to` *`target-turtle`*

Like `nw:path-to`, but the turtles on the path are reported, instead of the links, including the source turtle and target turtle.

Example:

```
to go
  clear-all
  create-turtles 5
  ask turtle 0 [ create-link-with turtle 1 ]
  ask turtle 1 [ create-link-with turtle 2 ]
  ask turtle 0 [ create-link-with turtle 3 ]
  ask turtle 3 [ create-link-with turtle 4 ]
  ask turtle 4 [ create-link-with turtle 2 ]
  ask turtle 0 [ show nw:turtles-on-path-to turtle 2 ]
end
```

Will output:

```
(turtle 0): [(turtle 0) (turtle 1) (turtle 2)]
```

nw:weighted-path-to

`nw:weighted-path-to` *`target-turtle weight-variable`*

Like `nw:path-to`, but takes link weight into account.

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:weighted-path-to turtle 2 weight ]
end
```

Will output:

```
(turtle 0): [(link 0 3) (link 3 4) (link 2 4)]
```

nw:turtles-on-weighted-path-to

`nw:turtles-on-weighted-path-to` *`target-turtle weight-variable`*

Like `nw:turtles-on-path-to`, but takes link weight into account.

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:weighted-path-to turtle 2 weight ]
end
```

Will output:

```
(turtle 0): [(turtle 0) (turtle 3) (turtle 4) (turtle 2)]
```

nw:mean-path-length

`nw:mean-path-length`

Reports the average shortest-path length between all distinct pairs of nodes in the current context.

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
  create-turtles 1 ; create a new, disconnected turtle
  show nw:mean-path-length
end
```

Will ouput:

```
observer: 1.3333333333333333
observer: false
```

nw:mean-weighted-path-length

`nw:mean-weighted-path-length` ***`weight-variable`***

Like `nw:mean-path-length`, but takes into account link weights.

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: 2.6666666666666665
observer: false
```

nw: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.

nw: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.

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.

nw: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.

nw: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.

nw:weighted-closeness-centrality

`nw:weighted-closeness-centrality` *`link-weight-variable`*

This is identical to `nw:closeness-centrality`, except that weights provided by the given variable are treated as the distances of links.

nw: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 turtles
  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
```

nw:modularity

`nw:modularity`

Modularity is a measurement of community structure in the network. It is defined based on the number of in-community links versus the number of between-community links. This primitive takes as input a list of agentsets, where each of the agentsets is one the communities that you're separating the network into.

This measurement works on undirected, directed, and mixed-directedness networks. In the case of mixed-directedness, undirected links are treated essentially the same as two opposing directed links. It does not take weight into account.

Example:

```
nw:modularity (list (turtles with [ color = blue ]) (turtles with [ color = red ]))
```

nw: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.

nw: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.

nw:louvain-communities

`nw:louvain-communities`

Detects community structure present in the network. It does this by maximizing `modularity` using the Louvain method. The communities are reported as a list of turtle-sets.

Often you'll want to tell turtles about the community that they are in. You can do this like so:

```
turtles-own [ community ]

...

foreach nw:louvain-communities [ [comm] ->
  ask comm [ set community comm ]
]
```

You can give each community its own color with something like this:

```
let communities nw:louvain-communities
let colors sublist 0 (length communities) base-colors
(foreach communities colors [ [community col] ->
  ask community [ set color col ]
])
```

nw: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.

nw: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.

nw:generate-preferential-attachment

`nw:generate-preferential-attachment` *`turtle-breed link-breed num-nodes min-degree optional-command-block`*

Generates a new network using a version of 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.

Generation works as follows turtles are added, one by one, each forming `min-degree` links to a previously added turtles, until `num-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.

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 1 [ set color red ]
```

nw:generate-random

`nw:generate-random` *`turtle-breed link-breed num-nodes connection-probability optional-command-block`*

Generates a new random network of *num-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 ]
```

nw:generate-watts-strogatz

`nw:generate-watts-strogatz` *`turtle-breed link-breed num-nodes neighborhood-size rewire-probability optional-command-block`*

Generates a new Watts-Strogatz small-world network.

The algorithm begins by creating a ring of nodes, where each node is connected to `neighborhood-size` nodes on either side. Then, each link is rewired with probability `rewire-prob`.

If you specify an *optional-command-block*, it is executed for each turtle in the newly created network. Furthermore, the turtles are generated in the order they appear as in `create-ordered-turtles`. So, in order to lay the ring out as a ring, you can do something like:

```
nw:generate-watts-strogatz turtles links 50 2 0.1 [ fd 10 ]
```

nw: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. Note that `nw:generate-watts-strogatz` generates a more traditional small-world network.

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 ]
```

The turtles are generated in the order that they appear in the lattice. So, for instance, to generate a kleinberg lattice accross the entire world, and lay it out accordingly, try the following:

```
nw:generate-small-world turtles links world-width world-height 2.0 false
(foreach (sort turtles) (sort patches) [ [t p] -> ask t [ move-to p ] ])
```

nw: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 ]
```

The turtles are generated in the order that they appear in the lattice. So, for instance, to generate a lattice accross the entire world, and lay it out accordingly, try the following:

```
nw:generate-lattice-2d turtles links world-width world-height false
(foreach (sort turtles) (sort patches) [ [t p] -> ask t [ move-to p ] ])
```

nw:generate-ring

`nw:generate-ring` *turtle-breed link-breed num-nodes optional-command-block*

Generates a ring network of *num-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 ]
```

nw:generate-star

`nw:generate-star` *`turtle-breed link-breed num-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 ]
```

nw:generate-wheel

`nw:generate-wheel` *`turtle-breed link-breed num-nodes optional-command-block`*

Variants:

- `nw:generate-wheel-inward`
- `nw:generate-wheel-outward`

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 ]
```

nw: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.

nw: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 used 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 ]
```

nw: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. Note that the breed is stored in its plural form.
- 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*.

nw: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. The load expects the *plural form* of the breed for a turtle or link, it will not recognize the singular form.

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 exten-

sion 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
```

nw:load

`nw:load` *`file-name default-turtle-breed default-link-breed optional-command-block`*

Filetype specific variants:

- `nw:load`
- `nw:load-dl`
- `nw:load-gdf`
- `nw:load-gexf`
- `nw:load-gml`
- `nw:load-vna`

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`.

nw:load-from-string

`nw:load-from-string` *`format data default-turtle-breed default-link-breed optional-command-block`*

Load network `data` from an in-memory string rather than from a file. This is useful in NetLogo Web and other contexts where the data comes from somewhere other than the file system, such as the `fetch` or `resource` extensions.

Because there is no file name to infer the file-type from, the first argument, `format`, names the format explicitly. It is case-insensitive and an optional leading dot is ignored, so `"gml"`, `"GML"`, and `".gml"` are all equivalent. The supported formats are:

- `dl`
- `gdf`

- `gexf`
- `gml`
- `graphml`
- `matrix`
- `vna`

Aside from taking its data from a string and its format from the `format` argument, `nw:load-from-string` behaves exactly like the corresponding `nw:load-*` primitive, including matching node and edge attributes to turtle and link variables and honoring `breed`.

For example, to load a GML network fetched from a URL:

```
fetch:url-async "https://example.com/network.gml" [ [gml] ->
  nw:load-from-string "gml" gml turtles links
] [ [err] -> print err ]
```

nw:save

`nw:save` *`file-name`*

Filetype specific variants:

- `nw:save-dl`
- `nw:save-gdf`
- `nw:save-gexf`
- `nw:save-gml`
- `nw:save-vna`

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`.

nw:save-to-string

`nw:save-to-string` *`format`*

Serialize the current network context to a string in the given format and report it, rather than writing to a file. This is the counterpart to `nw:load-from-string`, and is useful in NetLogo Web and other contexts where the result

needs to go somewhere other than the file system.

The `format` argument names the format explicitly and takes the same values as `nw:load-from-string`. It is case-insensitive and an optional leading dot is ignored, so `"gml"`, `"GML"`, and `".gml"` are all equivalent. The supported formats are:

- `dl`
- `gdf`
- `gexf`
- `gml`
- `graphml`
- `matrix`
- `vna`

Aside from reporting a string instead of writing to a file, `nw:save-to-string` behaves exactly like the corresponding `nw:save-*` primitive, with the same limitations regarding attributes and multigraphs.

For example, to save a network and hand it off to a browser download in NetLogo Web:

```
let gml nw:save-to-string "gml"
send-to:file "my-network.gml" gml
```

CSV

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 [ [ [col-total new-val] -> col-total + new-val] 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

Looking for the primitive reference for the CSV extension? You can find the full reference here.

Parsing CSV input to NetLogo data

`csv:from-row` · `csv:from-string` · `csv:from-file`

Formatting NetLogo data as CSV

`csv:to-row` · `csv:to-string` · `csv:to-file`

All Primitives

csv:from-row

csv:from-row *netlogo-string*
csv:from-row *netlogo-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]
```

csv:from-string

`csv:from-string` *`netlogo-string`*
`csv:from-string` *`netlogo-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]]
```

csv:from-file

`csv:from-file` *`csv-file`*
`csv:from-file` *`csv-file 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]
```

csv:to-row

csv:to-row *netlogo-list*
csv:to-row *netlogo-list delimiter*

Reports the given list as a CSV row. For example:

```
observer> show csv:to-row ["one" 2 true]
observer: "one,2,true"
```

csv:to-string

csv:to-string *netlogo-list*
csv:to-string *netlogo-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"
```

csv:to-file

csv:to-file *csv-file netlogo-list*
csv:to-file *csv-file netlogo-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
```

Profiler

Using the 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.

How to use

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
```

Another way to use the profiler is to export its raw data using the `csv` extension and the `profiler:data` primitive:

```
extensions [ csv profiler ]

to profile
  setup                                       ;; set up the model
  profiler:start                              ;; start profiling
  repeat 20 [ go ]                            ;; run something you want to measure
  profiler:stop                               ;; stop profiling
  csv:to-file "profiler_data.csv" profiler:data  ;; save the results
  profiler:reset                              ;; clear the data
end
```

Running the above procedure will write a `profiler_data.csv` file that you can then load into your favorite data analysis program. Here is an example data file produced using the Wolf Sheep Predation model:

```
procedure,calls,inclusive_time,exclusive_time
EAT-SHEEP,1085,1.237283,11.637364
```

```
DISPLAY-LABELS,20,17.389512,17.389512
MOVE,3578,14.164451,14.164451
GO,20,683.414619,604.773837
GRASS,3,0.05384,0.05384
DEATH,1085,3.172517,3.172517
REPRODUCE-SHEEP,2493,10.265135,10.265135
REPRODUCE-WOLVES,1039,5.599509,5.599509
```

Thanks to Roger Peppe for his contributions to the code.

Primitives

Looking for the primitive reference for the Profiler extension? You can find the full reference here.

`profiler:calls` · `profiler:exclusive-time` · `profiler:inclusive-time` · `profiler:start` · `profiler:stop` · `profiler:reset` · `profiler:report` · `profiler:data`

All Primitives

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
```

profiler:data

`profiler:data`

Reports a list of lists containing the results of the profiler in a format that is suitable for exporting with the `csv` extension.

The first sublist contains the name of the data columns: `procedure`, `calls`, `inclusive_time` and `exclusive_time`. This is followed by one sublist containing the profiler data for each user-defined procedure. The reported times are in milliseconds.

Python

Using

As with all NetLogo extensions, you must declare that you're using this extension in your NetLogo code with:

```
extensions [
  py
  ; ... your other extensions
]
```

The general workflow of this extension is to run `py:setup py:python` to initialize the Python session that NetLogo will talk to, and then use `py:run`, `py:runresult`, and `py:set` to interact with that Python session. By default, `py:python` will report the latest version of Python that the extension finds on your system. You can also use `py:python3` or `py:python2` to use Python 3 or 2 specifically. See the Configuring section below to specify exactly which Python installations to use.

Here's an example to get you started:

```
observer> py:setup py:python
observer> show py:runresult "1 + 1"
observer: 2
observer> py:run "print('hi')"
hi
observer> py:run "import math"
observer> show py:runresult "[math.factorial(i) for i in range(10)]"
observer: [1 1 2 6 24 120 720 5040 40320 362880]
observer> py:set "patch_xs" [ pxcor ] of patches
observer> show py:runresult "max(patch_xs)"
observer: 16
observer> py:run "print(min(patch_xs))"
-16
```

See the documentation for each of the particular primitives for details on, for instance, how to multi-line statements and how object type conversions work. See the demo models included in the `demo` folder for some examples of using libraries such as `numpy` and `tensorflow`.

See the documentation on `py:set` to learn how to have the extension serialize entire agents and agentsets into Python dictionaries.

There is also a separate interactive Python console that can be found under Python > Interactive Python Console. This console is connected to the same Python session as all the Python NetLogo primitives, so you can define a variable in your model and access it in the interactive Python console window.

Error handling

Python errors will be reported in NetLogo as "Extension exceptions". For instance, this code:

```
py:run "raise Exception('hi')"
```

will result in the NetLogo error "Extension exception: hi". To see the Python stack trace of the exception, click "Show internal details". If you then scroll down, you will find the Python stack trace in the middle of the Java stack trace.

Configuring

By default, the `py:python2`, `py:python3`, and `py:python` commands will attempt to find a Python executable of the appropriate version. If you'd like to change which Python executable they use, or they can't find a Python executable, you should configure which Python executables to use. You can do this by either:

- Using the configuration menu under the Python toolbar menu that appears when you use a model that uses the Python extension.
- Editing the `python.properties` file that appears in the Python extension installation folder as follows:

```
python3=/path/to/python3
python2=/path/to/python2
```

Primitives

Looking for the primitive reference for the Python extension? You can find the full reference here.

`py:setup` · `py:python` · `py:python2` · `py:python3` · `py:run` · `py:runresult` · `py:set`

All Primitives

py:setup

`py:setup` *`python-executable`*

Create the Python session that this extension will use to execute code. The session will be started with the given Python executable. This command *must* be run before running any other Python extension primitive. Running this command again will shutdown the current Python environment and start a new one.

The executable may be specified as a relative path, absolute path, or just the executable name if it is on your PATH. Furthermore, this extension offers a few helper primitives for getting particular versions of Python in system independent ways.

In general, unless working with a virtual environment or a specific system setup, you should do:

```
py:setup py:python  ; if your code works with either Python 2 or 3
py:setup py:python3 ; for Python 3
py:setup py:python2 ; for Python 2
```

`py:setup` may be invoked by directly referring to different Pythons as well. For instance:

```
py:setup "python3" ; if `python3` is on your PATH
py:setup "python"  ; if `python` is on your PATH
```

If you use virtualenv or Conda, simply specify the path of the Python executable in the environment you wish to use:

```
py:setup "/path/to/myenv/bin/python"
```

The path may be relative or absolute. So, if you have a virtual environment in the same folder as your model, you can do:

```
py:setup "myenv/bin/python"
```

py:python

`py:python`

Reports either the path to the latest version of Python configured in the `python.properties` file or, if that is blank, looks for a Python executable on your system's PATH. For Windows, there is an installation option for including Python on your PATH. For MacOS and Linux, it will likely already be on your PATH. The output of this reporter is meant to be used with `py:setup`, but you may also use it to see which Python installation this extension will use by default.

For example, on MacOS with Homebrew installed Python 3:

```
observer> show py:python
observer: "/usr/local/bin/python3"
```

py:python2

`py:python2`

Reports either the path to Python 2 configured in the `python.properties` file or, if that is blank, looks for a Python 2 executable on your system's PATH. For Windows, there is an installation option for including Python on your PATH. For MacOS and Linux, it will likely already be on your PATH. The output of this reporter is meant to be used with `py:setup`, but you may also use it to see which Python 2 installation this extension will use by default.

For example, on MacOS with Homebrew installed Python 2:

```
observer> show py:python2
observer: "/usr/local/bin/python2"
```

py:python3

`py:python3`

Reports either the path to Python 3 configured in the `python.properties` file or, if that is blank, looks for a Python 3 executable on your system's PATH. For Windows, there is an installation option for including Python on your PATH. For MacOS and Linux, it will likely already be on your PATH. The output of this reporter is meant to be used with `py:setup`, but you may also use it to see which Python 3 installation this extension will use by default.

For example, on MacOS with Homebrew installed Python 3:

```
observer> show py:python3
observer: "/usr/local/bin/python3"
```

py:run

`py:run` *`python-statement`*

Runs the given Python statements in the current Python session. To make multi-line Python code easier to run, this command will take multiple strings, each of which will be interpreted as a separate line of Python code. For instance:

```
(py:run
  "import matplotlib"
  "matplotlib.use('TkAgg')"
  "import numpy as np"
  "import matplotlib.pyplot as plt"
```

```
  "for i in range(10):"
  "    plt.plot([ x ** i for x in np.arange(-1, 1, 0.1) ])"
  "plt.show()"
)
```

`py:run` will wait for the statements to finish running before continuing. Thus, if you have long running Python code, NetLogo will pause while it runs.

py:runresult

`py:runresult` *`python-expression`*

Evaluates the given Python expression and reports the result. `py:runresult` attempts to convert from Python data types to NetLogo data types. Numbers, strings, and booleans convert as you would expect. Any list-like object in Python (that is, anything with a length that you can iterate through) will be converted to a NetLogo list. For instance, Python lists and NumPy arrays will convert to NetLogo lists. Python dicts (and dict-like objects) will convert to a NetLogo list of key-value pairs (where each pair is represented as a list). `None` will be converted to `nobody`. Other objects will simply be converted to a string representation.

Note that due a current issue, dict keys will always be reported as strings. If you need to report non-string keys, report the `.items()` of the dict instead of the dict itself.

py:set

`py:set` *`variable-name value`*

Sets a variable in the Python session with the given name to the given NetLogo value. NetLogo objects will be converted to Python objects as expected.

All vanilla NetLogo objects are supported, but objects from other extensions, even other bundled extensions, are not supported.

```
py:set "x" [1 2 3]
show py:runresult "x" ;; Shows [1 2 3]
```

Agents are converted into dictionaries with elements for each agent variable. Agentsets are converted into lists of agent dictionaries.

```
breed [goats goat]
goats-own [energy ]
create-goats 1 [ set heading 0 set color 75 ]
ask goat 0 [ set energy 42 ]
py:set "goat" goat 0
py:runresult "str(goat)" ;; Should output: "{'WHO': 0, 'COLOR': 75, 'HEADING': 0, 'XCOR': 0,
'YCOR': 0, 'SHAPE': 'default', 'LABEL': '', 'LABEL-COLOR': 9.9, 'BREED': 'GOATS', 'HIDDEN?':
False, 'SIZE': 1, 'PEN-SIZE': 1, 'PEN-MODE': 'up', 'ENERGY': 42}"
```

Agents with variables containing references to agentsets will have those variables converted into the string representation of that agentset.

Time

The Time extension provides NetLogo with three kinds of capabilities for models that use discrete-event simulation or represent time explicitly. The package provides tools for common date and time operations, discrete event scheduling, and using time-series input data.

- Quickstart
- What is it?
- Installation
- Examples
- Behavior
- Format
- Primitives
  - Date/Time Utilities
  - Time Series Tool
  - Discrete Event Scheduler
- Building
- Authors
- Feedback
- Credits
- Terms of use

Quickstart

Include the extension in your NetLogo model (at the top):

```
extensions [time]
```

Date/Time Utilities

Create a global date/time and initialize in the setup procedure:

```
globals[dt]
to setup
  set dt time:create "2000/01/01 10:00"
end
```

From the console, execute setup and then print a formatted version of your date/time to the console:

```
setup
print time:show dt "EEEE, MMMM d, yyyy"
;; prints "Sunday, January 2, 2000"
```

Print the hour of the day, the day of the week, and the day of the year:

```
print time:get "hour" dt    ;; prints 10
print time:get "dayofweek" dt   ;; prints 6
```

```
print time:get "dayofyear" dt  ;; prints 1
```

Add 3 days to your date/time and print the date/time object to the screen:

```
set dt time:plus dt 3 "days"
print dt
```

Compare your date/time to some other date/time:

```
ifelse time:is-after? dt (time:create "2000-01-01 12:00") [print "yes"][print "no"]
```

Time Series Tool

NOTE: The time series tool is not currently included in the extension. However, all of the functionality (plus some) has been reproduced in NetLogo code in the **time-series.nls file in this repo** which you can download by clicking that link and then right clicking the "raw" button on the upper right of the file and selecting "download linked file". You might have to remove a '.txt' file extension so that the file ends with the '.nls' extension. You can then include that file with the **__includes** primitive.

Download this example time series file and place in the same directory as your NetLogo model. Here are the first 10 lines of the file:

```
; meta data at the top of the file
; is skipped when preceded by
; a semi-colon
timestamp,flow,temperature
2000-01-01 00:00:00,1000,10
2000-01-01 01:00:00,1010,11
2000-01-01 03:00:00,1030,13
2000-01-01 04:00:00,1040,14
2000-01-01 05:00:00,1050,15
…
…
```

Create a global to store a LogoTimeSeries object. In your setup procedure, load the data from the CSV file:

```
globals[time-series]

set time-series ts-load "time-series-data.csv"
```

Create a LogoTime and use it to extract the value from the "flow" column that is nearest in time to that object:

```
let current-time time:create "2000-01-01 01:20:00"

let current-flow ts-get time-series current-time "flow"

;; By default, the nearest record in the time series is retrieved (in this case 1010),
;; you can alternatively require an exact match or do linear interpolation.
```

Discrete Event Scheduler

Create a few turtles and schedule them to go forward at tick 10, then schedule one of them to also go forward at tick 5.

```
create-turtles 5

time:schedule-event turtles ([ [] -> fd 1]) 10
time:schedule-event (turtle 1) ([ [] -> fd 1]) 5
```

Execute the discrete event schedule (all events will be executed in order of time):

```
time:go

;; turtle 1 will go foward at tick 5,
;; then all 5 turtles will go forward at tick 10
```

back to top

What is it?

This package contains the NetLogo **time extension**, which provides NetLogo with three kinds of capabilities for models that use discrete-event simulation or represent time explicitly. The package provides tools for common date and time operations, discrete event scheduling, and using time-series input data.

Dates and Times

The extension provides tools for representing time explicitly, especially by linking NetLogo's ticks to a specific time interval. It allows users to do things such as starting a simulation on 1 January of 2010 and end on 31 December 2015, have each tick represent 6 hours, and check whether the current simulation date is between 1 and 15 March.

This extension is powered by the Java Time Library, which has very sophisticated and comprehensive date/time facilities. A subset of these capabilities have been extended to NetLogo. The **time extension** makes it easy to convert string representations of dates and date/times to a **LogoTime** object which can then be used to do many common time manipulations such as incrementing the time by some amount (e.g. add 3.5 days to 2001-02-22 10:00 to get 2001-02-25 22:00).

Time Series Utilities

Modelers commonly need to use time series data in NetLogo. The **time extension** no longer provides time series functionality (plus some), but the same functionality is included the time-series.nls file in this repo which you can download by clicking that link and then right clicking the "raw" button on the upper right of the file and selecting "download linked file" (you might have to remove a '.txt' file extension so that the file ends with the '.nls' extension). You can then include that file with the **__includes** primitive. The time-series.nls file includes convenient procedures for handling time series data. With a single command, you can load an entire time series data set from a text file. The first column in that text file holds dates or datetimes. The remaining columns can be numeric or string values. You then access the data by time and by column heading, akin to saying "get the flow from May 8, 2008".

Users can also create and record a time series of events within their model, access that series during simulations, and export it to a file for analysis. For example, a market model could create a time series object into which is recorded the date and time, trader, price, and size of each trade. The time series utilities let model code get (for example) the mean price over the previous day or week, and save all the trades to a file at the end of a run.

Discrete Event Scheduling

Note: Formerly this capability was published as the **Dynamic Scheduler Extension**, but that extension has been merged into the **time extension** in order to integrate the functionality of both.

The **time extension** enables a different approach to scheduling actions in NetLogo. Traditionally, a NetLogo modeler puts a series of actions or procedure calls into the "go" procedure, which is executed once each tick. Sometimes it is more natural or more efficient to instead say "have agent X execute procedure Y at time Z". This is what discrete event scheduling (also know as "dynamic scheduling"") enables. Discrete event simulation has a long history and extensive literature, and this extension makes it much easier to use in NetLogo.

When is discrete event scheduling useful? Discrete event scheduling is most useful for models where agents spend a lot of time sitting idle despite knowing when they need to act next. Sometimes in a NetLogo model, you end up testing a certain condition or set of conditions for every agent on every tick (usually in the form of an "ask"), just waiting for the time to be ripe... this can get cumbersome and expensive. In some models, you might know in advance exactly when a particular agent needs to act. Dynamic scheduling cuts out all of those superfluous tests. The action is performed only when needed, with no condition testing and very little overhead.

For example, if an agent is a state machine and spends most of the time in the state “at rest” and has a predictable schedule that knows that the agent should transition to the state “awake” at tick 105, then using a dynamic scheduler allows you to avoid code that looks like: “if ticks = 105 [do-something]”, which has to be evaluated every tick!

A second common use of discrete event scheduling is when it is important to keep track of exactly when events occur in continuous time, so the simplifying assumption that all events happen only at regular ticks is not appropriate. One classic example is queuing models (e.g., how long customers have to stand in line for a bank teller), which use a continuous random number distribution (e.g., an exponential distribution) to determine when the next agent enters the queue.

back to top

Installation

The Time extension comes bundled with NetLogo. Simply include `extensions [time]` at the top of the model and it will be loaded. New versions of the extension are available through the Netlogo extensions manager as they are released.

Alternatively, download the latest version of the extension (Note that this extension is compiled for NetLogo 6.1 and may not work for earlier versions of NetLogo) Unzip the archive and rename the directory to “time”. Move the renamed directory to the “extensions” directory inside your NetLogo application folder (i.e. NETLOGO/extensions/). Or you can place the time directory under the same directory holding the NetLogo model in which you want to use this extension.

For more information on NetLogo extensions: http://ccl.northwestern.edu/netlogo/docs/extensions.html

back to top

Examples

See the example models in the time extension’s subfolder “examples” for thorough demonstrations of usage.

Data Types

The **time extension** introduces some new data types (more detail about these is provided in the behavior section):

- **LogoTime** - A LogoTime object stores a time stamp; it can track a full date and time, or just a date (with no associated time).
- **LogoEvent** - A LogoEvent encapsulates a who, a what, and a when. It allows you to define, for example, that you want turtle 7 to execute the go-forward procedure at tick 10. When scheduling an event using the **time extension** you pass the who, what, and when as arguments (e.g. “time:schedule-event (turtle 1) fd 5”).
- **Discrete Event Schedule** - A discrete event schedule is a sorted list of LogoEvents that is maintained by this extension and manages the dispatch (execution) of those events. Users do not need to manipulate or manage this schedule directly, but it is useful to understand that it stores and executes LogoEvents when the “time:go” or “time:go-until” commands are issued. As the schedule is executed, the **time extension** automatically updates the NetLogo ticks to match the current event in the schedule.

The time-series.nls file in this repo provides one more datatype.

- **LogoTimeSeries** - A LogoTimeSeries object stores a table of data indexed by LogoTime. The time series can be read in from a file or recorded by the code during a simulation. It is currently implemented in NetLogo code using a list of lists.

back to top

Behavior

The **time extension** has the following notable behavior:

- **LogoTimes can store DATETIMEs, DATEs, or DAYs** - A LogoTime is a flexible data structure that will represent your time data as one of three varieties depending on how you create the LogoTime object. A LogoTime can be a DATETIME, a DATE, or a DAY:
  - A DATETIME is a fully specified instant in time, with precision down to a millisecond (e.g. January 2, 2000 at 3:04am and 5.678 seconds).
  - A DATE is a fully specified day in time but lacks any information about the time of day (e.g. January 2, 2000).
  - A DAY is a generic date that does not specify a year (e.g. January 2).

The behavior of the **time extension** primitives depend on which variety of LogoTime you are storing. For example, the difference between two DATETIMES will have millisecond resolution, while the difference between two DATES or two DAYS will only have resolution to the nearest whole day.

As another example, a DAY representing 01/01 is always considered to be before 12/31. Because there's no wrapping around for DAYs, they are only useful if your entire model occurs within one year and doesn't pass from December to January. If you need to wrap, use a DATE and pick a year for your model, even if there's no basis in reality for that year. (DAY LogoTime variables are useful, however, as parameters defining the date when something happens every simulated year. You can, for example, use time extension commands to say that a species' breeding season starts on one DAY—15 May—and ends on another—25 June—each year.)

- **You create LogoTime objects by passing a string** - The time:create primitive was designed to both follow the standard used by the Java time library, and to make date time parsing more convenient by allowing a wider range of delimiters and formats. For example, the following are all valid DATETIME strings:
  - "2000-01-02T03:04:05.678"
  - "2000-01-02T3:04:05.678"
  - "2000-01-02 03:04:05"
  - "2000-01-02 3:04:05"
  - "2000-01-02 03:04"
  - "2000-01-02 3:04"
  - "2000-01-02 03"
  - "2000-01-02 3"
  - "2000/01/02 03:04:05.678"
  - "2000-1-02 03:04:05.678"
  - "2000-01-2 03:04:05.678"
  - "2000-1-2 03:04:05.678"

The following are all valid DATE strings:

- "2000-01-02"
- "2000-01-2"
- "2000-1-02"
- "2000/1/02"

The following are all valid DAY strings:

- "01-02"
- "01-2"
- "1-02"
- "1/2"

Note that if you do not include a time in your string, the **time extension** will assume you want a DATE. If you want a DATETIME that happens to be at midnight, specify the time as zeros: "2000-01-02 00:00".

- **Time extension recognizes "period types"** - In order to make it easy to specify a time period like "2 days" or "4 weeks", the **time extension** will accept strings to specify a period type. The following is a table of the period types and strings that **time** recognizes (note: any of these period type strings can be pluralized and are case **in**sensitive):

PERIOD TYPE	Valid string specifiers
YEAR	"year"
MONTH	"month"
WEEK	"week"
DAY	"day", "dayofmonth", "dom"
DAYOFYEAR	"dayofyear", "doy", "julianday", "jday"
DAYOFWEEK	"dayofweek", "dow", "weekday", "wday"
HOUR	"hour"
MINUTE	"minute"
SECOND	"second"
MILLI	"milli"

- **Time extension has millisecond resolution** - This is a fundamental feature of the Java Time library and cannot be changed. Milliseconds are represented as fractions of a second, truncated to the third decimal place to prevent parsing issues with the Java Time library. This truncation can create artifacts; for example if time:plus is used to add a value that is less than one whole millisecond, it does nothing.
- **Daylight savings time is ignored** - All times are treated as local, or "zoneless", and daylight savings time (DST) is ignored. It is assumed that most NetLogo users don't need to convert times between time zones or be able to follow the rules of DST for any particular locale. Instead, users are much more likely to need the ability to load a time series and perform date and time operations without worrying about when DST starts and whether an hour of their time series will get skipped in the spring or repeated in the fall. It should be noted that the Time library definitely can handle DST for most locales on Earth, but that capability is not extended to NetLogo here and won't be unless by popular demand.
- **Leap days are included** - While we simplify things by excluding time zones and DST, leap days are kept to allow users to reliably use real-world time series in their NetLogo model. LogoTime variable type DAY assumes a leap year, so DAY variables can include 2/29. (If you add one day to a DAY with value of 28 February, the result is 29 February, not 1 March.)
- **LogoTimes are mutable when anchored** - If you anchor a LogoTime (using the *time:anchor-to-ticks* primitive) you end up with a variable whose value changes as the value of Netlogo ticks changes. If you have an anchored variable called "anchored-time" and you assign it to another variable "set new-time anchored-time", your new variable will *also be mutable* and change with ticks. If what you want is a snapshot of the anchored-time that doesn't change, then use the time:copy primitive: "set new-time time:copy anchored-time".
- **Decimal versus whole number time periods** - In this extension, decimal values can be used by the *plus* and *anchor-to-ticks* primitives for seconds, minutes, hours, days, and weeks (milliseconds can't be fractional because they are the base unit of time). These units are treated as *durations* because they can unambiguously be converted from a decimal number to a whole number of milliseconds. But there is ambiguity in how many milliseconds there are in 1 month or 1 year, so month and year increments are treated as *periods* which are by definition whole number valued. So if you use the *time:plus* primitive to add 1 month to the date "2012-02-02", you will get "2012-03-02"; and if you add another month you get "2012-04-02" even though February and March have different numbers of days. If you try to use a fractional number of months or years, it will be rounded to the nearest integer and then added. If you want to increment a time variable by one and a half 365-day years, then just increment by 1.5 * 365 days instead of 1.5 years.
- **LogoEvents are dispatched in order, and ties go to the first created** - If multiple LogoEvents are scheduled for the exact same time, they are dispatched (executed) in the order in which they were added to the discrete event schedule.

- **LogoEvents can be created for an agentset** - When an agentset is scheduled to perform an anonymous command (before NetLogo 6.0 these were called tasks) at the same time, the individual agents execute the procedure in a non-random order, which is different from *ask* which shuffles the agents. Not shuffling the agents may save some execution time. To shuffle the order, use the *add-shuffled* primitive, which will execute the actions in random order with low overhead.
- **LogoEvents won't break if an agent dies** - If an agent is scheduled to perform an anonymous command in the future but dies before the event is dispatched, the event will be silently skipped.
- **LogoEvents can be scheduled to occur at a LogoTime** - LogoTimes are acceptable alternatives to specifying tick numbers for when events should occur. However, for this to work the discrete event schedule must be "anchored" to a reference time so it knows a relationship between ticks and time. See *time:anchor-schedule** below for an example of anchoring.

back to top

Format

The time extension utilizes rules and standards, provided by the Java 8 time library, to parse and accept valid dates. The extension provides default formats for quickly creating time objects, but includes an option for specifying a custom format. In addition, the extension adheres to the ISO 8601 standard, following the Java 8 STRICT format and 24 hour clock format. The same formats are used for both input and output.

Date Format

Date formats are encoded strings with unique characters that represent various units of time and their position (date and time). Each character is meant to be placed in a contiguous group to indicate its expected location and type. If a user provides a date string that does not follow the parsing format or bounds, then an exception will be thrown. With create primitives, the selected default format can create a date-time, date, and day object depending on the string provided. The set of default formats are below. The default delimiter between months, days and years is '/', between hours, minutes and seconds is ":" and between seconds and milliseconds is ".". Using any other delimiter requires using a custom formatter (see User Defined Formatting).

- **DateTime Default Formatter**

```
MM/dd/yyyy HH:mm:ss.SSS
```

For date-time formatters, the format specifies all 7 units (month,day,year,etc) will be available for parsing and generating a date-time object.

- **Date Default Formatter**

```
MM/dd/yyyy
```

For date formatters, the month, day, and year need to be specified to obtain a date object.

- **Day Default Formatter**

```
MM/dd
```

For day formatters, the month and day need to be specified to obtain a day object. Day objects are based on the year 2000, which contains a leap day, which could lead to miscalculations if only 28 days are expected in February.

Supported Format Characters

For supported format characters ('H','m',etc), there are three main modes for parsing: shorthand, sized, and full. Each mode controls the number of acceptable charcters for parsing digits.

- **Shorthand** For all units, except the millisecond and year fields, "shorthand" formats use a single unique character meant to accept 1 or 2 digits for the corresponding unit. The short-hand case is a lenient option for values that can fluctuate values between 1 or 2 digits. An example:

```
M/d
```

- **Sized** For year and millisecond, the number of characters for a specific unit of time determines the number of acceptable numbers. For example, a format of "MM/dd/yyyy HH:mm:ss.S" assumes that that the input/output has tenths of seconds while "MM/dd/yyyy HH:mm:ss.SSS" assumes that the input/output has miliseconds.

```
MM/dd/yyyy HH:mm:ss.SSS
MM/dd/yyyy HH:mm:ss.S
```

- **Full** For all units of time, the maximum number of representable formatting characters can be used to provide a verbose and accurrate representation of the time objects. If the string does not follow the format, an exception will be thrown. Example:

```
MM/dd/yyyy HH:mm:ss.SSS
```

Month	Day	Year	Hour	Minute	Second	Millisecond
`MM` or `M`	`dd` or `d`	`yyyy` or `yy`	`HH` or `H`	`mm` or `m`	`ss` or `s`	`SSS` or `SS` or `S`

NOTE: generally 'yyyy' (or 'uuuu') should be used for years and not 'YYYY' which is a "week based year." This means that a date at the very end of one year, e.g. 12/31/2000 might be given the year 2001 if 'YYYY' is used, because it is counted as part of the first week of 2001.

Date-time Bounds

Since not all dates are representable within the time extension, a strict formatting is enforced to minimize the effects of invalid input. Dates are bounded by the Gregorian calendar while the time is bounded with their respective unit of time.

Month	Day	Year	Hour	Minute	Second	Millisecond
1 - 12	1 - 31 or 30 or 29 or 28	1000-9999	0-23	0-59	0-59	0 - 999

User Defined Formatting

The time extension supports user-defined formatters for brevity and control. User defined formatters allow for reordering format characters for units of time and using alternative delimiters from the defaults.

```
M:yyyy:d
MM/d/yyyy HH:mm
```

back to top

Primitives

Date/Time Utilities

time:create

time:create time-string

Reports a LogoTime created by parsing the *time-string* argument. A LogoTime is a custom data type included with this extension, used to store time in the form of a DATETIME, a DATE, or a DAY. All other primitives associated with this extension take one or more LogoTimes as as an argument. See the "Behavior" section above for more information on the behavior of LogoTime objects. The `time:create` primitive raises an error if time-string does not represent a real date or time, e.g., if the day exceeds the number of days in the month or the month is not between 1 and 12. Hours must have values 0-23, and minutes and seconds must be 0-59. Seconds with milliseconds must be between 0.000 and 59.999.

```
;; Create a datetime, a date, and a day
let t-datetime time:create "2000-01-02 03:04:05.678"
let t-date time:create "2000/01/02"
let t-day time:create "01-02"
```

---

time:create-with-format

time:create-with-format time-string format-string

Like `time:create`, but parses the *time-string* argument using the *format-string* argument as the format specifier.

```
;; Create a datetime, a date, and a day using American convention for dates: Month/Day/Year
let t-datetime time:create-with-format "01-02-2000 03:04:05.678" "MM-dd-yyyy HH:mm:ss.SSS"
let t-date time:create-with-format "01/02/2000" "MM/dd/yyyy"
let t-day time:create-with-format "01-02" "MM-dd"
```

See the following link for a full description of the available format options:

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

---

time:show

time:show logotime string-format

Reports a string containing the *logotime* formatted according the *string-format* argument.

```
let t-datetime time:create "2000-01-02 03:04:05.678"

print time:show t-datetime "EEEE, MMMM d, yyyy"
;; prints "Sunday, January 2, 2000"

print time:show t-datetime "yyyy-MM-dd HH:mm"
;; prints "2000-01-02 03:04"
```

See the following link for a full description of the available format options:

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

Note In the case for DATE and DAY variables, time:get returns default values for unsupported values. Day has a default year of 2000 to calculate for leap years.

```
let t-date (time:create "2000-01-01")
let t-day (time:create "01-01")

print time:show t-date "HH"
print time:show t-day "HH"
;; print 00

print time:show t-date "mm"
print time:show t-day "mm"
;; print 00

print time:show t-date "ss"
print time:show t-day "ss"
;; print 00

print time:show t-date "SSS"
print time:show t-day "SSS"
;; print 000

print time:show t-day "yyyy"
;; print 2000
```

---

time:get

time:get period-type-string logotime

Retrieves the numeric value from the *logotime* argument corresponding to the *period-type-string* argument. For DATETIME variables, all period types are valid; for DATEs, only period types of a day or higher are valid; for DAYs, the only valid period types are “day” and “month”. For accessing seconds, seconds are not rounded to the nearest integer, but truncated. When time:get is used with a sub-day period type (“hour”, “minute”) and a logotime that is a DATE or DAY instead of a DATETIME, it returns zero. Getting the year of a DAY variable returns the default of 2000.

```
let t-datetime (time:create "2000-02-02 03:04:05.678")

print time:get "year" t-datetime
;;prints "2000"

print time:get "month" t-datetime
;;prints "1"

print time:get "day" t-datetime
;;prints "2"

print time:get "dayofyear" t-datetime
;;prints "33"

print time:get "hour" t-datetime
;;prints "3"

print time:get "second" t-datetime
;;prints "5"

print time:get "milli" t-datetime
;;prints "678"
```

Note In the case for Day and Date time formats, time:get returns default values for unsupported values. Day has a default year of 2000 to calculate for leap years.

```
let t-date (time:create "2000-01-01")
let t-day (time:create "01-01")

print time:get "hour" t-date
```

```
print time:get "hour" t-day
;; print 0

 print time:get "minute" t-date
 print time:get "minute" t-day
;; print 0

print time:get "second" t-date
print time:get "second" t-day
;; print 0

print time:get "milli" t-date
print time:get "milli" t-day
;; print 0

print time:get "year" t-day
;; print 2000
```

---

time:plus

time:plus logotime number period-type-string

Reports a LogoTime resulting from the addition of some time period to the *logotime* argument. The time period to be added is specified by the *number* and *period-type-string* arguments. Valid period types are YEAR, MONTH, WEEK, DAY, DAYOFYEAR, HOUR, MINUTE, SECOND, and MILLI. To subtract time, use a negative value of number. When applying additions to LogoTimes, addition is applied through converting the format to datetime, adding the values, and reconverting LogoTime to its respective format. This can lead to incorrectly applying time updates and lost information.

```
let t-datetime (time:create "2000-01-02 03:04:05.678")

;; Add some period to the datetime
print time:plus t-datetime 10 "milli"
;; prints "{{time:logotime 2000-01-02 03:04:06.688}}"

print time:plus t-datetime 1.0 "seconds"
;; prints "{{time:logotime 2000-01-02 03:04:06.678}}"

print time:plus t-datetime 1.0 "minutes"
;; prints "{{time:logotime 2000-01-02 03:05:05.678}}"

print time:plus t-datetime (60.0 * 24) "minutes"
;; prints "{{time:logotime 2000-01-03 03:04:05.678}}"

print time:plus t-datetime 1 "week"
;; prints "{{time:logotime 2000-01-09 03:04:05.678}}"

print time:plus t-datetime 1.0 "weeks"
;; prints "{{time:logotime 2000-01-09 03:04:05.678}}"

print time:plus t-datetime 1.0 "months"
;; note that decimal months or years are rounded to the nearest whole number
;; prints "{{time:logotime 2000-02-02 03:04:05.678}}"

print time:plus t-datetime 1.0 "years"
;; prints "{{time:logotime 2001-01-02 03:04:05.678}}"
```

Addition with Decimals

- Week, Day, DayOfYear, Hour, Minute, and Second all support decimal addition so you can add half a day, a quarter of a second, or another subvalue. You cannot add fractions of milliseconds because LogoTime variables do not have sub-millisecond resolution. You can add fractions of days (or hours, minutes, etc.) to LogoTime variables of type DATE and DAY, but the result will be truncated to an integer day. This causes some information to be lost. For example:

```
print time:plus (time:create "2000-01-01 00:00:00.000") 2.3 "day" => "2000-01-03 07:11:59.999"
print time:plus (time:create "2000-01-01") 2.3 "day" => "2000-01-03"
```

```
print time:plus (time:create "2000-01-04 00:00:00.000") -2.3 "day" => "2000-01-01 2000-01-01
16:48:00.001"
print time:plus (time:create "2000-01-04") -2.3 "day" => "2000-01-01"
```

- Year and Month both round the provided number to the nearest integer. There is no decimial addition support for year and month.

Addition with Negative Decimal Numbers

- Year, Month, Week, Day, DayOfYear, Hour, Minute, Second, and Millisecond all support subtraction through the use of negative numbers. Subtracting with whole numbers works as expected, but when subtracting with decimals the results may produce unexpected behavior. When subtracting with DateTime, DateTime objects maintain accurate times without truncation or rounding. With DATE and DAY variables, when decimals are subtracted two conversions are applied. First the DATE/DAY is converted to a DATETIME starting at midnight (00:00:00), tthen the subraction iis made, and the resulting DATETIME is converted back the original DATE or DAY type by setting the hours, minutes, etc. to zero. One example is with subtracting an hour from a DATE. Subtracting one hour from a DATE variable equal to 01/02/2000 would generate a date of 01/01/2000 because of the default (00:00:00) conversion and the truncation after applying the addition.

```
"2000-01-02" (convert) => "2000-01-02 00:00:00.000" (subtract hour) => "2000-01-01 23:00:00.000"
(application) => "2000-01-01" (restore type)
```

How Rounding, Truncating and Converting works

- Week, Day, DayOfYear, Hour, Minute, Second, and Millisecond: convert user provided time/unit to milliseconds as a long integer ⇒ apply addition ⇒ convert to initial format (DateTime, Date, or Day) ⇒ Remove unsupported units (Date, Day)
- Year and Month: round user provided time/value to the nearest integer ⇒ apply addition ⇒ convert to initial foramt ⇒ Remove Unsupported units (Day)

Carryover

- Week, Day, DayOfYear, Hour, Minute, Second, and Millisecond: All time types allow carrying over values from their subunits. This means that if 24 hours (or 100,000 seconds) are added to a DATE or DAY, result is the next day. If 23 hours are added to a DATE or DAY, its value will not change thanks to truncation. DATETIME variables can have carryover with decimal addition since a DATETIME can represent decimal values.
- Year and Month numbers are rounded to the nearest integer before being added.

---

time:is-before?
time:is-after?
time:is-equal?
time:is-between?

time:is-before? logotime1 logotime2
time:is-after? logotime1 logotime2
time:is-equal? logotime1 logotime2
time:is-between? logotime1 logotime2 logotime3

Note: Prior versions of the time extension included these primitives without the "?" at the end. If you have used an older version of the time extension, you will need to update your code accordingly.

Reports a boolean for the test of whether *logotime1* is before/after/equal-to *logotime2*. The is-between? primitive returns true if *logotime1* is between *logotime2* and *logotime3*. All LogoTime arguments must be of the same variety (DATETIME, DATE, or DAY).

```
print time:is-before? (time:create "2000-01-02") (time:create "2000-01-03")
;;prints "true"

  print time:is-before? (time:create "2000-01-03") (time:create "2000-01-02")
```

```
;;prints "false"

  print time:is-after?  (time:create "2000-01-03") (time:create "2000-01-02")
;;prints "true"

  print time:is-equal?  (time:create "2000-01-02") (time:create "2000-01-02")
;;prints "true"

  print time:is-equal?  (time:create "2000-01-02") (time:create "2000-01-03")
;;prints "false"

 print time:is-between? (time:create "2000-03-08")  (time:create "1999-12-02") (time:create "2000-05-
03")
;;prints "true"
```

---

time:difference-between

time:difference-between logotime1 logotime2 period-type-string

Reports the amount of time between *logotime1* and *logotime2* in units of *period-type-string*. Note that if the period type is YEAR or MONTH, then the reported value will be a whole number based soley on the month and year components of the LogoTimes. If *logotime2* is smaller (earlier than) *logotime1*, the reported value will be negative.

This primitive is useful for recording the elapsed time between model events because (unlike time:get) it reports the total number of time units, including fractions of units. For example, if *start-time* is a LogoTime variable for the time a simulation starts and *end-time* is when the simulation stops, then use *show time:difference-between start-time end-time "days"* to see how many days were simulated.

** Note **

For period type of "MONTH", difference-between reports the difference in the month number (1-12) of logotime1 and logotime2, not the total number of months between the two dates, even if the two times are DATETIME or DATE values. This difference is calculated as the number of times the day number (1-31) of logotime1 occurs between the two dates. For example, if *logotime1* is 2000-01-02 then difference-between reports 0 months for *logotime2* = 2000-02-01, 1 month for 2000-02-02, and -1 month for logotime2 = 1999-12-01. However, the value reported is never greater than 11 or less than -11; it reverts to zero when logotime2 reaches one year from logotime1 and then is 1 when logotime2 is 13 months after logotime1, etc.

For "YEAR", difference-between reports the number integer of years there are between two logotime instances. The difference is calculated through matching the day and month while truncating values around that date. This should apply for leap years, as well. For example, if logotime1 is 2000-01-02 then `time:difference-between` reports 0 years for *logotime2* = 2001-01-01, 1 year for 2000-01-02, and 5 years for 2005-01-02. The total number of months between two dates can therefore be calculated as 12 times the value of `time:difference-between` in years plus the value of `time:difference-between` in months.

```
print time:difference-between (time:create "2000-01-02 00:00") (time:create "2000-02-02 00:00") "days"
;;prints "31"

  print time:difference-between (time:create "2000-01-02") (time:create "2001-02-02") "days"
;;prints "397"

  print time:difference-between (time:create "01-02") (time:create "01-01") "hours"
;;prints "-24"

print time:difference-between (time:create "2000-01-02") (time:create "2000-02-15") "months"
;;prints "1"
```

---

time:anchor-to-ticks

time:anchor-to-ticks logotime number period-type

Reports a new LogoTime object which is "anchored" to the native time tracking mechanism in NetLogo (i.e the value of *ticks*). Once anchored, this LogoTime object will always hold the value of the current time as tracked by *ticks*. Any of the

three varieties of LogoTime can be achored to the tick. The time value of the *logotime* argument is assumed to be the time at tick zero. The *number* and *period-type* arguments describe the time represented by one tick (e.g. a tick can be worth 1 day or 2 hours or 90 seconds, etc.)

Note: *time:anchor-to-ticks* is a one-way coupling. Changes to the value of *ticks* (e.g. when using the *tick* or *tick-advance* commands) will automatically update the anchored LogoTime, but changing the value of the LogoTime variable will not update the value of *ticks*; in fact it is a bad idea for anything other than ticks to change an anchored variable. However, see the discrete event scheduling capability and the *time:anchor-schedule* command to influence the value of *ticks* through the use of LogoTimes.

```
set tick-datetime time:anchor-to-ticks (time:create "2000-01-02 03:04:05.678") 1 "hour"
set tick-date time:anchor-to-ticks (time:create "2000-01-02") 2 "days"
set tick-day time:anchor-to-ticks (time:create "01-02") 3 "months"

reset-ticks
tick
print (word "tick " ticks)  ;; prints "tick 1"
print (word "tick-datetime " tick-datetime)  ;; prints "tick-dateime {{time:logotime 2000-01-02
04:04:05.678}}"
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-04}}"
print (word "tick-day " tick-day)  ;; prints "tick-day {{time:logotime 04-02}}"


tick
print (word "tick " ticks)  ;; prints "tick 2"
print (word "tick-datetime " tick-datetime)  ;; prints "tick-dateime {{time:logotime 2000-01-02
05:04:05.678}}"
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-06}}""
print (word "tick-day " tick-day)  ;; prints "tick-day {{time:logotime 07-02}}""
```

back to top

---

time:copy

time:copy logotime

Returns a new LogoTime object that holds the same date/time as the *logotime* argument. The copy will not be anchored regardless of the argument, making this the recommended way to store a snapshot of an anchored LogoTime.

```
set tick-date time:anchor-to-ticks (time:create "2000-01-02") 2 "days"
reset-ticks
tick
print (word "tick " ticks)  ;; prints "tick 1"
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-04}}"

set store-date time:copy tick-date

tick
print (word "tick " ticks)  ;; prints "tick 1"
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-06}}"
print (word "store-date " store-date)  ;; prints "store-date {{time:logotime 2000-01-04}}"
```

Time Series Tool

NOTE: The time series tool is not currently included in the extension. However, all of the functionality (plus some) has been reproduced in NetLogo code. It is in the time-series.nls file in this repo which you can download by clicking that link and then right clicking the “raw” button on the upper right of the file and selecting “download linked file” (you might have to remove a ‘.txt’ file extension so that the file ends with the ‘.nls’ extension). You can then include that file

with the __includes primitive. Since it is not part of the extension, these “primitives” don’t have the “time:” prefix. Also, the NetLogo file must have the csv extension for the time-series.nls functionality to work.

ts-create

ts-create column-name-list

Reports a new, empty LogoTimeSeries. The number of data columns and their names are defined by the number and values of *column-name-list* parameter, which must be a list of strings. The first column, which contains dates or times, is created automatically. The words “ALL” and “LOGOTIME” (in any combination of lower- or upper-case) cannot be used as column names because they have special meanings in the ts-get and ts-get-range primitives.

```
let turtle-move-times (ts-create ["turtle-show" "new-xcor" "new-ycor"])
```

---

ts-add-row*NOTE*: This is the only time-series functionality in time-series.nls that is not backwards compatible with the old time extension. Since time-series.nls implements time series with native NetLogo lists, they are immutable. So, ts-add-row returns a new timeseries, it does not mutate the old one. This means that variables holding the time series must be re-assigned.

ts-add-row logotimeseries row-list

Returns a new LogoTimeSeries with row-list added to *logotimeseries*. This primitive is how data are added to a LogoTimeSeries. The *row-list* should be a list containing a LogoTime as the first element and the rest of the data corresponding to the number of columns in the LogoTimeSeries object. Columns are either numeric or string valued.

```
;; A turtle records the time and destination each time it moves
;; model-time is a DATETIME variable anchored to ticks.
set turtle-move-times ts-add-row turtle-move-times (sentence model-time who xcor ycor)
```

---

ts-get

ts-get logotimeseries logotime column-name

Reports the value from the *column-name* column of the *logotimeseries* in the row matching *logotime*. If there is not an exact match with *logotime*, the row with the nearest date/time will be used. If there are multiple rows with the same logotime, only one will be returned. For LogoTimeSeries containing multiple rows with the same logotime value, using ts-get-range to return a list of all times within a range is recommended instead of *ts-get*. If “ALL” or “all” is specified as the column name, then the entire row, including the logotime, is returned as a list.

```
print ts-get ts (time:create "2000-01-01 10:00:00") "flow"
;; prints the value from the flow column in the row containing a time stamp of 2000-01-01 10:00:00
```

---

ts-get-interp

ts-get-interp logotimeseries logotime column-name

Behaves like ts-get, except that if there is not an exact match with the *logotime*, then the value returned is a linear interpolation between the two values with date/times nearest (just before and just after) *logotime*. This command will throw an exception if the values in the column are strings instead of numeric.

```
print ts-get-interp ts (time:create "2000-01-01 10:30:00") "flow"
```

---

ts-get-exact

ts-get-exact logotimeseries logotime column-name

Behaves like ts-get, except that if there is not an exact match with the date/time stamp, then an exception is thrown. If there are multiple rows with the same logotime, only one will be returned. In such a case, usinig `ts-get-range` to return a list of all times within a range is recommended instead.

```
print ts-get-exact ts (time:create "2000-01-01 10:30:00") "flow"
```

---

ts-get-range

ts-get-range logotimeseries logotime1 logotime2 column-name

Reports a list of all of the values from the *column-name* column of the *logotimeseries* in the rows between *logotime1* and *logotime2* (inclusively). If "ALL" or "all" is specified as the column name, then a list of lists is reported, with one sub-list for each column in *logotimeseries*, including the date/time column. If "LOGOTIME" or "logotime" is specified as the column name, then the date/time column is returned.

If in the event *logotime1* is after *logotime2*, `ts-get-range` will reverse their values, so it does not matter which of these inputs is smaller.

```
print ts-get-range time-series time:create "2000-01-02 12:30:00" time:create "2000-01-03 00:30:00"
"all"
```

---

ts-has-repeat-times?*ts-has-repeat-times? logotimeseries*

Reports whether there are any repeated times in the logotimeseries (i.e. two rows with the same time). This can be helpful for determining if it is appropriate to use ts-get or ts-get-range.

---

ts-has-repeat-of-time?*ts-has-repeat-of-time? logotimeries logotime*

Reports whether *logotime* appears more than once in *logotimeseries*.

---

ts-load

ts-load filepath

Loads time series data from a text input file (comma or tab separated) and reports a new LogoTimeSeries object that contains the data.

```
let ts ts-load "time-series-data.csv"
```

Each input file and LogoTimeSeries object can contain one or more variables, which are accessed by the column names provided on the first line of the file. The first line of the file must therefore start with the date/time column name, followed by the names of the variables (other columns) in the file. Do not use "all" or "ALL," or "logotime" or "LOGOTIME" for a column name as thse are reserved keywords (see `ts-get` and `ts-get-range`).

The first column of the file must be timestamps that can be parsed by this extension (see the **behavior section** for acceptable string formats). Finally, if the timestamps do not appear in chronological order in the text file, they will be automatically sorted into order when loaded.

Comment lines can appear at the start of input file, but nowhere else in it. Comment lines must start with a semicolon.

The following is an example of hourly river flow and water temperature data that is formatted correctly:

```
; Flow and temperature data for Big Muddy River
timestamp,flow,temperature
2000-01-01 00:00:00,1000,10
2000-01-01 01:00:00,1010,11
2000-01-01 03:00:00,1030,13
```

back to top

---

ts-load-with-format

ts-load filepath format-string

Identical to ts-load except that the first column is parsed based on the *format-string* specifier.

```
let ts ts-load "time-series-data-custom-date-format.csv" "dd-MM-yyyy HH:mm:ss"
```

See Behavior and Format above concerning format-string options.

---

ts-write

ts-write logotimeseries filepath

Writes the time series data to a text file in CSV (comma-separated) format.

```
ts-write ts "time-series-output.csv"
```

The column names will be written as the header line, for example:

```
timestamp,flow,temperature
2000-01-01 00:00:00,1000,10
2000-01-01 01:00:00,1010,11
2000-01-01 03:00:00,1030,13
```

back to top

Discrete Event Scheduler

time:anchor-schedule

time:anchor-schedule logotime number period-type

Anchors the discrete event schedule to the native time tracking mechanism in NetLogo (i.e the value of *ticks*). Once anchored, LogoTimes can be used for discrete event scheduling (e.g. schedule agent 3 to perform some anonymous command on June 10, 2013). The value of the *logotime* argument is assumed to be the time at tick zero. The *number* and *period-type* arguments describe the length of one tick (e.g. a tick can represent 1 day, 2 hours, 90 seconds, etc.)

```
time:anchor-schedule time:create "2013-05-30" 1 "hour"
```

---

time:schedule-event

time:schedule-event agent anonymous-command tick-or-time
time:schedule-event agentset anonymous-command tick-or-time
time:schedule-event "observer" anonymous-command tick-or-time

Add an event to the discrete event schedule. The order in which events are added to the schedule is not important; they will be dispatched in order of the times specified as the last argument of this command. An *agent*, an *agentset*, or the string "observer" can be passed as the first argument along with an *anonymous command* as the second. The anony-

mous command is executed by the agent(s) or the observer at *tick-or-time* (either a number indicating the tick or a LogoTime), which is a time greater than or equal to the present moment (*>= ticks*).*.

If *tick-or-time* is a LogoTime, then the discrete event schedule must be anchored (see time:anchor-schedule). If *tick-or-time* is in the past (less than the current tick/time), a run-time error is raised. (The *is-after?* primitive can be used to defend against this error: add an event to the schedule only if its scheduled time is after the current time.)

Once an event has been added to the discrete event schedule, there is no way to remove or cancel it.

```
time:schedule-event turtles [ [] -> go-forward ] 1.0
time:schedule-event turtles [ [] -> fd 1 ] 1.0
time:schedule-event "observer" [ [] -> print "hello world" ] 1.0
```

---

time:schedule-event-shuffled

time:schedule-event-shuffled agentset anonymous-command tick-or-time

Add an event to the discrete event schedule and shuffle the agentset during execution. This is identical to *time:schedule-event* but the individuals in the agentset execute the action in randomized order.

```
time:schedule-event-shuffled turtles [ [] -> go-forward ] 1.0
```

---

time:schedule-repeating-event
time:schedule-repeating-event-with-period

time:schedule-repeating-event agent anonymous-command tick-or-time interval-number
time:schedule-repeating-event agentset anonymous-command tick-or-time-number interval-number
time:schedule-repeating-event “observer” anonymous-command tick-or-time interval-number
time:schedule-repeating-event-with-period agent anonymous-command tick-or-time period-duration period-type-string
time:schedule-repeating-event-with-period agentset anonymous-command tick-or-time-number period-duration period-type-string
time:schedule-repeating-event-with-period “observer” anonymous-command tick-or-time period-duration period-type-string

Add a repeating event to the discrete event schedule. This primitive behaves almost identically to *time:schedule-event* except that after the event is dispatched it is immediately rescheduled *interval-number* ticks into the future using the same *agent* (or *agentset*) and *anonymous-command*. If the schedule is anchored (see time:anchor-schedule), then *time:schedule-repeating-event-with-period* can be used to expressed the repeat interval as a period (e.g. 1 “day” or 2.5 “hours”). Warning: repeating events can cause an infinite loop to occur if you execute the schedule with time:go. To avoid infinite loops, use time:go-until.

```
time:schedule-repeating-event turtles [ [] -> go-forward ] 2.5 1.0
time:schedule-repeating-event-with-period turtles [ [] -> go-forward ] 2.5 1.0 "hours"
```

---

time:schedule-repeating-event-shuffled
time:schedule-repeating-event-shuffled-with-period

time:schedule-repeating-event-shuffled agentset anonymous-command tick-or-time-number interval-number
time:schedule-repeating-event-shuffled-with-period agentset anonymous-command tick-or-time-number interval-number

Add a repeating event to the discrete event schedule and shuffle the agentset during execution. This is identical to *time:schedule-repeating-event* but the individuals in the agentset execute the action in randomized order. If the schedule is anchored (see time:anchor-schedule), then *time:schedule-repeating-event-shuffled-with-period* can be used to expressed the repeat interval as a period (e.g. 1 “day” or 2.5 “hours”). Warning: repeating events can cause an infinite loop to occur if you execute the schedule with time:go. To avoid infinite loops, use time:go-until.

```
time:schedule-repeating-event-shuffled turtles [ [] -> go-forward ] 2.5 1.0
time:schedule-repeating-event-shuffled-with-period turtles [ [] -> go-forward ] 2.5 1.0 "month"
```

---

time:clear-schedule

time:clear-schedule

Clear all events from the discrete event schedule.

```
time:clear-schedule
```

---

time:go

time:go

Dispatch (execute) all of the events in the discrete event schedule. When each event is executed, NetLogo's tick counter (and any LogoTime variables anchored to ticks) is updated to that event's time. Note that this command will continue to dispatch events until the discrete event schedule is empty. If repeating events are in the discrete event schedule or if procedures in the schedule end up scheduling new events, it's possible for this to become an infinite loop.

```
time:go
```

---

time:go-until

time:go-until halt-tick-or-time

Dispatch all of the events in the discrete event schedule that are scheduled for times up until *halt-tick-or-time*. If the temporal extent of your model is known in advance, this variant of *time:go* is the recommended way to dispatch your model. This primitive can also be used to execute all the events scheduled before the next whole tick, which is useful if other model actions take place on whole ticks.

```
time:go-until 100.0
;; Execute events up to tick 100

time:go-until time:plus t-datetime 1.0 "hour"
;; Execute events within the next hour; t-datetime is the current time.
```

---

time:show-schedule

time:show-schedule

Reports all of the events in the schedule as a single string in tab-separated format with three columns: tick,semi-colon-separated-list-of-agents,anonymous-command.

```
print time:show-schedule
```

---

time:size-of-schedule

time:size-of-schedule

Reports the number of events in the discrete event schedule.

```
if time:size-of-schedule > 0[
  time:go
```

```
]
```

---

back to top

Building with SBT

Using SBT, create a package that will generate the time.jar. For example:

```
sbt package
```

If compilation succeeds, `time.jar` will be created. See Installation for instructions on where to put your compiled extension.

Authors

Colin Sheppard, Steve Railsback and Jacob Kelter

Feedback? Bugs? Feature Requests?

Please visit the github issue tracker to submit comments, bug reports, or feature requests. I'm also more than willing to accept pull requests.

Credits

This extension is inspired by the Ecoswarm Time Manager Library. Allison Campbell helped benchmark discrete event scheduling versus static scheduling. The extension was funded in part by the Swarm Development Group.

Terms of Use

The NetLogo dynamic scheduler extension is in the public domain. To the extent possible under law, Colin Sheppard and Steve Railsback have waived all copyright and related or neighboring rights.

back to top

Primitives

Looking for the primitive reference for the Time extension? You can find the full reference here.

All Primitives

Matrix

Using

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.

Example

```
let m matrix:from-row-list [[1 2 3] [4 5 6]]
print m
=> {{matrix:  [ [ 1 2 3 ][ 4 5 6 ] ]}}
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
=> {{matrix:  [ [ 1 2 3 ][ 4 5 10 ] ]}}

let m2 matrix:make-identity 3
print m2
```

```
=> {{matrix:  [ [ 1 0 0 ][ 0 1 0 ][ 0 0 1 ] ]}}
print matrix:times m m2 ;; multiplying by the identity changes nothing
=> {{matrix:  [ [ 1 2 3 ][ 4 5 10 ] ]}}

;; make a new matrix with the middle 1 changed to -1
let m3 (matrix:set-and-report m2 1 1 -1)
print m3
=> {{matrix:  [ [ 1 0 0 ][ 0 -1 0 ][ 0 0 1 ] ]}}
print matrix:times m m3
=> {{matrix:  [ [ 1 -2 3 ][ 4 -5 10 ] ]}}

print matrix:to-row-list (matrix:plus m2 m3)
=> [[2 0 0] [0 0 0] [0 0 2]]
```

Primitives

Looking for the primitive reference for the Matrix extension? You can find the full reference here.

Advanced features

`matrix:solve` · `matrix:forecast-linear-growth` · `matrix:forecast-compound-growth` · `matrix:forecast-continuous-growth` · `matrix:regress`

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:trace`

Matrix creation and 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`

All Primitives

matrix:make-constant

`matrix:make-constant` ***`n-rows n-cols initial-value`***

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` *`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]]
=> {{matrix:  [ [ 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
=> {{matrix:  [ [ 1 2 3 ][ 4 5 6 ][ 7 8 9 ] ]}}
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
=> {{matrix:  [ [ 100 2 3 ][ 4 5 6 ][ 7 8 9 ] ]}}
```

```
print m2
=> {{matrix:  [ [ 100 2 3 ][ 4 5 6 ][ 7 8 9 ] ]}}
;;Notice that m2 was also changed, when m1 was changed!

print m3
=> {{matrix:  [ [ 1 2 3 ][ 4 5 6 ][ 7 8 9 ] ]}}
```

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:get

`matrix:get` ***`matrix row-i col-j`***

Reports the (numeric) value at location *row-i* (second argument), *col-j* (third argument), in the given matrix given in the first argument

matrix:get-row

`matrix:get-row` ***`matrix row-i`***

Reports a simple (not nested) NetLogo list containing the elements of *row-i* (second argument) of the matrix supplied in the first argument.

matrix:get-column

`matrix:get-column` ***`matrix col-j`***

Reports a simple (not nested) NetLogo list containing the elements of *col-j* of the matrix supplied in the first argument.

matrix:set

`matrix:set` ***`matrix row-i col-j new-value`***

Changes the given *matrix* by setting the value at 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 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:  [ [ 2 3 ][ 5 6 ] ]}}
```

matrix:map

`matrix:map` *`reporter matrix`*
`matrix:map` *`reporter matrix matrices ...`*

Reports a new matrix which results from applying *reporter* (an anonymous reporter or the name of a reporter) 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 reporter 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`*

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 netlogo-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 a 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:trace

`matrix:trace` ***`matrix`***

Reports the trace of the *matrix*, which is simply the sum of the main diagonal elements.

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
=> {{matrix:  [ [ 4 ][ 2.0000000000000004 ] ]}}
;; 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

$Y = constant * growth\text{-}proportion^{\,t}$.
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-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 [ x -> x ]) ; 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 [ x -> x ]) ; 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 [ x -> x ]) ; 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.
```

GoGo

NetLogoLab and the GoGo Board Extension for sensors and robotics

This is the extension for physical computing, using sensors, motors, etc in NetLogo. It interfaces with GoGo boards running Human Interface Driver (HID) firmware, and it replaces the old GoGo Extension, which used USB-serial communications. (The USB-serial GoGo extension is still available for NetLogo 5.3.1 and earlier here.)

This extension comes bundled with NetLogo 5.2 and later. The source code is hosted online at https://github.com/NetLogo/GoGo-HID-Extension. ,# 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, 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 a 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. For more information see the the gogo board about page.

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. See the Gogo docs here for info on the the sensors that are included with a GoGo board kit.

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. See the Using Output Ports section of the GoGo docs for more info.

The GoGo board also comes with Terminal Connectors to wire in additional sensor that don't come included. Additional sensors and actuators can be found through online retailers such as Digikey, Mouser, Phidgets, Spark Fun, and Solarbotics.

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?

Gogo Boards can be purchased from the SEED Foundation.

Installing and testing the GoGo Extension

The GoGo Board connects with the computer via the USB port. 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

Plug in your GoGo board and go to https://code.gogoboard.org/. The website should prompt you to download the gogo plugin for Windows. You can now try using the NetLogo GoGo extension.

Mac OS X

The GoGo extension requires no special installation on Mac OS X.

Linux

Many versions of Linux require no special installation. ,## Usage

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.

If you are using a GoGo board 6, you will also need to initialize the extension by typing the following command into the command center:

```
gogo:init "gogo6"
```

You can see if the GoGo board is connected by seeing if it will beep when you type into the command center:

```
gogo:beep
```

Or, you can see whether one or more HID-based gogos are on and attached to the computer by typing the following into the command center:

```
gogo:howmany-gogos
```

For examples that use the GoGo extension see GoGoMonitor and GoGoMonitorSimple models in the Models Library and the example projects at the bottom of this page.

Changes

Compared to previous versions of the GoGo extension, this version offers:

- **Improved robustness**. With prior versions of the GoGo extension, crashes were fairly common due to problems in the USB-Serial stack across platforms. The switch to HID improved robustness, and the new extension also uses a "daemon" architecture which shields NetLogo from any problems that may occur in direct communication with the GoGo board. The result is a substantial reduction in the number of crashes of NetLogo.
- **No Installation of Drivers**. Because the new GoGo firmware presents the board as an HID device, the extension could be written so as not to require installing drivers. This means there is no need for the user to have administrator rights on the computer.
- **Directionality for Motors**. The board now has polarity-ensuring output connectors, so that "counterclockwise" or "clockwise" can now be specified in code.

Primitives

Looking for the primitive reference for the GoGo extension? You can find the full reference here.

General

`gogo:primitives` · `gogo:howmany-gogos` · `gogo:init`

Other Outputs

`gogo:led` · `gogo:beep`

Outputs and Servos

`gogo:talk-to-output-ports` · `gogo:talk-to-servo-ports` · `gogo:set-output-port-power` · `gogo:output-port-on` · `gogo:output-port-off` · `gogo:output-port-clockwise` · `gogo:output-port-counterclockwise` · `gogo:set-servo`

Sensors

`gogo:read-sensors` · `gogo:read-sensor`

Utilities

`gogo:read-all`

All Primitives

gogo:primitives

`gogo:primitives`

Returns a list of the primitives of this extension.

gogo:howmany-gogos

`gogo:howmany-gogos`

Reports the number of USB HID devices visible to the computer and having the correct vendor and product ID to be a GoGo board. A board will only be detected if it is both connected and powered on. Using this primitive is one way to determine quickly whether a GoGo board has the HID firmware loaded. (A USB-Serial version of the board will not be detected.).

gogo:init

`gogo:init`

Initialize the extension. The parameter `name` is a string representing the name of the backend to initialize. There are currently 2 backends. One is the `"gogo"` backend that supports GoGo board 5 and earlier versions. Another is the `"gogo6"` backend that supports GoGo board 6. You only need to use this primitive if you are using a GoGo board 6. The extension will automatically initialize itself with the `"gogo"` backend if this primitive is not called.

gogo:talk-to-output-ports

`gogo:talk-to-output-ports` *`list-of-portnames`*

Establishes a list of output ports that will be controlled with subsequent output-port commands. See below...

gogo:talk-to-servo-ports

`gogo:talk-to-servo-ports` *`list-of-portnames`*

Establishes a list of servo ports that will be controlled with subsequent output-port commands. See below...

This command is only available with the “gogo6” backend.

gogo:set-output-port-power

`gogo:set-output-port-power` *`power-level`*

`power-level` is a number between 0 and 100, reflecting the percentage of maximum power. Sets the amount of power that will be fed to the output ports indicated in `talk-to-output-ports`. This will not affect the on-off state of the output ports. So, for example, if a motor is already connected to an output port and running, changing its power will change its speed. If the motor is not running, changing the power level will not turn it on; instead, it will affect the speed at which the motor starts when it is turned on with `output-port-on`.

gogo:output-port-on

`gogo:output-port-on`

Turns on the output ports which have been indicated with talk-to-output-ports. If none have been set with talk-to-output-ports, no ports will be turned on.

gogo:output-port-off

`gogo:output-port-off`

Turns off the output ports which have been indicated with talk-to-output-ports. If none have been set with talk-to-output-ports, no ports will be turned off.

gogo:output-port-clockwise

`gogo:output-port-clockwise`

Sets the polarity of the output port(s) that have been specified with talk-to-output-ports, so that a motor attached to one of these ports would turn clockwise.

gogo:output-port-counterclockwise

`gogo:output-port-counterclockwise`

Sets the polarity of the output port(s) that have been specified with `talk-to-output-ports`, so that a motor attached to one of these ports would turn counterclockwise.

gogo:set-servo

`gogo:set-servo` *`netlogo-number`*

Sets the Pulse-Width Modulation (PWM) proportion of the output port(s) that have been specified with talk-to-output-ports. Note that the servo connectors are the male pins next to the standard motor connectors. Different servos respond to different PWM ranges, but all servos read PWM proportions and set the position of their main gear accordingly.

gogo:led

`gogo:led` *`on-or-off`*

Turns the user-LED on or off, depending on the argument. gogo:led 1 turns the LED on; gogo:led 0 turns it off.

gogo:beep

`gogo:beep`

Causes the GoGo board to beep.

gogo:read-sensors

`gogo:read-sensors`

Reports a list containing the current readings of all eight sensors ports of the GoGo.

gogo:read-sensor

`gogo:read-sensor` *`which-sensor`*

Reports the value of sensor number *which-sensor*, where *which-sensor* is a number between 0-7.

gogo:read-all

`gogo:read-all`

Reports all data available from the board, in a raw-list form useful for debugging.

gogo:send-bytes

`gogo:send-bytes` *`netlogo-list`*

Sends a list of bytes to the GoGo board. Useful for debugging or for testing any new or future functionality that is added to the GoGo board with new firmware updates.

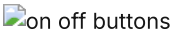
Examples of NetLogoLab models

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]
```

Now let's 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. For this sensing project we do not need motors, but we will need another device: a temperature sensor. A standard GoGo board comes with a temperature sensor. It is labeled (TEMP Sensor). More info on the sensors that come standard with a gogo board can be found in the GoGo Docs. The simplest use of a temperature sensor, obviously, is to display the temperature. We could achieve this by plugging the sensor in to input 1 and adding a monitor to the NetLogo interface with this code:

```
gogo:read-sensor 1
```

The GoGo docs say, “Note that the readout is not in Celsius or Fahrenheit. The value is reflects the resistance of the sensor. Mappings between the resistance and the standard units must be done manually.” So, to figure out how these values map to a real temperature, you will need to use a thermometer and see how the reading on the temperature sensor corresponds to the temperature in degrees. If you found out that the conversion is to divide by 30, the monitor on the NetLogo interface could be changed to:

```
gogo:read-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:read-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.

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:read-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:read-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:read-sensor 1)
```

View2.5D

The View2.5D extension offers visualization for Patch and Turtle reporters, in real time, in a simulation's context.

How to Use

The view2.5d extension is pre-installed in NetLogo.

To use the view2.5d extension in your model, add a line to the top of your Code tab:

```
extensions [view2.5d]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `view2.5d` to the list.

For more information on using NetLogo extensions, see the Extensions Guide

Incorporating Into Models

open a window using either the `view2.5d:patch-view` or `view2.5d:turtle-view` commands (it can be a good idea to put these in your 'SETUP' procedure or a separate button).

update your window's view using one of the update commands (put these in your 'GO' procedure).

See the View2.5d Code Examples in the NetLogo models library.

Feedback

Send comments, bugs, or other feedback to CCL Feedback.

Primitives

Looking for the primitive reference for the View2.5D extension? You can find the full reference here.

`view2.5d:patch-view` · `view2.5d:decorate-patch-view` · `view2.5d:undecorate-patch-view` · `view2.5d:turtle-view` · `view2.5d:update-all-patch-views` · `view2.5d:update-patch-view` · `view2.5d:update-turtle-view` · `view2.5d:get-z-scale` · `view2.5d:set-z-scale` · `view2.5d:set-turtle-stem-thickness` · `view2.5d:set-turtle-stem-color` · `view2.5d:show-links-xy-plane` · `view2.5d:show-links-xyz` · `view2.5d:get-observer-angles` · `view2.5d:set-observer-angles` · `view2.5d:get-observer-xy-focus` · `view2.5d:set-observer-xy-focus` · `view2.5d:get-observer-distance` · `view2.5d:set-observer-distance` · `view2.5d:remove-patch-view` · `view2.5d:remove-turtle-view` · `view2.5d:remove-all-patch-views` · `view2.5d:remove-all-turtle-views` · `view2.5d:count-windows`

All Primitives

view2.5d:patch-view

`view2.5d:patch-view` *`title reporter`*

This command must be called from the Observer context. (Attempting to call from another context causes an error) The Title is a string, which will be used to label the new Window and to call for subsequent updates and modifications. Specification of the Reporter uses the NetLogo anonymous procedure syntax, from the Observer perspective.

Example:

```
view2.5d:patch-view "Test"  [ [the-patch] -> [pxcor] of the-patch ]
```

view2.5d:decorate-patch-view

`view2.5d:decorate-patch-view` *`title`*

This command must be called from the Observer context. (Attempting to call from another context causes an error) The Title is a string, the label of an existing Patch View Window. Effect: draws the turtles of the model at their current location, on top of the Patch view display

> NOTE: only has an effect in the “structures” patch view (in the others, the patch value is inclined based on neighbors & gradient)

> NOTE: for negative patch values, the turtle shapes are drawn below (orbit underneath to see them)

Example:

```
view2.5d:decorate-patch-view "Test"
```

view2.5d:undecorate-patch-view

`view2.5d:undecorate-patch-view` *`title`*

This command must be called from the Observer context. (Attempting to call from another context causes an error)

The Title is a string, the label of an existing Patch View Window. Effect: STOPS drawing the turtles of the model at their current location, on top of the Patch view display

Example:

```
view2.5d:undecorate-patch-view "Test"
```

view2.5d:turtle-view

`view2.5d:turtle-view` *`title agents reporter`*

This command must be called from the Observer context. (Attempting to call from another context causes an error) The Title is a string, which will be used to label the new Window and to call for subsequent updates. The turtle-set is any selector for turtles. Reporter is an anonymous reporter that should take a turtle as input, and report some number from it.

Example:

```
view2.5d:turtle-view "Test" turtles with [color = red] [ the-turtle -> [energy] of the-turtle]
; This would create a new 2.5d window, plotting the ENERGY value of all turtles that are red.
```

view2.5d:update-all-patch-views

`view2.5d:update-all-patch-views`

This command must be called from the Observer context. Updates all existing patch-view windows according to the latest values.

view2.5d:update-patch-view

`view2.5d:update-patch-view` ***`title`***

This command must be called from the Observer context. Updates only the patch-view window with the specified title (if any).

view2.5d:update-turtle-view

`view2.5d:update-turtle-view` ***`title agents`***

This command must be called from the Observer context. Updates only the turtle-view window with the specified title (if any). The turtle-set selector must be supplied to refresh the set of turtles.

view2.5d:get-z-scale

`view2.5d:get-z-scale` ***`title`***

This reporter must be called from the Observer context. Returns the current z-scale of the turtle-view or patch-view window with the specified title (if any).

view2.5d:set-z-scale

`view2.5d:set-z-scale` ***`title new-z-scale`***

This command must be called from the Observer context. Updates only the turtle-view or patch-view window with the specified title (if any). The view is now displayed with the new z-scale.

view2.5d:set-turtle-stem-thickness

`view2.5d:set-turtle-stem-thickness` ***`title thickness`***

This command must be called from the Observer context. Updates only the turtle-view window with the specified title (if any). Turtles are now drawn with “pins” or “stems” that have the specified thickness (instead of the hairline default).

view2.5d:set-turtle-stem-color

`view2.5d:set-turtle-stem-color` ***`title color-reporter`***

This command must be called from the Observer context. Updates only the turtle-view window with the specified title (if any). colorReporter is an anonymous reporter that should take a turtle as input, and report some number representing a color from it. Turtles are now drawn with “pins” or “stems” that have the specified color (instead of the grey default).

Example:

```
;; setup view with 2 turtles
crt 1 [ set color green  setxy 2 3]
crt 1 [ set color red  setxy 5 4]
view2.5d:turtle-view "Test" turtles [ the-turtle -> [ xcor] of the-turtle ]
view2.5d:set-observer-distance "Test" 40
view2.5d:set-z-scale "Test" 2
view2.5d:set-observer-angles "Test" 25 30

;; increase the stem thickness
view2.5d:set-turtle-stem-thickness "Test" .2

;; change the stem color to  match the turtle-color
view2.5d:set-turtle-stem-color "Test" [ the-turtle -> [ color ] of the-turtle ]

; now make the stems orange
view2.5d:set-turtle-stem-color "Test" [ orange ]
```

view2.5d:show-links-xy-plane

`view2.5d:show-links-xy-plane` *`title`*

This command must be called from the Observer context. Updates only the turtle-view or patch-view window with the specified title (if any). Links are drawn in the xy-plane (instead of the turtle to turtle default). This option can be selected using the Link Options xy-plane radio Button in the 3D view.

view2.5d:show-links-xyz

`view2.5d:show-links-xyz` *`title`*

This command must be called from the Observer context. Updates only the turtle-view or patch-view window with the specified title (if any). Links are drawn from turtle to turtle (this is the default way links are displayed). This option can be selected using the Link Options xyz radio Button in the 3D view.

view2.5d:get-observer-angles

`view2.5d:get-observer-angles` *`title`*

This reporter must be called from the Observer context. Returns a list reflecting the observer's angular perspective { heading pitch } (the place on an imaginary sphere at the zoom distance is updated to obey heading & pitch given)

view2.5d:set-observer-angles

`view2.5d:set-observer-angles` *`title heading pitch`*

This command must be called from the Observer context. Updates only the turtle-view window with the specified title (if any). Sets the observer's angular perspective (the place on an imaginary sphere at the zoom distance is updated to obey heading & pitch given)

view2.5d:get-observer-xy-focus

`view2.5d:get-observer-xy-focus` *`title`*

This reporter must be called from the Observer context. Returns a list reflecting the x and y coordinates the observer is “looking at” in the patch plane.

view2.5d:set-observer-xy-focus

`view2.5d:set-observer-xy-focus` *`title netlogo-number ycor`*

This command must be called from the Observer context. Updates only the turtle-view window with the specified title (if any). Sets the x and y coordinates the observer that is “looking at” in the patch plane.

view2.5d:get-observer-distance

`view2.5d:get-observer-distance` *`title`*

This reporter must be called from the Observer context. Returns the observer’s distance from its “focus point”

view2.5d:set-observer-distance

`view2.5d:set-observer-distance` *`title new-distance`*

This command must be called from the Observer context. Updates only the turtle-view window with the specified title (if any). Sets the observer’s distance from its “focus point”

view2.5d:remove-patch-view

`view2.5d:remove-patch-view` *`title`*

This command closes and removes the specified patch view programmatically (equivalent to closing the window manually).

view2.5d:remove-turtle-view

`view2.5d:remove-turtle-view` *`title`*

This command closes and removes the specified turtle view programmatically (equivalent to closing the window manually).

view2.5d:remove-all-patch-views

`view2.5d:remove-all-patch-views`

This command closes and removes all patch views programmatically (equivalent to closing the windows manually).

view2.5d:remove-all-turtle-views

`view2.5d:remove-all-turtle-views`

This command closes and removes all turtle views programmatically (equivalent to closing the windows manually).

view2.5d:count-windows

`view2.5d:count-windows`

This reporter returns the number of turtle and patch views that are currently active.

Table

Using

The table extension is pre-installed in NetLogo.

To use the table extension in your model, add a line to the top of your Code tab:

```
extensions [table]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `table` to the list.

For more information on using NetLogo extensions, see the Extensions Guide

When to Use

In general, anything you can do with an table in NetLogo, you could also just use a list for. But you may want to consider using an table instead for speed reasons. Lists and tables have different performance characteristics, so you may be able to make your model run faster by selecting the appropriate data structure.

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, tables are "mutable". That means that you can actually modify them directly, rather than constructing an altered copy as with lists. If the 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.

Example

```
let dict table:make
table:put dict "turtle" "cute"
table:put dict "bunny" "cutest"
print dict
=> {{table: "turtle" -> "cute", "bunny" -> "cutest" }}
print table:length dict
=> 2
print table:get dict "turtle"
=> "cute"
print table:get dict "leopard"
=> (error)
print table:keys dict
=> ["turtle" "bunny"]
```

Manipulating Tables

If the same key is used with `table:put` more than once for the same table, the value provided to *last* call of `table:put` will be the value shown when `table:get` is used. Here is an example:

```
let dict table:make
table:put dict "a" 5
table:put dict "a" 3
print table:get dict "a"
=> 3
```

Because tables are mutable, manipulating existing values should be done by calling `table:get` or `table:get-or-default` on a key, transforming the returned value, and then calling `table:put` to update the transformed value in the table. Here is an example procedure which increments a value in a table at a given key. If the key doesn't exist, it puts a 1 at that key instead.

```
to increment-table-value [ dict key ]
  let i table:get-or-default dict key 0
  table:put dict key i + 1
end
```

As `increment-table-value` shows, when a table is given as an input for a procedure, modifications made to it with `table:put`, `table:remove`, or `table:clear` are reflected in the original value outside of the procedure. This is different behavior than with list values, which are immutable and so cannot be changed when given as inputs. Caution needs to be exercised when using `let` or `set`, as they can give a different variable name to the same table. If you change the value for a key in the table using one variable, any other variables assigned that same table will also reflect the change.

```
let dict table:make
table:put dict "a" 5
let alt dict
table:put alt "a" 3
print table:get dict "a"
=> 3 ; changed because `dict` and `alt` refer to the same table
print table:get alt "a"
=> 3
```

If you want to create a copy of a table instead of assigning the same table to multiple variable names, here is a simple reporter for creating a duplicate table from an existing one:

```
to-report copy-table [ orig ]
  let copy table:make
  foreach ( table:keys orig ) [
    [key] -> table:put copy key ( table:get orig key )
  ]
  report copy
end
```

And here is a sample usage of the `copy-table` reporter:

```
let dict table:make
table:put dict "a" 5
let alt copy-table dict
table:put alt "a" 3
print table:get dict "a"
=> 5 ; `dict` is not changed because we created a new table copy for `alt`
print table:get alt "a"
=> 3
```

Key Restrictions

Table keys are limited to the following NetLogo types:

- Numbers
- Strings
- Booleans
- Lists containing only elements which are themselves valid keys

If you attempt to use an illegal value, the table extension will raise an exception, as shown in the following example.

```
crt 1
let dict table:make
table:put dict (one-of turtles) 10
;; Errors with the following message:
;; (turtle 0) is not a valid table key (a table key may only be a number, a string, true or false, or
a list whose items are valid keys)
```

Primitives

Looking for the primitive reference for the Table extension? You can find the full reference here.

`table:clear` · `table:counts` · `table:group-agents` · `table:group-items` · `table:from-list` · `table:from-json` · `table:from-json-file` · `table:get` · `table:get-or-default` · `table:has-key?` · `table:keys` · `table:length` · `table:make` · `table:put` · `table:remove` · `table:to-list` · `table:to-json` · `table:values`

All Primitives

table:clear

`table:clear` ***`table`***

Removes all key-value pairs from *table*.

table:counts

`table:counts` ***`netlogo-list`***

Counts the occurrences of each element of the given list and reports the counts in a table.

table:group-agents

`table:group-agents` ***`agentset reporter`***

Groups the agents in the agentset based on the given reporter. Agents that report the same thing for reporter will be grouped together. The results of the reporter will be used as the keys in the resulting table and the groups will be agentsets.

For example:

```
observer> create-turtles 100 [ set color one-of [ red green blue ] ]
observer> show table:group-agents turtles [ color ]
```

```
observer: {{table: [[105 (agentset, 38 turtles)] [55 (agentset, 32 turtles)] [15 (agentset, 30
turtles)]]}}
```

table:group-items

`table:group-items` *`netlogo-list anonymous-reporter`*

Groups the items of the list based on the given reporter. The reporter should take a single argument, which will be the items of the list. Items that report the same thing when passed to the reporter will be grouped together. The results of the reporter will be used as the keys in the resulting table and the groups will be lists.

For example:

```
observer> show table:group-items range 10 [ num -> num mod 3 ]
observer: {{table: [[0 [0 3 6 9]] [1 [1 4 7]] [2 [2 5 8]]]}}
```

table:from-list

`table:from-list` *`netlogo-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:from-json

`table:from-json` *`netlogo-string`*

```
Reads in the JSON object from the given string argument, converts it to a table, and reports the
table.
The JSON object in the argument must be an object, not an array (i.e. the outmost characters
must be `{` and `}`, not `[` and `]`),
but there can be arrays inside the JSON object which will be converted to NetLogo lists.
```

table:from-json-file

`table:from-json-file` *`filename`*

```
Reads in the JSON object from *filename.json*, converts it to a table and reports the table.
The JSON object in *filename.json* must be an object, not an array (i.e. the outmost characters
of the file must be `{` and `}`, not `[` and `]`),
but there can be arrays inside the JSON object which will be converted to NetLogo lists.
```

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:get-or-default

`table:get-or-default` *`table key default-value`*

Reports the value that *key* is mapped to in the table. Reports the default-value 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.

table:to-json

`table:to-json table`

Reports a string with the contents of the given table as a JSON string.

table:values

`table:values` *`table`*

Reports a list with the entries of *table*. The entries will appear in the same order they were inserted, with duplicates included.

Vid

Concepts

Video Source

The `vid` extension has a built-in concept of a video source. At the moment, the only video sources available are movies in the directory the model lives in and cameras attached to the computer. The `vid` extension opens a new video source with the `vid:<source>-open` and `vid:<source>-select`. These primitives change the source to the selected source. If a source is already open, it closes it before opening a new one.

Source Lifecycle

Movie sources are "stopped" after being created by `vid:movie-select` or `vid:movie-open`. Camera sources start off as "playing" after being created by `vid:camera-select` or `vid:camera-open`. If a source is in status "stopped" it can be started with `vid:start`. Conversely, if the source is "playing" it can be stopped with `vid:stop`. When a source is "stopped", each call to `vid:capture-image` will return the same image.

Video Recorder

The `vid` extension also has the concept of a recording, a series of frames which can be sewn into an "mp4" movie. The recorder status can be queried using `vid:recorder-status`. The recorder status is "inactive" until started with `vid:start-recorder`, which sets it to "recording". While the recorder is "recording" the `vid:record-view`, `vid:record-interface`, and `vid:record-source` can be used to save frames to the recording. You can choose to save the recording while recording using `vid:save-recording` which saves the movie to the specified file and reset the recording status to "inactive". If you would prefer to throw away the recorded frames without saving, use `vid:reset-recorder`.

Known Issues

When running the extension on macOS and using `vid:camera-open` or `vid:camera-select` and then starting a different program (like Zoom) and using the same camera will crash NetLogo. If you start the other program first and then start NetLogo things should work better.

Primitives

Looking for the primitive reference for the Vid extension? You can find the full reference here.

`vid:camera-names` · `vid:camera-open` · `vid:camera-select` · `vid:movie-select` · `vid:movie-open` · `vid:movie-open-remote` · `vid:close` · `vid:start` · `vid:stop` · `vid:status` · `vid:capture-image` · `vid:set-time` · `vid:show-player` · `vid:hide-player` · `vid:record-view` · `vid:record-interface` · `vid:record-source` · `vid:recorder-status` · `vid:reset-recorder` · `vid:start-recorder` · `vid:save-recording`

All Primitives

vid:camera-names

`vid:camera-names`

Provides a list of all available cameras.

Example:

```
vid:camera-names => []
vid:camera-names => ["Mac Camera"]
vid:camera-names => ["Logitech Camera"]
```

vid:camera-open

`vid:camera-open`

Opens the named camera as a video source. If no name is provided, opens the first camera that would be listed by `camera-names`.

Example:

```
vid:camera-open ; opens first camera
(vid:camera-open "Logitech Camera")
```

Errors:

- Message `"vid: no cameras found"`: no cameras are available.
- Message `"vid: camera "\<name\>" not found"`: if the named camera is not available.

vid:camera-select

`vid:camera-select`

Prompts the user to select a camera as video source. This command does not error if the user cancels. Use `vid:status` to see if a user selected a camera.

Example:

```
vid:camera-select
```

Errors:

- Message “vid: no cameras found”: no cameras are available.

vid:movie-select

`vid:movie-select`

Prompts the user to select a movie to use as a video source. The formats supported are those supported by JavaFX2. This command does not error if the user cancels. Use `vid:status` to see if the user selected a movie.

Example:

```
vid:movie-select
```

Errors:

- Message `"vid: format not supported"`: the user selected a movie with an unsupported format.

vid:movie-open

`vid:movie-open` *`filename`*

Opens a video from the file system. If the provided path is not absolute the extension searches for the given path relative to the current model directory. If the provided path is absolute the extension opens the file.

Example:

```
vid:movie-open "foo.mp4"      ; Opens foo.mp4 in the directory containing the model
vid:movie-open user-file      ; Opens a dialog for the user to select a movie
vid:movie-open "/tmp/foo.mp4" ; Opens a movie from the "/tmp" directory
```

Errors:

- Message `"vid: no movie found"`: the movie could not be found.
- Message `"vid: format not supported"`: the user selected a movie with an unsupported format.

vid:movie-open-remote

`vid:movie-open-remote` *`url`*

Opens a remote video from a website or ftp server.

Example:

```
vid:movie-open-remote "http://example.org/foo.mp4"
```

Errors:

- Message `"vid: no movie found"`: The specified URL could not be loaded or errored while loading.
- Message `"vid: format not supported"`: The file type of the remote movie is not supported.
- Message `"vid: protocol not supported"`: The movie was at an unsupported URL protocol. Supported protocols are `ftp` and `http`.

vid:close

`vid:close`

Closes the currently selected video source. Has no effect if there is no active video source.

Example:

```
vid:close
```

vid:start

`vid:start`

Starts the selected video source. A video source must have been selected before calling `vid:start`.

Example:

```
vid:start
```

Errors:

- Message `"vid: no selected source"`: There is no currently selected video source. Select a source with `vid:movie-open`, `vid:movie-select`, `vid:camera-open`, or `vid:camera-select`.

vid:stop

`vid:stop`

Stops the currently running video source.

Example:

```
vid:stop
```

vid:status

`vid:status`

Reports the current status of an active video. Note that after calling `vid:movie-open` or `vid:movie-select` the status will be set to “stopped”, while after calling `vid:camera-open` or `vid:camera-select` the status will be “playing”.

Example:

```
vid:status     ; => "inactive"

vid:movie-open "foobar.mp4"
vid:status      ; => "stopped"

vid:movie-start
vid:status       ; => "playing"
```

vid:capture-image

`vid:capture-image` *`width height`*

Captures an image from the currently selected active source.

If width and height are not specified, the image is captured at the current source resolution.

Example:

```
extensions [ vid bitmap ]

to capture
  ; capture an image if a video source is open,
  ; have the user select a camera if no video source found
  carefully [
    ; when camera open, take an image
    let image vid:capture-image ; returns image suitable for use with bitmap extension
    bitmap:copy-to-drawing image 0 0
  ] [
    if error-message = "Extension exception: vid: no selected source" [
      vid:camera-select
      vid:start
      let image vid:capture-image
      bitmap:copy-to-drawing image 0 0
    ]
  ]
end
```

If you want to capture images at a different resolution, simply replace `vid:capture-image` with, e.g., `(vid:capture-image 640 480)`.

Errors:

- Message `"vid: no selected source"`: There is no currently selected video source. Select a source with `vid:movie-open`, `vid:movie-select`, `vid:camera-open`, or `vid:camera-select`.
- Message `"vid: invalid dimensions"`: The selected dimensions are invalid (one of the dimensions is zero or negative).

vid:set-time

`vid:set-time` *`seconds`*

Sets the time of the current video source to `*seconds*`. This has no effect when the current video source is a camera.

Example:

```
vid:set-time 100
```

Errors:

- Message `"vid: no selected source"`: There is no currently selected video source. Select a source with `vid:movie-open`, `vid:movie-select`, `vid:camera-open`, or `vid:camera-select`.
- Message `"vid: invalid time"`: The currently active video does not contain the specified second. The second may be negative, or greater than the length of the video.

vid:show-player

`vid:show-player` *`width height`*

Shows a player in a separate window. If there is no video source, the window will be an empty black frame. If there is an active video source, it will be displayed in the window with the specified width and height. If there is a playing video source, it will be displayed in the window at its specified width and height. If width and height are omitted, the video will be displayed in its native resolution.

Example with native resolution:

```
vid:show-player
```

Example with custom resolution:

```
(vid:show-player 640 480)
```

Errors:

- Message `"vid: invalid dimensions"`: The selected dimensions are invalid (one of the dimensions is zero or negative).

vid:hide-player

`vid:hide-player`

Hides the player if open. Does nothing if there is no player window.

Example:

```
vid:hide-player
```

vid:record-view

`vid:record-view`

Records the current image shown in the NetLogo view to the active recording.

Example:

```
vid:record-view
```

Errors:

- Message `"vid: recorder not started"`: The recorder has not been started. Start the recorder with `vid:start-recorder`.

vid:record-interface

`vid:record-interface`

Records the NetLogo interface view to the active recording.

Example:

```
vid:record-interface
```

Errors:

- Message `"vid: recorder not started"`: The recorder has not been started. Start the recorder with `vid:start-recorder`.
- Message `"vid: export interface not supported"`: The calling NetLogo version does not support interface exports. This will occur when running NetLogo headlessly.

vid:record-source

`vid:record-source`

Records a frame to the active recording from the currently active source.

Example:

```
vid:record-source
```

Errors:

- Message `"vid: recorder not started"`: The recorder has not been started. Start the recorder with `vid:start-recorder`.
- Message `"vid: no selected source"`: There is no currently selected video source. Select a source with `vid:movie-open`, `vid:movie-select`, `vid:camera-open`, or `vid:camera-select`.

vid:recorder-status

`vid:recorder-status`

Reports the current status of the recorder. Initially and after the recorder is saved (via `vid:save-recording`) or reset (via `vid:reset-recorder`) the recorder status is “inactive”. After calling `vid:start-recorder` the status will be “recording”.

Example:

```
vid:recorder-status ; => "inactive"

vid:start-recorder
vid:recorder-status ; => "recording"

vid:reset-recorder
vid:recorder-status ; => "inactive"
```

vid:reset-recorder

`vid:reset-recorder`

Stops the current recording, discards any recorded frames without saving them, and forgets the currently set recording resolution.

vid:start-recorder

`vid:start-recorder`

Starts the recorder. If the recorder is already running this will cause an error to be raised. If desired, a recording width and height can be supplied. If width and height are not supplied, they will be determined from the first frame recorded.

Example:

```
vid:start-recorder
(vid:start-recorder 640 480)
```

Errors:

- Message `"vid: recorder already started"`: The recorder has already been started. The existing recording should be saved or reset before starting the recording.
- Message `"vid: invalid dimensions"`: The selected dimensions are invalid (one of the dimensions is zero or negative).

vid:save-recording

`vid:save-recording` *`filename`*

Saves the recording to the specified path. If the recorder is not running this will cause an error to be raised. Note that at present the recording will always be saved in the "mp4" format. If the supplied filename does not end in ".mp4", the ".mp4" suffix will be added. Note that `vid:save-recording` *will* overwrite existing files of the same name. `vid:save-recording` will error if the recorder has not been started or if the file cannot be written since the containing directory does not exist.

Example:

```
vid:save-recording "foo.mp4"      ; Saves to foo.mp4 in the directory containing the model
vid:save-recording user-new-file  ; Opens a dialog for the user to select a save path
vid:save-recording "/tmp/foo.mp4" ; Saves the recording to the "/tmp" directory
```

Errors:

- Message `"vid: recorder not started"`: The recorder has not been started. Start the recorder with `vid:start-recorder`.
- Message `"vid: no such directory"`: The directory containing the specified save file does not exist.
- Message `"vid: no frames recorded"`: You tried to save a recording with no frames recorded. Check that you are recording properly or use `vid:reset-recording` to to change the recording format without saving.

Rnd

This extension adds the ability to do roulette wheel selection in NetLogo. It provides a simpler way to accomplish the same thing as the Lottery Example from the NetLogo Models Library.

Usage

Which primitive to use depends on whether you want to select an item from a list or from an agenset. It also depends on whether you want one or many items and, if you want many, if repeats are allowed or not. The following table summarizes the situation:

	From an AgentSet	From a List
One item	`rnd:weighted-one-of`	`rnd:weighted-one-of-list`
Many items, without repeats	`rnd:weighted-n-of`	`rnd:weighted-n-of-list`
Many items, with repeats	`rnd:weighted-n-of-with-repeats`	`rnd:weighted-n-of-list-with-repeats`

(**Note:** the initial version of the extension had a single set of primitives for both lists and agentsets, but it turned out to be confusing, so we changed it. If you were using the old version of the extension, you will need to modify your code to use the new primitives.)

In all cases, you will need to provide two things to the primitive:

- The "candidates": the items that the primitive will select from.
- The "weight": how likely it is for each candidate to be selected.

If you want to select more than one items, you will also need to tell it:

- How many items to select.

A note about performance

The extension uses Keith Schwarz's implementation of Vose's Alias Method (see Schwarz's Darts, Dice, and Coins page). Assuming you are choosing *n* candidates for a collection of size *m* **with repeats**, this method has an initialization cost of *O(m)* followed by a cost of *O(1)* for each item you pick, so *O(m + n)* overall.

For example, in the following code:

```
let candidates n-values 500 [ [n] -> n ]
rnd:weighted-n-of-list-with-repeats 100 candidates [ [w] -> w ]
n-values 100 [ rnd:weighted-one-of-list candidates [ [w] -> w ] ]
```

...the line using `rnd:weighted-n-of-list-with-repeats` will likely run 100 times faster than the line using a combination of `n-values` and `rnd:weighted-one-of-list`. This is because `rnd:weighted-n-of-list-with-repeats` only initializes the algorithm once and `rnd:weighted-one-of` does it each time it is called.

(Note that composing `n-values` with `rnd:weighted-one-of-list` does not preserve the order of the original candidate list, while `rnd:weighted-n-of-list-with-repeats` does.)

Things are a bit more complicated if you are choosing **without repeats**, however. In this case, the algorithm may have to discard some picks because the candidates have already been selected. When this starts happening too often (maybe because some weights are much bigger than others), the extension re-initializes the algorithm with the already-picked candidates excluded. This should not happen too often, however, so while picking without repeats has an upper bound of *O(m * n)* in theory, it should usually not be much more than *O(m + n)* in practice.

The previous remarks apply to agentset primitives as much as they apply to list primitives.

Primitives

Looking for the primitive reference for the Rnd extension? You can find the full reference here.

AgentSet Primitives

`rnd:weighted-one-of` · `rnd:weighted-n-of` · `rnd:weighted-n-of-with-repeats`

List Primitives

`rnd:weighted-one-of-list` · `rnd:weighted-n-of-list` · `rnd:weighted-n-of-list-with-repeats`

All Primitives

rnd:weighted-one-of

`rnd:weighted-one-of` ***`agentset reporter`***

Reports a random agent from

agentset

.

The probability of each agent being picked is proportional to the weight given by the

reporter

for that agent. The weights must not be negative.

If the agentset is empty, it reports `nobody`.

Here is a full rewrite of the **Lottery Example** model using the `rnd:weighted-one-of` primitive:

```
extensions [ rnd ]

to setup
  clear-all
  ; create a turtle on every fifth patch
  ask patches with [ pxcor mod 5 = 0 and pycor mod 5 = 0 ] [
    sprout 1 [
      set size 2 + random 6 ; vary the size of the turtles
      set label 0           ; start them out with no wins
      set color color - 2   ; make turtles darker so the labels stand out
    ]
  ]
  reset-ticks
end

to go
  ask rnd:weighted-one-of turtles [ size ] [
```

```
    set label label + 1
  ]
  tick
end
```

rnd:weighted-n-of

`rnd:weighted-n-of` ***`size agentset [ reporter ]`***

Reports an agentset of the given

size
randomly chosen from the
agentset
, with no repeats.
The probability of each agent being picked is proportional to the weight given by the

reporter
for that agent. The weights must be non-negative numbers.
It is an error for

size
to be greater than the size of the
agentset
.
If, at some point during the selection, there remains only candidates with a weight of `0.0`, they all have an equal probability of getting picked.

rnd:weighted-n-of-with-repeats

`rnd:weighted-n-of-with-repeats` ***`size agentset [ reporter ]`***

Reports a **list** of the given

size
randomly chosen from the
agentset
, with repeats. (Why a list instead of an agentset? Because an agentset cannot contain the same agent more than once.)
The probability of each agent being picked is proportional to the weight given by the

reporter
for that agent. The weights must be non-negative numbers.
It is **not** an error for

size
to be greater than the size of the
agentset
, but there has to be at least one candidate.
If, at some point during the selection, there remains only candidates with a weight of `0.0`, they all have an equal probability of getting picked.

If all weights are `0.0`, each candidate has an equal probability of being picked.

rnd:weighted-one-of-list

`rnd:weighted-one-of-list` ***`netlogo-list anonymous-reporter`***

Reports a random item from

list
.

The probability of each item being picked is proportional to the weight given by the

anonymous-reporter
for that item. The weights must not be negative. The first argument passed to the anonymous procedure refers to the list item. (See the Anonymous Procedures section of the Programming Guide for more details.)

It is an error for the list to be empty.

A common way to use the primitive is to have a list of lists, where the first item of each sublist is the thing you want to choose and the second item is the weight. Here is a short example:

```
let pairs [ [ "A" 0.2 ] [ "B" 0.8 ] ]
repeat 25 [
  ; report the first item of the pair selected using
  ; the second item (i.e., `last p`) as the weight
  type first rnd:weighted-one-of-list pairs [ [p] -> last p ]
]
```

This should print `B` roughly four times more often than it prints `A`.

If you happen to have your items and your weights in two separate lists, you can combine them into pairs by using a combination of `map` and `list`:

```
let items [ "A" "B" "C" ]
let weights [ 0.1 0.2 0.7 ]
let pairs (map list items weights)
```

Since we apply `map` to both the `items` list and the `weights` list, the parentheses are needed in `(map list items weights)`. We also use the concise anonymous procedure syntax (see the programming guide) to pass `list` as the reporter for `map`. The same thing could have been written `(map [ [a b] -> list a b ] items weights)`.

rnd:weighted-n-of-list

`rnd:weighted-n-of-list` *`size netlogo-list anonymous-reporter`*

Reports a list of the given

size
randomly chosen from the
list
of candidates, with no repeats.

The probability of each item being picked is proportional to the weight given by the

anonymous-reporter
for that item. The weights must not be negative. The first argument passed to the anonymous procedure refers to the list item. (See the Anonymous Procedures section of the Programming Guide for more details.)

It is an error for

size
to be greater than the size of the
list of candidates
.

If, at some point during the selection, there remains only candidates with a weight of `0.0`, they all have an equal probability of getting picked.

The items in the resulting list appear in the same order that they appeared in the list of candidates. (If you want them in random order, use `shuffle` on the result).

Example:

```
let candidates n-values 8 [ [n] -> 2 ^ (n + 1) ] ; make a list with the powers of two
print rnd:weighted-n-of-list 4 candidates [ [w] -> w ]
```

This should print a list of four numbers, where the bigger numbers (32, 64, 128, 256) have a much better chance to show up than the smaller ones (2, 4, 8, 16).

rnd:weighted-n-of-list-with-repeats

`rnd:weighted-n-of-list-with-repeats` ***`size netlogo-list anonymous-reporter`***

Reports a list of the given

size
randomly chosen from the
list
of candidates, with repeats.

The probability of each item being picked is proportional to the weight given by the

anonymous-reporter
for that item. The weights must not be negative. The first argument passed to the anonymous procedure refers to the list item. (See the Anonymous Procedures section of the Programming Guide for more details.)

It is **not** an error for

size
to be greater than the size of the
list
of candidates, but there has to be at least one candidate.

If, at some point during the selection, there remains only candidates with a weight of `0.0`, they all have an equal probability of getting picked.

If all weights are `0.0`, each candidate has an equal probability of being picked.

The items in the resulting list appear in the same order that they appeared in the list of candidates. (If you want them in random order, use `shuffle` on the result).

Example:

```
let pairs [ [ "A" 0.2 ] [ "B" 0.8 ] ]
print map first rnd:weighted-n-of-list-with-repeats 25 pairs [ [p] -> last p ]
```

This should print a list of 25 `A`s and `B`s, with roughly four times as many `B`s than `A`s.

LevelSpace

LevelSpace fundamentals

LevelSpace must be loaded in a model using `extensions [ls]` at the top of your model. Once this is done, a model will be able to load up other models using the LevelSpace primitives, run commands and reporters in them, and close them down when they are no longer needed.

Asking and reporting in LevelSpace is conceptually pretty straight forward: You pass blocks of code to child models, and the child models respond as if you had typed that code into their Command Center. LevelSpace allows you to report strings, numbers, and lists from a child to its parent. It is not possible to directly report turtles, patches, links, or any of their respective sets. Further, it is not possible to push data from a child to its parent - parents must ask their children to report. This mimicks the way in which turtles cannot "push" data to the observer, but rely on the observer to ask them for it.

In general, the LevelSpace syntax has been designed to align with existing NetLogo primitives whenever possible.

Headless and Interactive Models

LevelSpace has two different child model types; headless models and interactive models. They each have their strengths and weaknesses:

Interactive models

- are full-fledged models that give full access to their interface and widgets,
- run a bit slower, and use more memory
- are visible by default

Headless Models

- only give you access to their view and command center
- are faster and use less memory than interactive models.
- are hidden by default

Typically you will want to use headless models when you are running a large number of models, or if you simply want to run them faster. Interactive models are good if you run a small amount of models, if you are writing a LevelSpace model and need to be able to debug, or if you need access to widgets during runtime.

Keeping Track of Models

Child models are kept track of in the extension with an id number, starting with 0, and all communication from parent to child is done by referencing this number, henceforth referred to as `model-id`.

The easiest way to work with multiple models is to store their `model-id` in a list, and use NetLogo's list primitives to sort, filter, etc. them during runtime.

Keeping track of models is important: Most LevelSpace primitives will fail and cause a runtime interruption if provided a `model-id` to a non-existing model. You can use `ls:model-exists? model-id` to check if `model-id` refers to an existing model.

A general use case: Asking and Reporting

This use case is based on the Model Visualizer and Plotter Example-model from the NetLogo Models Library.

A simple thing we can do is to open up some models, run them concurrently, and calculate the average of some reporter. Let's say that we are interested in finding the mean number of sheep in a bunch of Wolf Sheep Predation models. First we would open up some of these models, and set them up:

```
to setup
  ls:reset
  ca
  ls:create-models 30 "Wolf Sheep Predation.nlogox"
  ls:ask ls:models [ set grass? true setup ]
  reset-ticks
end
```

We then want to run all our child models, and then find out what the mean number of sheep is:

```
to go
    ls:ask ls:models [ go ]
    show mean [ count sheep ] ls:of ls:models
end
```

A general use case: Inter-Model Interactions

This use case is based on the Model Interactions Example-model from the NetLogo Models Library.

Let's imagine that we have two models: a Wolf Sheep Predation-model called `WSP`, and a Climate Change model called `CC`. Now let's imagine that we want the regrowth time in the wSP model to depend on the temperature in the CC model. Using LevelSpace's primitives, we could do something like this:

```
  ; save new regrowth time in a temporary LevelSpace let-variable
  ls:let new-regrowth-time 25 + ( abs [ temperature - 55 ] ls:of CC ) / 2

  ; remove decimals, pass it to the wolf sheep predation model and change the time
  ls:ask WSP [
    set grass-regrowth-time round new-regrowth-time
  ]

  ; finally ask both models to go
  ls:ask ls:models [ go ]
```

A general Usecase: Tidying up "Dead" Child Models

As previously mentioned, it is important to keep track of "living" and "dead" models when you dynamically create and dispose of models. Let us imagine we have some lists of models of different kinds, and we want to make sure that we only keep the models that are alive. After running code that kills child models we can use the `ls:model-exists?` primitive to clean up our list of models like this:

```
to-report remove-dead-models [list-of-models]
  report filter [ [ model-id ] -> ls:model-exists model-id] list-of-models
end
```

We then reassign each list of models with this, e.g.

```
set a-list-of-models remove-dead-models a-list-of-models
set another-list-of-models remove-dead-models a-list-of-models
```

,## Citing LevelSpace in Research

If you use LevelSpace in research, we ask that you cite us,

Hjorth, A. Head, B. & Wilensky, U. (2015). “LevelSpace NetLogo extension”. http://ccl.northwestern.edu/rp/levelspace/index.shtml Evanston, IL: Center for Connected Learning and Computer Based Modeling, Northwestern University.

Primitives

Looking for the primitive reference for the LevelSpace extension? You can find the full reference here.

Commanding and Reporting

`ls:ask` · `ls:of` · `ls:report` · `ls:with` · `ls:let` · `ls:assign`

Logic and Control

`ls:models` · `ls:show` · `ls:show-all` · `ls:hide` · `ls:hide-all` · `ls:path-of` · `ls:name-of` · `ls:model-exists?`

Opening and Closing Models

`ls:create-models` · `ls:create-interactive-models` · `ls:close` · `ls:reset`

All Primitives

ls:create-models

`ls:create-models` ***`netlogo-number path`***
`ls:create-models` ***`netlogo-number path command`***

Create the specified number of instances of the given model. The path can be absolute, or relative to the main model. Compared with `ls:create-interactive-models`, this primitive creates lightweight models that are hidden by default. You should use this primitive if you plan on having many instances of the given model. The models may be shown using `ls:show`; when visible, they will have a view and command center, but no other widgets, e.g. plots or monitors.

If given a command, LevelSpace will call the command after loading each instance of the model with the `model-id` as the argument. This allows you to easily store model ids in a variable or list when loading models, or do other initialization. For example, to store a model id in a variable, you can do:

```
let model-id 0
(ls:create-models "My-Model.nlogox" [ [id] -> set model-id id ])
```

Child model RNGs are seeded from the parent models RNG when they are created. Thus, if you seed the parent’s model RNG before child model before child models are created, the simulation as a whole will be reproducible. Use

the `ls:random-seed` primitive to seed the model system's RNGs after child models have been created.

ls:create-interactive-models

`ls:create-interactive-models` *`netlogo-number path`*
`ls:create-interactive-models` *`netlogo-number path command`*

Like `ls:create-models`, creates the specified number of instances of the given model. Unlike `ls:create-models`, `ls:create-interactive-models` creates models that are visible by default, and have all widgets. You should use this primitive if you plan on having only a handful of instances of the given model, and would like to be able to interact with the instances through their interfaces during runtime.

Child model RNGs are seeded from the parent models RNG when they are created. Thus, if you seed the parent's model RNG before child model before child models are created, the simulation as a whole will be reproducible. Use the `ls:random-seed` primitive to seed the model system's RNGs after child models have been created.

ls:close

`ls:close` *`model-or-list-of-models`*

Close the model or models with the given `model-id`.

ls:reset

`ls:reset`

Close down all child models (and, recursively, their child models). You'll often want to call this in your setup procedure.

Note that `clear-all` does *not* close LevelSpace models.

ls:ask

`ls:ask` *`model-or-list-of-models command argument`*

Ask the given child model or list of child models to run the given command. This is the primary of doing things with child models. For example:

```
ls:ask model-id [ create-turtles 5 ]
```

You can also ask a list of models to all do the same thing:

```
ls:ask ls:models [ create-turtles 5 ]
```

You may supply the command with arguments, just like you would with anonymous commands:

```
let turtle-id 0
let speed 5
(ls:ask model-id [ [t s] -> ask turtle t [ fd s ] ] turtle-id speed)
```

Note that the commands cannot access variables in the parent model directly. You must either pass information in through arguments or using `ls:let`.

ls:of

reporter `ls:of` *model-or-list-of-models*

Run the given reporter in the given model and report the result.

`ls:of` is designed to work like NetLogo's inbuilt `of`: If you send `ls:of` a `model-id`, it will report the value of the reporter from that model. If you send it a list of model-ids, it will report a list of values of the reporter string from all models. You cannot pass arguments to `ls:of`, but you can use `ls:let`.

```
[ count turtles ] ls:of model-id
```

ls:report

`ls:report` *model-or-list-of-models reporter argument*

Run the given reporter in the given model and report the result. This form exists to allow you to pass arguments to the reporter.

```
let turtle-id 0
(ls:report model-id [ [a-turtle] -> [ color ] of turtle a-turtle ] turtle-id)
```

ls:with

list-of-models `ls:with` *reporter*

Reports a new list of models containing only those models that report `true` when they run the reporter block.

```
ls:models ls:with [ count turtles > 100 ]
```

ls:let

`ls:let` *variable-name value*

Creates a variable containing the given data that can be accessed by the child models.

```
ask turtles [
  ls:let my-color color
  ls:ask my-model [
    ask turtles [ set color my-color ]
  ]
]
```

`ls:let` works quite similar to `let` in that the variable is only locally accessible:

```
ask turtles [
  ls:let my-color color
]
;; my-color is innaccessible here
```

`ls:let` is very similar to `let`, except in a few cases.

- `ls:let` will overwrite previous values in the variable

If you do

```
ls:let my-var 5
ls:let my-var 6
```

`my-var` will be set equal to `6`. There is no `ls:set`.

- `ls:let` supports variable shadowing

If you do

```
ls:let my-var 5
ask turtles [
  ls:let my-var 6
  ls:ask child-model [ show my-var ]
]
ls:ask child-model [ show my-var ]
```

`child-model` will show `6` and then `5`. This is known as variable shadowing.

- The parent model cannot directly read the value of an ls variable

For example, this does *not* work.

```
ls:let my-var 5
show my-var
```

This is intentional. ls variables are meant to be used for sharing data with child models. The parent model already has access to the data.

Furthermore, changing the value of an ls let variable in a child model will not affect it in any other model. For example:

```
ls:let my-var 0
ls:ask ls:models [
  set my-var my-var + 1
  show my-var
]
```

All models will print `1`.

ls:assign

`ls:assign` *`model-or-list-of-models global-variable value`*

Sets the given global variable in child model to given value. For instance

```
ls:assign ls:models glob1 count turtles
```

sets the global variable `glob1` in all models to the parent's model `count turtles`.

ls:models

`ls:models`

Report a list of model-ids for all existing models.

ls:show

`ls:show` *`model-or-list-of-models`*

Makes all of the given models visible.

ls:show-all

`ls:show-all` *`model-or-list-of-models`*

Makes all of the given models *and their descendents* visible.

ls:hide

`ls:hide` *`model-or-list-of-models`*

Hide all of the given models. Hiding models is a good way of making your simulation run faster.

ls:hide-all

`ls:hide-all` *`model-or-list-of-models`*

Hide all of the given models *and their descendents*. Hiding models is a good way of making your simulation run faster.

ls:path-of

`ls:path-of` *`model-or-list-of-models`*

Report the full path, including the file name of the model. If a list of models is given, a list of paths is reported.

ls:name-of

`ls:name-of` *`model-or-list-of-models`*

Reports the name of the model file. This is the name of the window in which the model appears when visible. If a list of models is given, a list of names is reported.

ls:model-exists?

`ls:model-exists?` *`model-or-list-of-models`*

Report a boolean value for whether there is a model with that model-id. This is often useful when you are dynamically generating models, and want to ensure that you are not asking models that no longer exist to do stuff.

ls:random-seed

`ls:random-seed` *`seed`*

Behaves exactly like NetLogo's built-in primitive `random-seed`, except that child models have their RNGs seeded based on the given seed as well (as well their child models, and their child models' child models, and so forth). This primitive should almost always be used instead of NetLogo's built-in one for seeding RNG when using LevelSpace.

Arduino

Using

For a first use without compiling code, do the following:

1. Acquire the NetLogo software. The Arduino extension comes pre-installed with NetLogo 5.2.1 and later.
2. Acquire an Arduino board and install the arduino IDE
3. Use the Arduino IDE to edit the Sketch (if desired) and send to the board. (See elaborate comments in the sketch for recommendations about what to comment out/leave in depending on your setup & circuit on the board.)
4. Once the Arduino has the sketch loaded on it, it will run that sketch whenever it is powered on.
5. Open the test "Arduino Example" model in the NetLogo Models library (it's in the "IABM Textbook" > "Chapter 8" folder)
6. Connect the Arduino to a USB port on the computer if it is not still connected from step 3.
7. Press OPEN to choose the port to communicate with and establish the connection.
8. Use the buttons to send byte commands; use the interface to inspect variable value(s) that your sketch is sending.
9. Note that by typing `arduino:primitives` you can get a list of the available commands in the extension.

Notes

A NetLogo model using this extension must work in conjunction with an Arduino Sketch. These two endpoints communicate by way of an application protocol that they define. For example, if the NetLogo model sends a byte '1' over the wire this may mean something to the Arduino Sketch, which will respond accordingly. The Arduino Sketch for its own part may send name-value pairs over the serial port, which then can be looked up asynchronously by the NetLogo model.

The modeler is free to build as simple or as complex an application protocol on top of this raw communication mechanism.

The asynchronous nature of the board-to-computer communications has one notable limitation. If you choose to try to simulate a synchronous, BLOCKING READ communications pattern, (e.g., by sending a byte-based signal to the board, which triggers a response in a known name-value pair), then you are likely to be 'off by one' response. That is, if you do the following in NetLogo code:

```
arduino:write-byte b
show arduino:get "varname"
```

You are likely to get the value of `varname` from the PRIOR command represented by writing the byte `b`. This is because the second line of NetLogo code will execute while the Arduino is off generating a new value for `varname`.

There are ways of getting around this (simulating a blocking interface by polling on a value to indicate fresh "news" on `varname`). But this extension works best in settings where the Arduino Sketch is "chatty" and the NetLogo model samples this stream when it needs data.

Compatibility

This code has been tested on Windows 7 and 10 with 32-bit NetLogo and on Mac OS X. You are likely to encounter issues when running this with 64-bit NetLogo in Windows 8 or Windows 10, so if you have Windows 8 or 10, please

download the *32-Bit* version of NetLogo if you plan on using the Arduino extension. We strive for cross-platform compatibility across Mac, Win, and Linux. So if you have troubles, please let us know.

Questions

If you run into problems or have questions about the extension, please email ccl-feedback or cbrady@inquirelearning.com.

Primitives

Looking for the primitive reference for the Arduino extension? You can find the full reference here.

`arduino:primitives` · `arduino:ports` · `arduino:open` · `arduino:close` · `arduino:get` · `arduino:write-string` · `arduino:write-int` · `arduino:write-byte` · `arduino:is-open?` · `arduino:debug-to-arduino` · `arduino:debug-from-arduino`

All Primitives

arduino:primitives

`arduino:primitives`

Reports a list of primitives available in the extension, with basic hints about their syntax.

arduino:ports

`arduino:ports`

Reports a list of port names

arduino:open

`arduino:open` *`port-name`*

Opens the port named *port-name*.

arduino:close

`arduino:close`

Closes the currently open port.

arduino:get

`arduino:get` *`var-name`*

Reads and reports the value associated with *var-name* on the Arduino board. If there is no value associated with *var-name*, returns false. Note: *var-name* is case insensitive.

arduino:write-string

`arduino:write-string` *`string-message`*

Writes a string message to the currently open port.

arduino:write-int

`arduino:write-int` *`int-message`*

Writes a integer message to the currently open port.

arduino:write-byte

`arduino:write-byte` *`byte-message`*

Writes a byte message to the currently open port.

arduino:is-open?

`arduino:is-open?`

Reports a boolean value (`true` or `false`) indicating if a port is open.

arduino:debug-to-arduino

`arduino:debug-to-arduino`

Reports a list of the last messages sent from NetLogo to the Arduino, up to a maximum of 5 messages. Each entry in this list is a string beginning with “s:” if the message sent was a string, “i:” if the message sent was an int, and “b:” if the message sent was a byte.

arduino:debug-from-arduino

`arduino:debug-from-arduino`

Reports a list of lists containing any errant messages sent from NetLogo to the Arduino, up to a maximum of 10 errant messages. Each sublist contains the raw message as its first element and a message describing the error as the second element.

Palette

Using the Palette Extension

The NetLogo palette extension offers the user more control over their colors. The colors go beyond NetLogo colors, including ColorBrewer color schemes and arbitrary RGB colors. Additionally, users can control specific components of their color such as alpha, hue and red.

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:hue-of
palette:set-saturation

palette:scale-gradient
palette:scale-scheme
```

Background

Review of color representation in NetLogo

Colors can be represented as NetLogo colors or RGB or RGBA colors

- A NetLogo color (NLC or NL color) is a number in the range 0 to 140, with the exception of 140 itself, other values are 'wrapped.'
  - NetLogo colors are shades of a base color which has units digit 5, and is in the middle of the shade range.
- An RGB color is a list of 3 numbers.
  - The numbers represent red, green and blue and are in the range 0 to 255.
- An RGBA color is a list of 4 numbers.
  - The fourth number is called alpha and is in the range 0 to 255.
  - Alpha represents transparency with 0 fully transparent, and 255 fully opaque.
- RGB/A will be used to denote a color that is either RGB or RGBA.

Transparency

The degree to which one can see through a color can also be specified by referring to *transparency*. Transparency is in the range 0 to 100, with 0 fully opaque and 100 fully transparent. Transparency is not stored as part of a color, but is mathematically converted to and from alpha.

HSB Color Specification

Colors can also be specified (but not stored as color values) by reference to HSB - hue, saturation and brightness.

- Hue ranges 0 to 360 and forms a cycle.
- Saturation ranges 0 to 100.
- Brightness ranges 0 to 100.
- For more information, search the web for 'hsb color system.'

More control over the color

Palette primitives allow the user to:

- Report a specific component of a color
  - e.g. palette:hue-of
- Report the result of changing a specific component of a color
  - e.g. palette:with-hue
- Report a specific component of an agent's color
  - e.g. palette:hue
- Change a specific component of an agent's color
  - e.g. palette:set-hue

The user can also create color gradients and access ColorBrewer color schemes.

Varying an Agent's Transparency or Color

You can make a turtle more transparent by decreasing alpha, or increasing transparency. A similar approach can be used to adjust color components such as brightness. Here are two examples:

```
ask turtles [ palette:set-transparency palette:transparency  + 10 ]
ask turtles [ palette:set-alpha .9 * palette:alpha ]
```

Because an error will result if the value passed to a set command is not in the correct range, a little additional code is sometimes needed.

```
ask turtles [ palette:set-transparency min list  100 (palette:transparency + 10) ]
ask turtles [ palette:set-alpha max list 0 (palette:alpha - 25.5) ]
```

How do I choose a color scheme?

ColorBrewer has three kinds of color schemes: Sequential, Divergent and Qualitative. Although choosing a color palette is a design problem that does not have a single solution we offer the following guidelines for choosing colors for a NetLogo model:

Decide first whether to use a Sequential, Divergent or Qualitative color scheme

- Sequential color schemes are best for models of continuous natural phenomena such as as heat diffusion in physics or fire in earth sciences.
- Divergent color schemes are useful for highlighting a middle value in a model. Therefore they could be used with the heat diffusion model if the goal were to highlight regions with the middle temperature.
- Qualitative colors are best for choosing colors in models where color denotes category and not value.

Additional color considerations

- For agents that cover large areas avoid strong colors and try to use pastel colors. However, for a few small isolated agents try to use strong colors such as such an 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. For example, use different hues of pastel for patches and accent for turtles.

You can learn more about the use of these color collections in the original ColorBrewer paper (Harrower, Brewer 2003), which focuses on their application to maps. See (Kornhauser, Wilensky, and Rand 1999) for design guidelines for visualization of Agent Based Models.

Should I use a continuous color gradient or just a discrete color set?

The answer depends on the focus of the user experience.

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 and focusing on the big picture. Thus, discrete colors can be a better choice for a written document for which 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 are a few examples of using the palette primitives in the Code Examples section of the models library:

- Palette Example
- Color Bubbles
- Color Painting
- Color Reveal

And one Sample Model that uses the extension:

- Heat Diffusion - Alternative Gradient

Further Reading

- Be sure to check the ColorBrewer web page.
- To get a deeper understanding of how to use the color schemes read the ColorBrewer paper (Harrower, Brewer 2003).

Primitives

Looking for the primitive reference for the Palette extension? You can find the full reference here.

`palette:alpha-of` · `palette:with-alpha` · `palette:alpha` · `palette:set-alpha` · `palette:transparency-of` · `palette:with-transparency` · `palette:transparency` · `palette:set-transparency` · `palette:hue-of` ·

`palette:with-hue` · `palette:hue` · `palette:set-hue` · `palette:saturation-of` · `palette:with-saturation` · `palette:saturation` · `palette:set-saturation` · `palette:brightness-of` · `palette:with-brightness` · `palette:brightness` · `palette:set-brightness` · `palette:R-of` · `palette:with-R` · `palette:R` · `palette:set-R` · `palette:G-of` · `palette:with-G` · `palette:G` · `palette:set-G` · `palette:B-of` · `palette:with-B` · `palette:B` · `palette:set-B` · `palette:scale-gradient` · `palette:scale-gradient-hsb` · `palette:scheme-colors` · `palette:scale-scheme` · `palette:scheme-dialog`

All Primitives

palette:alpha-of

`palette:alpha-of` *`color`*

Reports a value in the range 0 to 255 representing the alpha of the given NL, RGB or RGBA *color*. The alpha value of an NL or RGB color is 255.

Example:

```
show palette:alpha-of red ;; 255
show palette:alpha-of [3 14 159 26] ;; 26
```

palette:with-alpha

`color`* `palette:with-alpha` *`number`

Reports an RGBA color with alpha component equal to *number* and the RGB content of *color*. An error message results if alpha is not in the range from 0 to 255.

Example:

```
show [12 34 55 6] palette:with-alpha 99 ;; [12 34 55 99]
```

palette:alpha

`palette:alpha`

Get the alpha value of the color of the agent. The alpha value of an NL or RGB color is 255.

Example:

```
show [palette:alpha] of turtles
```

palette:set-alpha

`palette:set-alpha` *`number`*

Set the color of the agent to the RGBA color with alpha value *number* and the RGB content of the color of the turtle or link (or patch in NetLogo3D). An error message results if *number* is not in the range 0 to 255.

Example:

```
ask turtles [palette:set-alpha 100]
```

palette:transparency-of

`palette:transparency-of` *`color`*

Reports a value in the range 0 to 100 representing the percent transparency of the given NL, RGB or RGBA *color*. The percent transparency of an NL or RGB color is 0.

Example:

```
show palette:transparency-of [100 100 100 100] ;; 60.7843137254902
show palette:transparency-of red ;; 0
```

palette:with-transparency

`color`* `palette:with-transparency` *`number`

Reports an RGBA color with an alpha value equivalent to the transparency *number* and the RGB content of *color*. An error message results if *number* is not in the range 0 to 100.

Example:

```
show red palette:with-transparency 100 ;; [215 50 41 0]
```

palette:transparency

`palette:transparency`

Reports the transparency equivalent of the agent's alpha.

Example:

```
show [palette:transparency] of turtles
```

palette:set-transparency

`palette:set-transparency` *`number`*

Set the color of the agent to the RGBA color with alpha value equivalent to the transparency *number* and the RGB content of the color of the turtle or link. An error message results if *number* is not in the range 0 to 100.

Example:

```
ask turtles [palette:set-transparency 30]
```

palette:hue-of

`palette:hue-of` *`color`*

Reports a value in the range 0 to 360 representing the hue component in HSB color space of the given NL, RGB or RGBA *color*.

Example:

```
show palette:hue-of red ;; 3.103
```

palette:with-hue

color palette:with-hue _number_

Reports an RGBA color with hue component in HSB color space equal to *number* and the other HSB components of the input *color* unchanged. An error message results if *number* is not in the range 0 to 360.

Example:

```
show red palette:with-hue 100 ;; [99 215 41]
```

palette:hue

palette:hue

Reports a number in the range 0 to 360 that represents the hue of an agent's color or pcolor.

Example:

```
crt 1 [set color red]
[palette:hue] of turtle 0 ;; 3.103
```

palette:set-hue

palette:set-hue _number_

Changes an agent's hue value to *number*, leaving the other HSB components unchanged. An error message results if *number* is not in the range from 0 to 360.

Example:

```
ask turtles [palette:set-hue 30]
```

palette:saturation-of

palette:saturation-of _color_

Reports a value in the range 0 to 100 representing the saturation component in HSB color space of the given NL, RGB or RGBA *color*.

Example:

```
show palette:saturation-of red ;; 80.93
```

palette:with-saturation

color palette:with-saturation _number_

Reports an RGBA color with saturation component in HSB color space equal to *number* and the other HSB components of the input *color* unchanged. An error message results if *number* is not in the range 0 to 100.

Example:

```
show red palette:with-saturation 50 ;; [215 113 108]
```

palette:saturation

`palette:saturation`

Reports a number in the range 0 to 100 that represents the saturation of an agent's color or pcolor.

Example:

```
crt 1 [set color red]
[palette:saturation] of turtle 0 ;; 80.93
```

palette:set-saturation

`palette:set-saturation` ***`number`***

Changes an agent's saturation leaving the other HSB components unchanged. An error message results if *number* is not in the range from 0 to 100.

Example:

```
ask turtles [palette:set-saturation 30]
```

palette:brightness-of

`palette:brightness-of` ***`color`***

Reports a value in the range 0 to 100 representing the brightness component in HSB color space of the given NL, RGB or RGBA *color*.

Example:

```
show palette:brightness-of red ;; 84.314
```

palette:with-brightness

`color` **`palette:with-brightness`** ***`number`***

Reports an RGBA color with brightness component in HSB color space equal to *number* and the other HSB components of the input *color* unchanged. An error message results if *number* is not in the range 0 to 100.

Example:

```
show red palette:with-brightness 50 ;; [128 30 24]
```

palette:brightness

`palette:brightness`

Reports a number in the range 0 to 100 that represents the brightness of an agent's color or pcolor.

Example:

```
crt 1 [set color red]
[palette:brightness] of turtle 0 ;; 84.314
```

palette:set-brightness

palette:set-brightness *number*

Changes an agent's brightness leaving the other HSB components unchanged. An error message results if *number* is not in the range from 0 to 100.

Example:

```
ask turtles [palette:set-brightness 30]
```

palette:R-of

palette:R-of *color*

Reports a value in the range 0 to 255 representing the red component in the RGB color space of the given NL, RGB or RGBA *color*.

Example:

```
show palette:R-of red ;; 215
```

palette:with-R

color* palette:with-R *number

Reports an RGBA color with red component in RGB color space equal to *number* and the other RGB components of the input *color* unchanged. An error message results if *number* is not in the range 0 to 255.

Example:

```
show red palette:with-R 50 ;; [50 50 41]
```

palette:R

palette:R

Reports a number in the range 0 to 255 that represents the red of an agent's color or pcolor.

Example:

```
crt 1 [set color red]
[palette:R] of turtle 0 ;; 215
```

palette:set-R

palette:set-R *number*

Changes an agent's red component of its RGB/A color leaving the other RGB components unchanged. An error message results if *number* is not in the range from 0 to 255.

Example:

```
ask turtles [palette:set-R 30]
```

palette:G-of

`palette:G-of` *`color`*

Reports a value in the range 0 to 255 representing the green component in the RGB color space of the given NL, RGB or RGBA *color*.

Example: `show palette:G-of red ;; 50`

palette:with-G

`color`* `palette:with-G` *`number`

Reports an RGBA color with green component in RGB color space equal to *number* and the other RGB components of the input *color* unchanged. An error message results if *number* is not in the range 0 to 255.

Example: `show red palette:with-G 56 ;; [215 56 41]`

palette:G

`palette:G`

Reports a number in the range 0 to 255 that represents the green of an agent's color or pcolor.

Example:

```
crt 1 [set color red]
[palette:G] of turtle 0 ;; 50
```

palette:set-G

`palette:set-G` *`number`*

Changes an agent's green component of its RGB/A color leaving the other RGB components unchanged. An error message results if *number* is not in the range from 0 to 255.

Example:

```
ask turtles [palette:set-G 30]
```

palette:B-of

`palette:B-of` *`color`*

Reports a value in the range 0 to 255 representing the blue component in the RGB color space of the given NL, RGB or RGBA *color*.

Example:

```
show palette:B-of red ;; 41
```

palette:with-B

`color`* `palette:with-B` *`number`

Reports an RGBA color with blue component in RGB color space equal to *number* and the other RGB components of the input *color* unchanged. An error message results if *number* is not in the range 0 to 255.

Example:

```
show red palette:with-B 56 ;; [215 50 56]
```

palette:B

palette:B

Reports a number in the range 0 to 255 that represents the blue of an agent's color or pcolor.

Example:

```
crt 1 [set color red]
[palette:B] of turtle 0 ;; 41
```

palette:set-B

palette:set-B *number*

Changes an agent's blue component of its RGB/A color leaving the other RGB components unchanged. An error message results if *number* is not in the range from 0 to 255.

Example:

```
ask turtles [palette:set-B 30]
```

palette:scale-gradient

palette:scale-gradient *rgb-color-list number range1 range2*

Reports an RGB color proportional to *number* using a gradient generated with *rgb-color-list*. The *rgb-color-list* is a list containing NL or RGB colors: e.g. NL1 r1 g1 b1 r2 g2 b2 r3 g3 b3 NL2 ....

When *range1* is less than or equal to *range2*, the color will be directly mapped to gradient colors. When *range2* is less than *range1*, the color gradient is inverted.

Let *min-range* be the minimum of *range1* and *range2*. If *number* is less than or equal to *min-range*, then the result is the same as if *number* was equal to *min-range*.

Let *max-range* be the maximum of *range1* and *range2*. If *number* is greater than *max-range*, then the result is the same as if *number* was equal to *max-range*.

Example:

```
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-gradient-hsb

palette:scale-gradient-hsb *rgb-color-list number range1 range2*

Reports an RGB color equivalent to a color in HSB space proportional to *number* using a gradient generated with *hsb-list*. The *hsb-list* is a list containing three-element lists of HSB colors: [[h1 s1 b1] h2 s2 b2 h3 s3 b3 ...].

When *range1* is less than or equal to *range2*, the HSB color will be directly mapped to gradient HSB colors. When range2 is less than range1, the color gradient is inverted.

Let *min-range* be the minimum of *range1* and *range2*. If *number* is less than or equal to *min-range*, then the result is the same as if *number* was equal to *min-range*.

Let *max-range* be the maximum of *range1* and *range2*. If *number* is greater than *max-range*, then the result is the same as if *number* was equal to *max-range*.

Example:

```
ask patches
[
ask patches [set pcolor palette:scale-gradient-hsb [[200 50 50] [100 60 70]] pxcor min-pxcor
max-pxcor]
]

;; colors each patch with a color proportional to the gradient
```

palette:scheme-colors

`palette:scheme-colors` *`scheme-type scheme-color number-of-classes`*

Reports a list of RGB colors using ColorBrewer scheme of type *scheme-type* with name *scheme-color* containing *number-of-classes* colors. Scheme types are “Sequential,” “Divergent” and “Qualitative.” The choice of scheme names depends on the scheme type and includes “Reds”, “Spectral” and “Set1”. The minimum number of colors is 3 and a maximum is between 8 and 12 depending on the color scheme. For more information go to http://www.colorbrewer.org.

Example:

```
show palette:scheme-colors "Divergent" "Spectral" 3
=> [[252 141 89] [255 255 191] [153 213 148]]


; The scheme-colors 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-scheme

`palette:scale-scheme` *`scheme-type scheme-color number-of-classes number range1 range2`*

Reports an RGB color from the ColorBrewer scheme of type *scheme-type* with name *scheme-color* containing *number-of-classes* colors. The color is chosen using the proportionality of *number* in the range specified by *range1* and *range2*.

When *range1* is less than or equal to *range2*, the colors are ordered as in the ColorBrewer scheme. When *range2* is less than *range1*, the color order is reversed.

Let *min-range* be the minimum of *range1* and *range2*. If *number* is less or equal to than *min-range*, then the result is the same as if *number* was equal to *min-range*. This will be the first color in the color list if *range1* is less than *range2*.

Let *max-range* be the maximum of *range1* and *range2*. If *number* is greater than *max-range*, then the result is the same as if *number* was equal to *max-range*. This will be the last color in the color list if *range1* is less than *range2*.

Example:

```
ask patches
[
  set pcolor palette:scale-scheme "Divergent" "Spectral" 8 pxcor min-pxcor max-pxcor
]

;; colors each patch with a color from the Color Brewer Schemes
```

palette:scheme-dialog

palette:scheme-dialog

Launches a dialog for previewing all the ColorBrewer color schemes.

Use the leftmost pulldown menu to select the scheme type: “Sequential,” “Divergent” or “Qualitative.” Use the next pulldown menu to select a named scheme. Finally select the number of colors. The maximum number depends on the color scheme. Use the *Copy* button to copy the scheme information into the clipboard. This information can be pasted into commands such as *palette:scheme-colors*. The *Close* button can be used to close the dialog.

Example:

```
  palette:scheme-dialog

  ;; In the dialog the user selects scheme type "Divergent," the scheme named "Spectral," and
the number 3, and clicks the *Copy* button.
  ;; In the Command Center the user types "show palette:scheme-colors " and then pastes the
clipboard contents, and hits *Enter*.

  show palette:scheme-colors "Divergent" "Spectral" 3

  => [[252 141 89] [255 255 191] [153 213 148]]
```

References

ColorBrewer 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.

KORNHAUSER, D, WILENSKY, U and RAND, W (1999). Design Guidelines for Agent Based Model Visualization. *Journal of Artificial Societies and Social Simulation* 12(2)1.

TUFTE, E (1983) The Visual Display of Quantitative Information, Graphics Press.

WARE, C (2004) Information Visualization, 2nd Ed., Morgan Kaufmann.

Bitmap

Using

The bitmap extension is pre-installed in NetLogo. For instructions on using it, or for more information about NetLogo extensions, see the NetLogo User Manual.

What does the Bitmap Extension do?

The Bitmap Extension allows you to manipulate and import images into the drawing and patches. It offers features not provided by the NetLogo core primitives, such as: scaling, manipulation of different color channels, and width and height reporters.

Getting started

To import and manipulate images you will need to include the bitmap extension in your NetLogo model.

```
extensions[ bitmap ]
```

The image file formats supported are determined by your Java virtual machine's imageio library. Typically this is PNG, JPG, GIF, and BMP. PNG is a good, standard choice that is likely to work everywhere.

If the image format supports transparency (alpha), that information will be imported as well.

Primitives

Looking for the primitive reference for the Bitmap extension? You can find the full reference here.

`bitmap:average-color` · `bitmap:channel` · `bitmap:copy-to-drawing` · `bitmap:copy-to-pcolors` · `bitmap:difference-rgb` · `bitmap:export` · `bitmap:from-base64` · `bitmap:to-base64` · `bitmap:from-view` · `bitmap:to-grayscale` · `bitmap:height` · `bitmap:import` · `bitmap:scaled` · `bitmap:width`

All Primitives

bitmap:average-color

`bitmap:average-color` ***`image`***

Reports a 3-element list describing the amount of R, G, and B in *image*, by summing across all pixels, and normalizing each component by the number of pixels in the image, so each component ranges from 0 to 255.

bitmap:channel

`bitmap:channel` *`image channel`*

Extracts either the alpha, red, green, or blue channel from an image. The input *channel* should be an integer 0-3 indicating the channel to remove (alpha=0, red=1, green=2, blue=3). The resulting image is a grayscale image representing specified channel.

bitmap:copy-to-drawing

`bitmap:copy-to-drawing` *`image x y`*

Imports the given image into the drawing without scaling the image at the given pixel coordinates.

bitmap:copy-to-pcolors

`bitmap:copy-to-pcolors` *`image boolean`*

Imports the given image into the pcolors, scaled to fit the world. The second input indicates whether the colors should be interpreted as NetLogo colors or left as RGB colors. `false` means RGB colors.

bitmap:difference-rgb

`bitmap:difference-rgb` *`image1 image2`*

Reports an image that is the absolute value of the pixel-wise RGB difference between two images. Note that image1 and image2 MUST be the same width and height as each other, or errors will ensue.

bitmap:export

`bitmap:export` *`image filename`*

Writes *image* to *filename*.

bitmap:from-base64

`bitmap:from-base64` *`base64-string`*

Turns a base64 encoded string into a bitmap image for use by the extension. The Fetch and Export-The extensions would be common sources of these encoded strings.

bitmap:to-base64

`bitmap:to-base64` *`base64-string`*

Turns a bitmap image into a base64 encoded string with a PNG format.

bitmap:from-view

`bitmap:from-view`

Reports an image of the current view.

bitmap:to-grayscale

`bitmap:to-grayscale` ***`image`***

Converts the given image to grayscale.

bitmap:height

`bitmap:height` ***`image`***

Reports the height of given *image*

bitmap:import

`bitmap:import` ***`filename`***

Reports a LogoBitmap containing the image at *filename*.

bitmap:scaled

`bitmap:scaled` ***`image width height`***

Reports a new image that is *image* scaled to the given *width* and *height*

bitmap:width

`bitmap:width` ***`image`***

Reports the width of the given *image*

Sound

Using

The Sound Extension lets NetLogo models make two kinds of sounds: MIDI sounds and playback of pre-recorded sound files.

The Java APIs used are `javax.sound.midi` and `java.applet.AudioClip`.

How to Use

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

Looking for the primitive reference for the Sound extension? You can find the full reference here.

`sound:drums` · `sound:instruments` · `sound:play-drum` · `sound:play-note` · `sound:play-note-later`

All Primitives

sound:drums

`sound:drums`

Reports a list of the names of the 47 drums for use with `sound:play-drum`.

sound:instruments

`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.

Example:

```
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.

Example:

```
;; in one second, play a trumpet at middle C for two seconds
sound:play-note-later 1 "TRUMPET" 60 64 2
```

Drum Names

```
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
47. Open Hi Hat
47. Low Mid Tom
48. Hi Mid Tom
49. Crash Cymbal 1
50. Hi Tom
51. Ride Cymbal 1
52. Chinese Cymbal
53. Ride Bell
54. Tambourine
55. Splash Cymbal
56. Cowbell
57. Crash Cymbal 2
58. Vibraslap
69. Cabasa
70. Maracas
71. Short Whistle
72. Long Whistle
73. Short Guiro
74. Long Guiro
75. Claves
76. Hi Wood Block
77. Low Wood Block
78. Mute Cuica
79. Open Cuica
80. Mute Triangle
81. Open Triangle

Instrument Names

Piano
1. Acoustic Grand Piano
2. Bright Acoustic Piano
3. Electric Grand Piano
4. Honky-tonk Piano
5. Electric Piano 1
6. Electric Piano 2
7. Harpsichord
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

Reed
65. Soprano Sax
66. Alto Sax
67. Tenor Sax
68. Baritone Sax
69. Oboe
70. English Horn
71. Bassoon
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

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

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

Resource

Using

Purpose

The resource extension allows you to retrieve the contents of files that are bundled with a model. Bundled resources, which were added in NetLogo 7, provide a way to use the contents of other files in a model without needing to send the files alongside the model. When you add a resource to a model, the contents of the source file are stored in the model file.

For details on bundled resources, see Resource Manager.

Getting Started

The resource extension comes pre-installed with NetLogo. To use the resource extension in your model, add a line to the top of your Code tab:

```
extensions [resource]
```

If your model already uses other extensions, then it already has an `extensions` line in it, so just add `resource` to the list.

For more information on using NetLogo extensions, see the Extensions Guide

Primitives

Looking for the primitive reference for the Resource extension? You can find the full reference here.

`resource:get` · `resource:list`

All Primitives

resource:get

`resource:get` ***`name`***

Reports the contents of the resource whose name matches the input string. If the resource is plain text, the contents will be reported as such. Otherwise, the contents will be reported as a Base64 string. An error will be thrown if the specified resource does not exist.

resource:list

`resource:list`

Reports a list of names of the bundled resources present in the model.

Simple R

Using

To run R code you must install R and have the `R` executable on your `PATH`. You can download R from their site.

To use this extension, you must first include it at the top of your NetLogo model code in an `extensions` declaration.

```
extensions [
  sr
  ; ... your other extensions
]
```

You must then initialize the R environment with `sr:setup`. This only needs to be done once per session. Any subsequent calls will reset your R environment.

Here's an example to get you started:

```
observer> sr:setup
;; sr:runresult evaluates R statements and returns the result back to NetLogo
observer> show sr:runresult "2 + 2"
observer: 4
;; sr:run runs R code
observer> sr:run "cat('hello world')"
hello world
;; any standard output gets forwarded to the command center output
;; sr:set sets R variables to values from NetLogo
observer> ask patch 0 0 [ set pcolor red ]
observer> sr:set "center_patch_color" [pcolor] of patch 0 0
observer> show sr:runresult "center_patch_color"
observer: 15 ;; the NetLogo representation of the color red
```

See the documentation for each of the particular primitives for details on, for instance, how to multi-line statements and how object type conversions work.

The extension also includes an interactive R console/REPL that is connected to the same R environment as the main window's NetLogo environment. It is useful for executing longer blocks of R code or quickly examining or modifying R values. This console can be opened via the menu bar under the SimpleR heading, or by using the `sr:show-console` command.

Error handling

R errors will be reported in NetLogo as "Extension exceptions". For instance, this code:

```
sr:run "stop('hi')"
```

will result in the NetLogo error "Extension exception: hi" appearing in an error dialog. ,## Citing Simple R in Research

If you use the Simple R extension in research, we ask that you cite us,

Hovet, J. Head, B. & Wilensky, U. (2022). "Simple R NetLogo extension". https://github.com/NetLogo/SimpleR-Extension Evanston, IL: Center for Connected Learning and Computer Based Modeling, Northwestern University.

Primitives

Looking for the primitive reference for the Simple R extension? You can find the full reference here.

`sr:setup` · `sr:run` · `sr:runresult` · `sr:set` · `sr:set-agent` · `sr:set-agent-data-frame` · `sr:set-data-frame` · `sr:set-list` · `sr:set-named-list` · `sr:set-plot-device` · `sr:r-home` · `sr:show-console`

All Primitives

sr:setup

`sr:setup`

Create the R session that this extension will use to execute code. This command *must* be run before running any other R extension primitive. Running this command again will shutdown the current R environment and start a new one.

sr:run

`sr:run` *`r-statement`*
`sr:run` *`r-statement repeatable`*

Runs the given R statements in the current session. To make multi-line R code easier to run, this command will take multiple strings, each of which will be interpreted as a separate line of R code. This requires putting the command in parentheses.

For instance:

```
(sr:run
  "domain <- seq(-3.14, 3.14, 0.01)"
  "range <- sin(domain)"
  "png('my_file.png')"
  "plot(domain, range, "
  "     pch = 20,"
  "     main = 'y = sin(x)',"
  "     xlab = 'x',"
  "     ylab = 'y')"
  "dev.off()"
)
```

`sr:run` will wait for the statements to finish before continuing. If you have long-running R code, NetLogo may freeze for a bit while it runs.

sr:runresult

`sr:runresult` *`r-expression`*

Evaluates the given R expression and reports the result. `rs:runresult` attempts to convert from R data types to NetLogo data types.

Numbers, strings, and booleans convert as you would expect, except for outliers like Infinity and NaN which will be converted into the strings 'Inf' and 'NaN', respectively.

R vectors and R lists will be converted to NetLogo lists. NA values will be converted into the string 'NA'.

R matrices will be flattened into one-dimensional lists using column-major order. If you want to convert a matrix into a list of lists before sending it to NetLogo, use the R `asplit` command. To convert into a list of column lists, use `asplit(<matrix>, 1)`; for a list of row lists, use `asplit(<matrix>, 2)`.

An R DataFrame will be converted into a list of lists, where the first item in each sublist is the name of the column and the second item is a list containing all that row data. For example, the first 6 rows of the `iris` dataset will be converted into NetLogo like so:

```
[
  ["Sepal.Length" [5.1 4.9 4.7 4.6 5 5.4]]
  ["Sepal.Width" [3.5 3 3.2 3.1 3.6 3.9]]
  ["Petal.Length" [1.4 1.4 1.3 1.5 1.4 1.7]]
  ["Petal.Width" [0.2 0.2 0.2 0.2 0.2 0.4]]
  ["Species" ["setosa" "setosa" "setosa" "setosa" "setosa" "setosa"]]
]
```

Other objects will be converted to a string representation if possible and and may throw an error if not.

sr:set

`sr:set` *`variable-name value`*

Sets a variable in the R session with the given name to the given NetLogo value. NetLogo objects will be converted to R objects as expected.

Note that lists in NetLogo are converted into lists in R if the elements are of different types. If all the elements of a NetLogo list are of the identical number, boolean, or string type then the data will be automatically converted into a vector in R.

```
sr:set "x" 42
sr:run "print(x)" ;; prints `[1] 42` to the command center
sr:set "y" [1 2 3]
sr:run "print(typeof(y))" ;; prints `[1] "double"` to the command center
sr:run "print(typeof(list(y)))" ;; prints `[1] "list"` to the command center
sr:run "print(y)" ;; prints `[1] 1 2 3` to the command center
show sr:runresult "y" ;; reports [1 2 3]
```

Agents are converted into lists with named elements for each agent variable.

Agentsets are converted into a list of the above lists. If you want to convert agents to a data frame, see `sr:set-agent-data-frame`. If you want to use `sr:set` and do the conversion manually, try the following:

```
my_data_frame <- as.data.frame(do.call(rbind, <agentset-list-of-lists>))
```

For example:

```
breed [goats goat]
goats-own [energy ]
create-goats 2 [ set color 75 ]
ask goat 0 [ set energy 42 set xcor 5]
ask goat 1 [ set energy -42 set xcor -5]

sr:set "goat" goat 0
sr:run "print(typeof(goat))" ;; prints `[1] "list"` to the command center
sr:run "print(goat)"
;; Should output:
;; $WHO
;; [1] 0
;;
;; $COLOR
;; [1] 75
;; (etc.)

sr:set "goats_list_of_lists" goats
```

```
sr:run "goats_data_frame <- as.data.frame(do.call(rbind, goats_list_of_lists))"
sr:run "print(goats_data_frame)"
;; Should output:
;;   WHO COLOR HEADING XCOR YCOR   SHAPE LABEL LABEL-COLOR BREED HIDDEN? SIZE
;; 1   0    75      82    5    0 default               9.9 GOATS   FALSE    1
;; 2   1    75     200   -5    0 default               9.9 GOATS   FALSE    1
;;   PEN-SIZE PEN-MODE ENERGY
;; 1        1       up     42
;; 2        1       up    -42
;;
```

Agents with variables containing references to agentsets will have those variables converted into the string representation of that agentset.

sr:set-agent

`sr:set-agent` *`r-variable-name agent or agentset agent-variable-name`*
`sr:set-agent` *`r-variable-name agent or agentset agent-variable-name1 agent-variable-name2...`*

Creates a new named list in R with the given variable name. If you want multiple agent variables make sure to surround the command in parenthesis.

```
clear-all
sr:setup
create-turtles 2
ask turtle 0 [ set color red set xcor 5]
ask turtle 1 [ set color blue set xcor -5]
(sr:set-agent "t0" turtle 0 "who" "color" "xcor" "hidden?")
sr:run "print(typeof(t0))"
;; [1] "list"
sr:run "print(t0)"
;; $who
;; [1] 0
;;
;; $color
;; [1] 15
;;
;; $xcor
;; [1] 5
;;
;; $`hidden?`
;; [1] FALSE
```

sr:set-agent-data-frame

`sr:set-agent-data-frame` *`r-variable-name agents agent-variable-name`*
`sr:set-agent-data-frame` *`r-variable-name agents agent-variable-name1 agent-variable-name2...`*

Creates a new data frame in R with the given variable name. The columns will have the names of the NetLogo agent variables used and each row will be one agent's data. If you want multiple agent variables make sure to surround the command in parenthesis.

```
clear-all
sr:setup
create-turtles 2
ask turtle 0 [ set color red set xcor 5]
ask turtle 1 [ set color blue set xcor -5]
(sr:set-agent-data-frame "turtles_data_frame" turtles "who" "color" "xcor" "hidden?")
sr:run "print(typeof(turtles_data_frame))"
```

```
;; [1] "list"
sr:run "print(is.data.frame(turtles_data_frame))"
;; [1] TRUE
sr:run "print(turtles_data_frame)"
;;   who color xcor hidden?
;; 1   0    15    5   FALSE
;; 2   1   105   -5   FALSE
```

sr:set-data-frame

`sr:set-data-frame` *`r-variable-name column-name list or anything`*
`sr:set-data-frame` *`variable-name column-name1 list or anything 1 column-name2 list or anything 2...`*

Creates a new data frame in R with the given variable name. The columns will have the names given. If the value for a column is a list, those will be the values for that column. If the value is a non-list, it will be used as the single item in that column. You can add additional column names and values by surrounding the command in parenthesis.

```
clear-all
sr:setup
let l1 [10 20 30 40]
let l2 [false true false false]
let l3 ["orange" "green" "blue" "purple"]
(sr:set-data-frame "df1" "score" l1 "enabled" l2 "color" l3)
sr:run "print(typeof(df1))"
;; [1] "list"
sr:run "print(is.data.frame(df1))"
;; [1] TRUE
sr:run "print(df1)"
;;   score enabled  color
;; 1    10   FALSE orange
;; 2    20    TRUE  green
;; 3    30   FALSE   blue
;; 4    40   FALSE purple
```

sr:set-list

`sr:set-list` *`r-variable-name anything`*
`sr:set-list` *`r-variable-name anything1 anything2...`*

Creates a new list in R with the given variable name. You can add additional values by surrounding the command in parenthesis.

sr:set-named-list

`sr:set-named-list` *`r-variable-name column-name list or anything`*
`sr:set-named-list` *`r-variable-name column-name1 list or anything 1 column-name2 list or anything 2...`*

Creates a new named list in R with the given variable name. The columns will have the names given. If the value for a column is a list, those will be the values for that column. If the value is a non-list, it will be used as the single item in that column. You can add additional column names and values by surrounding the command in parenthesis.

sr:set-plot-device

`sr:set-plot-device`

Activates the visual plot device for R, popping open a window if one is not already open.

sr:r-home

`sr:r-home`

Outputs the R home directory which is the top-level directory of the R installation being run.

```
observer> sr:setup
observer> show sr:r-home
observer: "/Library/Frameworks/R.framework/Resources"
```

sr:show-console

`sr:show-console`

Opens the R console. This console can be opened via the menu bar under the SimpleR heading.

Transitioning from the old R extension

As of version 2.0 of the Simple R extension, most primitives from the old R extension have a direct equivalent you can switch over to use, with a different name but identical syntax. One change in functionality is that when a named list with a single row and column is returned, in the R extension you'd get simply the value, in the Simple R extension you will get a list with the column name and the value as elements. If there is more than 1 element, you will get a list with the column name and a list of the values.

Using the Simple R code converter

- Open the code tab of your model.
- Change r to sr in the extensions line .
- Click the "Check" button.
- From the "SimpleR Extension" menu choose "Convert code from R extension".
- The primitives will be updated.
- You may need to add `sr:setup` in your setup procedure (or equivalent) in order too start a new R environment.
- You will need to change any code that returns a named list.

R Extension Primitive	Simple R Extension Primitive
`r:put`	`sr:set`
`r:get`	`sr:runresult`
`r:eval`	`sr:run`
`r:__evaldirect`	`sr:run`
`r:putList`	`sr:set-list`
`r:putNamedList`	`sr:set-named-list`

R Extension Primitive	Simple R Extension Primitive
`r:putDataFrame`	`sr:set-data-frame`
`r:putAgent`	`sr:set-agent`
`r:putAgentDf`	`sr:set-agent-data-frame`
`r:setPlotDevice`	`sr:set-plot-device`
`r:interactiveShell`	`sr:show-console`
`r:clear`	No exact equivalent, but `sr:setup` will start a new R environment
`r:clearLocal`	No exact equivalent, but `sr:setup` will start a new R environment
`r:gc`	No equivalent, functionality is no longer needed
`r:stop`	No equivalent, functionality is no longer needed
`r:jri-path`	No equivalent, functionality is no longer needed

Handling a named list

R's statistical functions often return lists. The R extension stripped out names to give you values. With the Simple R extension the user must extract the value

```
sr:run "c <- cor.test(turtles$weight, turtles$height, method = 'spearm', alternative = 'g')"

  ;; original code
  ;; let p sr:get "c$p.value"
  ;; let rho sr:get "c$estimate"           # sample value  0.673271955983285
  let estimate  sr:runresult "c$estimate"  # sample value  [[rho -0.121575984990619]]
  let p sr:runresult "c$p.value"
  ;; select result from list
  let rho item 1 (item 0 estimate)
```

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.)

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. 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 netlogo@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 Forum and 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?

So far, the NetLogo user interface has been localized in English, Spanish, Chinese, Portuguese, Russian, and Japanese. All six languages are included in the standard download.

There is a BehaviorSpace Guide en Español.

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 Forum](https://forum.netlogo.org/) and 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:** The installer is an MSI which can be run using one of the following commands (these may require admin privileges):

```
    msiexec.exe /i C:\Path\to\NetLogo.msi ALLUSERS=1 ADDLOCAL=Shortcuts   :: To install with desktop
shortcuts
```

```
    msiexec.exe /i C:\Path\to\NetLogo.msi ALLUSERS=1 ADDLOCAL=BaseInstall :: To install without
desktop shortcuts
```

We also maintain an advanced installation wiki page which may have additional tips and tricks. We encourage users who have problems with unattended installation or ideas on how it could be improved to email netlogo@northwestern.edu.

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 7.1.0-alpha1.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?

Since Mac OS X Lion, Apple have encouraged Mac application developers to bundle Java. NetLogo bundles Java because it allows us to deliver a consistent, convenient experience to our users. Bundling Java allows us to test for compatibility once and avoid any bugs caused by version mismatches or Java configuration incompatibilities.

If users are interested in using Java on their system instead of the version of Java bundled with NetLogo, they can configure NetLogo to use a different Java runtime. Please note that **this change is done at your own risk**. We are unable to offer support for problems caused by running NetLogo with an alternate Java Runtime. To change the Java runtime used by NetLogo, open the NetLogo.cfg file and modify the `app.runtime` property to the path of your preferred Java Runtime.

How come NetLogo won't start up on my Linux machine?

We bundle Bellsoft's OpenJDK-based 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 Where are the NetLogo Configuration Files? for more information).

In theory, any Java 11 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 https://help.ubuntu.com/community/Java.

When I try to install NetLogo on Windows, I see "Windows protected your PC"

Windows attempts to protect users from downloading malicious software by maintaining a list of "good" and "malicious" software. The first users to install any NetLogo release will see this warning. Later users may or may not see this warning. Before going any further, ensure you are protected. The CCL can only vouch for NetLogo downloads hosted on the CCL Website. NetLogo can be freely downloaded from the official NetLogo download page. If you obtain NetLogo from anywhere else, you install it at your own risk!

You can continue past the Windows prompt by taking the following steps:

- In the "Windows protected your PC" prompt, click "More Info", the prompt will change
- In the changed prompt, click "Run Anyway" and continue with installation as normal

Note in order to see the "More Info" or "Run Anyway" options, you will need to run the installer as an administrator. If you do not see those options, right-click and choose "Run as administrator" and/or change your user account to an administrator account before installing.

We at the CCL are working on solving this problem and hope to free our users of the added installation difficulty soon!

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.

NetLogo won't start on Mac OS Sierra (or later)

Some users have reported problems opening NetLogo 6 on Mac OS Sierra and later (We don't yet know if this is an issue with NetLogo 7). We've been unable to determine a root cause for this problem, but we're continuing to investigate. We're looking for assistance from you if you run into this problem. It would be extremely helpful if you could run the following command in the "Terminal" application and send the output in an email to netlogo@northwestern.edu.

```
xattr -pl com.apple.quarantine /Applications/NetLogo\ 7.1.0-alpha1/NetLogo\ 7.1.0-alpha1.app
```

running it should give you a single line of output (something like `com.apple.quarantine: 01e1;58ac6af2;Firefox;F2E0B1E2-D203-4B05-8DF9-ABA58B52EFEA`, but yours will have different numbers, letters, and words). Please copy and paste this string into the email you send us.

There is also a partial workaround! Running the following command in the Terminal will enable users to run NetLogo without turning Gatekeeper completely off. The command is:

```
sudo xattr -dr com.apple.quarantine /Applications/NetLogo\ 7.1.0-alpha1/NetLogo\ 7.1.0-alpha1.app
```

Note that if you plan to run NetLogo 3D or HubNet Client you may also need to re-run that command once for each of those applications. If the workaround was not effective for you, please send us an email and let us know.

NetLogo won't start on Windows or crashes suddenly on Mac OS Sierra

Some users on Mac OS with discrete graphics cards experience sudden crashes of NetLogo when switching between integrated and discrete graphics. It's possible to prevent these crashes by disabling automatic graphics switching in System Preferences.

Some Windows users may also be unable to open NetLogo due to graphics card settings or drivers. If you are unable to open NetLogo on Windows, you might be able to fix this by updating or reinstalling your graphics driver and/or disabling any graphics card utilities. Because NetLogo might be unable to start for a number of reasons, we encourage you to contact us if these steps aren't effective in resolving the problem.

How can I speed up the launching of NetLogo on a remote server that is behind a proxy?

In a few cases involving a remote server that is behind a proxy, NetLogo users have observed startup times of several minutes. So far all of these cases have been on Linux systems.

This happens when NetLogo makes a request over the network to update information about extensions and their latest versions and there is a problem getting the response. We offer the following workaround.

Turn off the request to update the extensions information by setting the java preference netlogo.libraries.disabled=true by editing the NetLogo configuration file. Add the following line to the section with java-options (order doesn't matter)

```
java-options=-Dnetlogo.libraries.disabled=true
```

If you want to allow the updates again, you can set the value to false, or remove that preference. (see Where are the NetLogo Configuration Files? for more information)

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 Running from the command line.

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. See Run options: parallel runs.

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 Forum and 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 path to the auto-save file depends on whether or not the NetLogo model has been saved.

For NetLogo models which have been saved, a hidden file with the name "*.filename*.tmp.nlogo" will be created in the same directory as the NetLogo model.

For unsaved NetLogo models, autosave files can be found in your OS-specific java temporary directory. 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. The exact path will depend on your operating system:

- On Mac OS, the temporary directory is `/var/folders/68/<30-character-alphanumeric-sequence>/T/`. The 30-character-alphanumeric-sequence is unique to each machine.
- On Windows the logs can be found in `C:\Users\<user>\AppData\Local\Temp`, where `<user>` is the logged in user.
- On most non-Mac Unix-like systems the temp directory is `/tmp`.

Why is HubNet Discovery Not Working?

HubNet discovery provides the ability for HubNet clients running on the same local network as the HubNet server to automatically display the IP address and current activity of the server. This uses IPv4 multicast over a local area network - it won't identify HubNet servers on the internet or on different networks. Some networks also do not support multicast traffic and it will simply not work on those networks. For HubNet Discovery to work, the server must be broadcasting to the same network that the client is polling for messages. Often, both the client and server are only connected to a single network and there is no need for further configuration.

Since NetLogo 6.0.3, we have tools in place to allow clients and/or servers connected to multiple networks to select the network on which they will broadcast/listen. When running a HubNet server, this is done by selecting the appropriate "broadcast network" via selection dropdown when starting a HubNet server. This preference will be remembered until NetLogo is restarted, so if you end up needing to broadcast on a different network, restart NetLogo. HubNet clients also see a similar dropdown when selecting which activity to join. Unlike HubNet server, clients may change the network on which they listen at any time. HubNet discovery will only work when both client and server are listening/broadcasting on the *same* network, so some trial and error may be required to figure out a working configuration. Generally speaking, the shared network is the one on which the client and the server have the most similar IP address prefixes. For instance, if the server selects to broadcast on a network where it has IP address 192.168.0.101, a client should prefer listening on a network where it has an IP like 192.167.0.203 over a network where it has an IP like 10.5.0.101. Even when the same network is selected, it is possible that that network may not support IPv4 multicast, in which case discovery will *not* work and clients must enter IP addresses manually.

Note that the change made in 6.0.3 simply enable selection of the network on which clients and servers broadcast/listen on. The HubNet server still listens on *all* network connections, so all listed IP addresses given in the server broadcast network selection dropdown are potentially valid for a HubNet client (even a client on a different network) to connect to. Similarly HubNet clients may connect to any HubNet server reachable from their computer.

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 have reports from NetLogo users that they have run models that required up to 50 gigabytes of RAM to use millions of patches and tens of thousands of agents, and they worked as desired with the proper Java settings. Theoretically the only limit is the RAM available on your system, but you might hit some limits that are inherent in the underlying Java runtime or the operating system (either designed-in limits, or bugs).

The NetLogo engine has no fixed limits on model size. By default NetLogo sets Java to use up to half of your available system memory. If your model exceeds that memory limit, you'll get an `OutOfMemoryError` dialog.

If you are using BehaviorSpace, note that doing runs in parallel will multiply your RAM usage accordingly. Similarly, using LevelSpace with many child models increases RAM usage greatly.

If you want to increase the memory limit Java uses, you can do so using the **NetLogo app configuration files**.

Locate the section of the file that looks like the following:

```
[JavaOptions]
# there may be one or more lines, leave them unchanged
java-options=-XX:MaxRAMPercentage=50
# there may be one or more lines, leave them unchanged
```

You can change the percentage number from 50 to whatever you want. Save the file and restart NetLogo for the setting to take effect. If you need to set an exact amount of memory (as opposed to a percentage), you can also comment out the `java-options=-XX:MaxRAMPercentage=50` line with a `#` at the start and add a new line with `java-options=-Xmx####m`. Replace the `####m` with whatever amount of memory you like; using `4096m` would limit NetLogo to 4096 megabytes of memory, or 4 gigabytes.

Where are the NetLogo Configuration Files?

Each platform contains configuration files (with the extension ".cfg") containing JVM settings. There is one configuration file for each sub-application (NetLogo, NetLogo 3D, HubNet Client, etc.). Windows also has a configuration file for the NetLogo_Console app; on macOS and Linux the configuration file for NetLogo_Console is just the NetLogo.cfg file. Although the file location varies by platform, the process for changing it is the same.

Platform specific notes follow:

- **Windows:** The file will typically be in `C:\Program Files\NetLogo 7.1.0-alpha1\app`, unless you are running 32 bit NetLogo on a 64 bit Windows, then it will be in `C:\Program Files (x86)\NetLogo 7.1.0-alpha1\app`.
- **Mac OS X:** The file for NetLogo will be located at: `/Applications/NetLogo 7.1.0-alpha1/NetLogo 7.1.0-alpha1.app/Contents/app/NetLogo 7.1.0-alpha1.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 configuration 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 configuration 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".

- **Linux:** The configuration files will be located in the `lib/app` directory of your NetLogo installation.

By default, Mac builds of NetLogo bundle a 64-bit JVM. 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 Groupand participants in the NetLogo Forum may be able to help with this.

You wil find useful advice in the article Improving Execution Speed of Models Implemented in NetLogo.

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.)

Why is the number value in my monitor widget changing even though nothing is happening in my model?

This is caused by a combination of a few things:

1. NetLogo's use of floating point numbers, which can produce small accuracy issues. See **Floating point accuracy** in the NetLogo programming guide.
2. Agentsets such as `turtles` are always returned in a random order.
3. Monitors re-run their reporter calculation constantly, even when you are not running any model code with a forever button or through the command center.

So the monitor constantly re-runs its `mean [xcor] of turtles` reporter, but the `turtles` agentset gives the turtles in a random order, and so the floating-point inaccuracies for `mean` will accumulate in a slightly different way each time due to the order differences. The end result is you see very slightly different numbers flashing through the monitor widget while nothing is happening.

You see the same problem doing `sum [xcor] of turtles`, `variance [xcor] of turtles`, or `standard-deviation [xcor] of turtles` - anytime you're reducing a bunch of floating point numbers from an agentset into a single value.

You can also see the problem running your reporter code directly in the command center repeatedly, without a monitor widget at all.

The fixes are straightforward:

- Sort your numbers before you calculate: `mean sort [xcor] of turtles`, `sum sort [xcor] of turtles`, `variance sort [xcor] of turtles`. If the numbers are in the same order you'll still have small floating-point inaccuracies, but they'll be the the same every time so you won't see the values change. This is probably the best solution, but it can be slow if you have a really large agentset.
- Change the `Decimal places` setting of your monitors to a number where you don't notice the changing values. Since the differences in results should be small, this is usually possible.

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] [ i ->
    if i = 2 [ stop ]
    print i
  ]
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 literal value"?

If a list contains only literal values, 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

You can find details about using BehaviorSpace at BehaviorSpace Guide and BehaviorSpace Guide en Español

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 Table Output section of the BehaviorSpace Guide.

How do I measure runs every n ticks?

Use `mod` in your experiment's Run Metrics when text box, e.g.:

```
ticks mod 10 = 0
```

to measure the run after every 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.

NetLogo 3D

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: Turtle - Patch - Links - Agentset - Color - Anonymous Procedures - Control/Logic - World Perspective - Input/Output - File - List - String - Math - Plotting - System - HubNet

Special: Variables - Keywords - Constants

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`) · `can-move?` · `clear-turtles` (`ct`) · `create-ordered-turtles` (`cro`) · `create-turtles` (`crt`) · `die` · `distance` · `distancexy` · `downhill` (`downhill4`) · `dx` (`dy`) · `face` · `facexy` · `forward` (`fd`) · `hatch` · `hide-turtle` (`ht`) · `home` · `inspect` · `is-agent?` (`is-agentset?`, `is-turtle?`, `is-turtle-set?`) · `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` · `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?` (`is-patch-set?`) · `myself` · `neighbors` (`neighbors4`) · `nobody` · `no-patches` · `of` · `other` · `patch` · `patch-ahead` · `patch-at` · `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` · `stop-inspecting` · `subject` · `turtles-here`

Link-related

`both-ends` · `clear-links` · `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`) · `die` · `directed-link-breed` · `hide-link` · `in-<breed>-neighbor?` (`in-link-neighbor?`) · `in-<breed>-neighbors` (`in-link-neighbors`) · `in-<breed>-from` (`in-link-from`) · `inspect` · `is-directed-link?` (`is-link?`, `is-link-set?`, `is-undirected-link?`) · `layout-radial` · `layout-spring` · `layout-tutte` · `link` · `link-heading` · `link-length` · `links` · `links-own` · `my-<breeds>` (`my-links`) · `my-in-<breeds>` (`my-in-links`) · `my-out-<breeds>` (`my-out-links`) · `<breed>-neighbors` (`link-neighbors`) · `<breed>-neighbor?` (`link-neighbor?`) · `no-links` · `other-end` · `out-<breed>-neighbor?` (`out-link-neighbor?`) · `out-<breed>-neighbors` (`out-link-neighbors`) · `out-<breed>-to` (`out-link-to`) · `show-link` · `stop-inspecting` · `tie` · `undirected-link-breed` · `untie` · `<breed>-with` (`link-with`)

Agentset

`all?` · `any?` · `ask` · `ask-concurrent` · `at-points` · `count` · `in-cone` · `in-radius` · `is-list?` · `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` · `up-to-n-of` · `who-are-not` · `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` · `startup` · `stop` · `to` · `to-report` · `wait` · `while` · `with-local-randomness` · `without-interruption` · `xor`

Anonymous Procedures

`filter` · `foreach` · `is-agent?` (`is-agentset?`, `is-anonymous-command?`, `is-anonymous-reporter?`, `is-boolean?`, `is-directed-link?`, `is-link?`, `is-link-set?`, `is-list?`, `is-number?`, `is-patch?`, `is-patch-set?`, `is-string?`, `is-turtle?`, `is-turtle-set?`, `is-undirected-link?`) · `map` · `n-values` · `reduce` · `run` (`runresult`) · `sort-by` · `-> (anonymous procedure)`

World

`clear-all` (`ca`) · `clear-drawing` (`cd`) · `clear-globals` · `clear-patches` (`cp`) · `clear-ticks` · `clear-turtles` (`ct`) · `display` · `import-drawing` · `import-pcolors` · `import-pcolors-rgb` · `max-pxcor` (`max-pycor`) · `min-pxcor` (`min-pycor`) · `no-display` · `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-fetch-message` · `hubnet-kick-client` · `hubnet-kick-all-clients` · `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`

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` · `home-directory` · `user-directory` · `user-file` · `user-new-file`

List

`but-first` (`butfirst`, `bf`, `but-last`, `butlast`, `bl`) · `empty?` · `filter` · `first` · `foreach` · `fput` · `histogram` · `insert-item` · `is-list?` · `item` · `last` · `length` · `list` · `lput` · `map` · `max` · `member?` · `min` · `modes` · `n-of` · `n-values` · `of` · `one-of` · `position` · `shuffle` · `sort` · `sort-by` · `sort-on` · `sublist` (`substring`) · `up-to-n-of`

String

`String Operators` (`<`, `>`, `=`, `!=`, `<=`, `>=`) · `empty?` · `first` · `insert-item` · `is-string?` · `item` · `last` · `length` · `member?` · `position` · `read-from-string` · `remove` · `remove-item` · `replace-item` · `reverse` · `sublist` (`substring`) · `word`

Mathematical

`abs` · `acos` · `Arithmetic Operators` (`+`, `*`, `-`, `/`, `^`, `<`, `>`, `=`, `!=`, `<=`, `>=`) · `asin` · `atan` · `ceiling` · `cos` · `exp` · `floor` · `int` · `is-number?` (`is-boolean?`) · `ln` · `log` · `max` · `mean` · `median` · `min` · `mod` · `modes` · `new-seed` · `precision` · `random` · `random-float` · `random-exponential` (`random-gamma`, `random-normal`, `random-poisson`) · `random-seed` · `remainder` · `round` · `sin` · `sqrt` · `standard-deviation` · `subtract-headings` · `sum` · `tan` · `variance`

Plotting

`autoplot?` (`autoplotx?`, `autoploty?`) · `auto-plot-off` (`auto-plot-x-off`, `auto-plot-y-off`, `auto-plot-on`, `auto-plot-x-on`, `auto-plot-y-on`) · `clear-all-plots` · `clear-plot` · `create-temporary-plot-pen` · `export-view` (`export-interface`, `export-output`, `export-plot`, `export-all-plots`, `export-world`) · `histogram` · `plot` · `plot-name` · `plot-pen-exists?` · `plot-pen-down` (`plot-pen-up`) · `plot-pen-reset` · `plotxy` · `plot-x-min` (`plot-x-max`, `plot-y-min`, `plot-y-max`) · `set-current-plot` · `set-current-plot-pen` · `set-histogram-num-bars` · `set-plot-background-color` · `set-plot-pen-color` · `set-plot-pen-interval` · `set-plot-pen-mode` · `setup-plots` · `set-plot-x-range` (`set-plot-y-range`) · `update-plots`

BehaviorSpace

`behaviorspace-experiment-name` · `behaviorspace-run-number`

System

`netlogo-version` · `netlogo-web?`

Built-In Variables

Turtle Variable

`breed` `color` `heading` `hidden?` `label` `label-color` `pen-mode` `pen-size` `shape` `size` `who` `xcor` `ycor`

Link Variable

`breed` `color` `end1` `end2` `hidden?` `label` `label-color` `shape` `thickness` `tie-mode`

Patch Variable

`pcolor` `plabel` `plabel-color` `pxcor`

Keywords

`breed` `directed-link-breed` `end` `export` `extensions` `globals` `import` `__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

Primitives and Commands

abs

Since 1.0

`abs` *`number`*

Reports the absolute value of *number*.

```
show abs -7
=> 7
show abs 5
=> 5
```

acos

Since 1.3

`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?

Since 4.0

`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

Since 1.0

`boolean1`* `and` *`boolean2`

Reports true if both *boolean1* and *boolean2* are true. Otherwise reports false.

Note that if *boolean1* is false, then *boolean2* will not be run (since it can't affect the result). See the programming guide for more information on logical operator precedence.

```
if (pxcor > 0) and (pycor > 0)
  [ set pcolor blue ]  ;; the upper-right quadrant of
                       ;; patches turn blue
```

any?

Since 2.0

`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

Since 4.0

`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

Since 4.0

`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.

```
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

+	*Since* 1.0
*****	*Since* 1.0
-	*Since* 1.0
/	*Since* 1.0
^	*Since* 1.0
<	*Since* 1.0
>	*Since* 1.0
=	*Since* 1.0
!=	*Since* 1.0
<=	*Since* 1.0
>=	*Since* 1.0

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 or link) 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
```

Many extension objects may be tested for equality and inequality using = and !=. For instance, the array, matrix, and table objects returned by their respective extensions may be compared for equality / inequality. Extension objects may not be tested using <, >, <=, or >=.

asin

Since 1.3

`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

Since 1.0

`ask` *`agentset commands`*
`ask` *`agent commands`*

The specified agent or agentset runs the given commands. Because agentset members are always read in a random order, when ask is used with an agentset each agent will take its turn in a random order. See Agentsetsfor more information.

```
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

Since 4.0

`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

Since 1.0

***`agentset`* `at-points [[`*`x1 y1`*`] `*`x2 y2`*` …]`**

Reports a subset of the given agentset that includes only the agents on the patches at the given coordinates (relative to this agent). The coordinates 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
            ;; coordinates (2,4), (1,2) and (10,15),
            ;; relative to the caller, move
```

atan

Since 1.0

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?

Since 1.0

autoplotx?

Since 7.0

autoploty?

Since 7.0

autoplot?
autoplotx?
autoploty?

Reports true if auto-plotting is on for the current plot or plot axis, false otherwise.

auto-plot-off

Since 1.0

auto-plot-x-off

Since 7.0

auto-plot-y-off

Since 7.0

auto-plot-on

Since 1.0

auto-plot-x-on

Since 7.0

auto-plot-y-on

Since 7.0

auto-plot-off
auto-plot-x-off
auto-plot-y-off
auto-plot-on
auto-plot-x-on
auto-plot-y-on

These pairs of commands are 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 when adding a new point with `plot` or `plotxy`. When using `histogram` to plot values, only the y axis will automatically update its ranges, the x axis will be unchanged. Each plot has an auto-plotting settings called *Auto xcale x-axis?* and *Auto scale y-axis?* in the user interface that determines if the plot will enable auto-plotting when the model starts. Auto-plotting is useful when wanting to show all plotted values in the current plot, regardless of the current plot ranges.

These commands will produce a runtime error if the current plot has not been set.

See also `plot`, `plotxy`, and `histogram`, and also the Plots section of the Interface Tab guide.

back

Since 1.0

bk

Since 1.0

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 `forward`, `jump`, `can-move?`.

base-colors

Since 4.0

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

Since 2.1

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

Since 5.2

`behaviorspace-experiment-name`

Reports the current experiment name in the current experiment.

If no BehaviorSpace experiment is running, reports "".

behaviorspace-run-number

Since 4.1.1

`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

Since 4.0

`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

Since 1.0

butfirst

Since 1.0

bf

Since 1.0

but-last

Since 1.0

butlast

Since 1.0

bl

Since 1.0

`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"
```

can-move?

Since 3.1

`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

Since 2.1

`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

Since 1.0

`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

Since 1.0

ca

Since 1.0

`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

Since 1.0

`clear-all-plots`

Clears every plot in the model. See `clear-plot` for more information.

clear-drawing

Since 3.0

cd

Since 3.0

`clear-drawing`

Clears all lines and stamps drawn by turtles.

clear-globals

Since 5.2

`clear-globals`

Sets all code-defined global variables (i.e., those defined inside of `globals [ ... ]`) to 0. Global variables defined by widgets are not affected by this primitive.

clear-links

Since 4.0

`clear-links`

Kills all links.

See also `die`.

clear-output

Since 1.0

`clear-output`

Clears all text from the model's output area, if it has one. Otherwise does nothing.

clear-patches

Since 1.0

cp

Since 1.0

`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`.

This command will produce a runtime error if the current plot has not been set.

clear-ticks

Since 5.0

`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. This is useful if you want to set the model to a “pre-setup” state with some forever buttons disabled.

See also `reset-ticks`.

clear-turtles

Since 1.0

ct

Since 1.0

`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 or 4 numbers with transparency). See details in the Colors section of the Programming Guide.

See also `pcolor`.

cos

Since 1.0

`cos` ***`number`***

Reports the cosine of the given angle. Assumes the angle is given in degrees.

```
show cos 180
=> -1
```

count

Since 1.0

`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

Since 4.0

cro

Since 4.0

`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.)

If *number* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
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 *Since* 4.0

create-links-to *Since* 4.0

create-link-from *Since* 4.0

create-links-from *Since* 4.0

create-link-with *Since* 4.0

create-links-with *Since* 4.0

```
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

Since 1.0

crt

Since 1.0

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.)

If *number* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
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

Since 1.1

`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

This command will produce a runtime error if the current plot has not been set.

See: `clear-plot`, `clear-all-plots`, and `set-current-plot-pen`.

date-and-time

Since 3.0

`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

Since 1.0

`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: `clear-turtles` `clear-links`

diffuse

Since 1.0

`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

Since 1.0

`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 `create-link(s)-to`, and `create-link(s)-from`, 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

Since 1.0

`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

Since 1.0

`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

Since 1.0

distancexy *x y*

Reports the distance from this agent to the point (*x*, *y*).

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

Since 1.0

downhill4

Since 1.0

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

Since 1.0

dy

Since 1.0

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.

empty?

Since 1.0

`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

Since 4.0

`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

Since 4.0

`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

Since 5.0

`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

Since 2.1

`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

Since 1.0

`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

Since 1.0

`exp` *`number`*

Reports the value of e raised to the *number* power.

Note: This is the same as e ^ *number*.

export

Since 7.1

`export` *`names`*

Make the given names available for importing. This command is only valid in an NLM file. See `import` for more information.

export-view *Since* 3.0

export-interface *Since* 2.0

export-output *Since* 1.0

export-plot *Since* 1.0

export-all-plots *Since* 1.2.1

export-world *Since* 1.0

`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 *plot-name* 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 Windows
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

Since 1.0

`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 0 to 100.

The given *color* can either be a NetLogo color in the range 0 to 140, not including 140 itself, or an RGB list of three values in the range 0 to 255 representing the levels of red, green, and blue.

```
show extract-hsb cyan
=> [180 57.143 76.863]
show extract-hsb red
=> [3.103 80.93 84.314]
show extract-hsb [255 0 0]
=> [0 100 100]
```

See also `approximate-hsb`, `approximate-rgb`, `extract-rgb`.

extract-rgb

Since 1.0

`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 or RGB(A) color.

```
show extract-rgb red
=> [215 50 41]
show extract-rgb cyan
=> [84 196 196]
show extract-rgb [31 41 59]
=> [31 41 59]
```

See also `approximate-rgb`, `approximate-hsb`, `extract-hsb`.

face

Since 3.0

`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

Since 3.0

`facexy x y`

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?

Since 2.0

`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

Since 2.0

`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

Since 2.0

`file-close-all`

Closes all files (if any) that have been opened previously with `file-open`.

See also `file-close`, `file-open`.

file-delete

Since 2.0

`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?

Since 2.0

`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

Since 4.0

`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

Since 2.0

`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

Since 2.0

`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`, `file-write`, and Output (/programming guide).

file-read

Since 2.0

`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
```

Note: This primitive is not compatible with NetLogo Web. If you wish to read the contents of a file with the same code and the same behavior in both NetLogo and NetLogo Web, see fetch:user-file-async.

See also `file-open` and `file-write`.

file-read-characters

Since 2.0

`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

Since 2.0

`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

Since 2.0

`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`, `file-write`, and Output (/programming guide).

file-type

Since 2.0

`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`, `file-write`, and Output (/programming guide).

file-write

Since 2.0

`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`, `file-type`, and Output (/programming guide).

filter

Since 1.3

`filter` *`reporter list`*

Reports a list containing only those items of *list* for which the reporter reports true — in other words, the items satisfying the given condition. *reporter* may be an anonymous reporter or the name of a reporter.

```
show filter is-number? [1 "2" 3]
=> [1 3]
show filter [ i -> i < 3 ] [1 3 2]
=> [1 2]
show filter [ s -> first s != "t" ] ["hi" "there" "everyone"]
=> ["hi" "everyone"]
```

See also `map`, `reduce`, `-> (anonymous procedure)`.

first

Since 1.0

`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

Since 1.0

`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

Since 3.0

follow *turtle*

Similar to ride, but, in the 3D view, the observer's vantage point is behind and above *turtle*.

The observer may only watch or follow a single subject. Calling `follow` will alter the highlight created by prior calls to `watch` and `watch-me`, highlighting the followed turtle instead.

See also `follow-me`, `ride`, `reset-perspective`, `watch`, `subject`.

follow-me

Since 3.0

follow-me

Asks the observer to follow this turtle.

The observer may only watch or follow a single subject. Calling `follow-me` will remove the highlight created by prior calls to `watch` and `watch-me`, highlighting this turtle instead.

See also `follow`.

foreach

Since 1.3

foreach *list command*
(foreach *list1* … *command*)

With a single list, runs the command for each item of *list*. *command* may be the name of a command, or an anonymous command created with `->`.

```
foreach [1.1 2.2 2.6] show
=> 1.1
=> 2.2
=> 2.6
foreach [1.1 2.2 2.6] [ x -> show (word x " -> " round x) ]
=> 1.1 -> 1
=> 2.2 -> 2
=> 2.6 -> 3
```

With multiple lists, runs *command* 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]
   [ [a b] -> show word "the sum is: " (a + b) ])
=> "the sum is: 3"
=> "the sum is: 6"
=> "the sum is: 9"
(foreach list (turtle 1) (turtle 2) [3 4]
  [ [the-turtle num-steps] -> ask the-turtle [ fd num-steps ] ])
;; turtle 1 moves forward 3 patches
;; turtle 2 moves forward 4 patches
```

See also `map`, `-> (anonymous procedure)`.

forward

Since 1.0

fd

Since 1.0

`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

Since 1.0

`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]
```

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.

hatch

Since 1.0

`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

Since 4.0

`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

Since 1.0

ht

Since 1.0

`hide-turtle`

The turtle makes itself invisible.

Note: This command is equivalent to setting the turtle variable “hidden?” to true.

See also `show-turtle`.

histogram

Since 1.0

`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
```

This command will produce a runtime error if either the current plot or the current pen has not been set.

See also `set-histogram-num-bars`, `set-plot-pen-interval`, `set-plot-x-range`.

home

Since 1.0

`home`

This turtle moves to the origin (0,0). Equivalent to `setxy 0 0`.

home-directory

Since 6.4.1

`home-directory`

Reports the user's home directory.

hsb

Since 1.0

`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. Input values will be clamped to these ranges before converting to RGB. The RGB list contains three integers in the range of 0-255.

See also `rgb`

hubnet-broadcast

Since 1.1

`hubnet-broadcast` *`tag-name value`*

This broadcasts *value* from NetLogo to the interface element with the name *tag-name* on the clients.

See the HubNet Authoring Guide for details and instructions.

hubnet-broadcast-clear-output

Since 4.1

`hubnet-broadcast-clear-output`

This clears all messages printed to the text area on every client.

See also: `hubnet-broadcast-message` , `hubnet-send-clear-output`

hubnet-broadcast-message

Since 4.1

`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

Since 4.1

hubnet-clear-overrides

Since 4.1

`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

Since 5.0

`hubnet-clients-list`

Reports a list containing the names of all the clients currently connected to the HubNet server.

hubnet-enter-message?

Since 1.2.1

`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?

Since 1.2.1

`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

Since 1.1

`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 Guidefor details.

hubnet-kick-client

Since 5.0

`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

Since 5.0

`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

Since 1.1

`hubnet-message`

Reports the message retrieved by `hubnet-fetch-message`.

See the HubNet Authoring Guidefor details.

hubnet-message-source

Since 1.1

`hubnet-message-source`

Reports the name of the client that sent the message retrieved by `hubnet-fetch-message`.

See the HubNet Authoring Guidefor details.

hubnet-message-tag

Since 1.1

`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 Guidefor details.

hubnet-message-waiting?

Since 1.1

`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 Guidefor details.

hubnet-reset

Since 1.1

`hubnet-reset`

Starts up the HubNet system. HubNet must be started to use any of the other hubnet primitives.

See the HubNet Authoring Guidefor details.

hubnet-reset-perspective

Since 4.1

`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

Since 1.1

`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 Guidefor details.

hubnet-send-clear-output

Since 4.1

`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

Since 4.1

`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.

A client may only watch or follow a single subject. Calling `hubnet-send-follow` will alter the highlight created by prior calls to `hubnet-send-watch`, highlighting the followed agent instead.

See also: `hubnet-send-watch`, `hubnet-reset-perspective`

hubnet-send-message

Since 4.1

`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

Since 4.1

`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

Since 4.1

`hubnet-send-watch` *`client-name agent`*

Tells the client associated with *client-name* to watch *agent*.

A client may only watch or follow a single subject. Calling `hubnet-send-watch` will undo perspective changes caused by prior calls to `hubnet-send-follow`.

See also: `hubnet-send-follow`, `hubnet-reset-perspective`

if

Since 1.0

`if` *`boolean commands`*

If *boolean* reports true then the *commands* are run, otherwise the commands are not run if *boolean* reports false.

If *boolean* reports a value other than true or false a runtime error will occur.

The *boolean* may report a different value for different agents when used with a primitive like `ask`, so some agents may run *commands* and others don't.

```
ask turtles [
  if xcor > 0 [ set color blue ]
  ;; turtles in the right half of the world
  ;; turn blue
]
```

See also `ifelse`, `ifelse-value`.

ifelse

Since 1.0

`ifelse` *`boolean1 commands1 elsecommands`*
`(ifelse` *`boolean1 commands1 boolean2 commands2 … elsecommands`*`)` `Since 6.1`

For the first *boolean* that reports true, runs the *commands* that follow.

If no *boolean* reports true, runs *elsecommands* or does nothing if *elsecommands* is not given. When using only one *boolean*you do not need to surround the entire *ifelse* primitive and its blocks in parentheses.

If a *boolean* reports a value other than true or false a runtime error will occur.

```
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
```

The reporters may report a different value for different agents, so some agents may run different command blocks. When using more than one *boolean* you must surround the whole *ifelse* primitive and its blocks in parentheses. This functionality was added in NetLogo 6.1.

```
ask patches [
  let choice random 4
  (ifelse
    choice = 0 [
      set pcolor red
      set plabel "r"
    ]
    choice = 1 [
      set pcolor blue
      set plabel "b"
    ]
    choice = 2 [
      set pcolor green
      set plabel "g"
    ]
    ; elsecommands
    [
      set pcolor yellow
      set plabel "y"
  ])
]
```

See also `if`, `ifelse-value`.

ifelse-value

Since 2.0

`ifelse-value` *`boolean1 reporter1 elsereporter`*

`(ifelse-value` *`boolean1 reporter1 boolean2 reporter2 … elsereporter`*`)` *Since 6.1*

For the first *boolean* that reports true, runs the *reporter* that follows and reports that result. When using only one *boolean*you do not need to surround the entire *ifelse-value* primitive and its blocks in parentheses.

If all *boolean*s report false, the result is the value of *elsereporter*. You may leave out the *elsereporter*, but if all *boolean*s report false then a runtime error will occur.

If a *boolean* reports a value other than true or false a runtime error will occur.

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 [ n -> ifelse-value (n < 5) [0] [1] ]
=> [0 0 0 0 0 1 1 1 1 1]
show reduce [ [a b] -> ifelse-value (a > b) [a] [b] ]
  [1 3 2 5 3 8 3 2 1]
=> 8
```

When using more than one *boolean* you must surround the whole *ifelse-value* primitive and its blocks in parentheses. This functionality was added in NetLogo 6.1.

```
ask patches [
  let choice random 4
  set pcolor (ifelse-value
    choice = 0 [ red ]
    choice = 1 [ blue ]
    choice = 2 [ green ]
               [ yellow ])
]
```

A runtime error can occur if there is no *elsereporter*.

```
ask patches [
  let x = 2
  set pcolor (ifelse-value
    x = 0 [ red ]
    x = 1 [ blue ]
    ; no final else reporter is given, and x is 2 so there will be a runtime error
  )
```

See also `if`, `ifelse`.

import *Since 7.1*

`import` *`module`* `as` *`prefix`*
`import` *`names`* `from` *`module`*

Make the procedures in the given module available for use in the current file. The file name of the module file (with the `.nlm` suffix) must be a valid NetLogo name, so there can be no space in the name.

For example:

```
import utils
```

This will import procedures in “utils.nls” in the same directory as the current file and make them available with a “utils:” prefix. As an example, if there is a reporter called `add` in “utils.nls”, then it will be made available for use under the name `utils:add`.

The prefix can be changed as follows:

```
import utils as u
```

This will instead make `add` available as `u:add`.

It is also possible to selectively import a subset of procedures or reporters from a module, for example:

```
import [add subtract] from utils
```

This will make `add` and `subtract` available in the current file. Note that no prefix is added when a module is imported this way.

Imported procedures and reporters can be renamed as follows:

```
import [add (subtract as sub)] from utils
```

This will instead make `subtract` available as `sub`.

import-drawing

Since 3.0

`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.

Note: This primitive is not compatible with NetLogo Web. If you wish to import a drawing with the same code and the same behavior in both NetLogo and NetLogo Web, see import-a:drawing.

import-pcolors

Since 3.0

`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.)

Note: This primitive is not compatible with NetLogo Web. If you wish to import patch colors with the same code and the same behavior in both NetLogo and NetLogo Web, see import-a:pcolors.

import-pcolors-rgb

Since 4.0

`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.)

Note: This primitive is not compatible with NetLogo Web. If you wish to import patch colors with the same code and the same behavior in both NetLogo and NetLogo Web, see import-a:pcolors-rgb.

import-world

Since 1.0

`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.

Note: This primitive is not compatible with NetLogo Web. If you wish to import a world with the same code and the same behavior in both NetLogo and NetLogo Web, see import-a:world.

in-cone

Since 3.0

`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?

Since 4.0

in-<breed>-neighbor? ***agent***
in-link-neighbor? ***turtle***

Reports true if there is a directed link going from *turtle*to the caller or an undirected link connecting *turtle* to the caller. You can think of this as "is there a link I can use to get 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

Since 4.0

in-<breed>-neighbors
in-link-neighbors

Reports the agentset of all the turtles that have directed links coming from them to the caller as well as all turtles that have an undirected link connecting them with the caller. You can think of this as "all the turtles that can get to the caller using a link."

```
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

Since 4.0

in-<breed>-from ***turtle***
in-link-from ***turtle***

Reports a directed link from *turtle* to the caller or an undirected link connecting the two. If no link exists then it reports nobody. If more than one such link exists, reports a random one. You can think of this as "give me a link that I can use to travel from *turtle* to the caller."

```
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

Since 4.0

`__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.

The file names must be strings, for example:

```
__includes [ "utils.nls" ]
```

Or, for multiple files:

```
__includes [ "utils1.nls" "utils2.nls" ]
```

in-radius

Since 1.0

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
```

insert-item

Since 6.0.2

`insert-item` *`index list value`*
`insert-item` *`index string1 string2`*

On a list, inserts an item in that list. *index* is the index where the item will be inserted. The first item has an index of 0. (The 6th item in a list would have an index of 5.)

Likewise for a string, but all characters in a multiple-character *string2*are inserted at *index*.

```
show insert-item 2 [2 7 4 5] 15
=> [2 7 15 4 5]
show insert-item 2 "cat" "re"
=> "caret"
```

inspect

Since 1.1

`inspect` *`agent`*

Opens an agent monitor for the given agent (turtle or patch or link).

```
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
```

```
inspect one-of links
;; an agent monitor opens for a random link
```

See `stop-inspecting` and `stop-inspecting-dead-agents`

int

Since 1.0

`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?

Since 1.2.1

is-agentset?

Since 1.2.1

is-anonymous-command?

Since 6.0

is-anonymous-reporter?

Since 6.0

is-boolean?

Since 1.2.1

is-directed-link?

Since 4.0

is-link?

Since 4.0

is-link-set?

Since 4.0

is-list?

Since 1.0

is-number?

Since 1.2.1

is-patch?

Since 1.2.1

is-patch-set?

Since 4.0

is-string?

Since 1.0

is-turtle?

Since 1.2.1

is-turtle-set?

Since 4.0

is-undirected-link?

Since 4.0

`is-agent?` *`value`*
`is-agentset?` *`value`*
`is-anonymous-command?` *`value`*
`is-anonymous-reporter?` *`value`*
`is-boolean?` *`value`*
`is-<breed>?` *`value`*
`is-<link-breed>?` *`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-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

Since 1.0

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.)

If *index* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
;; suppose mylist is [2 4 6 8 10]
show item 2 mylist
=> 6
show item 3 "my-shoe"
=> "s"
```

jump

Since 1.0

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?`.

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

Since 1.0

`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

Since 4.0

`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 [ [a b] -> [size] of a < [size] of b ] turtles 10
```

layout-radial

Since 4.0

`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

Since 4.0

`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

Since 4.0

`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 8
  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
    create-link-with turtle 6
    create-link-with turtle 7
  ]
  ; 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 [count link-neighbors > 1]) links 8 ]
end
```

left

Since 1.0

lt

Since 1.0

`left` *`number`*

The turtle turns left by *number* degrees. (If *number*is negative, it turns right.)

length

Since 1.0

`length` *`list`*
`length` *`string`*

Reports the number of items in the given list, or the number of characters in the given string.

let

Since 2.1

`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 ] ]
```

You can also create multiple local variables if you put the variable names in a list format. The values for the new variables will be taken from the list given as the second argument. This can be particular useful when you want to calculate multiple values in a reporter procedure, as you can easily create multiple variables with the results. A runtime error will occur if you do not give a list of values or the list you provide doesn't have enough values for all the variables.

```
let [x y z] [10 15 20]
show x ; prints 10
show y ; prints 15
show z ; prints 20

let [a b c] [10] ; causes a runtime error as we need at least 3 values in the list
```

link

Since 4.0

`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

Since 4.0

`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

Since 4.0

`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

Since 4.0

`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

Since 4.0

`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

Since 4.0

`links`

Reports the agentset consisting of all links. This is a special agentset that can grow as links are added to the world, see the programming guide for more info.

```
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

Since 1.0

`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

Since 1.0

`ln` *`number`*

Reports the natural logarithm of *number*, that is, the logarithm to the base e (2.71828...).

See also `e`, `log`.

log

Since 1.0

`log` *`number base`*

Reports the logarithm of *number* in base *base*.

```
show log 64 2
=> 6
```

See also `ln`.

loop

Since 1.0

`loop` *`commands`*

Repeats the commands forever, or until the enclosing procedure exits through use of the `stop` or `report` 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 Buttons in the Programming Guide.

lput

Since 1.0

`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]
```

map

Since 1.3

`map` *`reporter list`*
`(map` *`reporter list1`* `…)`

With a single *list*, the given reporter is run for each item in the list, and a list of the results is collected and reported. *reporter* may be an anonymous reporter or the name of a reporter.

```
show map round [1.1 2.2 2.7]
=> [1 2 3]
show map [ i -> i * i ] [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 [ [a b c] -> a + b = c ] [1 2 3] [2 4 6] [3 5 9])
=> [true false true]
```

See also `foreach`, `-> (anonymous procedure)`.

max

Since 1.0

`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

Since 4.0

`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

Since 1.0

`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

Since 3.1

max-pycor

Since 3.1

`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 by editing the view or using `resize-world`.

```
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

Since 1.0

`mean` *`list`*

Reports the statistical mean of the numeric items in the given list. Ignores non-numeric items. The mean is the average, i.e., the sum of the items divided by the total number of items.

In NetLogo 6.1.1 and earlier, mean would error when finding non-number values in the given list.

```
show mean [xcor] of turtles
;; prints the average of all the turtles' x coordinates
```

See this FAQ question for information on possible issues using *mean* with *agentsets*

median

Since 1.0

`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?

Since 1.0

`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

Since 1.0

`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

Since 4.0

`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

Since 1.0

`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

Since 3.1

min-pycor

Since 3.1

`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 by editing the view or using `resize-world`.

```
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

Since 1.0

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

Since 2.0

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] ]
=> __WIKILINK_0__]
show modes [pxcor] of turtles
;; shows which columns of patches have the most
;; turtles on them
```

mouse-down?

Since 1.0

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?

Since 3.0

mouse-inside?

Reports true if the mouse pointer is inside the world boundaries inside the current view, false otherwise. In the case of an unwrapped world and using `follow` on an agent, It's possible for the mouse pointer to be inside the view but not inside the world boundaries.

mouse-xcor

Since 1.0

mouse-ycor

Since 1.0

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

Since 4.0

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`.

my-<breeds>

my-links

Since 4.0

my-<breeds>
my-links

Reports an agentset of all links connected to the caller of the corresponding breed, regardless of directedness. Generally, you might consider using [[|my-out-links]] instead of this primitive, as it works well for either directed or undirected networks (since it excludes directed, incoming links).|my-out-links]] instead of this primitive, as it works well for either directed or undirected networks (since it excludes directed, incoming links).

```
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
```

If you only want the undirected links connected to a node, you can do `my-links with [ not is-directed-link? self ]`.

my-in-<breeds>

my-in-links

Since 4.0

my-in-<breeds>
my-in-links

Reports an agentset of all the directed links coming in from other nodes to the caller as well as all undirected links connected to the caller. You can think of this as "all links that you can use to travel *to* this node".

```
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

Since 4.0

my-out-<breeds>
my-out-links

Reports an agentset of all the directed links going out from the caller to other nodes as well as undirected links connected to the caller. You can think of this as "all links you can use to travel *from* this node".

```
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

Since 1.0

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-of

Since 3.1

`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.

If *size* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
ask n-of 50 patches [ set pcolor green ]
;; 50 randomly chosen patches turn green
```

See also `one-of` and `up-to-n-of`, a version that does not error with a *size* greater than the size of the input.

n-values

Since 2.0

`n-values` *`size reporter`*

Reports a list of length *size* containing values computed by repeatedly running the reporter. *reporter* may be an anonymous reporter or the name of a reporter.

If the reporter accepts inputs, the input will be the number of the item currently being computed, starting from zero.

If *size* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
show n-values 5 [1]
=> [1 1 1 1 1]
show n-values 5 [ i -> i ]
=> [0 1 2 3 4]
show n-values 3 turtle
=> [(turtle 0) (turtle 1) (turtle 2)]
show n-values 5 [ x -> x * x ]
=> [0 1 4 9 16]
```

See also `reduce`, `filter`, `-> (anonymous procedure)`, `range`.

neighbors

Since 1.1

neighbors4

Since 1.1

`neighbors`
`neighbors4`

Reports an agentset containing the 8 surrounding patches (neighbors) or 4 surrounding patches (neighbors4), if they exist. If the world is wrapping, the neighbors of a patch along the edge of the world will include the corresponding patches on the opposite edge of the world. If the world is not wrapping, any neighbors outside the boundaries of the world will be excluded from the set. The neighbor set will never include the target patch itself, even if the world only has one row or one column of patches.

```
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

Since 4.0

`<breed>-neighbors`
`link-neighbors`

Reports the agentset of all turtles found at the other end of any links (undirected or directed, incoming or outgoing) 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?

Since 4.0

`<breed>-neighbor?` *`turtle`*
`link-neighbor?` *`turtle`*

Reports true if there is a link (either directed or undirected, incoming or outgoing) 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-version

Since 3.0

`netlogo-version`

Reports a string containing the version number of the NetLogo you are running.

```
show netlogo-version
=> "{{version}}"
```

netlogo-web?

Since 5.2

`netlogo-web?`

Reports true if the model is running in NetLogo Web.

new-seed

Since 3.0

`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

Since 1.0

`no-display`

Turns off all updates to the current view until the display command is issued. This does not affect the display of plots. The primitive 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

Since 4.0

no-links

Reports an empty link agentset.

no-patches

Since 4.0

no-patches

Reports an empty patch agentset.

not

Since 1.0

not *boolean*

Reports true if *boolean* is false, otherwise reports false.

```
if not any? turtles [ crt 10 ]
```

no-turtles

Since 4.0

no-turtles

Reports an empty turtle agentset.

of

Since 4.0

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

Since 1.0

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`, `up-to-n-of`.

or

Since 1.0

boolean1* or *boolean2

Reports true if *boolean1* is true, *boolean2* is true, or both are true. Otherwise returns false if both *boolean*s are false.

Note that if *boolean1* is true, then *boolean2* will not be run (since it can't affect the result). See the programming guide for more information on logical operator precedence.

```
if (pxcor > 0) or (pycor > 0) [ set pcolor red ]
;; patches turn red except in lower-left quadrant
```

other

Since 4.0

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

Since 4.0

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?

Since 4.0

out-<breed>-neighbor? ***turtle***
out-link-neighbor? ***turtle***

Reports true if there is a directed link going from the caller to *turtle* or if there is an undirected link connecting the caller with *turtle*. You can think of this as "can I get from the caller to *turtle* using a link?"

```
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

Since 4.0

out-<breed>-neighbors
out-link-neighbors

Reports the agentset of all the turtles that have directed links from the caller, or undirected links with the caller. You can think of this as "who can I get to from the caller using a link?"

```
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

Since 4.0

`out-<breed>-to` *`turtle`*
`out-link-to` *`turtle`*

Reports a directed link from the caller to *turtle* or an undirected link connecting the two. If no link exists then it reports nobody. If more than one such link exists, reports a random one. You can think of this as "give me a link that I can use to travel from the caller to *turtle*."

```
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

Since 2.1

output-show

Since 2.1

output-type

Since 2.1

output-write

Since 2.1

`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.) See also Output (/programming guide).

patch

Since 1.0

`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

Since 2.0

`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

Since 1.0

`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

Since 2.0

`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

Since 1.0

`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

Since 2.0

patch-right-and-ahead

Since 2.0

`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

Since 4.0

`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

Since 4.1

`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

Since 1.0

`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. If you give an RGBA color with 4 numbers the transparency value will be ignored (except in NetLogo 3D).

See also `color`.

pen-down

Since 1.0

pd

Since 1.0

pen-erase

Since 3.0

pe

Since 3.0

pen-up

Since 1.0

pu

Since 1.0

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

Since 1.0

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.)

This command will produce a runtime error if the current plot has not been set.

plot-name

Since 1.0

`plot-name`

Reports the name of the current plot (a string)

This command will produce a runtime error if the current plot has not been set.

plot-pen-exists?

Since 4.0

`plot-pen-exists?` ***`string`***

Reports true if a plot pen with the given name is defined in the current plot. Otherwise reports false.

This command will produce a runtime error if the current plot has not been set.

plot-pen-down

Since 1.0

plot-pen-up

Since 1.0

`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.)

These commands will produce a runtime error if either the current plot or the current pen has not been set.

plot-pen-reset

Since 1.0

`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.

This command will produce a runtime error if either the current plot or the current pen has not been set.

plotxy

Since 1.0

`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).

This command will produce a runtime error if the current plot has not been set.

plot-x-min

Since 1.0

plot-x-max

Since 1.0

plot-y-min

Since 1.0

plot-y-max

Since 1.0

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.)

These commands will produce a runtime error if the current plot has not been set.

position

Since 1.0

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

Since 1.0

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

Since 1.0

`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`, `write`, `output-print`, and Output (/programming guide).

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`.

random

Since 1.0

`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

Since 2.0

`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

Since 1.2.1

random-gamma

Since 2.0

random-normal

Since 1.2.1

random-poisson

Since 1.2.1

`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

Since 3.1

random-pycor

Since 3.1

`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

Since 1.0

`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 Numberssection 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
```

See also the `new-seed` reporter that generates proper random seed values.

random-xcor

Since 3.1

random-ycor

Since 3.1

`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`.

range

Since 6.0

`range` *`stop`*
`(range` *`start stop`*`)`
`(range` *`start stop step`*`)`

Generates a list of numbers, starting at *start*, ending before *stop*, counting by *step*. *start* defaults to 0 and *step* defaults to 1.

```
show range 5
=> [0 1 2 3 4]
show (range 2 5)
=> [2 3 4]
```

```
show (range 2 5 0.5)
=> [2 2.5 3 3.5 4 4.5]
show (range 10 0 -1)
=> [10 9 8 7 6 5 4 3 2 1]
```

See also `n-values`

read-from-string

Since 1.1

`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
```

Note: This primitive is not compatible with NetLogo Web. If you wish to read user input with the same code and the same behavior in both NetLogo and NetLogo Web, see dialog:user-input.

reduce

Since 1.3

`reduce` *`reporter list`*

Reduces a list from left to right using the given reporter, resulting in a single value. This means, for example, that `reduce [ [a b] -> a + b] [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. *reporter* may be an anonymous reporter or the name of a reporter.

The first input passed to the reporter 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 [ [result-so-far next-item] -> next-item - result-so-far ] [1 2 3]
=> 2
show reduce [ [result-so-far ignored-item] -> result-so-far ] [1 2 3]
=> 1
show reduce [ [ignored next-item] -> next-item ] [1 2 3]
=> 3
show reduce sentence [[1 2] [3 [4]] 5]
=> [1 2 3 [4] 5]
show reduce [ [result-so-far next-item] -> fput next-item result-so-far ] (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
    [ [longest-so-far next-string] -> ifelse-value (length longest-so-far >= length next-string)
```

```
[longest-so-far] [next-string] ]
    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
    [ [occurrence-count next-item] -> ifelse-value (next-item = x) [occurrence-count + 1]
[occurrence-count] ] (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 [ [value coefficient] -> (x * value) + coefficient ] coefficients
end

;; evaluate 3x^2 + 2x + 1 at x = 4
show evaluate-polynomial [3 2 1] 4
=> 57
```

See also `filter`, `-> (anonymous procedure`.

remainder

Since 1.2.1

`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

Since 1.0

`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

Since 1.0

`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

Since 2.0

`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 *string* 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.)

If *index* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
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

Since 1.0

`repeat` *`number commands`*

Runs *commands number* times.

```
pd repeat 36 [ fd 1 rt 10 ]
;; the turtle draws a circle
```

replace-item

Since 1.0

`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.

If *index* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
show replace-item 2 [2 7 4 5] 15
=> [2 7 15 5]
show replace-item 1 "cat" "are"
=> "caret"
```

report

Since 1.0

`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

Since 3.0

rp

Since 3.0

`reset-perspective`

The observer stops watching, following, or riding any turtles (or patches or links). (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). Note: following and riding are valid for turtles only.

See also `follow`, `ride`, `watch`.

reset-ticks

Since 4.0

`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

Since 1.0

`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

Since 4.1

`resize-world` *`min-pxcor max-pxcor min-pycor max-pycor`*

Changes the size of the patch grid.

If any coordinate is fractional, it will be rounded to the nearest integer towards zero (4.5 becomes 4, 10.9 becomes 10, -2.9 becomes -2).

If the new patch grid coordinates are different than the ones in use, all turtles and links die, the existing patch grid is discarded, new patches created, and the tick counter is cleared. However, if the grid coordinates are the same, the world state will remain unchanged.

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

Since 1.0

`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

Since 1.0

`rgb` *`red green blue`*

Reports a RGB list when given three numbers describing an RGB color. The numbers are clamped to the range 0-255.

See also `hsb`

ride

Since 3.0

`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.

The observer may only watch or follow a single subject. Calling `ride` will remove the highlight created by prior calls to `watch` and `watch-me`, highlighting the ridden turtle instead.

See also `reset-perspective`, `watch`, `follow`, `subject`.

ride-me

Since 3.0

`ride-me`

Asks the observer to ride this turtle.

The observer may only watch or follow a single subject. Calling `ride-me` will remove the highlight created by prior calls to `watch` and `watch-me`, highlighting this turtle instead.

See also `ride`.

right

Since 1.0

rt

Since 1.0

`right` *`number`*

The turtle turns right by *number* degrees. (If *number* is negative, it turns left.)

round

Since 1.0

`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 `precision`, `ceiling`, `floor`.

run

Since 1.3

runresult

Since 1.3

`run` *`command`*
`(run` *`command input1`* `…)`
`run` *`string`*
`runresult` *`reporter`*
`(runresult` *`reporter input1`* `…)`
`runresult` *`string`*

The `run` form expects the name of a command, an anonymous command, or a string containing commands. This agent then runs them.

The `runresult` form expects the name of a reporter, an anonymous reporter, 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 an anonymous command.

Anonymous procedures 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.)

Anonymous procedures 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 anonymous procedures, you can provide them with inputs, if you surround the entire call with parentheses. For example:

```
(run [ [turtle-count step-count] -> crt turtle-count [ fd step-count ] ] 10 5)
;; creates 10 turtles and move them forward 5 steps
show (runresult [ [a b] -> a + b ] 10 5)
=> 15
;; adds 10 and 5
```

See also `foreach`, `-> (anonymous procedure)`.

scale-color

Since 1.0

`scale-color` *`color number range1 range2`*

Reports a shade of *color* proportional to the value of *number*.

When *range1* is less than or equal to *range2*, then the larger the number, the lighter the shade of *color*. However, if *range2*is less than *range1*, the color scaling is inverted.

Let *min-range* be the minimum of *range1* and *range2*. If *number* is less than or equal to *min-range*, then the result is the same as if *number* was equal to *min-range*. Let *max-range* be the maximum of *range1* and *range2*. If *number* is greater than *max-range*, then the result is the same as if *number* was equal to *max-range*.

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

Since 1.3

`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

Since 1.0

se

Since 1.0

`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 __WIKILINK_0__ __WIKILINK_1__
=> [[1 2] [3 4]]
show (sentence [1 2] 3 [4 5] (3 + 3) 7)
=> [1 2 3 4 5 6 7]
```

set

Since 1.0

`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.

Example:

```
ask turtles [
  set color red
  set size 2
  set shape "arrow"
]
```

You can also give a list of variable names as the first argument for `set` and they will be assigned the values from a list given as the second argument. This can be particular useful when you want to calculate multiple values in a reporter procedure, as you can easily set multiple variables with the results. A runtime error will occur if the second argument is not a list value or if there are not enough values in the list for all the variables specified.

```
ask turtles [
  set [color size shape] [red 2 "arrow"]
  show color ; prints 15
  show size ; prints 2
  show shape ; prints "arrow"
]
ask turtles [
  set [color size shape] [red] ; causes a runtime error as we need at least 3 values in the list
]
```

set-current-directory

Since 2.0

`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.

```
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

Since 1.0

`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

Since 1.0

`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. If the current plot has not been set, a runtime error occurs.

set-default-shape

Since 1.0

`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

Since 1.0

`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`.

This command will produce a runtime error if either the current plot or the current pen has not been set.

__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

Since 4.1

`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-background-color

Since 6.0.2

`set-plot-background-color` *`color`*

Sets the background color of the current plot. The color may be specified as a number or a list. See the Colors section of the programming guide for more details. This change is temporary and is not saved with the model. When the plot is cleared, the background color will revert to white.

Note: Plot backgrounds do not support transparency. If a list is used to set the color, the alpha component will be ignored.

This command will produce a runtime error if the current plot has not been set.

set-plot-pen-color

Since 1.0

`set-plot-pen-color` *`color`*

Sets the color of the current plot pen to *color*.

This command will produce a runtime error if either the current plot or the current pen has not been set.

set-plot-pen-interval

Since 1.0

`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.)

This command will produce a runtime error if either the current plot or the current pen has not been set.

set-plot-pen-mode

Since 1.0

`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).

This command will produce a runtime error if either the current plot or the current pen has not been set.

setup-plots

Since 5.0

`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

Since 1.0

set-plot-y-range

Since 1.0

`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.

These commands will produce a runtime error if the current plot has not been set.

set-topology

Since 7.0

`set-topology` *`wraps-in-x? wraps-in-y?`*

Sets the wrapping behavior of the world, such that X coordinates wrap if the first argument was *true* and Y coordinates wrap if the second argument was *true*.

Equivalent to adjusting the "World wraps horizontally" and "World wraps vertically" checkboxes in the "Model Settings" window.

```
set-topology true true
;; wrapping in both directions (torus)
set-topology true false
;; wrapping only in the X direction (vertical cylinder)
set-topology false true
;; wrapping only in the Y direction (horizontal cylinder)
set-topology false false
;; no wrapping (box)
```

setxy

Since 1.0

`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?

Since 1.0

`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](shapes)
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

Since 2.1

`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

Since 1.0

`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`, `write`, `output-show`, and Output (/programming guide).

show-turtle

Since 1.0

st

Since 1.0

`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

Since 4.0

`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

Since 2.0

`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

Since 1.0

`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

Since 1.0

`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. (Patches are sorted

with the top left-most patch first and the bottom right-most patch last, turtles are sorted by `who` number).

```
show sort [3 1 4 2]
=> [1 2 3 4]
show sort [2 1 "a"]
=> [1 2]
show sort (list "a" "c" "b" (patch 0 0))
=> ["a" "b" "c"]
show sort (list (patch 0 0) (patch 0 1) (patch 1 0))
=> [(patch 0 1) (patch 0 0) (patch 1 0)]

;; label patches with numbers in left-to-right, top-to-bottom order
let n 0
foreach sort patches [ the-patch ->
  ask the-patch [
    set plabel n
    set n n + 1
  ]
]

;; some additional examples to clarify behavior in strange cases
show sort (list patch 0 0 patch 0 1 patch 1 0 turtle 0 turtle 1) ; turtles are always sorted
lower than patches
=> [(turtle 0) (turtle 1) (patch 0 1) (patch 0 0) (patch 1 0)]
show sort (list nobody false true) ; booleans and nobody cannot be sorted
=> []
show sort (list [1 2 3] turtles) ; lists and agentsets are not included if they are inside a
list passed to sort
=> []
```

See also `sort-by`, `sort-on`.

sort-by

Since 1.3

`sort-by` *`reporter list`*
`sort-by` *`reporter 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. *reporter* may be an anonymous reporter or the name of a reporter.

The two inputs to *reporter* are the values being compared. The reporter should report true if the first argument comes strictly before the second 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 [ [string1 string2] -> length string1 < length string2 ] ["Grumpy" "Doc" "Happy"]
=> ["Doc" "Happy" "Grumpy"]
```

See also `sort`, `sort-on`, `-> (anonymous procedure)`.

sort-on

Since 5.0

`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
  [ the-turtle -> ask the-turtle [ do-something ] ]
;; turtles run "do-something" one at a time, in
;; ascending order by size
```

See also `sort`, `sort-by`.

sprout

Since 1.0

`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.

If *number* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
sprout 5
sprout-wolves 10
sprout 1 [ set color red ]
sprout-sheep 1 [ set color black ]
```

See also `create-turtles`, `hatch`.

sqrt

Since 1.0

`sqrt` *`number`*

Reports the square root of *number*.

stamp

Since 1.0

`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

Since 3.1

`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

Since 1.0

`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
```

See this FAQ question for information on possible issues using *standard-deviation* with *agentsets*

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

Since 1.0

`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

Since 5.2

`stop-inspecting` *`agent`*

Closes the agent monitor for the given agent (turtle or patch or link). 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
ask links [ stop-inspecting self ]
;; close all agent monitors for links
```

See `inspect` and `stop-inspecting-dead-agents`.

stop-inspecting-dead-agents

Since 5.2

`stop-inspecting-dead-agents`

Closes all agent monitors for dead agents. See `inspect` and `stop-inspecting`.

subject

Since 3.0

`subject`

Reports the turtle (or patch or link) that the observer is currently watching, following, or riding. Reports `nobody` if there is no such turtle (or patch or link). Note: following and riding are valid for turtles only.

See also `watch`, `follow`, `ride`.

sublist

Since 2.1

substring

Since 1.0

`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).

If either *position* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

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

Since 2.1

`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

Since 1.0

`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
```

See this FAQ question for information on possible issues using *sum* with *agentsets*

tan

Since 1.0

`tan` *`number`*

Reports the tangent of the given angle. Assumes the angle is given in degrees.

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

Since 4.0

`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

Since 4.0

`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

Since 4.0

`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

Since 4.0

`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

Since 1.0

`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

Since 1.0

`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

Since 1.0

`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

Since 1.0

`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

Since 4.0

`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

Since 1.0

`turtles`

Reports the agentset consisting of all turtles. This is a special agentset that can grow as turtles are added to the world, see the programming guide for more info.

```
show count turtles
;; prints the number of turtles
```

turtles-at

Since 1.0

`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

Since 1.0

`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

Since 2.0

`turtles-on` *`turtle or patch`*
`turtles-on` *`turtleset or agentset`*
`<breeds>-on` *`turtle or patch`*
`<breeds>-on` *`turtleset or patchset`*

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

Since 1.0

`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`, `write`, `output-type`, and Output (/programming guide).

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

Since 4.0

`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.

up-to-n-of

Since 6.1

`up-to-n-of` ***size agentset***
`up-to-n-of` ***size list***

From an agentset, reports an agentset of size *size*randomly chosen from the input set, with no repeats. If the input does not have enough agents to satisfy the *size*, reports the entire agentset.

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.) If the input does not have enough items to satisfy the *size*, reports the entire list.

If *size* is fractional, it will be rounded down to the nearest integer (4.5 becomes 4, 10.9 becomes 10).

```
ask up-to-n-of 50 patches [ set pcolor green ]
;; 50 randomly chosen patches turn green
;; if less than 50 patches exist, they all turn green
```

See also `n-of`, `one-of`.

update-plots

Since 5.0

`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

Since 1.0

uphill4

Since 1.0

`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

Since 3.1

`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

Since 3.1

`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

Since 3.1

`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`.

Note: This primitive is not compatible with NetLogo Web. If you wish to read the contents of a file with the same code and the same behavior in both NetLogo and NetLogo Web, see fetch:user-file-async.

user-input

Since 1.1

`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?"
```

See the User Interaction Primitives section of the Programming Guide for additional details.

user-message

Since 1.1

`user-message` *`value`*

Opens a dialog with *value* displayed as the message to the user.

value may be of any type, but is typically a string.

```
user-message (word "There are " count turtles " turtles.")
```

Note that if a user closes the `user-message` dialog with the "X" in the corner, the behavior will be the same as if they had clicked "OK".

See the User Interaction Primitives section of the Programming Guide for additional details.

user-one-of

Since 3.1

`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 ]
```

Note: This primitive is not compatible with NetLogo Web. If you wish to read a chooser value from the user with the same code and the same behavior in both NetLogo and NetLogo Web, see dialog:user-one-of.

See the User Interaction Primitives section of the Programming Guide for additional details.

user-yes-or-no?

Since 2.0

`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 ]
```

Note: This primitive is not compatible with NetLogo Web. If you wish to read a *true* or *false* value from the user with the same code and the same behavior in both NetLogo and NetLogo Web, see dialog:user-yes-or-no?.

See the User Interaction Primitives section of the Programming Guide for additional details.

variance

Since 1.0

`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
```

See this FAQ question for information on possible issues using *variance* with *agentsets*

wait

Since 1.0

`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

Since 3.0

`watch agent`

Puts a spotlight on *agent*. In the 3D view the observer will also turn to face the subject.

The observer may only watch or follow a single subject. Calling `watch` will undo perspective changes caused by prior calls to `follow`, `follow-me`, `ride`, and `ride-me`.

See also `follow`, `subject`, `reset-perspective`, `ride`, `ride-me`, `watch-me`.

watch-me

Since 3.0

`watch-me`

Asks the observer to watch this agent.

The observer may only watch or follow a single subject. Calling `watch` will undo perspective changes caused by prior calls to `follow`, `follow-me`, `ride`, and `ride-me`.

See also `follow`, `subject`, `reset-perspective`, `ride`, `ride-me`, `watch`.

while

Since 1.0

`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.

who-are-not

Since 6.3.1

agentset* who-are-not *agentset
agentset* who-are-not *agent

Takes an agentset on the left and an agentset or an agent on the right. Reports a new agentset containing all agents from the left-hand agentset that are not in the right-hand agentset (or are not the right-hand agent).

```
breed [frogs frog]
breed [mice mouse]

create-frogs 10
create-mice 10
create-turtles 10

; contains all the turtles who are not frogs
ask turtles who-are-not frogs [
  forward 1
]```

Another example:
```

ask turtles ; contains all the turtles this turtle is not linked to let targets (other turtles who-are-not link-neighbors) if count targets > 0 [create-link-with one-of targets] ```

with

Since 1.0

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

Since 4.0

<breed>-with *turtle*
link-with *turtle*

Reports a link between *turtle* and the caller (directed or undirected, incoming or outgoing). If no link exists then it reports nobody. If more than one such link exists, reports a random one.

```
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

Since 2.1

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

Since 2.1

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

Since 4.0

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

Since 1.1

`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

Since 1.0

`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:foofishbar.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

Since 3.1

world-height

Since 3.1

`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

Since 1.0

`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

Since 2.0

`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`, `type`, `output-write`, and Output (/programming guide).

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

Since 1.0

`boolean1` `xor` *`boolean2`*

Reports true if either *boolean1* or *boolean2* is true, but not when both are true. Otherwise returns false. See the programming guide for more information on logical operator precedence.

```
if (pxcor > 0) xor (pycor > 0)
  [ set pcolor blue ]
;; upper-left and lower-right quadrants turn blue
```

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`,

Since 6.0

args -> commands
args -> reporter

Creates and reports an anonymous procedure - a command or reporter

- depending on the input. Within *commands* or *reporter* the listed *args* may be used just as you would use `let` or procedure variables. The variable names in *args* have the same restrictions as variable names of commands and reporters. In addition, they must not match the name of any let or procedure variable in their procedure.

Anonymous procedures are commonly used with the primitives `foreach`, `map`, `reduce`, `filter`, `sort-by`, and `n-values`. See those entries for example usage.

See the Anonymous Procedures section of the Programming Guide for details.