



NetLogo 3.1.3 User Manual

Table of Contents

<u>What is NetLogo?</u>	1
<u>Features</u>	1
<u>Copyright Information</u>	3
<u>Third party licenses</u>	3
<u>What's New?</u>	7
<u>Version 3.1.3 (September 20, 2006)</u>	7
<u>Version 3.1.2 (August 9, 2006)</u>	7
<u>Version 3.1.1 (June 16, 2006)</u>	7
<u>Version 3.1 (April 14, 2006)</u>	8
<u>Version 3.0 (September 2005)</u>	10
<u>Version 2.1 (December 2004)</u>	10
<u>Version 2.0.2 (August 2004)</u>	11
<u>Version 2.0 (December 2003)</u>	11
<u>Version 1.3 (June 2003)</u>	11
<u>Version 1.2 (March 2003)</u>	11
<u>Version 1.1 (July 2002)</u>	11
<u>Version 1.0 (April 2002)</u>	12
<u>System Requirements</u>	13
<u>System Requirements: Application</u>	13
<u>Windows</u>	13
<u>Mac OS X</u>	13
<u>Mac OS 8 and 9</u>	13
<u>Other platforms</u>	13
<u>System Requirements: Saved Applets</u>	14
<u>System Requirements: 3D View</u>	14
<u>Operating Systems</u>	14
<u>Graphics Cards</u>	14
<u>Fullscreen mode</u>	15
<u>Library Conflicts</u>	15
<u>Removing an old JOGL</u>	15
<u>Known Issues</u>	17
<u>Known bugs (all systems)</u>	17
<u>Windows-only bugs</u>	17
<u>Macintosh-only bugs</u>	17
<u>Linux/UNIX-only bugs</u>	18
<u>Known issues with computer HubNet</u>	18
<u>Unimplemented StarLogoT primitives</u>	18
<u>Contacting Us</u>	21
<u>Web Site</u>	21
<u>Feedback, Questions, Etc.</u>	21
<u>Reporting Bugs</u>	21

Table of Contents

Sample Model: Party	23
<u>At a Party</u>	23
<u>Challenge</u>	25
<u>Thinking With Models</u>	26
<u>What's Next?</u>	26
Tutorial #1: Models	27
<u>Sample Model: Wolf Sheep Predation</u>	27
<u>Controlling the Model: Buttons</u>	28
<u>Adjusting Settings: Sliders and Switches</u>	29
<u>Gathering Information: Plots and Monitors</u>	31
<u>Plots</u>	31
<u>Monitors</u>	31
<u>Controlling the View</u>	31
<u>The Models Library</u>	35
<u>Sample Models</u>	36
<u>Curricular Models</u>	36
<u>Code Examples</u>	36
<u>HubNet Calculator & Computer Activities</u>	36
<u>What's Next?</u>	36
Tutorial #2: Commands	37
<u>Sample Model: Traffic Basic</u>	37
<u>The Command Center</u>	37
<u>Working With Colors</u>	40
<u>Agent Monitors and Agent Commanders</u>	42
<u>What's Next?</u>	45
Tutorial #3: Procedures	47
<u>Agents and procedures</u>	47
<u>Making the setup button</u>	47
<u>Making the go button</u>	50
<u>Experimenting with commands</u>	51
<u>Patches and variables</u>	52
<u>Turtle variables</u>	53
<u>Monitors</u>	55
<u>Switches and labels</u>	57
<u>More procedures</u>	59
<u>Plotting</u>	60
<u>Global variables</u>	63
<u>Some more details</u>	64
<u>What's next?</u>	65
<u>Appendix: Complete code</u>	66
Interface Guide	69
<u>Menus</u>	69
<u>Tabs</u>	71
<u>Interface Tab</u>	71

Table of Contents

Interface Guide

<u>Working with Interface Elements</u>	71
<u>Chart: Interface Toolbar</u>	72
<u>The 2D and 3D Views</u>	73
<u>Command Center</u>	77
<u>Plots</u>	79
<u>Procedures Tab</u>	80
<u>Information Tab</u>	82
<u>Information Tab Markup</u>	83
<u>WHAT IS IT</u>	83

Programming Guide.....85

<u>Agents</u>	85
<u>Procedures</u>	86
<u>Variables</u>	88
<u>Colors</u>	89
<u>Ask</u>	91
<u>Agentsets</u>	92
<u>Breeds</u>	94
<u>Buttons</u>	95
<u>Synchronization</u>	97
<u>Lists</u>	98
<u>Math</u>	102
<u>Random Numbers</u>	104
<u>Turtle shapes</u>	105
<u>Plotting</u>	105
<u>Strings</u>	108
<u>Output</u>	109
<u>File I/O</u>	110
<u>Movies</u>	111
<u>Perspective</u>	112
<u>Drawing</u>	113
<u>Topology</u>	114
<u>Links</u>	118
<u>Tie</u>	121

Shapes Editor Guide.....123

<u>Getting Started</u>	123
<u>Importing Shapes</u>	123
<u>Creating and Editing Shapes</u>	124
<u>Tools</u>	125
<u>Previews</u>	125
<u>Overlapping Shapes</u>	125
<u>Undo</u>	125
<u>Colors</u>	125
<u>Other buttons</u>	126
<u>Shape Design</u>	126
<u>Keeping a Shape</u>	126

Table of Contents

Shapes Editor Guide

<u>Using Shapes in a Model</u>	126
--------------------------------------	-----

BehaviorSpace Guide.....127

<u>What is BehaviorSpace?</u>	127
<u>Why BehaviorSpace?</u>	127
<u>Historical Note</u>	128
<u>How It Works</u>	128
<u>Managing experiment setups</u>	128
<u>Creating an experiment setup</u>	128
<u>Running an experiment</u>	131
<u>Advanced usage</u>	131
<u>Running from the command line</u>	131
<u>Setting up experiments in XML</u>	133
<u>Controlling API</u>	134
<u>Conclusion</u>	134

HubNet Guide.....135

<u>Understanding HubNet</u>	135
<u>NetLogo</u>	135
<u>HubNet Architecture</u>	135
<u>Computer HubNet</u>	136
<u>Activities</u>	136
<u>Requirements</u>	136
<u>Starting an activity</u>	136
<u>HubNet Control Center</u>	137
<u>Troubleshooting</u>	137
<u>Known Limitations</u>	138
<u>Calculator HubNet</u>	139
<u>Requirements</u>	139
<u>Teacher workshops</u>	139
<u>HubNet Authoring Guide</u>	139
<u>Getting help</u>	139

HubNet Authoring Guide.....141

<u>General HubNet Information</u>	141
<u>NetLogo Primitives</u>	141
<u>Setup</u>	141
<u>Data extraction</u>	142
<u>Sending data</u>	143
<u>Examples</u>	144
<u>Calculator HubNet Information</u>	144
<u>Saving</u>	144
<u>Computer HubNet Information</u>	145
<u>How To Make an Interface for a Client</u>	145
<u>View Updates on the Clients</u>	146
<u>Plot Updates on the Clients</u>	147
<u>Clicking in the View on Clients</u>	147

Table of Contents

HubNet Authoring Guide

<u>Text Area for Input and Display</u>	147
--	-----

Extensions Guide.....149

<u>Using Extensions</u>	149
<u>Applets</u>	150
<u>Writing Extensions</u>	150
<u>Summary</u>	150
<u>Tutorial</u>	150
<u>Extension development tips</u>	153
<u>Conclusion</u>	154

Controlling Guide.....155

<u>Note on memory usage</u>	155
<u>Example (with GUI)</u>	155
<u>Example (headless)</u>	156
<u>BehaviorSpace</u>	157
<u>Other Options</u>	158
<u>Conclusion</u>	158

GoGo Extension.....159

<u>What is the GoGo Board?</u>	159
<u>How to get a GoGo Board?</u>	159
<u>Installing the GoGo Extension</u>	159
<u>Mac OS X</u>	159
<u>Windows</u>	160
<u>Linux and others</u>	160
<u>Using the GoGo Extension</u>	160
<u>Primitives</u>	161
<u>gogo-close</u>	161
<u>gogo-open</u>	161
<u>gogo-open?</u>	161
<u>gogo-ports</u>	161
<u>output-port-coast</u>	162
<u>output-port-off</u>	162
<u>output-port-reverse</u>	162
<u>output-port-[that/this]way</u>	162
<u>talk-to-output-ports</u>	162
<u>ping</u>	163
<u>sensor</u>	163
<u>set-output-port-power</u>	164

Sound Extension.....165

<u>Using the Sound Extension</u>	165
<u>Primitives</u>	165
<u>drums</u>	165
<u>instruments</u>	165
<u>play-drum</u>	166

Table of Contents

Sound Extension

<u>play-note</u>	166
<u>play-note-later</u>	166
<u>start-note</u>	166
<u>stop-note</u>	167
<u>stop-instrument</u>	167
<u>stop-music</u>	167
<u>Sound names</u>	167
<u>Drums</u>	167
<u>Instruments</u>	168

FAQ (Frequently Asked Questions).....171

<u>Questions</u>	171
<u>General</u>	171
<u>Downloading</u>	171
<u>Applets</u>	171
<u>Usage</u>	171
<u>BehaviorSpace</u>	172
<u>Programming</u>	172
<u>General</u>	173
<u>Why is it called NetLogo?</u>	173
<u>What programming language was NetLogo written in?</u>	173
<u>How do I cite NetLogo in an academic publication?</u>	173
<u>How do I cite a model from the Models Library in a publication?</u>	173
<u>What license is NetLogo released under? Are there any legal restrictions on</u> <u>use, redistribution, etc.?</u>	173
<u>Is the source code to NetLogo available?</u>	173
<u>Do you offer any workshops or other training opportunities for NetLogo?</u>	174
<u>What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?</u>	174
<u>Has anyone built a model of <x>?</u>	174
<u>Are NetLogo models runs scientifically reproducible?</u>	174
<u>Are there any NetLogo textbooks?</u>	175
<u>Is NetLogo available in a Spanish version, German version, (your language here)</u> <u>version, etc.?</u>	175
<u>Is NetLogo compiled or interpreted?</u>	175
<u>Will NetLogo and NetLogo 3D remain separate?</u>	175
<u>Downloading</u>	176
<u>The download form doesn't work for me. Can I have a direct link to the software?</u>	176
<u>Downloading NetLogo takes too long. Is it available any other way, such as on a</u> <u>CD?</u>	176
<u>I downloaded and installed NetLogo but the Models Library has few or no models</u> <u>in it. How can I fix this?</u>	176
<u>Can I have multiple versions of NetLogo installed at the same time?</u>	176
<u>I'm on a UNIX system and I can't untar the download. Why?</u>	176
<u>How do I install NetLogo on Windows 2003 or Windows Server 2003?</u>	176
<u>Applets</u>	177
<u>I tried to run one of the applets on your site, but it didn't work. What should I do?</u>	177
<u>Can I make my model available as an applet while keeping the code secret?</u>	177

Table of Contents

FAQ (Frequently Asked Questions)

Can a model saved as an applet use import-world, file-open, and other commands that read files?.....	177
Usage.....	178
Can I run NetLogo from a CD?.....	178
Why is NetLogo so much slower when I unplug my laptop?.....	178
How do I change how many patches there are?.....	178
Can I use the mouse to "paint" in the view?.....	178
How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?.....	179
Can I import GIS data into NetLogo?.....	179
My model runs slowly. How can I speed it up?.....	180
I want to try HubNet. Can I?.....	180
Can I run NetLogo from the command line, without the GUI?.....	181
Can I have more than one model open at a time?.....	181
Does NetLogo take advantage of multiple processors?.....	181
Can I distribute NetLogo model runs across a cluster of computers?.....	181
Can I use max-pxcor or max-pycor, etc., as the minimum or maximum of a slider?.....	182
Can I change the choices in a chooser on the fly?.....	182
Can I divide the code for my model up into several files?.....	182
How do I show the legend in a plot?.....	182
Why does my code have strange characters in it?.....	182
BehaviorSpace.....	182
How do I measure runs every n ticks?.....	182
I'm varying a global variable I declared in the Procedures tab, but it doesn't work. Why?.....	182
Programming.....	183
How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?.....	183
How does the NetLogo language differ from other Logos?.....	183
My model from NetLogo 3.0 or earlier doesn't work (or looks funny) in 3.1. Help!.....	184
Why do I get a runtime error when I use setxy random world-width random world-height? It worked before.....	185
How do I take the negative of a number?.....	185
My turtle moved forward 1, but it's still on the same patch. Why?.....	185
patch-ahead 1 is reporting the same patch my turtle is already standing on. Why?.....	186
How do I give my turtles "vision"?.....	186
Can agents sense what's in the drawing layer?.....	186
Does NetLogo have a command like StarLogo's "grab" command?.....	186
I tried to put -at after the name of a variable, for example variable-at -1 0, but NetLogo won't let me. Why not?.....	187
I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?.....	187
The documentation says that random-float 1.0 might return 0.0 but will never return 1.0. What if I want 1.0 to be included?.....	187
How can I keep two turtles from occupying the same patch?.....	187
How can I find out if a turtle is dead?.....	187
How do I find out how much time has passed in my model?.....	187

Table of Contents

FAQ (Frequently Asked Questions)

<u>Does NetLogo have arrays?</u>	188
<u>Does NetLogo have associative arrays or lookup tables?</u>	188
<u>How can I use different patch "neighborhoods" (circular, Von Neumann, Moore, etc.)?</u>	188
<u>Can I connect turtles with lines, to indicate connections between them?</u>	188
<u>How can I convert an agentset to a list of agents, or vice versa?</u>	188
<u>How does NetLogo decide when to switch from agent to agent when running code?</u> ..	189
<u>How do I stop foreach?</u>	189
<u>How do I make an empty agentset?</u>	190

Primitives Dictionary.....191

<u>Categories of Primitives</u>	191
<u>Turtle-related</u>	191
<u>Patch-related primitives</u>	191
<u>Agentset primitives</u>	191
<u>Color primitives</u>	191
<u>Control flow and logic primitives</u>	192
<u>World primitives</u>	192
<u>Perspective primitives</u>	192
<u>HubNet primitives</u>	192
<u>Input/output primitives</u>	192
<u>File primitives</u>	192
<u>List primitives</u>	192
<u>String primitives</u>	192
<u>Mathematical primitives</u>	193
<u>Plotting primitives</u>	193
<u>Link primitives</u>	193
<u>Movie primitives</u>	193
<u>System primitives</u>	193
<u>Built-In Variables</u>	193
<u>Turtles</u>	193
<u>Patches</u>	193
<u>Other</u>	194
<u>Keywords</u>	194
<u>Constants</u>	194
<u>Mathematical Constants</u>	194
<u>Boolean Constants</u>	194
<u>Color Constants</u>	194
<u>A.</u>	194
<u>abs</u>	195
<u>acos</u>	195
<u>and</u>	195
<u>any?</u>	195
<u>Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=)</u>	195
<u>asin</u>	196
<u>ask</u>	196
<u>at-points</u>	196

Table of Contents

Primitives Dictionary

<u>atan</u>	197
<u>autoplot?</u>	197
<u>auto-plot-off auto-plot-on</u>	197
B	198
<u>back bk</u>	198
<u>beep</u>	198
<u>both-ends</u>	198
<u>breed</u>	199
<u>breed</u>	199
<u>but-first bf but-last bl</u>	200
C	200
<u>can-move?</u>	200
<u>carefully</u>	200
<u>ceiling</u>	201
<u>clear-all ca</u>	201
<u>clear-all-plots</u>	201
<u>clear-drawing cd</u>	201
<u>clear-output</u>	201
<u>clear-patches cp</u>	202
<u>clear-plot</u>	202
<u>clear-turtles ct</u>	202
<u>color</u>	202
<u>cos</u>	203
<u>count</u>	203
<u>create-<breed>-to create-<breed>-from create-<breed>-with</u>	203
<u>create-turtles crt create-<breeds></u>	204
<u>create-custom-turtles cct create-custom-<breeds> cct-<breeds></u>	204
<u>create-temporary-plot-pen</u>	205
D	206
<u>date-and-time</u>	206
<u>die</u>	206
<u>diffuse</u>	206
<u>diffuse4</u>	207
<u>display</u>	207
<u>distance</u>	208
<u>distancexy</u>	208
<u>downhill</u>	208
<u>downhill4</u>	209
<u>dx dy</u>	209
E	209
<u>empty?</u>	209
<u>end</u>	209
<u>end1</u>	210
<u>end2</u>	210
<u>error-message</u>	210
<u>every</u>	211
<u>exp</u>	211

Table of Contents

Primitives Dictionary

<u>export-view</u>	<u>export-interface</u>	<u>export-output</u>	<u>export-plot</u>	<u>export-all-plots</u>	
<u>export-world</u>					211
<u>extract-hsb</u>					212
<u>extract-rgb</u>					213
E					213
<u>face</u>					213
<u>facexy</u>					213
<u>file-at-end?</u>					214
<u>file-close</u>					214
<u>file-close-all</u>					214
<u>file-delete</u>					214
<u>file-exists?</u>					215
<u>file-open</u>					215
<u>file-print</u>					215
<u>file-read</u>					216
<u>file-read-characters</u>					216
<u>file-read-line</u>					217
<u>file-show</u>					217
<u>file-type</u>					217
<u>file-write</u>					217
<u>filter</u>					218
<u>first</u>					218
<u>floor</u>					218
<u>follow</u>					219
<u>follow-me</u>					219
<u>foreach</u>					219
<u>forward fd</u>					220
<u>fput</u>					220
G					220
<u>globals</u>					220
H					220
<u>hatch hatch-<breeds></u>					220
<u>heading</u>					221
<u>hidden?</u>					221
<u>hideturtle ht</u>					222
<u>histogram-from</u>					222
<u>histogram-list</u>					222
<u>home</u>					223
<u>hsb</u>					223
<u>hubnet-broadcast</u>					223
<u>hubnet-broadcast-view</u>					223
<u>hubnet-enter-message?</u>					224
<u>hubnet-exit-message?</u>					224
<u>hubnet-fetch-message</u>					224
<u>hubnet-message</u>					224
<u>hubnet-message-source</u>					224
<u>hubnet-message-tag</u>					224

Table of Contents

Primitives Dictionary

<u>hubnet-message-waiting?</u>	225
<u>hubnet-reset</u>	225
<u>hubnet-send</u>	225
<u>hubnet-send-view</u>	226
<u>hubnet-set-client-interface</u>	226
L	226
<u>if</u>	226
<u>ifelse</u>	227
<u>ifelse-value</u>	227
<u>import-drawing</u>	228
<u>import-pcolors</u>	228
<u>import-world</u>	228
<u>in-cone</u>	229
<u>in-<breed>-neighbor?</u>	229
<u>in-<breed>-neighbors</u>	230
<u>in-<breed>-from</u>	230
<u>in-radius</u>	231
<u>inspect</u>	231
<u>int</u>	231
<u>is-agent? is-agentset? is-boolean? is-<breed>? is-link? is-list? is-number?</u> <u>is-patch? is-patch-agentset? is-string? is-turtle? is-turtle-agentset?</u>	231
<u>item</u>	232
J	232
<u>jump</u>	232
L	233
<u>label</u>	233
<u>label-color</u>	233
<u>last</u>	233
<u>layout-circle</u>	234
<u>layout-magspring</u>	234
<u>layout-radial</u>	235
<u>layout-spring</u>	236
<u>layout-tutte</u>	237
<u>left lt</u>	238
<u>length</u>	238
<u>let</u>	238
<u>list</u>	239
<u>ln</u>	239
<u>locals</u>	239
<u>log</u>	239
<u>loop</u>	240
<u>lput</u>	240
M	240
<u>map</u>	240
<u>max</u>	241
<u>max-one-of</u>	241
<u>max-pxcor max-pycor</u>	241

Table of Contents

Primitives Dictionary

<u>mean</u>	241
<u>median</u>	242
<u>member?</u>	242
<u>min</u>	242
<u>min-one-of</u>	243
<u>min-pxcor min-pycor</u>	243
<u>mod</u>	243
<u>modes</u>	244
<u>mouse-down?</u>	244
<u>mouse-inside?</u>	244
<u>mouse-xcor mouse-ycor</u>	244
<u>movie-cancel</u>	245
<u>movie-close</u>	245
<u>movie-grab-view movie-grab-interface</u>	245
<u>movie-set-frame-rate</u>	245
<u>movie-start</u>	245
<u>movie-status</u>	246
<u>my-<breeds></u>	246
<u>my-in-<breeds></u>	246
<u>my-out-<breeds></u>	247
<u>myself</u>	247
N	248
<u>n-of</u>	248
<u>n-values</u>	248
<u>neighbors neighbors4</u>	248
<u><breed>-neighbors</u>	249
<u><breed>-neighbor?</u>	249
<u>netlogo-version</u>	250
<u>new-seed</u>	250
<u>no-display</u>	250
<u>nobody</u>	250
<u>not</u>	251
<u>nsum nsum4</u>	251
O	251
<u>-of</u>	251
<u>one-of</u>	252
<u>or</u>	252
<u>other-turtles-here other-<breeds>-here</u>	252
<u>other-end</u>	253
<u>out-<breed>-neighbor?</u>	253
<u>out-<breed>-neighbors</u>	254
<u>out-<breed>-to</u>	254
<u>output-print output-show output-type output-write</u>	255
P	255
<u>patch</u>	255
<u>patch-ahead</u>	255
<u>patch-at</u>	256

Table of Contents

Primitives Dictionary

<u>patch-at-heading-and-distance</u>	256
<u>patch-here</u>	256
<u>patch-left-and-ahead patch-right-and-ahead</u>	256
<u>patches</u>	257
<u>patches-from</u>	257
<u>patches-own</u>	257
<u>pcolor</u>	258
<u>pen-down pd pen-erase pe pen-up pu</u>	258
<u>pen-mode</u>	258
<u>pen-size</u>	258
<u>plabel</u>	259
<u>plabel-color</u>	259
<u>plot</u>	259
<u>plot-name</u>	259
<u>plot-pen-down ppd plot-pen-up ppu</u>	260
<u>plot-pen-reset</u>	260
<u>plotxy</u>	260
<u>plot-x-min plot-x-max plot-y-min plot-y-max</u>	260
<u>position</u>	260
<u>precision</u>	261
<u>print</u>	261
<u>pxcor pycor</u>	261
<u>R</u>	262
<u>random</u>	262
<u>random-float</u>	262
<u>random-exponential random-gamma random-normal random-poisson</u>	263
<u>random-int-or-float</u>	263
<u>random-pxcor random-pycor</u>	264
<u>random-seed</u>	264
<u>random-xcor random-ycor</u>	265
<u>read-from-string</u>	265
<u>reduce</u>	265
<u>remainder</u>	266
<u>remove</u>	267
<u>remove-duplicates</u>	267
<u>remove-item</u>	267
<u>remove-<breed>-from remove-<breed>-to remove-<breed>-with</u>	267
<u>repeat</u>	268
<u>replace-item</u>	268
<u>report</u>	269
<u>reset-perspective rp</u>	269
<u>reset-timer</u>	269
<u>reverse</u>	269
<u>rgb</u>	269
<u>ride</u>	270
<u>ride-me</u>	270
<u>right rt</u>	270

Table of Contents

Primitives Dictionary

<u>round</u>	270
<u>run</u>	271
<u>runresult</u>	271
S	271
<u>scale-color</u>	271
<u>self</u>	272
<u>:</u> (semicolon).....	272
<u>sentence se</u>	272
<u>set</u>	273
<u>set-current-directory</u>	273
<u>set-current-plot</u>	274
<u>set-current-plot-pen</u>	274
<u>set-default-shape</u>	274
<u>set-histogram-num-bars</u>	275
<u>set-line-thickness</u>	275
<u>set-plot-pen-color</u>	275
<u>set-plot-pen-interval</u>	275
<u>set-plot-pen-mode</u>	275
<u>set-plot-x-range set-plot-y-range</u>	276
<u>setxy</u>	276
<u>shade-of?</u>	276
<u>shape</u>	277
<u>shapes</u>	277
<u>show</u>	277
<u>show-turtle st</u>	277
<u>shuffle</u>	278
<u>sin</u>	278
<u>size</u>	278
<u>sort</u>	278
<u>sort-by</u>	279
<u>sprout sprout-<breeds></u>	279
<u>sqrt</u>	280
<u>stamp</u>	280
<u>stamp-erase</u>	280
<u>standard-deviation</u>	280
<u>startup</u>	280
<u>stop</u>	281
<u>subject</u>	281
<u>sublist substring</u>	281
<u>subtract-headings</u>	281
<u>sum</u>	282
T	282
<u>tan</u>	282
<u>tie</u>	282
<u>timer</u>	283
<u>to</u>	283
<u>to-report</u>	283

Table of Contents

Primitives Dictionary

<u>towards</u>	283
<u>towardsxy</u>	284
<u>turtle</u>	284
<u>turtles</u>	284
<u>turtles-at <breeds>-at</u>	284
<u>turtles-from</u>	285
<u>turtles-here <breed>-here</u>	285
<u>turtles-on <breeds>-on</u>	286
<u>turtles-own <breeds>-own</u>	286
<u>type</u>	287
<u>U</u>	287
<u>untie</u>	287
<u>uphill</u>	287
<u>uphill4</u>	288
<u>user-directory</u>	288
<u>user-file</u>	288
<u>user-new-file</u>	288
<u>user-input</u>	289
<u>user-message</u>	289
<u>user-one-of</u>	289
<u>user-yes-or-no?</u>	289
<u>V</u>	290
<u>value-from</u>	290
<u>values-from</u>	290
<u>variance</u>	290
<u>W</u>	290
<u>wait</u>	290
<u>watch</u>	291
<u>watch-me</u>	291
<u>while</u>	291
<u>who</u>	291
<u>with</u>	292
<u><breed>-with</u>	292
<u>with-max</u>	292
<u>with-min</u>	293
<u>without-interruption</u>	293
<u>word</u>	293
<u>world-width world-height</u>	294
<u>wrap-color</u>	294
<u>write</u>	294
<u>X</u>	295
<u>xcor</u>	295
<u>xor</u>	295
<u>Y</u>	295
<u>ycor</u>	295
<u>?</u>	295
<u>?</u>	295

What is NetLogo?

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It was authored by Uri Wilensky in 1999 and is in continuous development 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 independent "agents" all operating concurrently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals.

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 that students and teachers can easily run simulations or even build their own. And, it is advanced enough to serve as a powerful tool for researchers in many fields.

NetLogo has extensive documentation and tutorials. It also comes with a Models Library, which is a large collection of pre-written simulations that can be used and modified. These simulations address many 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 currently under development.

NetLogo can also power a classroom participatory-simulation tool called HubNet. Through the use of networked computers or handheld devices such as Texas Instruments (TI-83+) calculators, each student can control an agent in a simulation. Follow [this link](#) for more information.

NetLogo is the next generation of the series of multi-agent modeling languages that started with StarLogo. It builds off the functionality of our product [StarLogoT](#) and adds significant new features and a redesigned language and user interface. NetLogo is written in Java so it can run on all major platforms (Mac, Windows, Linux, et al). It is run as a standalone application. Individual models can be run as Java applets inside a web browser.

Features

You can use the list below to help familiarize yourself with the features NetLogo has to offer.

- System:
 - ◆ Cross-platform: runs on Mac, Windows, Linux, et al
- Language:
 - ◆ Fully programmable
 - ◆ Simple language structure
 - ◆ Language is Logo dialect extended to support agents and concurrency
 - ◆ Mobile agents (turtles) move over a grid of stationary agents (patches)
 - ◆ Create links between turtles to make networks
 - ◆ Unlimited numbers of agents and variables
 - ◆ Large vocabulary of built-in language primitives
 - ◆ Integer and double precision floating point math
 - ◆ Runs are exactly reproducible cross-platform

- Environment:
 - ◆ View your model in either 2D and 3D
 - ◆ Scalable and rotatable vector shapes
 - ◆ Turtle and patch labels
 - ◆ Interface builder w/ buttons, sliders, switches, choosers, monitors, text boxes
 - ◆ "Control strip" including speed slider
 - ◆ Powerful and flexible plotting system
 - ◆ Info tab for annotating your model
 - ◆ 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)
 - ◆ BehaviorSpace tool used to collect data from multiple runs of a model
 - ◆ System Dynamics Modeler
- Web:
 - ◆ Models can be saved as applets to be embedded in web pages (note: some features are not available from applets, such as some extensions and the 3D view)

Copyright Information

Copyright 1999 by Uri Wilensky. All rights reserved.

The NetLogo software, models and documentation are distributed free of charge for use by the public to explore and construct models. Permission to copy or modify the NetLogo software, models and documentation for educational and research purposes only and without fee is hereby granted, provided that this copyright notice and the original author's name appears on all copies and supporting documentation. For any other uses of this software, in original or modified form, including but not limited to distribution in whole or in part, specific prior permission must be obtained from Uri Wilensky. The software, models and documentation shall not be used, rewritten, or adapted as the basis of a commercial software or hardware product without first obtaining appropriate licenses from Uri Wilensky. We make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

To reference this software in academic publications, please use: Wilensky, U. (1999). NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

The project gratefully acknowledges the support of the National Science Foundation (REPP and ROLE Programs) — grant numbers REC #9814682 and REC #0126227.

Third party licenses

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.

Parts of NetLogo (specifically, the random-gamma primitive) are based on code from the Colt library (<http://hoschek.home.cern.ch/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.

NetLogo uses the MRJ Adapter library, which is Copyright (c) 2003 Steve Roy <sroy@roydesign.net>. The library is covered by the GNU LGPL (Lesser General Public License). The text of that license is included in the "docs" folder which accompanies the NetLogo download, and is also available from <http://www.gnu.org/copyleft/lesser.html>.

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

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

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

This software is Copyright 2003 by Sean Luke. Portions Copyright 2003 by Gabriel Catalin Balan, Liviu Panait, Sean Paus, and Dan Kuebrich. All Rights Reserved.

Developed in Conjunction with the George Mason University Center for Social Complexity

By using the source code, binary code files, or related data included in this distribution, you agree to the following terms of usage for this software distribution. All but a few source code files in this distribution fall under this license; the exceptions contain open source licenses embedded in the source code files themselves. In this license the Authors means the Copyright Holders listed above, and the license itself is Copyright 2003 by Sean Luke.

The Authors hereby grant you a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:

to use, reproduce, modify, display, perform, sublicense and distribute all or any portion of the source code or binary form of this software or related data with or without modifications, or as part of a

larger work; and under patents now or hereafter owned or controlled by the Authors, to make, have made, use and sell ("Utilize") all or any portion of the source code or binary form of this software or related data, but solely to the extent that any such patent is reasonably necessary to enable you to Utilize all or any portion of the source code or binary form of this software or related data, and not to any greater extent that may be necessary to Utilize further modifications or combinations.

In return you agree to the following conditions:

If you redistribute all or any portion of the source code of this software or related data, it must retain the above copyright notice and this license and disclaimer. If you redistribute all or any portion of this code in binary form, you must include the above copyright notice and this license and disclaimer in the documentation and/or other materials provided with the distribution, and must indicate the use of this software in a prominent, publically accessible location of the larger work. You must not use the Authors's names to endorse or promote products derived from this software without the specific prior written permission of the Authors.

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, NOR THEIR EMPLOYERS, NOR GEORGE MASON UNIVERSITY, 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.

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

Copyright (c) 1999–2001 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.

For graphics rendering, NetLogo uses JOGL, a Java API for OpenGL. For more information about JOGL, see <http://jogl.dev.java.net/>. The library is distributed under the BSD license:

Copyright (c) 2003 Sun Microsystems, Inc. All Rights Reserved.

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

- Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution 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 Sun Microsystems, Inc. or the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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 MICROSYSTEMS, INC. ("SUN") AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS 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 THIS SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You acknowledge that this software is not designed, licensed or intended for use in the design, construction, operation or maintenance of any nuclear facility.

Sun gratefully acknowledges that this software was originally authored and developed by Kenneth Bradley Russell and Christopher John Kline.

What's New?

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. Please send comments, suggestions, and questions to feedback@ccl.northwestern.edu, and bug reports to bugs@ccl.northwestern.edu.

Version 3.1.3 (September 20, 2006)

- fixed 3.1.2–only bug where exporting the world could cause a Java exception on Windows machines
- fixed bug where saving a model didn't update the expected location of data files

Version 3.1.2 (August 9, 2006)

- documentation:
 - ◆ rewritten and expanded Tutorial #3
- models:
 - ◆ improved models: 3D Shapes Example (bugfix), GasLab Single Collision (improved code), Tetris (bugfix), GasLab Circular Particles (bugfixes, added code for reversing time)
 - ◆ new Code Examples: Musical Phrase Example, Network Import Example
- engine fixes:
 - ◆ fixed 3.1–only bug where startup procedures didn't work properly
 - ◆ fixed 3.1–only bug where `in-radius` in non-wrapping worlds was slow
 - ◆ fixed compiler bug where repeated `<breeds>-own` declarations would not result in an error
 - ◆ fixed bug where `sprout 0` caused Java exception
 - ◆ fixed bug where using `sort` with an invalid input could cause a Java exception
- other fixes:
 - ◆ fixed Windows–only bug where saved model files could include inconsistent line-ending characters
- extensions
 - ◆ new command in the sound extension `play-note-later` can be used to play a musical phrase

Version 3.1.1 (June 16, 2006)

- content:
 - ◆ corrections to User Manual
 - ◆ new biology models: Daisyworld, Wolf Sheep Stride Inheritance
 - ◆ improved models: Ethnocentrism (clearer code, corrected reference), Bug Hunt Camouflage (upgraded, now verified), Fire (improved info tab), GasLab Two Gas (bug fix), Heatbugs (now verified), GasLab Single Collision (bug fix), Disease Solo (improved interface), Echo (improved)
 - ◆ new Code Example: Line of Sight Example
- engine fixes:
 - ◆ many extensions can now be used in saved applets (however, extensions that require additional external jars still don't work)

- ◆ fixed bug in `in-radius` where a large radius in a non-wrapping world could give incorrect results
- ◆ fixed bug where calling `movie-close` when no movie was open could cause NetLogo to hang
- ◆ fixed some error messages to more accurately report the location of the error
- interface fixes:
 - ◆ when opening models from old NetLogo versions, `user-choice` is now auto-translated to `user-one-of`
 - ◆ in the 3D view, patch and turtle labels are now positioned correctly even if the origin is off-center
 - ◆ link direction indicators now show up in the 3D view and in HubNet view mirroring
 - ◆ fixed bug where the Procedures tab could lose the keyboard focus when the compiler finds an error
 - ◆ fixed bug where using the black arrows in the control strip to resize the world could cause an incorrect world size to appear in the strip
 - ◆ fixed bug in BehaviorSpace where failure to create an output file caused an uninformative error message

Version 3.1 (April 14, 2006)

- system:
 - ◆ the 3D view now works on Intel-based Macs
 - ◆ for Windows users, our bundled Java version is now 1.5 (was 1.4)
 - ◆ for Mac users, Java 1.5 is now used if it is available
- models:
 - ◆ new earth science model: Grand Canyon
 - ◆ new EvoLab evolution model: Sunflower Biomorphs
 - ◆ new social science model: Ethnocentrism
 - ◆ new mathematics model: Voronoi
 - ◆ new physics model: DLA Simple
 - ◆ new computer science models: Perceptron, Artificial Neural Network
 - ◆ new biology/system dynamics models: Tabonuco Yagrumo, Tabonuco Yagrumo Hybrid
 - ◆ new code examples: Moore & Von Neumann Example, Intersecting Lines Example, Diffuse Off Edges Example, Tie System Example
 - ◆ improved ProbLab model: Dice Stalagmite
 - ◆ improved social science models: Traffic Grid, Scatter
 - ◆ improved chemistry/physics model: GasLab Circular Particles (runs faster)
 - ◆ improved networks models (using experimental link primitives): Giant Component, Preferential Attachment, Small Worlds
- features:
 - ◆ randomized agent ordering: every agentset is now always in random order (a different random order each time you use it)
 - ◆ world topologies:
 - ◇ the world isn't always a torus anymore; by turning vertical and horizontal wrapping on or off, you can choose between a torus, a rectangle, or a vertical or horizontal cylinder
 - ◇ using primitives ending in `-nowrap` is no longer necessary; just turn off world wrapping instead
 - ◇ new `can-move?` reporter lets turtles sense the world edges

- ◊ `screen-edge-x` and `screen-edge-y` have been renamed to `max-pxcor` and `max-pycor` (and new `min-pxcor` and `min-pycor` primitives have been added)
- ◊ `screen-size-x` and `screen-size-y` have been renamed to `world-width` and `world-height`
- ◊ see the Topology section of the Programming Guide for more information
- ◆ new, experimental suite of link primitives, useful for network models and others; see Links sections of Programming Guide and Primitives Dictionary
- ◆ model authors can now specify the singular form of a breed name:
 - ◊ the new syntax for declaring each breed is e.g. `breed [wolves wolf]`
 - ◊ you can ask for a bred turtle by who number, e.g. `wolf 0`
 - ◊ in model output, bred turtles also appear as e.g. `wolf 0`
 - ◊ added new reporter e.g. `is-wolf?`
 - ◊ this form also appears elsewhere in the user interface, e.g. when picking a turtle from the view, in turtle monitors, and so on
- ◆ new command `stamp-erase`
- ◆ new `__tie` and `__untie` primitives allow turtles to connect their movement to another turtle. See the [Tie Section](#) of the Programming Guide for details.
- other language changes:
 - ◆ the `sort` and `sort-by` primitives can now be used to convert an agentset to a sorted list of agents; if you use `sort`, turtles are sorted by who number, and patches are sorted left-to-right, top-to-bottom
 - ◆ new reporters `random-xcor`, `random-ycor`, `random-pxcor`, and `random-pycor` are handy for generating random coordinates
 - ◆ `member?` can now be used to check whether an agent is a member of an agentset (worked in 3.0, but was undocumented)
 - ◆ `random-one-of` and `random-n-of` have been renamed to just `one-of` and `n-of`
 - ◆ removed `no-label` from the language; use the empty string instead
 - ◆ renamed `user-choice` to `user-one-of`, `user-choose-file` to `user-file`, `user-choose-new-file` to `user-new-file`, `user-choose-directory` to `user-directory`
- interface changes:
 - ◆ minor improvements to look & feel, e.g. vertical buttons in toolbars instead of horizontal ones
 - ◆ the info tabs of saved applets are now more attractive, as in the application
 - ◆ `user-one-of` (formerly `user-choice`) now always shows the choices as a menu, never buttons
- engine fixes:
 - ◆ fixed 3.0-only bug where the `repeat` and `let` commands didn't always work correctly when used from the Command Center
 - ◆ in the System Dynamics Modeler, `system-dynamics-t` is now incremented at the end of each step, not the beginning
- interface fixes:
 - ◆ double clicking outside a bracket or parenthesis in the editor now selects all of the text between the matching brackets or parentheses
 - ◆ when shapes are off, shapes consisting of a single line now still draw as lines, not squares
 - ◆ fixed some incorrectly worded compiler error messages involving primitives that can take a variable number of inputs, such as `list` and `sentence`
 - ◆ fixed Windows-only bug where the Procedures menu had trouble showing very large number of procedure names

- ♦ fixed Mac-only bug where pasting text from another application could cause a compiler error
- BehaviorSpace changes:
 - ♦ you can now use the `stop` command in an experiment's go commands, or in the procedure called by your experiment's go commands, to stop a model run, the same way it works to stop a forever button
 - ♦ you can now choose to save no results at all
 - ♦ the Abort button is now more often able to halt a stuck model
 - ♦ the Procedures tab is now always recompiled before an experiment begins
- HubNet changes:
 - ♦ new activities: Bug Hunters Camouflage, Walking, Disease Doctors, Function, Guppy Spots
 - ♦ improved activities: Predator Prey Game (formerly Herbivore Carnivore), Root Beer Game (formerly Beer Game)
- API changes:
 - ♦ documentation now recommends using the server VM for best performance

Version 3.0 (September 2005)

- 3D view (for 2D models)
- System Dynamics Modeler
- `follow`, `ride` and `watch` commands for tracking particular agents
- drawing layer for marks left by turtles
- buttons take turns now (instead of interleaving their code with each other)
- more muted, harmonious color palette
- more attractive Information tab
- GoGo extension for interfacing NetLogo with physical devices
- Color Swatches dialog helps you choose colors
- substantial improvements to BehaviorSpace
- commands for importing image files
- turtles can have cones of vision
- expanded controlling API

Version 2.1 (December 2004)

- much larger and higher quality library of turtle shapes
- runs models "headless", with no GUI, from the command line
- editor now highlights matching parentheses and brackets
- "action keys" let buttons be triggered by keypresses
- makes QuickTime movies of models
- redesigned Command Center for greater usability
- optional "output area" in models
- greatly improved shapes editor
- easy capture of images from Interface tab
- multilevel "Undo" in editor
- new `let` command for easy creation of local variables
- new `carefully` command for trapping runtime errors
- computer HubNet:
 - ♦ substantially improved reliability
 - ♦ "server discovery" is now fully supported

- ◆ you may serve multiple activities simultaneously from the same computer
- ◆ improved client interface and Control Center

Version 2.0.2 (August 2004)

- new, experimental "extensions" API lets users write new commands and reporters in Java
- NetLogo can now make sounds and music; this is done with a new, experimental sound extension that is also an example of how to use the extensions API
- new "controlling" API lets users control NetLogo from external Java code (such as for automating multiple runs)

Version 2.0 (December 2003)

- full support for Mac OS X; improved Linux support
- minimum Java version is now 1.4.1; Windows 95, MacOS 8, MacOS 9 no longer supported
- increased overall reliability
- improved look and feel throughout the application
- faster and more flexible graphics (labels, turtle sizes, exact turtle positions all now fast, reliable, and flicker-free)
- suite of primitives for reading and writing external files
- "strict math" mode now always on, for reproducible results
- export graphics window or interface tab as image file
- revamped BehaviorSpace (various improvements made; some old features are missing)
- Mersenne Twister random number generator
- many new primitives
- computer HubNet:
 - ◆ improved reliability; no longer alpha or beta
 - ◆ improved graphics window mirroring features and performance

Version 1.3 (June 2003)

- graphics window control strip
- choosers
- strict math mode so results are identical on all platforms (requires Java 1.3 or higher)
- new primitives including `run/runresult` and `map/foreach/filter/reduce`
- some primitives now accept a variable number of inputs

Version 1.2 (March 2003)

- alpha release of computer HubNet: formerly HubNet required the TI Navigator calculator network to operate; now you can use it over TCP/IP with networks of laptop or desktop computers
- new primitives and other language improvements
- display of coordinates when mousing over plots

Version 1.1 (July 2002)

- "Save as Applet" lets you embed your model in any web page

- printer support
- Procedures menu
- scrollable Interface tab
- contextual menus in Interface tab
- new primitives

Version 1.0 (April 2002)

- initial release (after a series of betas)

System Requirements

NetLogo is designed to run on almost any type of computer, but some older or less powerful systems are not supported. The exact requirements are summarized below. If you have any trouble with NetLogo not working on your system, we would like to offer assistance. Please write bugs@ccl.northwestern.edu.

System Requirements: Application

On all systems, approximately 25MB of free hard drive space is required.

Windows

- Windows NT, 98, ME, 2000, or XP
- 64 MB RAM (or probably more for NT/2000/XP)

You can choose to include a suitable Java Virtual Machine when downloading NetLogo. If you want to use a JVM that you install separately yourself, it must be version 1.4.1 or later. 1.5.0_05 or later is preferred.

Windows 95 is no longer supported by the current version of NetLogo. Windows 95 users should use NetLogo 1.3.1 instead. We will continue to support NetLogo 1.3.1.

Mac OS X

- OS X version 10.2.6 or later (10.3 or later is recommended)
- 128 MB RAM (256 MB RAM strongly recommended)

On OS X, the Java Virtual Machine is supplied by Apple as part of the operating system. OS X 10.3 includes an appropriate JVM. OS X 10.2 users must install Java 1.4.1 Update 1, which is available from Apple through Software Update.

For OS X 10.3 users, installing Java 1.4.2 Update 1 is recommended, for improved application reliability. The update is available from Apple through Software Update.

Mac OS 8 and 9

These operating systems are no longer supported by the current version of NetLogo. MacOS 8 and 9 users should download NetLogo 1.3.1 instead. We will continue to support NetLogo 1.3.1.

Other platforms

NetLogo should work on any platform on which a Java Virtual Machine, version 1.4.1 or later, is available and installed. Version 1.5.0_05 or later is preferred. If you have trouble, please contact us (see above).

System Requirements: Saved Applets

NetLogo models saved as Java applets should work on any web browser and platform on which a Java Virtual Machine, version 1.4.1 or later, is available. If you have trouble, please contact us (see above).

On Mac OS X, the Internet Explorer browser does not make use of the 1.4.1 JVM, so it cannot run saved applets. We suggest using Apple's Safari browser instead, or another web browser which uses the newer JVM.

Note that the 3D view is not available in applets.

System Requirements: 3D View

NetLogo's 3D view is a new feature, and hasn't been tested on every configuration. Below is information about configurations that we have tested so far.

Operating Systems

We've tested the 3D view on:

- Linux 2.6.8 (Debian i386)
- Linux 2.6.8 (Debian amd64)
- Mac OS X 10.3.9, 10.4.3
note: Java 1.4.2 is required
- Windows 2000
- Windows XP

If you use the 3D view on an operating system that we haven't tested, we'd like to hear about it. Please let us know at feedback@ccl.northwestern.edu. Please include the information in the System section of About NetLogo.

Graphics Cards

We've tested the 3D view on many different graphics cards and controllers, including:

- ATI Radeon 7500
- ATI Radeon 9200
- ATI Radeon 9600
- ATI Radeon 9800 XT
- ATI Radeon IGP 345
- ATI Radeon Mobility
- ATI FireGL V3100
- Intel 82830M
- nVidia GeForce MX
- nVidia GeForce FX 5200
- nVidia GeForce FX Go5650
- nVidia Quadro NVS

If you use the 3D view with a graphics card that we haven't tested, we'd like to hear about it. Please let us know at feedback@ccl.northwestern.edu. Please include the information in the System section of About NetLogo.

Fullscreen mode

Fullscreen mode does not work with some graphics cards and controllers, including the ATI Radeon IGP 345 and the Intel 82845.

Some users with older computers, especially laptops, have reported that entering fullscreen mode caused NetLogo to crash. If you experience this problem, please let us know.

Library Conflicts

NetLogo includes JOGL version 1.1.0 for the 3D View.

On Mac OS X and Windows, NetLogo uses the version of JOGL that comes with NetLogo, even if you have a different version of JOGL on your computer. If for some reason NetLogo is unable to find and use the correct version, it will warn you. If you get such a warning, you may need to remove your separate JOGL installation in order for NetLogo's 3D View to work.

On a Linux machine, if NetLogo is finding the wrong version of JOGL, trying running with the `-Djava.ext.dirs=` command line option, like this:

```
java -Djava.ext.dirs= -jar NetLogo.jar
```

That should fix the problem. If it doesn't, try removing your JOGL installation.

Removing an old JOGL

If NetLogo tells you you need to remove your JOGL installation, here's how to do it. You need to remove the `jogl.jar` file and one or two native library files:

- Remove `jogl.jar` from `lib/ext` in your Java home directory.
- On Mac OS X, remove `libjogl.jnilib` from `/Library/Java/Extensions` or `~/Library/Java/Extensions`.
- On Windows, remove `jogl.dll` and `jogl_cg.dll` from `jre/bin` in your Java home.
- On Linux, remove `libjogl.so` from your always-checked Java native libraries directory.

Known Issues

If NetLogo malfunctions, please send us a bug report. See the ["Contact Us"](#) section for instructions.

Known bugs (all systems)

- Integers in NetLogo must lie in the range `-2147483648` to `2147483647`; if you exceed this range, instead of a runtime error occurring, you get incorrect results
- Out-of-memory conditions are not handled gracefully
- The `stop` and `report` commands do not work properly if used inside `without-interruption` (we are already working on fixing this)
- The `uphill` and `downhill` reporters sometimes return incorrect answers for turtles which are standing on patch boundaries; we are already working on fixing this, but in the meantime you may wish to use `uphill4` and `downhill4` instead
- If you use "Export World" to suspend a model run and then resume it later with "Import World", this may change the outcome of the model run if your model involves turtles dying and new turtles being born, because the export/import may change what who numbers get assigned to new turtles (we are already working on fixing this)
- "Export World" does not include the contents of plots (we are already working on fixing this)
- Extensions that require additional external jars don't work from models saved as applets (we are already working on fixing this)
- The 3D View doesn't work on some graphics configurations; on others the 3D View works but 3D full screen mode doesn't
- A bug in Java causes patch colors imported using `import-pcolors` to be brighter than the original if the original image has a grayscale palette. To work around this issue, convert the image file to an RGB palette.

Windows-only bugs

- The "User Manual" item on the Help menu does not work on every machine (Windows 98 and ME are most likely to be affected, newer Windows versions less so)
- Drawing and then erasing a line in the drawing may not erase every pixel exactly.
- On some laptops, the Procedures and Info tabs may become garbled when you scroll them. To avoid this bug, reduce the size of the NetLogo window and/or reduce the color depth of your monitor (e.g. change from 32-bit to 16- or 8-bit color). This is a bug in Java itself, not in NetLogo per se. For technical details on the bug, see <http://developer.java.sun.com/developer/bugParade/bugs/4763448.html> (free registration required). NetLogo users are encouraged to visit that site and vote for Sun to fix this bug.

Macintosh-only bugs

- On Mac OS X 10.4 only, the "Copy View" and "Copy Interface" items may not work: the resulting image is distorted. The workaround is to use the "Export View" and "Export Interface" items instead. This issue will go away if you use Software Update to get the latest Java from Apple.
- When opening a model from the Finder (by double-clicking on it, or dragging it onto the NetLogo icon), if NetLogo is not already running, then the model may or may not open; the bug is intermittent. (If NetLogo is already running, the model always opens.)

- On versions of Mac OS X prior to 10.4, it is possible for NetLogo's menus to get confused so that the "Quit" item does not work. If this happens, you can quit NetLogo by pressing the red close button on the left end of the NetLogo's title bar.
- On Mac OS X 10.2 only, the "User Manual" item on the Help menu will sometimes launch a web browser other than your default browser
- On Mac OS X 10.2 only, opening the Models Library can trigger an error if you have malformed fonts installed. If this happens you should determine which fonts in /System/Library/Fonts and other font directories are causing the problem and remove them.

Linux/UNIX-only bugs

- User Manual always opens in Mozilla, not your default browser. One possible workaround is to bookmark the file docs/index.html in your favorite browser. Another workaround is to make a symlink that's called "mozilla" (that's the command name NetLogo tries to run), but actually runs a different browser.
- We have discovered a problem on Linux where the "exp" reporter sometimes returns a slightly different answer (differing only in the last decimal place) for the same input. According to an engineer at Sun, this should only happen on Linux kernel versions 2.4.19 and earlier, but we have observed the problem on more recent kernel versions. We assume the problem is Linux-specific and does not happen on other Unix-based systems. We are not sure if the problem ever occurs in practice during actual NetLogo model runs, or only occurs in the context of our testing regimen. The bug in the Sun's Java VM, and not in NetLogo itself. We hope that only the "exp" reporter is affected, but we can't be entirely certain of this. NetLogo users are encouraged to visit <http://developer.java.sun.com/developer/bugParade/bugs/5023712.html> (free registration required) and vote for Sun to fix this bug.
- If NetLogo cannot find the font Lucida, menus will be illegible. This has been known to happen on Fedora Core 3, after upgrading packages. Restarting the X Font Server (xfs) has resolved the problem in all reported cases.
- Sun's 1.5.0 Java runtime has display problems with GTK 2.0 and NetLogo. Issues may include windows not updating properly, interface elements being strangely sized, menus being cut-off at the bottom, and weird characters appearing on the view. To avoid these issues, you can use Sun's j2sdk1.4.2_10. Also, the JDK6 beta release from Sun does quite well.

Known issues with computer HubNet

See the [HubNet Guide](#) for a list of known issues with computer HubNet.

Unimplemented StarLogoT primitives

The following StarLogoT primitives are not available in NetLogo. (Note that many StarLogoT primitives, such as `count-turtles-with`, are intentionally not included in this list because NetLogo allows for the same functionality with the new `agentset` syntax.)

- `maxint`, `minint`, `maxnum`, `minnum`
- `import-turtles`, `import-patches`, `import-turtles-and-patches` (note that NetLogo adds `import-world`, though)

NetLogo 3.1.3 User Manual

- `bit`, `bitand`, `bitneg`, `bitor`, `bitset`, `bitstring`, `bitxor`, `make-bitarray`,
 `rotate-left`, `rotate-right`, `shift-left`, `shift-right`
- `camera-brightness`, `camera-click`, `camera-init`, `camera-set-brightness`
- `netlogo-directory`, `project-directory`, `project-name`, `project-pathname`,
 `save-project`

Contacting Us

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

Web Site

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

Feedback, Questions, Etc.

If you have general feedback, suggestions, or questions, write to feedback@ccl.northwestern.edu.

If you need help with your model, you should also consider posting to the NetLogo users group at <http://groups.yahoo.com/group/netlogo-users/>.

Reporting Bugs

If you would like to report a bug that you find in NetLogo, write to bugs@ccl.northwestern.edu. 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.
- Your system information: NetLogo version, OS version, Java version, and so on. This information is available from NetLogo's "About NetLogo" menu item. In saved applets, the same information is available by control-clicking (Mac) or right-clicking the white background of the applet.
- Any error messages that were displayed.

Sample Model: Party

This activity is designed to get you thinking about computer modeling and how you can use it. It also gives you some insight into the NetLogo software. We encourage beginning users to start with this activity.

At a Party

Have you ever been at a party and noticed how people cluster in groups? You may have also noticed that people do not stay within one group, but move throughout the party. As individuals move around the party, the groups change. If you watched these changes over time, you would notice patterns forming.

For example, in social settings, people tend to exhibit different behavior than when they are 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 quiet and reserved at work may be the "party starter" with friends.

The patterns may also depend on what kind of gathering it is. 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.

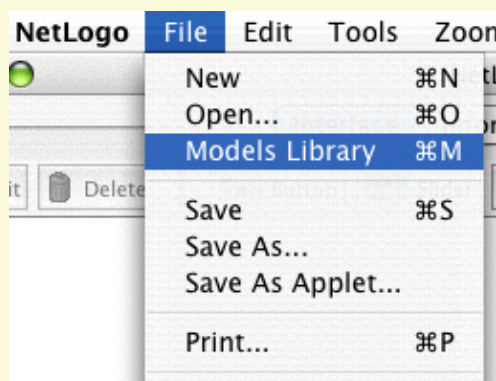
Is there any type of pattern to this kind of grouping?

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:

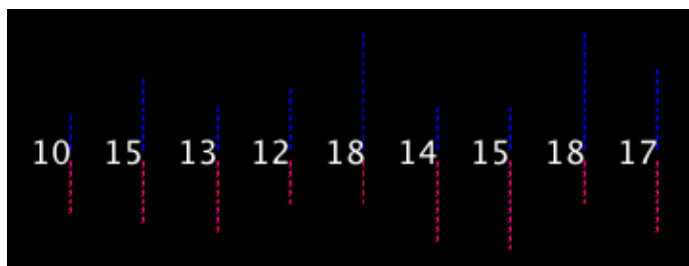
1. Start NetLogo.
2. Choose "Models Library" from the File menu.



3. Open the "Social Science" folder.
4. Click on the model called "Party".

5. Press the "open" button.
6. Wait for the model to finish loading
7. (optional) Make the NetLogo window bigger so you can see everything.
8. Press the "setup" button.

In the view, you will see pink and blue lines with numbers:



These lines represent mingling groups at a party. Men are represented in blue, women in pink. The numbers are the total number of people in each group.

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:

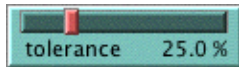
1. Press the "go" button. (Pressing "go" again will stop the model manually.)
2. Observe the movement of people until the model stops.
3. Watch the plots to see what's happening in another way.

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. Feel free to manipulate more than one variable at a time.

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 situations like this party scenario allows you to experiment with a system in a rapid and flexible way that would be difficult to do in a real world situation. 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](#), where you can learn how to alter and extend existing models to give them new behaviors, and 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 interact with 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.

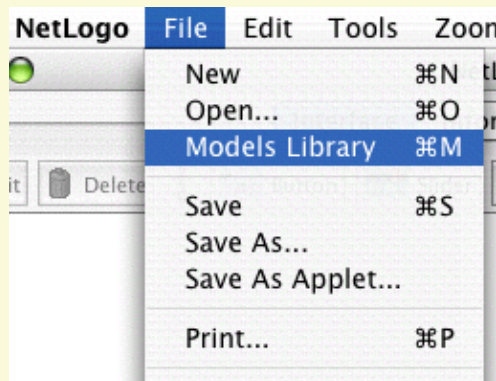
Throughout all of the tutorials, we'll be asking you to make predictions about what the effects of making changes to the models will be. Keep in mind that the effects are often surprising. We think these surprises are exciting and provide excellent opportunities for learning.

Some people have found it helpful to print out the tutorials in order to work through them. When the tutorials are printed out, there's more room on your computer screen for the NetLogo model you're looking at.

Sample Model: Wolf Sheep Predation

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

- Open the Models Library from the File menu.



- Choose "Wolf Sheep Predation" from the Biology section and press "Open".

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

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

When you first open the model, you will notice that the view 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?

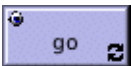
- Press the "go" button to start the simulation.

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

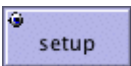
- Press the "go" button to stop the model.

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.

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 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 stop a model. It's safe to pause a model by stopping a forever button, then make it go on 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.

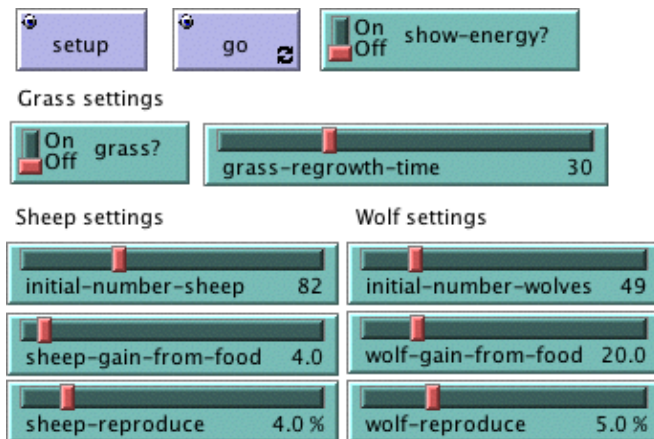
- If you like, experiment with the "setup" and "go" buttons in the Wolf Sheep Predation model.

Do you ever get different results if you run the model several times with the same settings?

Adjusting Settings: Sliders and Switches

The settings within a model give you an opportunity to work out different scenarios or hypotheses. Altering the settings and then running the model to see how it reacts to those changes can give you a deeper understanding of the phenomena being modeled. Switches and sliders give you access to a model's settings.

Here are the switches and sliders in Wolf Sheep Predation:



Let's experiment with their effect on the behavior of the model.

- Open Wolf Sheep Predation if it's not open already.
- Press "setup" and "go" and let the model run for about a 100 time-ticks. (Note: there is a readout of the number of ticks right above the plot.)
- 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 one of the settings.

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

What did this switch do to the model? Was the outcome the same as your previous run?

Just like buttons, switches have information attached to them. Their information is set up in an on/off format. Switches turn on/off a separate set of directions. These directions are usually not necessary for the model to run, but might add another dimension to the model. Turning the "grass?" switch on affected the outcome of the model. Prior to this run, the growth of the grass stayed constant. This is not a realistic look at the predator-prey relationship; so by setting and turning on a grass growth rate, we were able to model all three factors: sheep, wolf and grass populations.

Another type of setting is called a slider.

Sliders are a different type of setting than a switch. A switch has two values: on or off. A slider has a range of numeric values that can be adjusted. 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.

- Read the contents of the Information tab, located above the toolbar, to learn what each of this models' sliders represents.

The Information tab is extremely helpful for gaining 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 Information tab before running a model, or you might want to just start experimenting, then look at the Information tab later.

What would happen to the sheep population if there was more initial sheep and less initial wolves at the beginning of the simulation?

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

Try running the model several times with these settings.

What happened to the sheep population?

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

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

What happened to the wolves in this run?

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

(Note: in addition to sliders and switches, some models have a third kind of setting, called a chooser. The Wolf Sheep Predation doesn't have any of these, though.)

Gathering Information: Plots and Monitors

A purpose to modeling is to gather data on a subject or topic that would be very difficult to do in a laboratory situation. NetLogo has two main ways of displaying data to the user: plots and monitors.

Plots

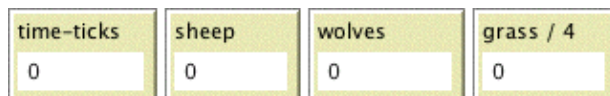
The plot in Wolf Sheep Predation contains three lines: sheep, wolves, and grass / 4. (The grass count is divided by four so it doesn't make the plot too tall.) The lines show what's happening in the model over time. To see which line is which, click on "Pens" in the upper right corner of the plot window to open the plot pens legend. A key appears that indicates what each line is plotting. In this case, it's the population counts.

When a plot gets close to becoming filled up, the horizontal axis increases in size 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 program, you can use the "Export Plot" item on the File menu. It saves this information to your computer in a format that can be read back by spreadsheet and database programs such as Excel. You can also export a plot by control-clicking (Mac) or right-clicking (Windows) it and choosing "Export..." from the popup menu.

Monitors

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



The monitor labeled "time-ticks" tells us how much time has passed in the model. The other 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 update continuously as the model runs, whereas the plots show you data from the whole course of the model run.

Note that NetLogo has also another kind of monitor, called "agent monitors". These will be introduced in Tutorial #2.

Controlling the View

If you look at the view, you'll see a strip of controls along the top edge. The control strip lets you control various 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 slider in the control strip back and forth.

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 all the way to the right again.
- Now try pressing and unpressing the red arrowhead in the control strip.
- Also try pressing and unpressing the on/off switch in the control strip.

What happens?

The shapes button and the freeze button are useful if you're impatient and want a model to run faster. When shapes are turned off, turtles are drawn as solid squares; it takes less work for NetLogo to draw squares than special shapes, so the model runs faster.

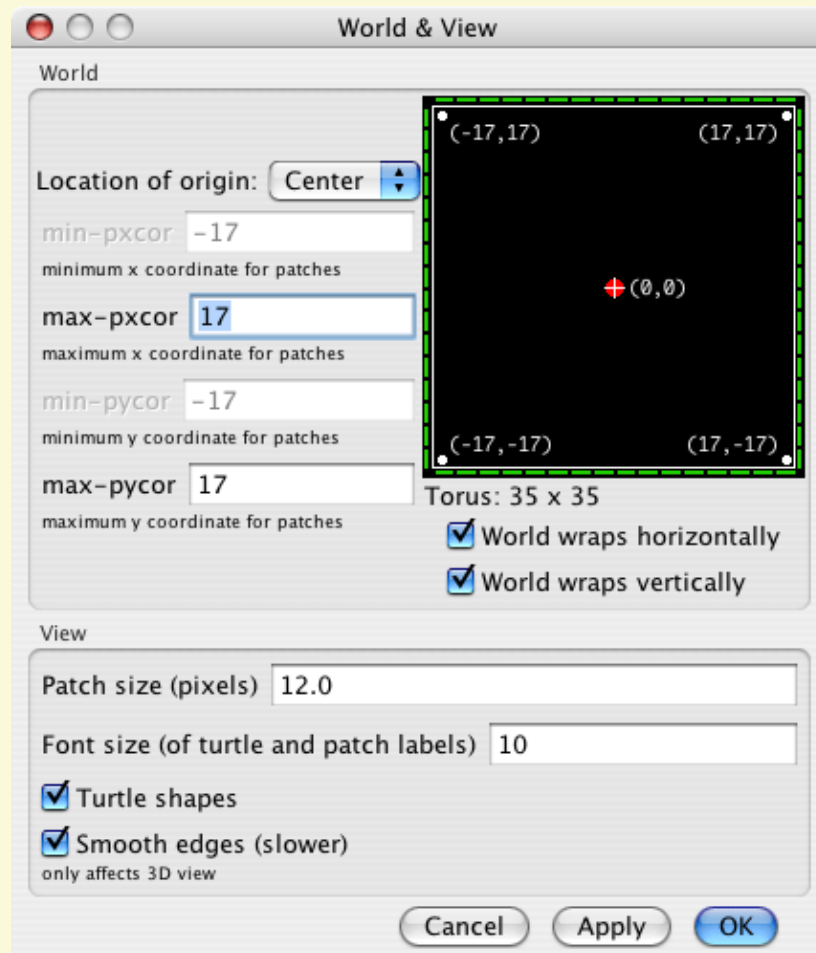
The freeze button "freezes" the view. The model continues to run in the background, and plots and monitors still update; but if you want to see what's happening, you need to unfreeze the view by turning the switch back on. Most models run much faster when the view is frozen.

The size of the view is determined by three separate settings: X Edge, Y Edge, 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 world and view settings than there's room for in the control strip. The "Edit..." button lets you get to the rest of the settings.

- Press the "Edit..." button in the control strip.

A dialog box will open containing all the settings for the view:



What are the current settings for max-pxcor, min-pxcor, max-pycor, min-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 "Edit..." button again and look at the settings.

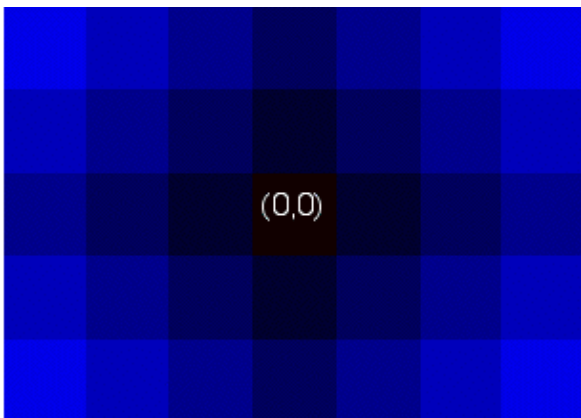
What numbers changed?

What numbers didn't change?

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

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

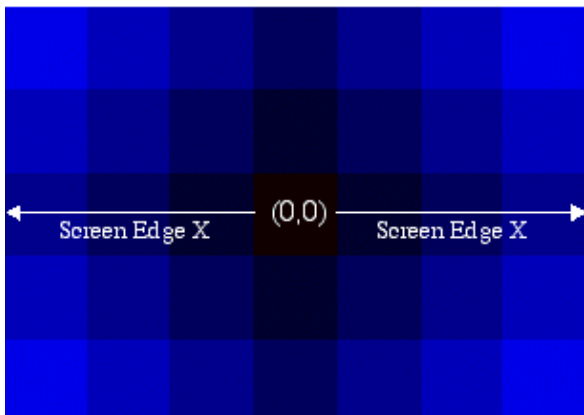
Think of the patches as being like square tiles in a room with a tile floor. By default, exactly in the middle of the room is a tile labeled (0,0); meaning that if the room was divided in half one way and then the other way, these two dividing lines would intersect on this tile. We now have a coordinate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

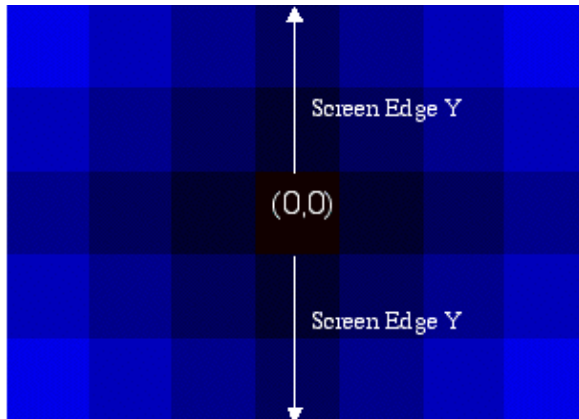


How many tiles away is the (0,0) tile from the right side of the room?

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 on the screen.

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

- Using the Edit 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 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, including the font size of labels and whether the view uses shapes. Feel free to experiment with these and other settings as well.

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.

The Models Library

The library contains five sections: Sample Models, Curricular Models, Code Examples, HubNet Calculator Activities, HubNet Computer Activities.

Sample Models

The Sample Models section is organized by subject area and currently contains more than 180 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 put a histogram within your model, you'd look at "Histogram Example" to find out how.

HubNet Calculator & Computer Activities

This section contains participatory simulations for use in the classroom. 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 your own models.

Tutorial #2: Commands

In Tutorial #1, you had the opportunity to view some of the NetLogo models, and you have successfully navigated your way through opening and running models, pressing buttons, changing slider and switch values, 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

- Go to the Models Library (File menu).
- Open up Traffic Basic, found in the "Social Science" section.
- Run the model for a couple minutes to get a feel for it.
- Consult the Information tab for any questions you may have about this model.

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 any cause such as an accident, a broken bridge, or an overturned truck. No "centralized cause" is needed for a traffic jam to form.

You may alter the settings and observe a few runs to get a full 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 cosmetic and would enhance the look of the model while the 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 Procedures tab.)

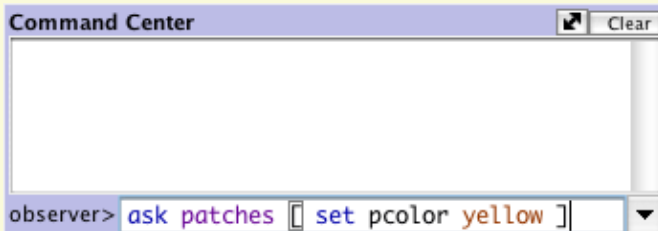
To make these simple changes we will be using the Command Center.

The Command Center

The Command Center is located in the Interface Tab and allows you to enter commands or directions to the model. Commands are instructions you can give to NetLogo's agents: turtles, patches, and the observer. (Refer to the [Interface Guide](#) for details explaining the different parts of the Command Center.)

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?

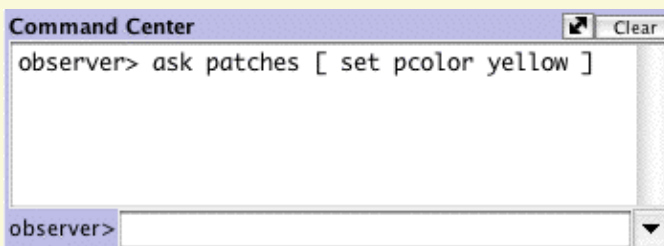
You may have noticed the background of the View has turned all yellow and the street has disappeared.

Why didn't the cars turn yellow too?

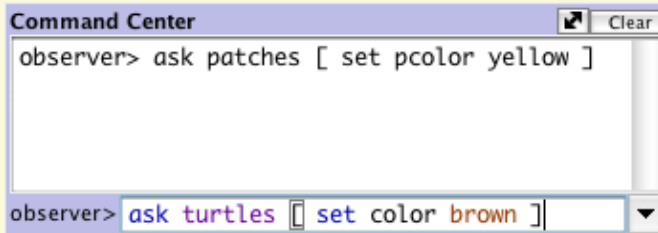
Looking back at the command that was written, we asked only the patches to change their color. In this model, the cars are represented by a different kind of agent, called "turtles". Therefore, the cars did not received these instructions and thus did not change.

What happened in the Command Center?

You may have noticed that the command you just typed is now displayed in the white box in the middle of the Command Center as shown below:

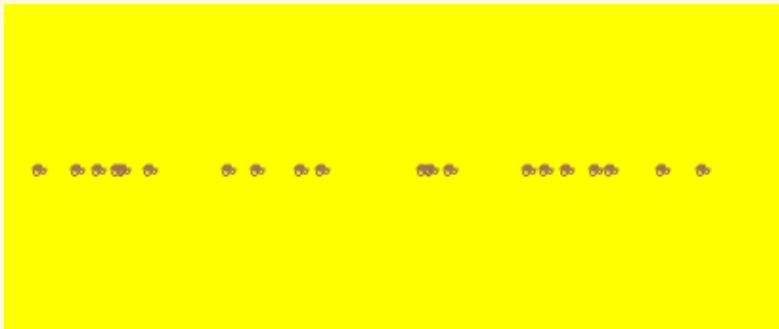


- Type in the white box at the bottom of the Command Center the text shown below:



Was the result what you expected?

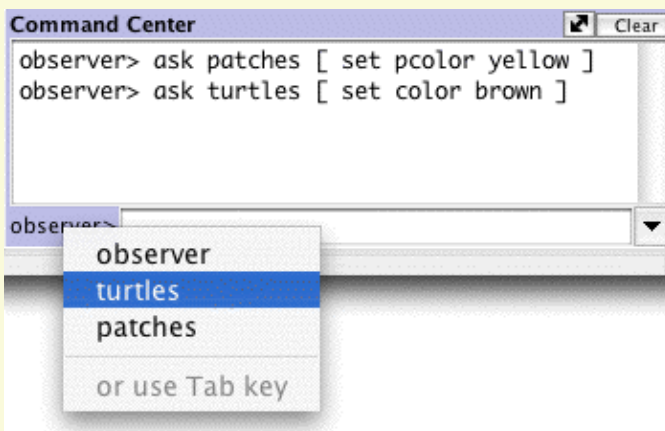
Your View should have a yellow background with a line of brown cars in the middle:



The NetLogo world is a two dimensional world that is made up of turtles, patches and an observer. The patches create the ground in which the turtles can move around on and the observer is a being that oversee everything that is going on in the world. (For a detailed description and specifics about this world, refer to the [NetLogo Programming Guide](#).)

In the Command Center, we have the ability to give the observer a command, the turtles a command, or the patches a command. We choose between these options by using the popup menu located in the bottom left corner of the Command Center. You can also use the tab key on your keyboard to cycle through the different options.

- 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 Procedures tab. The Command Center is not often used to permanently change the model. It is most often used as a tool to customize current models and allows for you to manipulate the NetLogo world to further answer those "What if" questions that pop up as you are investigating the models. (The Procedures 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`.

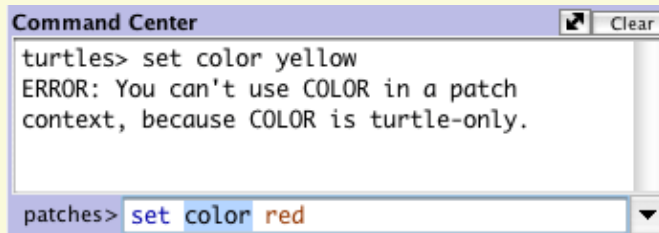
What is the difference between `color` and `pcolor`?

- 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:



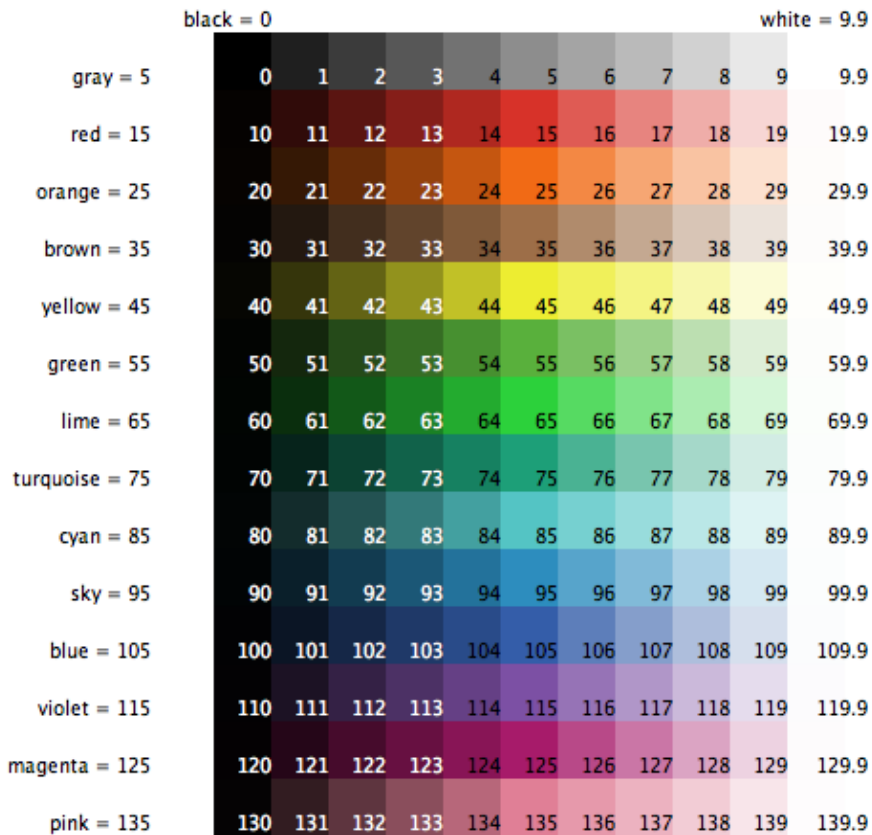
- Type `set pcolor red` instead and press return.

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, all 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 a color that doesn't have its own name, you just refer to it by a number instead, or by adding or subtracting a number from a name. For example, when you type `set color red`, this does the same thing as if you had typed `set color 15`. And you can get a lighter or darker version of the same color by using a number that is a little larger or a little smaller, as follows.

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

By subtracting from red, you make it darker.

- Type `set pcolor red + 2`

By adding to red, you make it lighter.

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

Agent Monitors and Agent Commanders

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

- Press "setup" to get the red car to reappear.
- If you are on a Macintosh, hold down the Control key and click on the red car. On other operating systems, click on the red car with the right mouse button.
- From the popup menu that appears, choose "inspect turtle 0"

A turtle monitor for that car will appear:

turtle 0	
who	0
color	15.0
heading	90.0
xcor	6.772186068086915
ycor	0.0
shape	"car"
label	
label-color	9.9999
breed	turtles
hidden?	false
size	1.0
pen-size	1.0
pen-mode	"up"
speed	6.1
speed-limit	1
speed-min	0

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 place that holds a value that can be changed. Remember when it was mentioned that all colors are represented in the computer as numbers? The same is true for the agents. For example, turtles have an ID number we call their "who" number.

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 who that has a who number of 0, a color of 15 (red — see above chart), and the shape of a car.

There are two other ways to open a turtle monitor besides right-clicking (or control-clicking, depending on your operating system). 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 (Macintosh) or upper right hand corner (other operating systems).

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

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

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

What happens in the View?

Did anything change in the Turtle Monitor?

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

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

What happened?

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

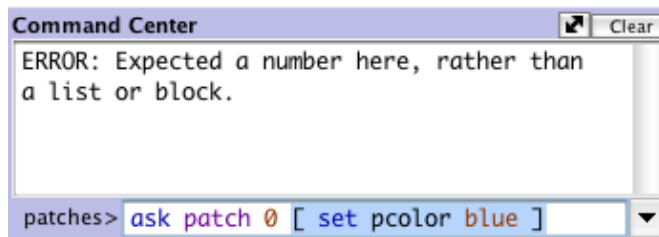
- In the Command Center, select "observer" from the popup menu (or use the tab key).
- Type `ask turtle 0 [set color blue]` and press return.

What happens?

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

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

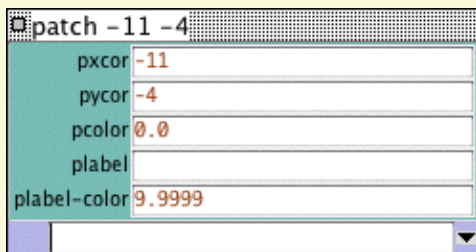
If you try to have the observer ask `patch 0 [set pcolor blue]`, you'll get an error message:



To ask an individual turtle to do something, we use its who number. But patches don't have who numbers, therefore we need to refer to them some other way.

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

- Open a patch monitor for any patch.



The monitor shows that for the patch in the picture, its `pxcor` variable is `-11` and its `pycor` variable is `-4`. If we go back to the analogy of the coordinate plane and wanted to plot this point, the point would be found in the lower left quadrant of the coordinate plane where $x=-11$ and $y=-4$.

To tell this particular patch to change color, use its coordinates.

- 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 -11 -4 [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

NetLogo User Manual

This tutorial leads you through the process of building a complete model, built up stage by stage, 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 Procedures tab.

You've already used types of agents you can give commands to in NetLogo: patches, turtles, and the observer. Patches are stationary and arranged in a grid. Turtles move over that grid. The observer oversees everything that's going on and does whatever the turtles and patches can't do for themselves.

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

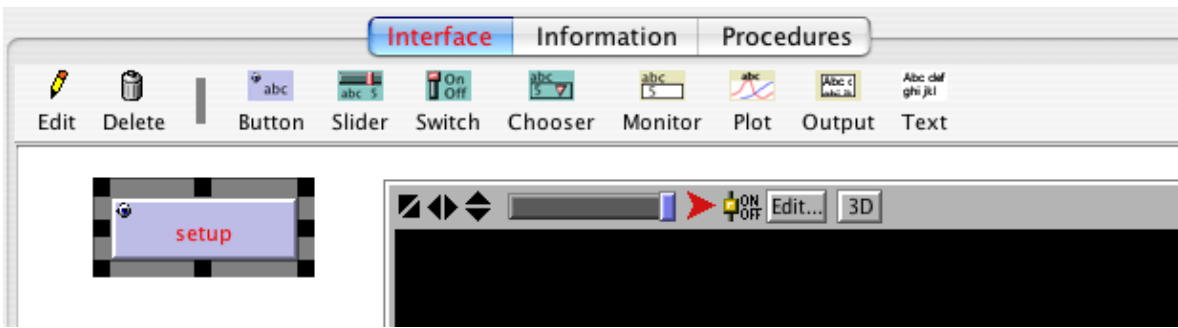
You will now learn to write procedures that make turtles move, eat, reproduce, and die. You will also learn how to make monitors, sliders, and plots. The model we'll build is a simple ecosystem model not unlike parts of Wolf Sheep Predation from Tutorial #1.

Making the setup button

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

- Click the "Button" icon in the toolbar at the top of the Interface tab.
- 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 haven't defined that procedure yet (we will soon). Because the button refers to a procedure that doesn't exist yet, the button turns red:



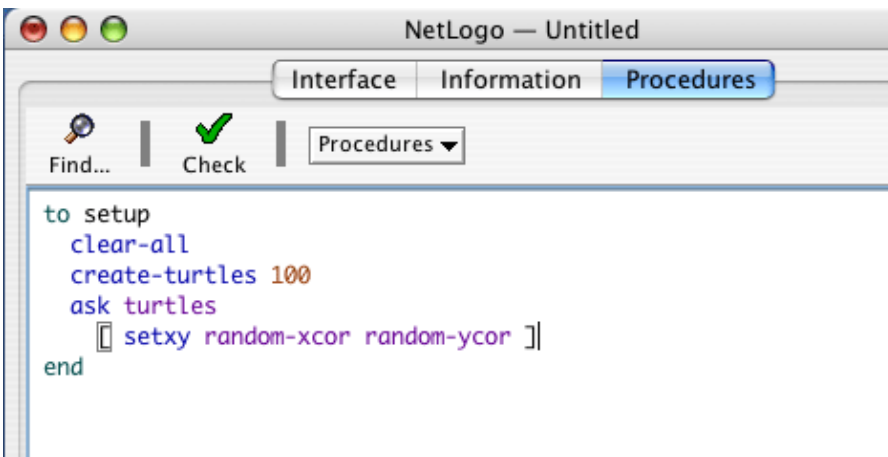
If you want to see the actual error message, click the button.

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

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

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

When you're done, the Procedures tab looks like this:



Note that the lines are indented different amounts. Most people find it helpful to indent their code like this, but it is not mandatory. It makes the code easier to read and change. Your procedure began with the word `to` and ended with the word `end`. Every new procedure you create will begin and end with these two words.

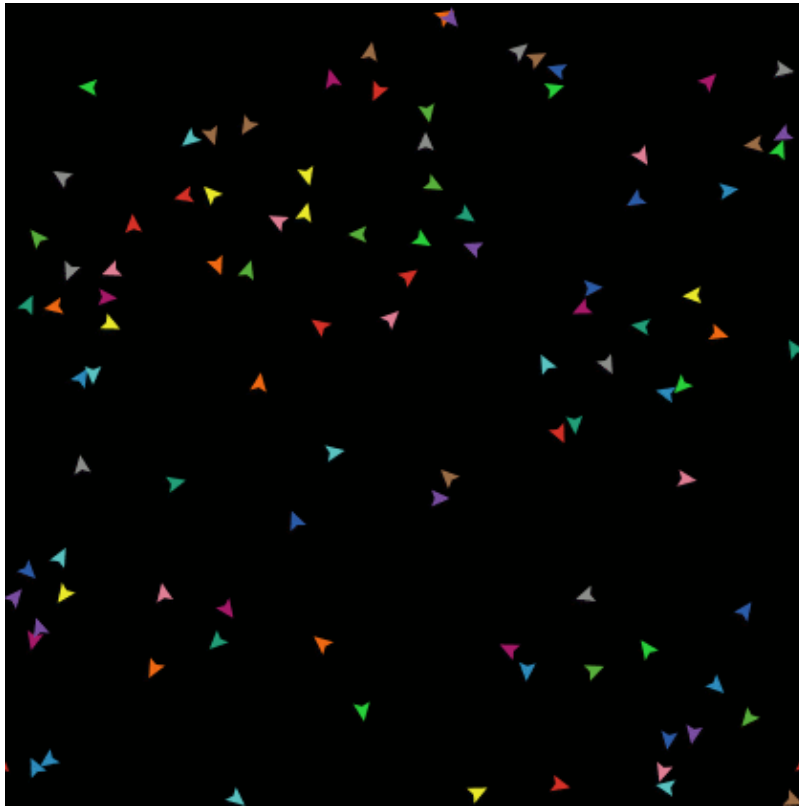
Let's look at what you typed in and see what each line of your procedure does:

- `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.
- `ask turtles [...]` tells each turtle to run, independently, the commands inside the brackets. (Every command in NetLogo is run by some agent. `ask` is a command too. Here, the observer is running the `ask` command itself, in turn causing the turtles to run commands.)
- `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.
- `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.

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.



The "forever" checkbox makes the button stay down once pressed, so its commands run over and over again, not just once.

- Then add a go procedure to the Procedures tab:

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

But what is `move-turtles`? Is it a primitive (in other words, built in to NetLogo), like `clear-all` is? 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
end

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

Note there are no spaces around the dash 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>`, 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
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-xxcor 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 Procedures tab, before all the procedures. Call it `energy`:

```
turtles-own [energy]
```

```

to go
  move-turtles
  eat-grass
end

```

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

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

```

to go
  move-turtles
  eat-grass
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 the grass.

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

- Rewrite `move-turtles` as follows:

```

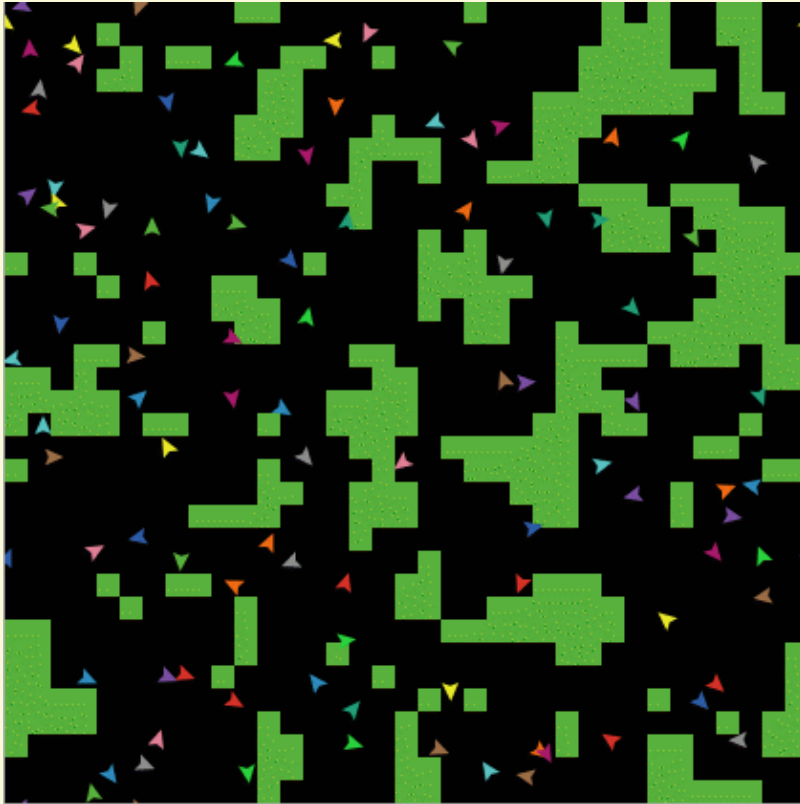
to move-turtles
  ask turtles [
    right random 360
    forward 1
    set energy energy - 1
  ]
end

```

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.



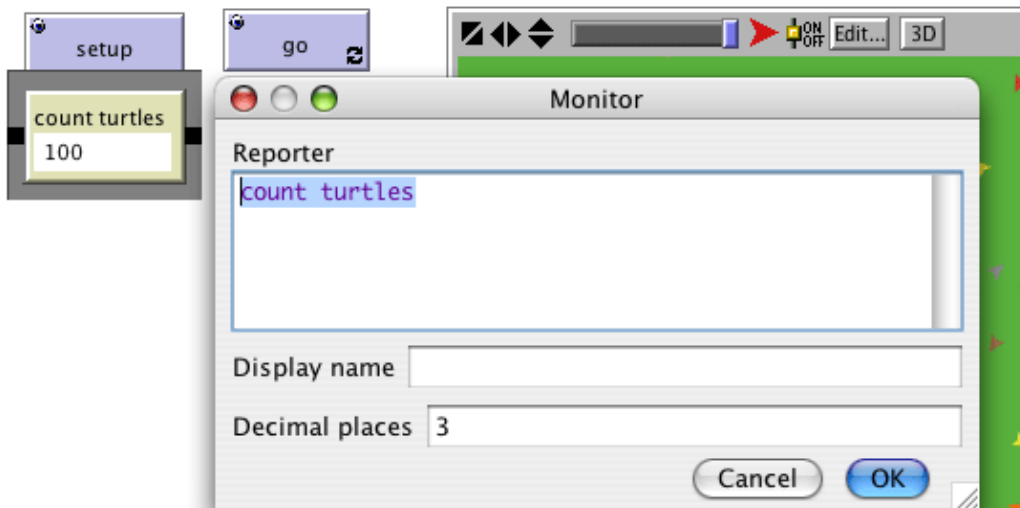
Monitors

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

- Create a monitor, using the monitor icon on the Toolbar and click on an open spot in the Interface.

A dialog box will appear.

- In the dialog box type: `count turtles` (see image below).
- Press the OK button to close the dialog box.



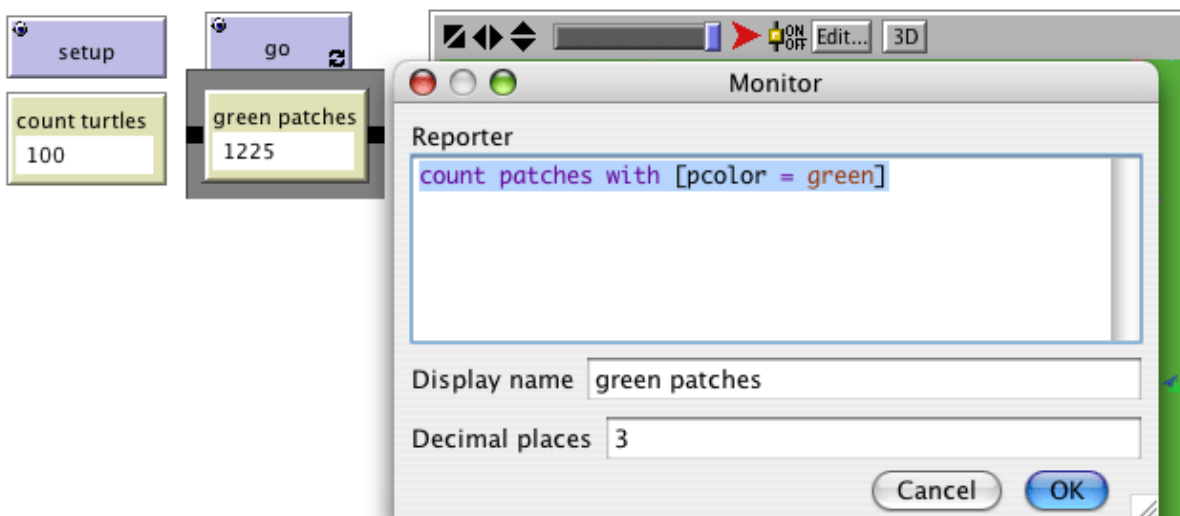
`turtles` is an "agentset", the set of all turtles. `count` tells us how many agents are in that set.

Let's make the second monitor now:

- Create a monitor, using the monitor icon on the Toolbar and click 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

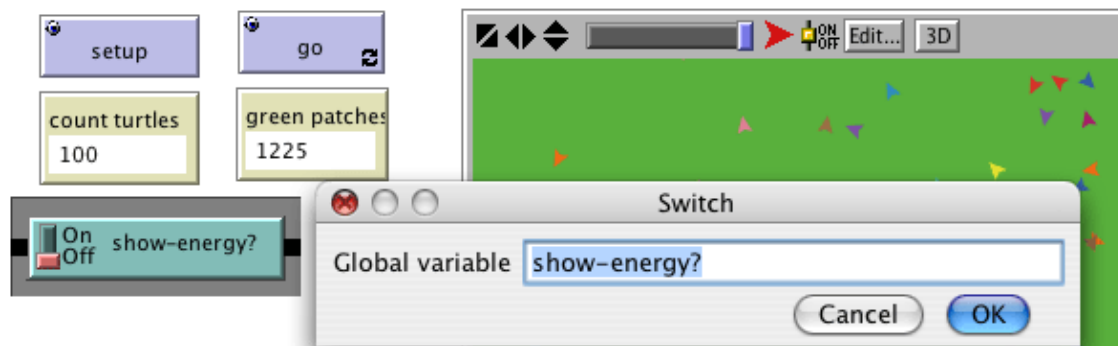
The turtles aren't just turning the patches black; they're also gaining and losing energy. As the model runs, try using a turtle monitor to watch one turtle's energy go up and down.

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.

- To create a switch, click on the switch icon on the Toolbar (in the Interface tab) and click on an open spot in the Interface.

A dialog box will appear.

- In the Global variable section of the dialog box 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 Procedures tab with the Toolbar.
- Rewrite the `eat-grass` procedure as follows:

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

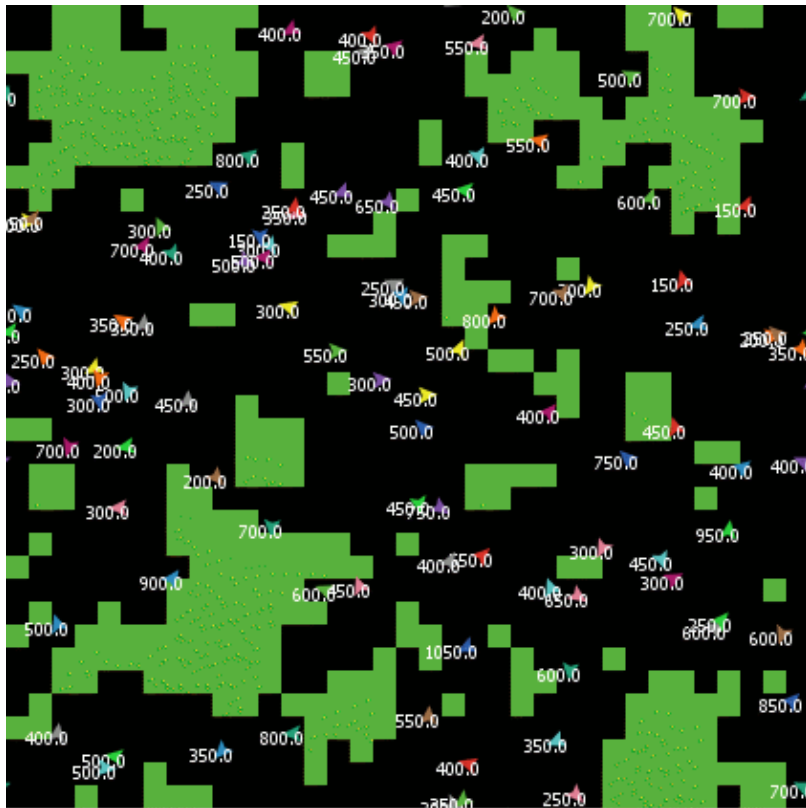
```
        set energy (energy + 10)
      ]
    ifelse show-energy?
      [ set label energy ]
      [ set label "" ]
    ]
  end
```

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". A string is a sequence of letters and 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. Let's also make the grass grow back. We'll add all three of these behaviors now, by making three separate procedures, one for each behavior.

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

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

```

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 `reproduce`, 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 turtle 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 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 set some settings in it. Then we'll add one more procedure to the Procedures tab, which will update the plot for us.

Let's do the Procedures tab part first.

- Change `setup` to call the new procedure, `do-plots`, which we're about to add:

```
to setup
  clear-all
  setup-patches
  setup-turtles
  do-plots
end
```

- Also, change `go` to call the `do-plots` procedure:

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

- Now add the new procedure. What we're plotting will be the number of turtles and the number of green patches versus time. At each time step (a single run through the `go` procedure) these values are added to the plot.

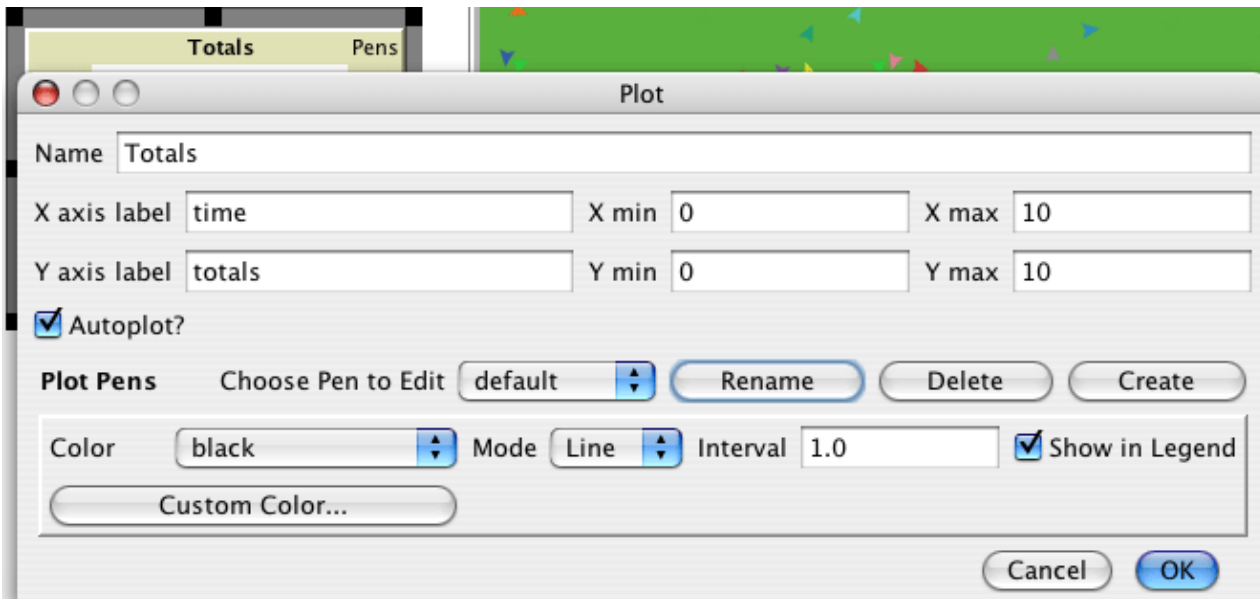
```
to do-plots
  set-current-plot "Totals"
  set-current-plot-pen "turtles"
  plot count turtles
  set-current-plot-pen "grass"
  plot count patches with [pcolor = green]
end
```

Note that we use the `plot` command to add the next point to a plot. However, before doing that, we need to tell NetLogo two things. First, we need to specify what plot we will be using (since later our model might have more than one plot) and second, we need to specify which pen we want to plot with (we will be using two pens on this plot).

The `plot` 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.

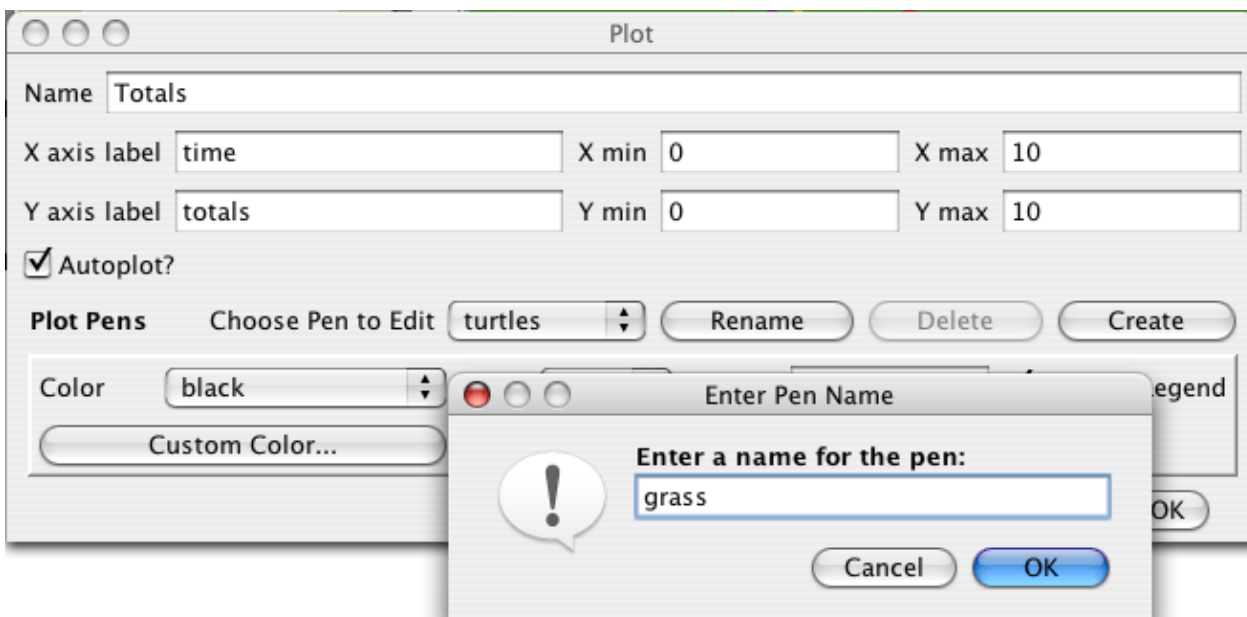
In order for `set-current-plot "Totals"` to work, you'll have to add a plot to your model in the Interface tab, then edit it so its name is the same name used in the procedures. Even one extra space in the name will throw it off — it must be exactly the same in both places.

- Create a plot, using the plot icon on the Toolbar and click 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 "total"



Next you will need to create two pens.

- With the Plot dialog box still open, press the 'Create' button in the Plot dialog, to create a new pen.
- Enter the name of this pen as "turtles" and press OK in the "Enter Pen Name" dialog. (see image below)
- Press the 'Create' button in the Plot dialog again, to create a second new pen.
- Enter the name of this pen as "grass" and press OK in the "Enter Pen Name" dialog. (see image below)
- Select the color for this pen and change it to green.
- Select OK in the Plot dialog box.

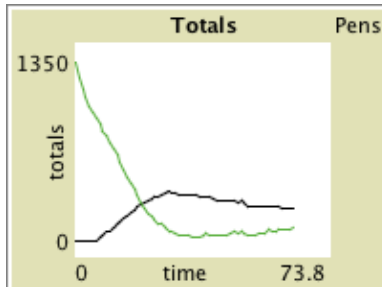


Note that when you create the plot you can also set the minimum and maximum values on the X and Y axes. You'll want to leave the "Autoplot?" checkbox checked, so that if anything you plot exceeds the minimum and maximum values for the axes, the axes will automatically grow so you can see all the data.

- 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 "Autoplot?" on. This allows the plot to readjust itself when it runs out of room.



If you forget which pen is which, click on the Pens label on the top right corner of the plot. You might try running the model several times to see what aspects of the plot are the same and which are different.

Global variables

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.

To keep track of this, we will need to add a global variable.

- Change the first few lines of your procedures to include a new first line that declares a global variable:

```
globals [ticks]
turtles-own [energy]
```

- Next, change the `setup` procedure:

```
to setup
  clear-all
  set ticks 0
```

```

    setup-patches
    setup-turtles
    do-plots
  end

```

- Finally, change the `go` procedure:

```

to go
  if ticks > 500 [ stop ]
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
  do-plots
  set ticks (ticks + 1)
end

```

- Now setup and run the model.

The graph and model won't keep running forever. They should stop automatically when the ticks counter reaches 500.

How would you add a monitor to report the ticks? How would you change your procedures so that the model stops when the ticks counter reaches 100?

Some more details

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

- Make a slider variable called 'number', using the monitor icon on the Toolbar 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
  ask turtles [ setxy random-xxcor 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 plots. 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 [Primitives 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 Procedures tab, comments are gray, so your eyes can pick them out easily.

```
globals [ticks]          ;; for keeping track of how many times the
                          ;; go procedure has been run
turtles-own [energy]      ;; for keeping track of when the turtle is ready
                          ;; to reproduce and when it will die

to setup
  clear-all
  set ticks 0             ;; resets the counter to zero when setup is pressed
  setup-patches
  setup-turtles
  do-plots
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-xxcor random-yyor ]
end

to go
  if ticks > 500 [ stop ]  ;; only run these procedures if ticks is less than 500
  move-turtles
  eat-grass
  reproduce
  check-death
  regrow-grass
  do-plots
  set ticks (ticks + 1)    ;; increase the tick counter by 1 each time through
end

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

```

]
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

to do-plots
  set-current-plot "Totals" ;; which plot we want to use next
  set-current-plot-pen "turtles" ;; which pen we want to use next
  plot count turtles ;; what will be plotted by the current pen
  set-current-plot-pen "grass" ;; which pen we want to use next
  plot count patches with [pcolor = green] ;; what will be plotted by the current pen
end

```


Interface Guide

This section of the manual walks you through every element of the NetLogo interface in order and explains its function.

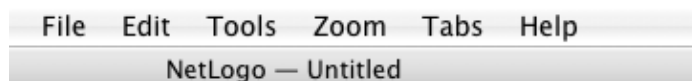
In NetLogo, you have the choice of viewing models found in the Models Library, adding to existing models, or creating your own models. The NetLogo interface was designed to meet all these needs.

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

- [Menus](#)
- [Tabs](#)
- [Interface Tab](#)
 - ◆ [Working with Interface Elements](#)
 - ◆ [The 2D and 3D Views](#)
 - ◆ [Command Center](#)
 - ◆ [Plots](#)
- [Procedures Tab](#)
- [Information Tab](#)

Menus

On Macs, if you are running the NetLogo application, the menubar is located at the top of the screen. On other platforms, the menubar 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

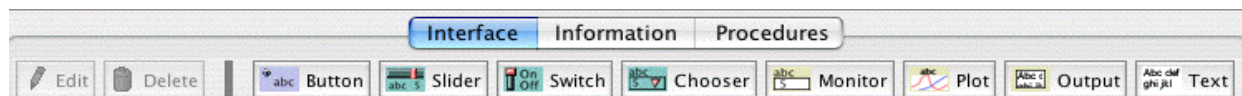
File		
	New	Starts a new model.
	Open	Opens any NetLogo model on your computer.
	Models Library	A collection of demonstration models.
	Save	Save the current model.
	Save As	Save the current model using a different name.
	Save As Applet	Used to save a web page in HTML format that has your model embedded in it as a Java "applet".
	Print	Sends the contents of the currently showing tab to your printer.
	Export World	Saves all variables, the current state of all turtles and patches, the drawing and the output area to a file.
	Export Plot	Saves the data in a plot to a file.
	Export All Plots	Saves the data in all the plots to a file.
	Export View	Save a picture of the current view (2D or 3D) to a file (in PNG format).

	Export Interface	Save a picture of the current Interface tab. (in PNG format)
	Export Output	Save the contents of the output area or the output section of the command center to a file.
	Import World	Load a file that was saved by Export World.
	Import Patch Colors	Load an image into the patches, see the import-pcolors command.
	Import Drawing	Load an image into the drawing, see the import-drawing command.
	Quit	Exits NetLogo. (On Macs, this item is on the NetLogo menu instead.)
Edit		
	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.
	Undo	Undo last text editing action you performed.
	Redo	Redo last undo action you performed.
	Select All	Select all the text in the active window.
	Find	Finds a word or sequence of characters within the Information or Procedures tabs.
	Find Next	Find the next occurrence of the word or sequence you last used Find with.
	Shift Left / Shift Right	Used in the Procedures tab to change the indentation level of code.
	Comment / Uncomment	Used in the Procedures tab to add or remove semicolons from code (semicolons are used in NetLogo code to indicate comments).
Tools		
	Halt	Stops all running code, including buttons and the command center. (Warning: since the code is interrupted in the middle of whatever it was doing, you may get unexpected results if you try to continue running the model without first pressing "setup" to start the model run over.)
	Globals Monitor	Displays the values of all global variables.
	Turtle Monitor	Displays the values of all of the variables in a particular turtle. You can can also edit the values of the turtle's variables and issue commands to the turtle. (You can also open a turtle monitor via the View; see the View section below.)
	Patch Monitor	Displays the values of all of the variables in a particular patch. You can can also edit the values of the patch's variables and issue commands to the patch. (You can also open a patch monitor via the View; see the View section below.)
	Hide/Show Command Center	Makes the command center visible or invisible. (Note that the command center can also be shown or hidden, or resized, with the mouse.)
	3D View	Opens the 3D view. See the View section for more information.
	Color Swatches	

		Opens the Color Swatches. See the Color Section of the Programming Guide for details.
	Shapes Editor	Draw turtle shapes. See the Shapes Editor Guide for more information.
	BehaviorSpace	Runs the model over and over with different settings. See the BehaviorSpace Guide for more information.
	System Dynamics Modeler	Opens the System Dynamics Modeler. See the System Dynamics Modeler Guide for more details.
	HubNet Control Center	Disabled if no HubNet activity is open. See the HubNet Guide for more information.
Zoom		
	Larger	Increase the overall screen size of the model. Useful on large monitors or when using a projector in front of a group.
	Normal Size	Reset the screen size of the model to the normal size.
	Smaller	Decrease the overall screen size of the model.
Tabs		This menu offers keyboard shortcuts for each of the tabs. (On Macs, it's Command 1 through Command 3. On Windows, it's Control 1 through Control 3.)
Help		
	About NetLogo	Information on the current NetLogo version the user is running. (On Macs, this menu item is on the NetLogo menu instead.)
	User Manual	Opens this manual in a web browser.

Tabs

At the top of NetLogo's main window are three tabs labeled "Interface", "Information" and "Procedures". 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 buttons. The buttons available vary from tab to tab.

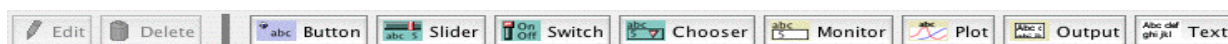
Interface Tab

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

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

Working with Interface Elements

The toolbar on the Interface tab contains buttons that let you edit, delete, and create items in the Interface tab (such as buttons and sliders).



The buttons in the toolbar are described below.

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

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

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

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


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

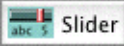


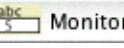

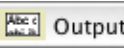
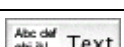
Resizing: Select the interface element, then drag the black "handles" in the selection border.

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

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

Chart: Interface Toolbar

Icon & Name	Description
	Buttons can be either <i>once-only</i> buttons or <i>forever</i> buttons. When you click on a once button, it executes its instructions once. The forever button executes the instructions over and over, until you click on the button again to stop the action. If you have assigned an action key to the button, pressing the corresponding keyboard key will act just like a button press when the button is in focus. Buttons with action keys have a letter in the upper right corner of the button to show what the action key is. If the input cursor is in another interface element such as the Command Center, pressing the action key won't trigger the button. The letter in the upper right hand corner of the button will be dimmed in this situation. To enable action keys, click in the white background of the Interface tab.

 Slider	Sliders are global variables, which are accessible by all agents. They are used in models as a quick way to change a variable without having to recode the procedure every time. Instead, the user moves the slider to a value and observes what happens in the model.
 Switch	Switches are a visual representation for a true/false variable. The user is asked to set the variable to either on (true) or off (false) by flipping the switch.
 Chooser	Choosers let the user choose a value for a global variable from a list of choices, presented in a drop down menu.
 Monitor	Monitors display the value of any expression. The expression could be a variable, a complex expression, or a call to a reporter. Monitors automatically update several times per second.
 Plot	Plots are real-time graphs of 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.
 Text	Text boxes lets you add informative text labels to the Interface tab. The contents of text boxes do not change as the model runs.

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, another visual representation of the world, by clicking on the "3D" button in the View Control Strip.



There are a number of settings associated with the Views. There are a few ways of changing the settings: by using the control strip along the top edge of the View, or by editing the 2D View, as described in the "Working With Interface Elements" section above, or pressing the "Edit..." button in the control strip.

The 3D View has a similar control strip but it looks slightly different and as you may notice a few of the controls are missing. However, the controls that are present work exactly the same as the 2D View Control Strip.

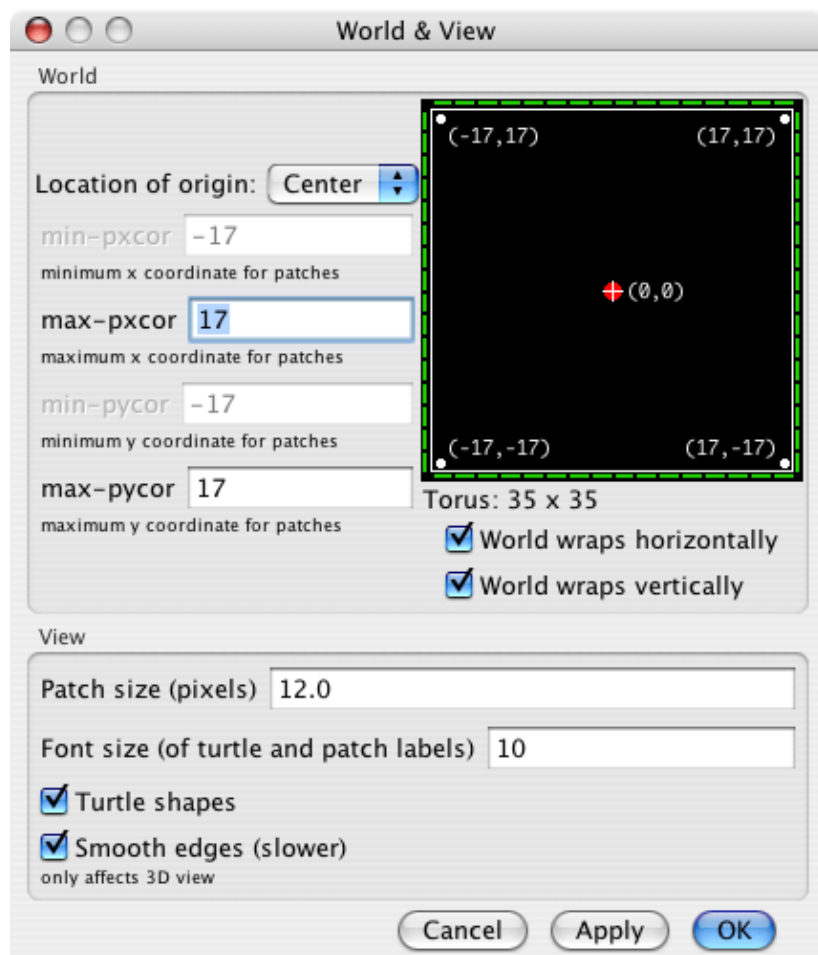


The controls in the control strip work as follows:

- The three sets of black arrows let you change the size of the world. When the origin is centered the world will grow in increments of two, adding one to the maximum and subtracting one from the minimum. If one of the edges is set to 0 the world will grow by one in the other direction to keep the origin along the edge. If the origin is at a custom location the black arrows will be disabled.
- The slider lets you control how fast the model runs — this is valuable since some models run so fast that it's hard to see what's going on.

- The button with the arrowhead lets you turn turtle "shapes" on and off. If shapes are off, turtles appear as colored squares, instead of having special shapes. The squares are less work for the computer to draw, so turning shapes off makes models run faster.
- The on-off switch lets you temporarily "freeze" the display. The model keeps running, but the contents of the view don't change until you unfreeze it by flipping the switch again. Most models run much faster when the view is frozen.
- The 3D button switches to the 3D View (see below).

Here are the settings for the View (accessible by editing the View, or by pressing the "Edit..." button in the control strip):



Notice that the settings are broken up into two groups. There are World setting and View settings. World settings affect the properties of the world that the turtles live in (changing them may require resetting the world). View settings only affect the appearance of view, changing them will not affect the outcome of the model.

The world settings allow you to define the boundaries and topology of the world. At the top of the left side of the world panel you can choose a location for the origin of the world either "Center", "Corner", "Edge", or "Custom". By default the world has a center configuration where (0,0) is at the center of the world and the user defines 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 Min-Pxcor will automatically be set to -10 thus there are 10 patches to the left of the origin and 10 patches to the right of patch 0 0.

A Corner configuration allows the user 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 the user to place the origin at any location in the world, though patch 0 0 must still exist in the world.

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

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

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

The "Turtle Shapes" checkbox performs the same function as the shapes button in the control strip, discussed above.

The "Smooth edges" checkbox controls the use of anti-aliasing in the 3D view only. It will make the lines appear less jagged but it will slow down the model.

Turtle and patch monitors are easily available through the View, just control-click (Macintosh) or right-click (other systems) 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 and patch monitors can also be opened from the Tools menu or by using the `inspect` command.)

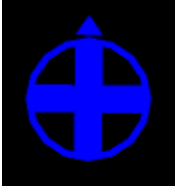
Some NetLogo models let you interact with the turtles and patches with your mouse by clicking and dragging in the View.

Manipulating the 3D View

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



A blue cross appears at the current focus point as you are adjusting these settings. The little blue triangle will always point up the positive y-axis, so you can orient yourself in case you get lost. It's easy to do!



To look at the world from a different angle, press the "rotate" button click and drag the mouse up, down, left, or right. 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 along the 3D View. (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 up, down, left, and right inside the 3D View while holding down the mouse button.

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

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

Fullscreen Mode

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

Note: Fullscreen mode doesn't work on some computers. It depends on what kind of graphics card you have. See the [System Requirements](#) for details.

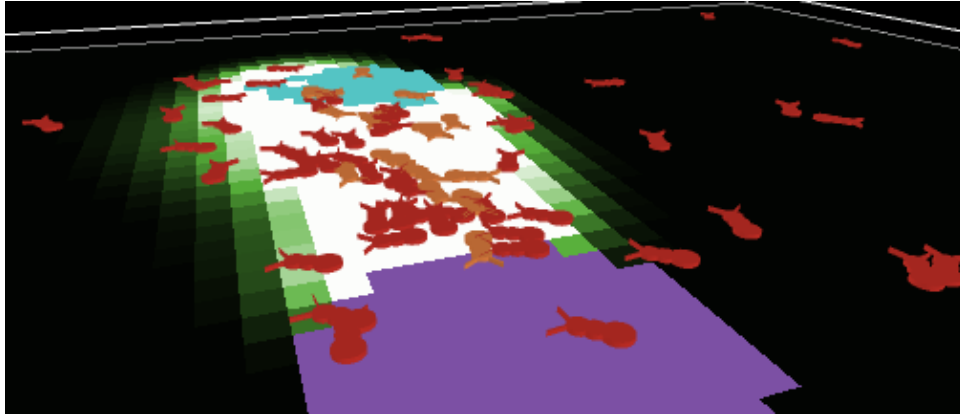
3D Shapes

Some shapes have true 3D counterparts (a 3D circle is actually a sphere) in the 3D view so they are automatically mapped to that shape.

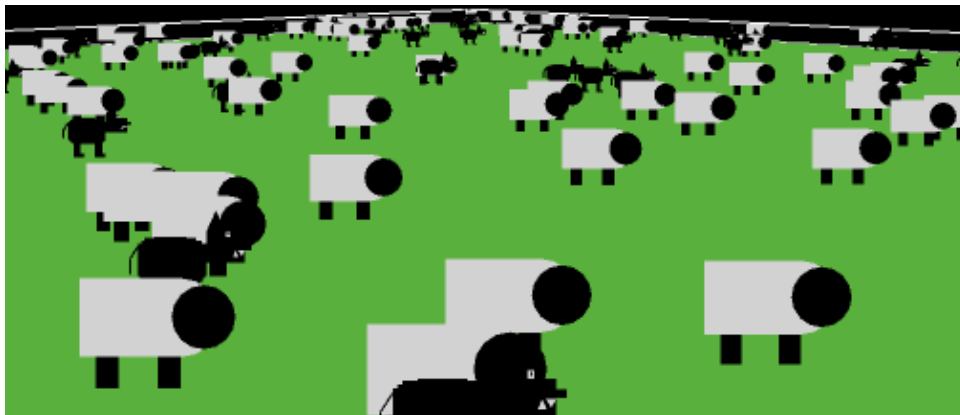
Shape name	3D shape
default	3D turtle shape
circle	sphere
dot	small sphere
square	cube
triangle	cone
line	3D line
cylinder	3D cylinder

line-half	3D line-half
car	3D car

All other shapes are interpreted from their 2D shapes. 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.



If a shape is non-rotatable it is assumed to be a side view so it is drawn always facing the observer (and with no thickness), as in Wolf Sheep Predation.

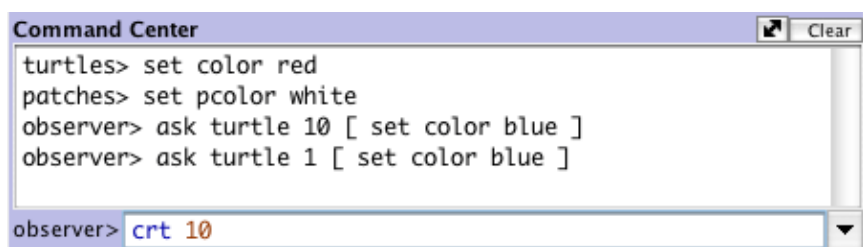


Command Center

The Command Center allows you to issue commands directly, without adding them to the model's procedures. (Commands are instructions you give to the agents in your model.) 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 text box, below the large box, is where you type a command. After typing it press the Return or Enter key to run it.

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

Tip: a quicker way to change between observer, turtles, and patches is to use the tab key on your keyboard.

Accessing previous commands

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

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

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

Clearing

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

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

Arranging

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

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

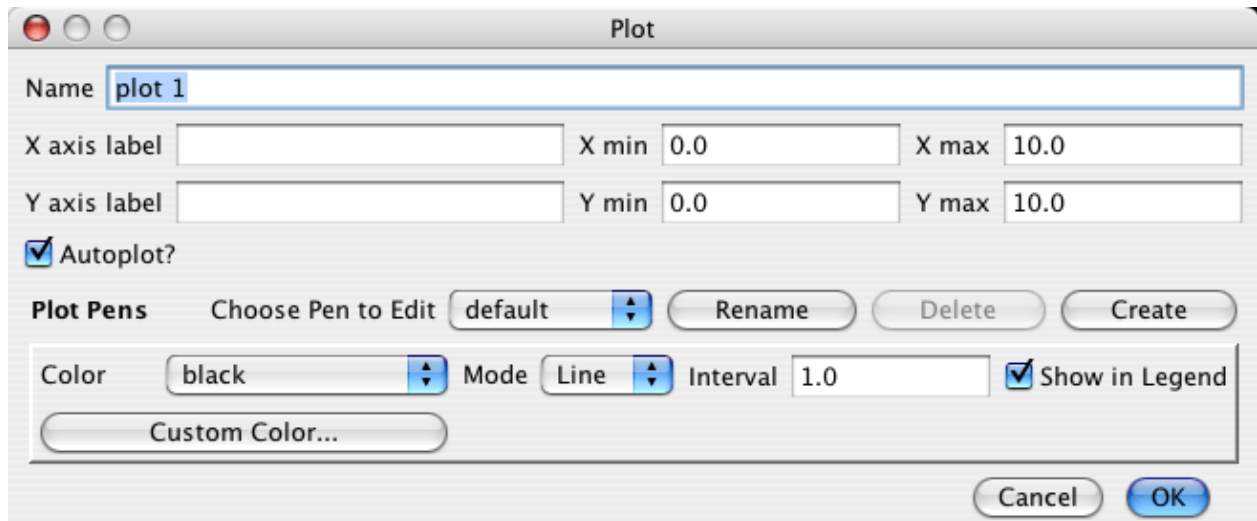
To switch between a vertical command center and a horizontal one, click the button with the double-headed arrow, just to the left of "Clear".

Plots

To show or hide a plot's pens legend, click on the word "Pens" in the upper right corner of a plot.

If you move the mouse over the white area of a plot, the x and y coordinates of the mouse location will 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, the name of the plot, labels for the x and y axes, and ranges for the axes.

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

In the plot pens section of the dialog you can create and customize different pens in this plot. You must always have a least one pen in every plot. You start out with one named "default" you probably want to rename it something that is meaningful in the model.

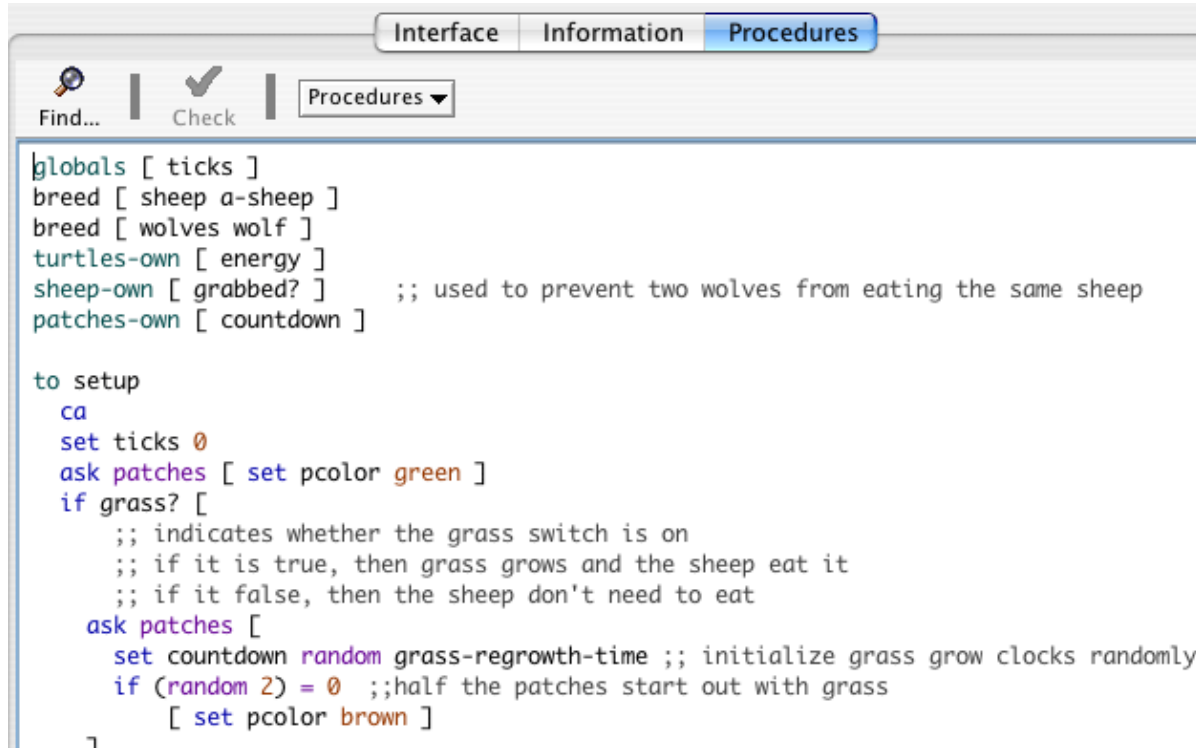
All the items in the box below the pen name are settings relevant to that particular pen.

- Set the color to one of the NetLogo base hues or a custom color using the color swatches.
- Mode allows you to change the appearance of the plot pen, line, bar (like a bar chart), or point (like line except the points are not connected)
- Interval is the amount by which x advances every time you use plot y
- If the Show in Legend checkbox is checked the selected pen will be a part of the legend in the upper right hand corner of the plot (which can be revealed by clicking on the word "Pens" on the plot itself).

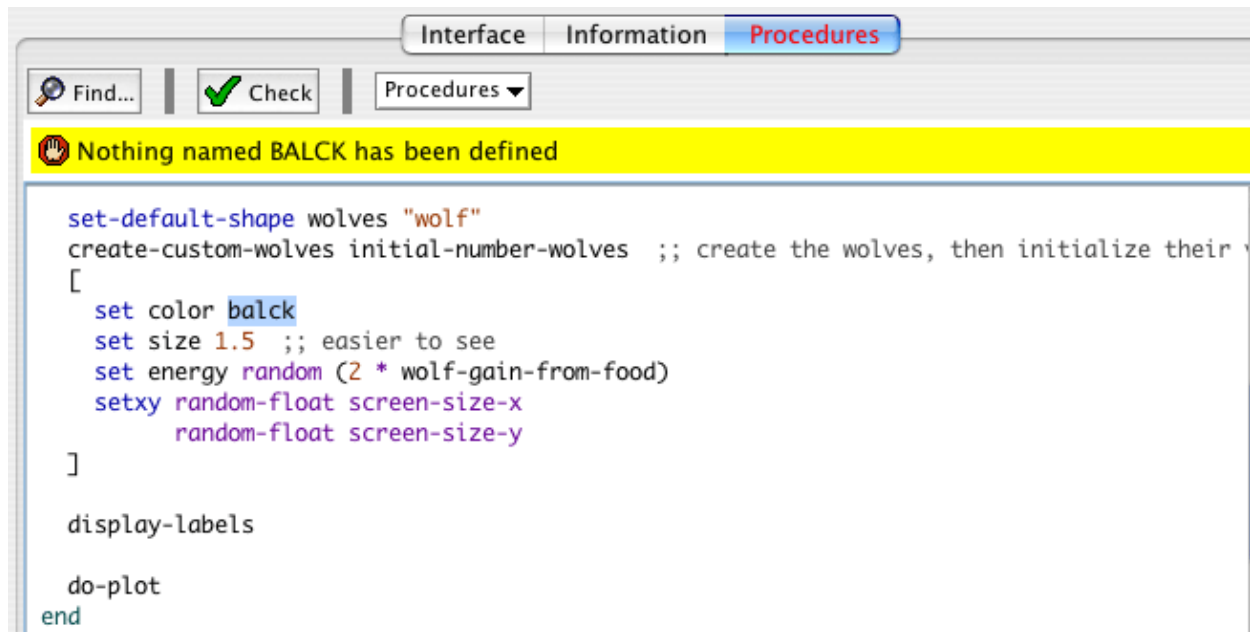
For more detailed information on how each of these features works you can see the [Plotting Section](#) of the Programming Guide.

Procedures Tab

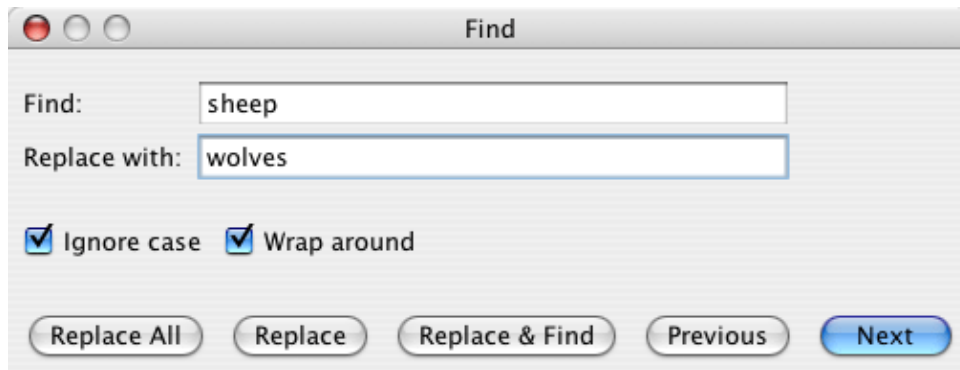
This tab is the workspace where the code for the model is stored. Commands you only want to use immediately go in the Command Center; commands you want to save and use later, over and over again, are found in the Procedures tab.



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



To find a fragment of code in the procedures, click on the "Find" button in the Procedures Toolbar and the Find dialog will appear.



You may enter either a word or phrase to find or a word or phrase to find and one 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 Procedures 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 Procedures 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 the find phrase in the search area with the replace phrase.

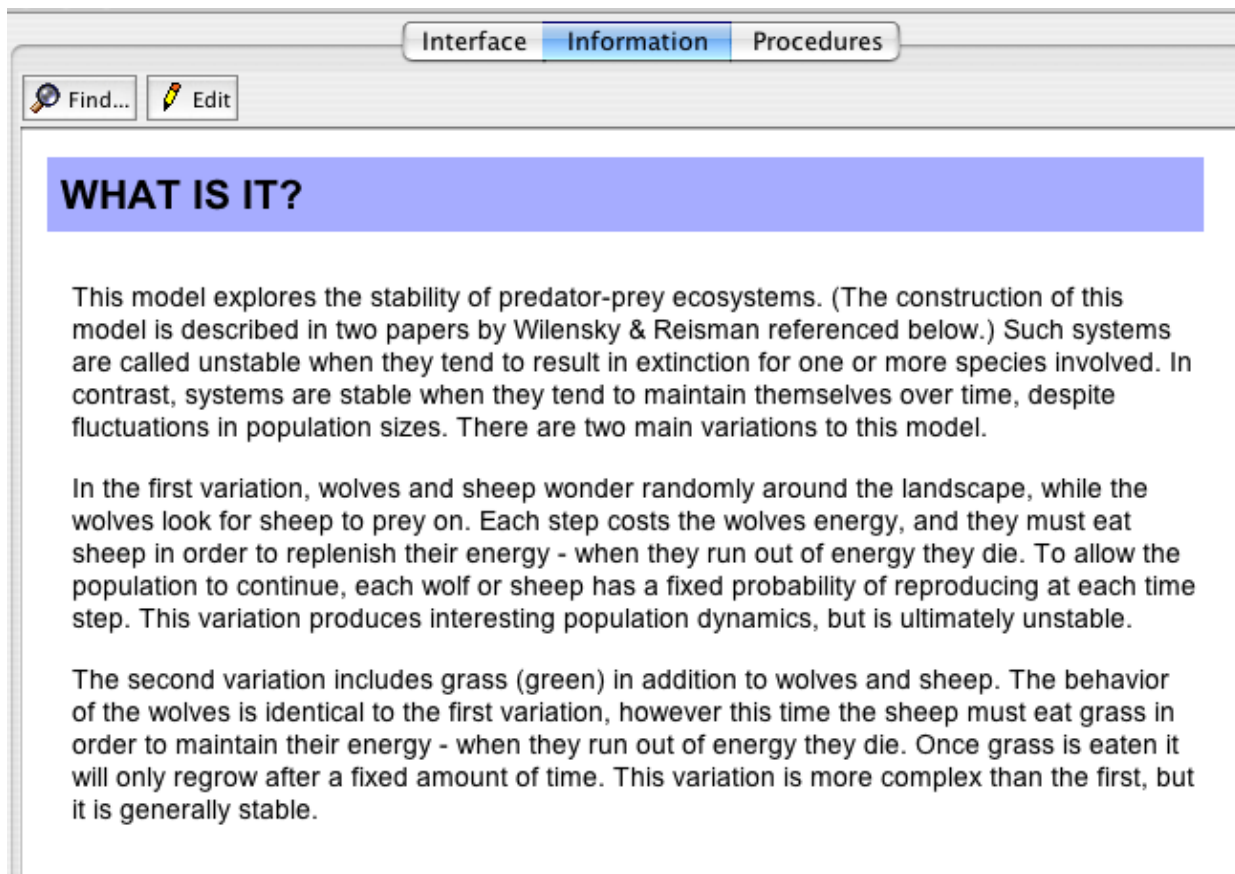
To find a particular procedure definition in your code, use the "Procedures" popup menu in the Procedures Toolbar. The menu lists all procedures in alphabetical order.

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

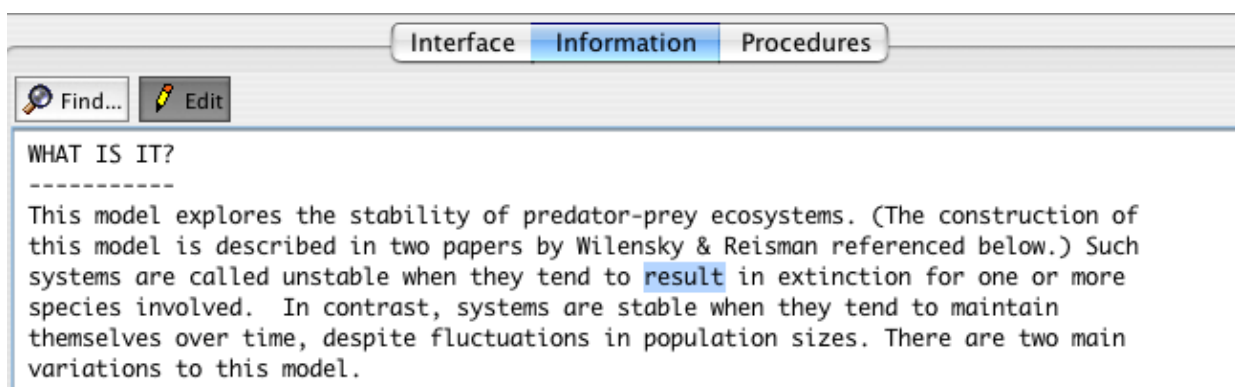
For more information about writing procedures, read [Tutorial #3: Procedures](#) and the [Programming Guide](#).

Information Tab

The Information tab provides an introduction to the model and an explanation of how to use it, things to explore, possible extensions, and NetLogo features. It is very helpful when you're first exploring a model.



We recommend reading the Information tab before starting the model. The Information tab explains what principle is being modeled and how the model was created. This display of the Information tab is not editable. To edit the content of the Info tab click the "Edit" button or double click on a word which will also scroll you to the location you clicked on and highlight the word.



You can edit the text in this view as in any text editor. However, a few different forms will be displayed specially when you switch out of the edit view.

Information Tab Markup

Description	Edit Mode	View Mode
Lines that come after blank lines and contain capital letters and no lower case letters become section headers.	WHAT IS IT	WHAT IS IT
Any line that has only dashes is omitted.	-----	
Anything beginning with "http://" becomes a clickable hyperlink.	http://ccl.northwestern.edu	http://ccl.northwestern.edu
E-mail addresses become clickable "mailto:" links.	bugs@ccl.northwestern.edu	bugs@ccl.northwestern.edu
Lines that begin with the pipe ' ' (shift + backslash '\ ') become monospaced text. This is useful for diagrams and complicated formulas, among other things.	this is preformatted text you can put spaces in it	this is preformatted text you can put spaces in it

To return to the normal view, click the edit button.

Programming Guide

The following material explains some important features of programming in NetLogo.

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

- [Agents](#)
- [Procedures](#)
- [Variables](#)
- [Colors](#)
- [Ask](#)
- [Agentsets](#)
- [Breeds](#)
- [Synchronization](#)
- [Buttons](#)
- [Lists](#)
- [Math](#)
- [Random Numbers](#)
- [Turtle Shapes](#)
- [Plotting](#)
- [Strings](#)
- [Output](#)
- [File I/O](#)
- [Movies](#)
- [Perspective](#)
- [Drawing](#)
- [Topology](#)
- [Links](#)
- [Tie](#)

Agents

The NetLogo world is made up of agents. Agents are beings that can follow instructions. Each agent can carry out its own activity, all simultaneously.

In NetLogo, there are three types of agents: turtles, patches, 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. The observer doesn't have a location — you can imagine it as looking out over the world of turtles and patches.

When NetLogo starts up, there are no turtles yet. 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 and the observer are.)

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 `-17`, `17`, `-17`, and `17` respectively. This means that `pxcor` and `pycor` both range from `-17` to `17`, so there are 35 times 35, or 1225 patches total. (You can change the number of patches by editing NetLogo's view.)

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.

The world of patches can have different shapes. 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 shape of the world by adjusting the wrap settings by editing the view. 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. See the [Topology](#) section for more information.

Procedures

In NetLogo, commands and reporters tell agents what to do. A **command** is an action for an agent to carry out. A **reporter** computes a result and report it.

Most commands begin with verbs ("create", "die", "jump", "inspect", "clear"), while most reporters are nouns or noun phrases.

Commands and reporters built into NetLogo are called **primitives**. [The Primitives 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`. 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.

Examples: Here are two command procedures:

```
to setup
  ca                ;; clear the world
  crt 10            ;; make 10 new turtles
end

to go
  ask turtles
  [ fd 1            ;; all turtles move forward one step
    rt random 10    ;; ...and turn a random amount
    lt random 10 ]
end
```


Note the use of semicolons to add "comments" to the program. Comments make your program easier to read and understand.

In this program,

- `setup` and `go` are user-defined commands.
- `ca` ("clear all"), `crt` ("create turtles"), `ask`, `lt` ("left turn"), and `rt` ("right turn") 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. 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 must specify which agents -- turtles, patches, or the observer -- 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.

`ca` and `crt` 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`, 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

Your own procedures can take inputs, just like primitives do. To create a procedure that accepts inputs, include a list of input names in square brackets after the procedure name. For example:

```
to draw-polygon [num-sides len]
  pd
  repeat num-sides
    [ fd len
      rt (360 / num-sides) ]
end
```

Elsewhere in the program, you could ask turtles to each draw an octagon with a side length equal to its ID-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 0 - number ]
```

end

Variables

Variables are places to store values (such as numbers). A variable can be a global variable, a turtle variable, or a patch variable.

If a variable is a global variable, there is only one value for the variable, and every agent can access it. But each turtle has its own value for every turtle variable, and each patch has its own value for every patch variable.

Some variables are built into NetLogo. For example, all turtles 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.) If you set the variable, the turtle or patch changes color. (See next section for details.)

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

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

```
globals [ clock ]
```

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

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

These variables can then be used freely in your model. Use the `set` command to set them. (If you don't set them, they'll start out storing a 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 or set a different agent's variable, you put `-of` after the variable name and then specify which agent you mean. Examples:

```
set color-of turtle 5 red
;; turtle with ID number 5 turns red
set pcolor-of patch 2 3 green
;; patch with pxcor of 2 and pycor of 3 turns green
ask turtles [ set pcolor-of patch-at 1 0 blue ]
;; every turtle turns the patch to its east blue
ask patches with [any? turtles-here]
  [ set color-of one-of turtles-here yellow ]
;; on every patch, a random turtle turns yellow
```

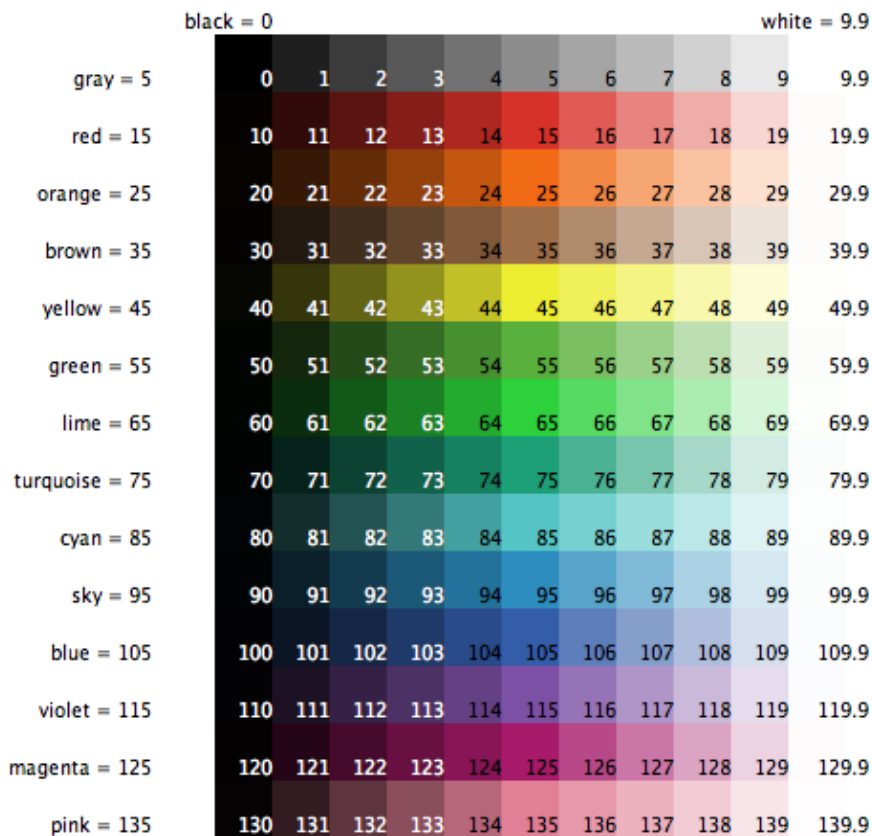
Local variables

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. You can use this command anywhere. If you use it 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
  set (color-of turtle1) (color-of turtle2)
  set (color-of turtle2) temp
end
```

Colors

NetLogo represents colors as numbers in the range 0 to 140, with the exception of 140 itself. Below is a chart showing the range of colors you can use in NetLogo.



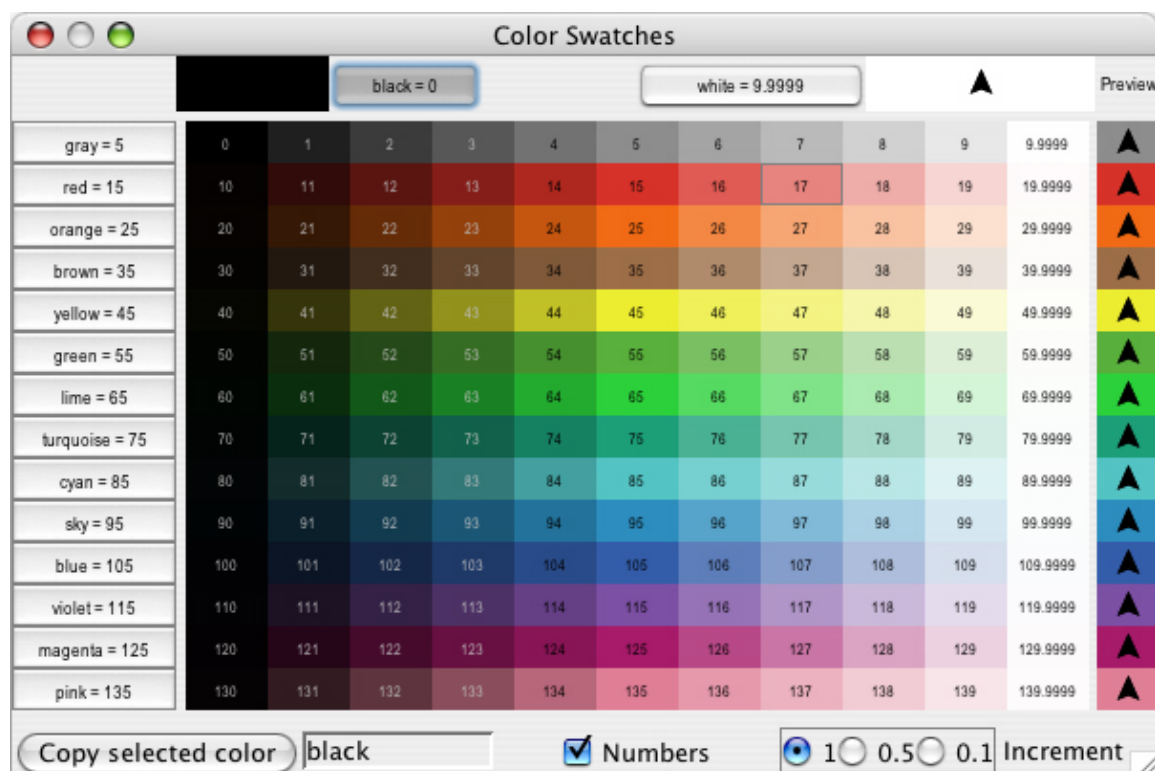
The chart shows that:

- Some of the colors have names. (You can use these names in your code.)
- Every named color except black and white has a number ending in 5.
- On either side of each named color are darker and lighter shades of the color.
- 0 is pure black. 9.9 is pure white.

- 10, 20, and so on are all so dark they appear black. 19.9, 29.9 and so on are all so light they appear white.

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

You can also view a similar chart, and experiment with the colors, by opening the Color Swatches from the Tools Menu.



When you click on any one of the color swatches (or the color buttons) that color will be displayed against all of the other standard colors along the right edge of the dialog and black and white along the top. In the bottom left corner the value of the currently selected color is displayed so you can copy the color and easily insert it into your code. In the bottom right corner there are three increment options, 1 , 0.5 , and 0.1. These numbers indicate the difference between two adjacent swatches. When the increment is 1 there are 10 different shades in each row so when the increment is 0.1 there are 100 different shades in each row.

Note: 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 can be found 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. A fixed set of discrete

hues. 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, for display color values are rounded to the nearest 0.1, so for example there's no visible difference between 26.5 and 26.52.

There are a few primitives that are helpful for working with color shades. The `scale-color` primitive is useful for converting numeric data into colors. And `shade-of?` will tell you if two colors are "shades" of the same basic hue. For example, `shade-of? orange 27` is true, because 27 is a lighter shade of orange.

Code Example: Scale-color Example shows you how to use the scale-color reporter.

For many models, the NetLogo color system is a convenient way of expressing colors. But sometimes you'd like to be able to specify colors the conventional way, by specifying HSB (hue/saturation/brightness) or RGB (red/green/blue) values. The `hsb` and `rgb` primitives let you do this. `extract-hsb` and `extract-rgb` let you convert colors in the other direction.

Since the NetLogo color space doesn't include all hues, `hsb` and `rgb` can't always give you the exact color you ask for, but they try to come as close as possible.

Code Example: You can use the HSB and RGB Example model to experiment with the HSB and RGB color systems.

Ask

NetLogo uses the `ask` command to specify commands that are to be run by turtles or patches. 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`.

The same goes for patches and the observer, except that code to be run by the observer must not be inside any `ask`.

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

```
to setup
  ca
  crt 100          ;; create 100 turtles
  ask turtles
  [ set color red   ;; turn them red
    rt random-float 360 ;; give them random headings
    fd 50 ]        ;; spread them around
```

```
ask patches
  [ if (pxcor > 0)          ;; patches on the right side
    [ set pcolor green ] ] ;; of the view turn green
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 or all patches to run commands. You can also use `ask` to have an individual turtle or patch run commands. The reporters `turtle`, `patch`, and `patch-at` are useful for this technique. For example:

```
to setup
  ca
  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
  ]
end
```

Every turtle created has an ID number. The first turtle created has ID 0, the second turtle ID 1, and so forth. The `turtle` primitive reporter takes an ID number as an input, and reports the turtle with that ID number. The `patch` primitive reporter takes values for `pxcor` and `pycor` and reports the patch with those coordinates. And the `patch-at` primitive reporter takes *offsets*: distances, in the x and y directions, *from* the first agent. In the example above, the turtle with ID number 0 is asked to get the patch east (and no patches north) of itself.

You can also select a subset of turtles, or a subset of patches, and ask them to do something. This involves a concept called "agentsets". The next section explains this concept in detail.

Agentsets

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

An agentset is not in any particular order. In fact, its 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 or patches 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, and the `patches` primitive, which reports the agentset of all patches.

But what's powerful about the agentset concept is that you can construct agentsets that contain only *some* turtles or *some* patches. 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. 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 particular patch. 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 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
```

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 `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:

```
set color-of one-of turtles 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-from` command.
- Use `values-from` 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 the richest turtle is, you could say

```
show max values-from turtles [sum assets]
```

- Use `turtles-from` and `patches-from` reporters to make new agentsets by gathering together the results reported by other agents.

This only scratches the surface — see the Models Library for many more examples, and consult the Primitives Guide and Primitives 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. In developing familiarity with programming in NetLogo, it is important to begin to think of compound commands in terms of how each element passes information to the next one. Agentsets are an important part of this conceptual scheme and provide the NetLogo developer with a lot of power and flexibility, as well as being more similar to natural language.

Code Example: Ask Agentset Example

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.

Breeds

NetLogo allows you to define different "breeds" of turtles. 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.

You define breeds using the `breed` keyword, at the top of the Procedures 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>-neighbor?`

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`, `create-custom-sheep` (`cct-sheep` for short), `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.

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](#).

Here is a quick example of using breeds:

```
breed [ mice mouse ]
breed [ frogs frog ]
mice-own [cheese]
to setup
  ca
  create-custom-mice 50
    [ set color white
      set cheese random 10 ]
  create-custom-frogs 50
    [ set color green ]
end
```

Code Example: Breeds and Shapes Example

Buttons

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

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

A button may be either a "once button", or a "forever button". You can control this by editing the button and checking or unchecking the "Forever" checkbox. Once buttons run their code once, then stop and pop back up. Forever buttons keep running their code over and over again, until either the code hits the `stop` command, or you press the button again to stop it. If you stop the button, the code doesn't get interrupted. The button waits until the code has finished, then pops up.

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 Procedures 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. (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.

Buttons and view updates

When you edit a button, there is a checkbox called "Force view update after each run". Below the checkbox is a note that reads "Checking this box produces smoother animation, but may make the button run more slowly."

Most of the time, it's enough to know that if you prefer smooth animation check the box and if you prefer speed uncheck it. In some models, the difference is dramatic; in others, it's hardly noticeable. It depends on the model.

What follows is a more detailed explanation of what's really going on with this checkbox.

To understand why this option is offered, you need to understand a little about how NetLogo updates the view. When something changes in the world, for example if a turtle moves or a patch changes color, the change does not always immediately become visible. NetLogo would run too slowly if changes always immediately became visible. So NetLogo waits until a certain amount of time has passed, usually about 1/5 of a second, and then redraws the view, so that all the changes that have happened so far become visible. This is sometimes called "skipping frames", by analogy with movies.

Skipping frames is good because each frame takes NetLogo time to draw, so your model runs faster if NetLogo can skip some of them. But skipping frames may be bad if the frames skipped contained information that you wanted to see. Sometimes the way a model looks when frames are being skipped can be misleading.

Even when the checkbox is on for a button, NetLogo will still skip frames while the code in the button is running. Checking the box only ensures that NetLogo will draw a frame when the code is done.

In some contexts, you may want to force NetLogo to draw a frame even in the middle of button code. To do that, use the `display` command; that forces NetLogo to refresh the view immediately.

In other contexts, you may want to force NetLogo *never* to draw a frame in the middle of button code, only at the end. To ensure that, put `no-display` at the beginning of the code and `display` at the end. Note also that NetLogo will never draw on-screen when inside a `without-interruption` block.

Turtle and patch forever buttons

There is a subtle difference between putting commands in a turtle or patch forever button, and putting the same commands in an observer button that does `ask turtles` or `ask patches`. 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 and patch 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 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.

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

Synchronization

In both StarLogoT and NetLogo, commands are executed asynchronously; each turtle or patch does its list of commands as fast as it can. In StarLogoT, one could make the turtles "line up" by putting in a comma (,). At that point, the turtles would wait until all were finished before any went on.

The equivalent in NetLogo is to come to the end of an ask block. If you write it this way, the two steps are not synced:

```
ask turtles
  [ fd random 10
    do-stuff ]
```

Since the turtles will take varying amounts of time to move, they'll begin "do-stuff" at different times.

But if you write it this way, they are:

```
ask turtles [ fd random 10 ]
ask turtles [ do-stuff ]
```

Here, some of the turtles will have to wait after moving until all the other turtles are done moving. Then the turtles all begin "do-stuff" at the same time.

This latter form is equivalent to this use of the comma in StarLogoT:

```
fd random 10 ,
do-stuff ,
```

Lists

In the simplest models, each variable holds only one piece of information, usually a number or a string. The list feature lets you store multiple pieces of information in a single variable by collecting those pieces of 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 [Primitives 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 Numbers 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 values-from 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 values-from turtles [...]
sum values-from 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 block of commands, like this:

```
foreach [2 4 6]
  [ crt ?
    show "created " + ? + " turtles" ]
=> created 2 turtles
=> created 4 turtles
=> created 6 turtles
```

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

Here are some more examples of foreach:

```
foreach [1 2 3] [ ask turtles [ fd ? ] ]
;; turtles move forward 6 patches
foreach [true false true true] [ ask turtles [ if ? [ fd 1 ] ] ]
;; turtles move forward 3 patches
```

map is similar to foreach, but it is a reporter. It takes an input list and another reporter. Note that unlike foreach, the reporter comes first, like this:

```
show map [round ?] [1.2 2.2 2.7]
;; prints [1 2 3]
```

map reports a list containing the results of applying the reporter to each item in the input list. Again, use ? to refer to the current item in the list.

Here is another example of map:

```
show map [? < 0] [1 -1 3 4 -2 -10]
;; prints [false true false false true true]
```

foreach and map won't necessarily be useful in 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 sort-by primitive uses a similar syntax to map and foreach, except that since the reporter needs to compare two objects, the two special variables ?1 and ?2 are used in place of ?.

Here is an example of sort-by:

```
show sort-by [?1 < ?2] [4 1 3 2]
;; prints [1 2 3 4]
```

Varying Numbers 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 commands has a default number of inputs for which no parentheses are required. The primitives which have this capability are list, word, sentence, map, and foreach.

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 need descending order instead, you can combine reverse with sort, for example `reverse sort turtles`.

If you want your agents to be ordered by some other criterion than the standard ones sort uses, you'll need to use sort-by instead.

Here's an example:

```
sort-by [size-of ?1 <size-of ?2] turtles
```

This returns a list of turtles sorted in ascending order by their turtle variable size.

Asking a list of agents

Once you have a list of agents, you might want to ask them each to do something. To do this, use the foreach and ask commands in combination, like this:

```
foreach sort turtles [
  ask ? [
    ...
```

```
]
]
```

This will ask each turtle in ascending order by who number. Substitute "patches" for "turtles" to ask patches in left-to-right, top-to-bottom order.

If you use `foreach` like this, the agents in the list run the commands inside the `ask` sequentially, not concurrently. Each agent finishes the commands before the next agent begins them.

Note that you can't use `ask` directly on a list of turtles. `ask` only works with agentsets and single agents.

Math

NetLogo supports two different kinds of math, integer and floating point.

Integers have no fractional part and may range from -2^{31} to $2^{31}-1$. Integer operations that exceed this range will not cause runtime errors, but will produce incorrect answers.

Floating point numbers are numbers containing a decimal point. In NetLogo, they operate according to the IEEE 754 standard for double precision floating point numbers. These 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. Any operation which produces the special quantities "infinity" or "not a number" will cause a runtime error.

In NetLogo, integers and floating point numbers are interchangeable, in the sense that as long as you stay within legal ranges, it is never an error to supply 3 when 3.0 is expected, or 3.0 when 3 is expected. In fact, 3 and 3.0 are considered equal, according to the `=` (equals) operator. If a floating point number 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.

Scientific notation

Very large or very small floating point numbers are displayed by NetLogo using "scientific notation". Examples:

```
O> show 0.0000000000001
observer: 1.0E-12
O> show 5000000000000000000.0
observer: 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:

```
O> show 1.0 * 10 ^ -12
observer: 1.0E-12
```

You can also use scientific notation yourself in NetLogo code:

```
O> show 3.0E6
observer: 3000000.0
```



```
O> show 3.0E7
observer: 3.0E7
O> show 8.0E-3
observer: 0.0080
O> show 8.0E-4
observer: 8.0E-4
```

These examples show that numbers are displayed using scientific notation if the exponent is less than -3 or greater than 6 .

When entering a number using scientific notation, you must include the decimal point. For example, `1E8` will not be accepted. Instead you must write `1.0E8` or `1.E8`:

```
O> show 1.0E8
observer: 1.0E8
O> show 1.E8
observer: 1.0E8
O> show 1E8
ERROR: Illegal number format
```

When entering a number, the letter E may be either upper or lowercase. When printing a number, NetLogo always uses an uppercase E:

```
O> show 4.5e10
observer: 4.5E10
```

Floating point accuracy

When using floating point numbers, you should be aware that due to the limitations of the binary representation for floating point numbers, you may get answers that are slightly inaccurate. For example:

```
O> show 0.1 + 0.1 + 0.1
observer: 0.30000000000000004
O> show cos 90
observer: 6.123233995736766E-17
```

This is an inherent issue with floating point arithmetic; it occurs in all programming languages that support floating point.

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 can be any integer. 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 95, 7, and 54.

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 we change the random number generator. For example, NetLogo 2.0 has a different generator than NetLogo 1.3 did. 2.0's generator (which is known as the "Mersenne Twister") is faster and generates numbers that are statistically more "random" than 1.3's (Java's built-in "linear congruential" generator).

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. And it never reports the same number 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.

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 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

Plotting

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

Before you can plot, you need to create one or more plots in the Interface tab. Each plot should have a unique name. You'll be using its name to refer to it in your code in the Procedures tab.

For more information on using and editing plots in the Interface tab, see the [Interface Guide](#).

Specifying a plot

If you only have one plot in your model, then you can start plotting to it right away. But if you have more than one plot, you have to specify which one you want to plot to. To do this, use the `set-current-plot` command with the name of the plot enclosed in double quotes, like this:

```
set-current-plot "Distance vs. Time"
```

You must supply the name of the plot 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.)

Specifying a pen

When you make a new plot, it just has one pen in it. If the current plot only has one plot pen, then you can start plotting to it right away.

But you can also have multiple pens in a plot. You can create additional pens by editing the plot and using the controls in the "Plot Pens" section at the bottom of the edit dialog. Each pen should have a unique name. You'll be using its name to refer to it in your code in the Procedures tab.

For a plot with multiple pens, you have to 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, use the `set-current-plot-pen` command with the name of the pen enclosed in double quotes, like this:

```
set-current-plot-pen "distance"
```

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 1.0; 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:

```
to setup
  ...
  plot count turtles
end

to go
  ...
  plot count turtles
end
```

Note that in this example we plot from both the "setup" and "go" procedures. That's because we want our plot to include the initial state of the system. 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.

If you need to specify both the x and y values of the point you want plotted, then use `plotxy` instead.

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 (ppu for short). 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 (ppd for short).

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-from command, like this:

```
histogram-from turtles [age]
```

If the data you want to histogram don't come from an agentset but from a list of numbers, use the histogram-list command instead.

Note that using the histogram commands 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.)

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 only want to remove only the points that the current plot pen has drawn, use plot-pen-reset.

When a whole plot is cleared, or when a pen is reset, that doesn't just remove the data that has been plotted. It also restores the plot or pen to its default settings, as they were specified in the

Interface tab when the plot was created or last edited. Therefore, the effects of such commands as `set-plot-x-range` and `set-plot-pen-color` are only temporary.

Autoplotting

By default, all NetLogo plots have the "autoplotting" 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.

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 Autoplot checkbox. At present, it is not possible to enable or disable this feature only on one axis; it always applies to both axes.

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. Once the pen has been created, you can use it like any ordinary pen. By default, the new pen is down, is black in color, has an interval of 1.0, and plots in line mode. Commands are available to change all of these settings; see the Plotting section of the Primitives Dictionary.

Using a Legend

You can show the legend of a plot by clicking on the word "Pens" in the upper right corner of the plot. 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 in the plot edit dialog.

Conclusion

Not every aspect of NetLogo's plotting system has been explained here. See the Plotting section of the Primitives 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

Strings

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:

```
butfirst "string" => "tring"
butlast "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.

To concatenate strings, that is, combine them into a single string, you can also use the + (plus) operator, like this:

```
"tur" + "tle" => "turtle"
```

If you need to embed a special character in a string, use the following escape sequences:

- \n = newline
- \t = tab
- \" = double quote
- \\ = backslash

Output

This section is about output to the screen. Output to the screen can also be later saved to a file using the export-output command. If you need a more flexible method of writing data to external files, see the next section, File I/O.

The basic commands for generating output to the screen in NetLogo are print, show, type, and write. These commands send their output to the Command Center.

For full details on these four commands, see their entries in the Primitives Dictionary. Here is how

they are typically used:

- print is useful in most situations.
- show lets you see which agent is printing what.
- type lets you print several things on the same line.
- write lets you print values in a format which can be read back in using file-read.

A NetLogo model may optionally have an "output area" in its Interface tab, separate from the Command Center. To send output there instead of the Command Center, use the output-print, output-show, output-type, and output-write commands.

The output area can be cleared with the clear-output command and saved to a file with export-output. The contents of the output area will be saved by the export-world command. The import-world command will clear the output area and set its contents to the value in imported world file. It should be noted that large amounts of data being sent to the output area can increase the size of your exported worlds.

If you use output-print, output-show, output-type, output-write, clear-output, or export-output in a model which does not have a separate output area, then the commands apply to the output portion of the Command Center.

File I/O

In NetLogo, there is a set of primitives that give you the power to interact with outside files. They all begin with the prefix **file-**.

There are two main modes when dealing with files: reading and writing. The difference is the direction of the flow of data. When you are reading in information from a file, data that is stored in the file flows into your model. On the other hand, writing allows data to flow out of your model and into a file.

When a NetLogo model runs as an applet within a web browser, it will only be able to read data from files which are in the same directory on the server as the model file. Applets cannot write to any files.

When working with files, always begin by using the primitive file-open. This specifies which file you will be interacting with. None of the other primitives work unless you open a file first.

The next **file-** primitive you use dictates which mode the file will be in until the file is closed, reading or writing. To switch modes, close and then reopen the file.

The reading primitives include file-read, file-read-line, file-read-characters, and file-at-end?. Note that the file must exist already before you can open it for reading.

Code Examples: File Input Example

The primitives for writing are similar to the primitives that print things in the Command Center, except that the output gets saved to a file. They include file-print, file-show, file-type,

and [file-write](#). Note that you can never "overwrite" data. In other words, if you attempt to write to a file with existing data, all new data will be appended to the end of the file. (If you want to overwrite a file, use [file-delete](#) to delete it, then open it for writing.)

Code Examples: File Output Example

When you are finished using a file, you can use the command [file-close](#) to end your session with the file. If you wish to remove the file afterwards, use the primitive [file-delete](#) to delete it. To close multiple opened files, one needs to first select the file by using [file-open](#) before closing it.

```
;; Open 3 files
file-open "myfile1.txt"
file-open "myfile2.txt"
file-open "myfile3.txt"

;; Now close the 3 files
file-close
file-open "myfile2.txt"
file-close
file-open "myfile1.txt"
file-close
```

Or, if you know you just want to close every file, you can use [file-close-all](#).

Two primitives worth noting are [file-write](#) and [file-read](#). These primitives are designed to easily save and retrieve NetLogo constants such as numbers, lists, booleans, and strings. [file-write](#) will always output the variable in such a manner that [file-read](#) will be able to interpret it correctly.

```
file-open "myfile.txt" ;; Opening file for writing
ask turtles
  [ file-write xcor file-write ycor ]
file-close

file-open "myfile.txt" ;; Opening file for reading
ask turtles
  [ setxy file-read file-read ]
file-close
```

Code Examples: File Input Example and File Output Example

Letting the user choose

The [user-directory](#), [user-file](#), and [user-new-file](#) primitives are useful when you want the user to choose a file or directory for your code to operate on.

Movies

This section describes how to capture a QuickTime movie of a NetLogo model.

First, use the movie-start command to start a new movie. The filename you provide should end with `.mov`, the extension for QuickTime movies.

To add a frame to your movie, use either movie-grab-view or movie-grab-interface, depending on whether you want the movie to show just the current view, or the entire Interface tab. In a single movie, you must use only one **movie-grab-** primitive or the other; you can't mix them.

When you're done adding frames, use movie-close.

```
;; export a 30 frame movie of the view
setup
movie-start "out.mov"
movie-grab-view ;; show the initial state
repeat 30
[ go
  movie-grab-view ]
movie-close
```

By default, a movie will play back at 15 frames per second. To make a movie with a different frame rate, call movie-set-frame-rate with a different number of frames per second. You must set the frame rate after movie-start but before grabbing any frames.

To check the frame rate of your movie, or to see how many frames you've grabbed, call movie-status, which reports a string that describes the state of the current movie.

To throw away a movie and delete the movie file, call movie-cancel.

NetLogo movies are exported as uncompressed QuickTime files. To play a QuickTime movie, you can use QuickTime Player, a free download from Apple.

Since the movies are not compressed, they can take up a lot of disk space. You will probably want to compress your movies with third-party software. The software may give you a choice of different kinds of compression. Some kinds of compression are lossless, while others are lossy. "Lossy" means that in order to make the files smaller, some of the detail in the movie is lost. Depending on the nature of your model, you may want to avoid using lossy compression, for example if the view contains fine pixel-level detail.

Code Example: Movie Example

Perspective

The 2D and the 3D view show the world from the perspective of the observer. By default the observer is looking down on the world from the positive z-axis at the origin. You can change the perspective of the observer by using the follow, ride and watch observer commands and follow-me, ride-me and watch-me turtle commands. When in follow or ride mode the observer moves with the subject agent around the world. The difference between follow and ride is only visible in the 3D view. In the 3D view the user can change the distance behind the agent using the mouse. When the observer is following at zero distance from the agent it is actually riding the agent. When the observer is in watch mode it tracks the movements of one turtle without moving. In both views you will see a spotlight appear on the subject and in the 3D view the observer will turn to face

the subject. To determine which agent is the focus you can use the subject reporter.

Code Example: Perspective Example

Drawing

The drawing is a layer where turtles can make visible marks.

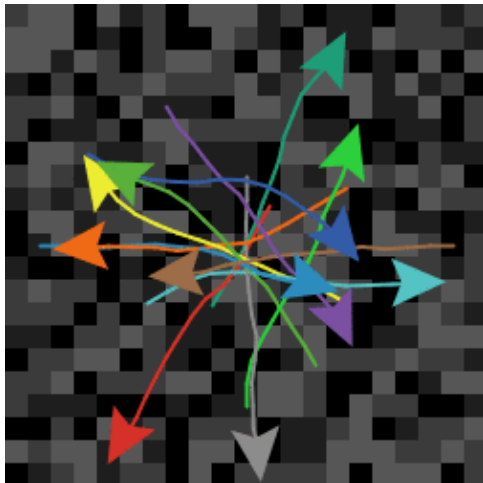
In the view, the drawing appears on top of the patches but underneath the turtles. Initially, the drawing is empty and transparent.

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

Turtles can draw and erase lines in the drawing using the pen-down and pen-erase commands. When a turtle's pen is down (or erasing), the turtle draws (or erases) a line behind it whenever it moves. The lines are the same color as the turtle. To stop drawing (or erasing), use pen-up.

Lines drawn by turtles are normally one pixel thick. If you want a different thickness, set the pen-size turtle variable to a different number before drawing (or erasing). In new turtles, the variable is set to 1.0.

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 2.0:



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 command 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` instead.

Comparison to other Logos

Drawing works somewhat differently in NetLogo than some other Logos.

Notable differences include:

- New turtles' pens are up, not down.
- Instead of using a `fence` command to confine the turtle inside boundaries, in NetLogo you edit the world and turn wrapping off.
- There is no `screen-color`, `bgcolor`, or `setbg`. You can make a solid background by coloring the patches, e.g. `ask patches [set pcolor blue]`.

Drawing features not supported by NetLogo:

- There is no `window` command. This is used in some other Logos to let the turtle roam over an infinite plane.
- There is no `flood` or `fill` command to fill an enclosed area with color.

Topology

The 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, as were all NetLogo worlds before NetLogo 3.1.

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.

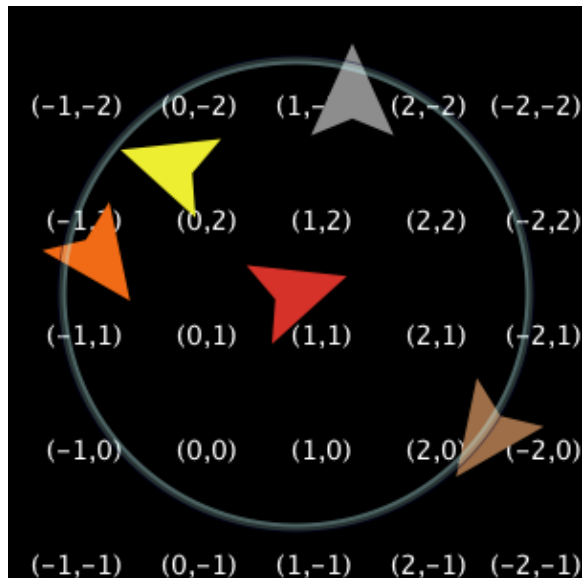
Horizontal and vertical cylinders wrap in one direction but not the other. A horizontal cylinder wraps vertically, so the top of the world is connected to the bottom. but the left and right edges are bounded. A vertical cylinder is the opposite; it wraps horizontally so the left and right edges are connected, but the top and bottom edges are bounded.

Code Example: Neighbors Example

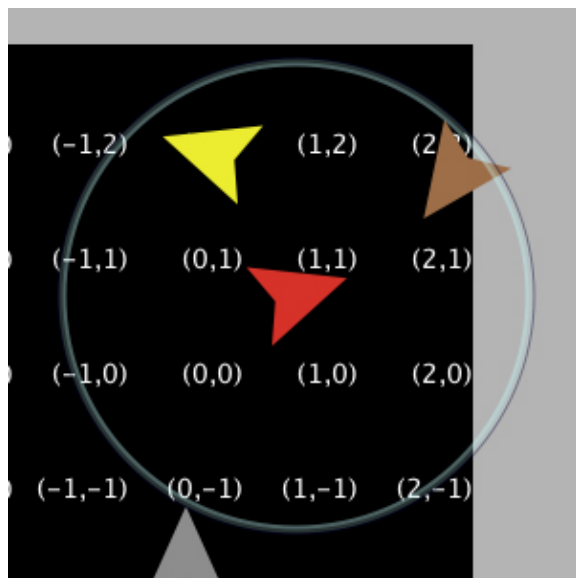
Since NetLogo 3.0 there have been settings to enable wrapping visually, so if a turtle shape extends past an edge, part of the shape will appear on the other edge of the view. (Turtles themselves are points that take up no space, so they cannot be on both sides of the world at once, but in the view,

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)

Instead of 3.0's settings that only control the *appearance* of wrapping in the view, NetLogo 3.1 has settings that control whether the world *actually* wraps or not, that is, whether opposite edges are in fact connected. These new wrapping settings determine the world topology, that is, whether the world is a torus, box, or cylinder. This affects the behavior and not just the visual appearance of the

model.

In the past, model authors were required to write extra code to simulate a box world, with the aid of special "no-wrap" primitives. No-wrap versions were provided for distance(xy), in-radius, in-cone, face(xy), and towards(xy). In 3.1 the special no-wrap versions are no longer necessary. Instead, the topology controls whether the primitives wrap or not. They always use the shortest path allowed by the topology. For example, the distance from the center of the patches in the bottom right corner (min-pxcor, min-pycor) and the upper left corner (max-pxcor, max-pycor) will be as follows for each topology given that the min and max pxcor and pycor are +/-2:

- Torus – $\sqrt{2} \sim 1.414$ (this will be the same for all world sizes since the patches are directly diagonal to each other in a torus.)
- Box – $\sqrt{\text{world-width}^2 + \text{world-height}^2} \sim 7.07$
- Vertical Cylinder – $\sqrt{\text{world-height}^2 + 1} \sim 5.099$
- Horizontal Cylinder – $\sqrt{\text{world-width}^2 + 1} \sim 5.099$

All the other primitives will act similarly to distance. If you formerly used no-wrap primitives in your model we recommend removing them and changing the topology of the world instead.

There are a number of reasons to change your model to use topologies rather than no-wrap primitives.

First, we expect if you are using no-wrap primitives, you are actually modeling a world that is not a torus. If you use a topology that matches the world you are modeling NetLogo does automatic bounds checking for you, it should make your life easier, your code simpler to understand and it adds visual cues to help the model user understand what you are modeling. Note that even with no-wrap primitives it was very difficult to model cylinders since the no-wrap primitives report the distance or heading when wrapping is not allowed in either direction.

You might have bugs in your model. If you are using a combination of no-wrap and wrap primitives, either it doesn't matter for some reason or there is a bug in your model (we found a few bugs in our models). For example, the Conductor model compared distance-no-wrap to distance to determine whether the next position is wrapped around the world, in which case the electron exits the system. This is a clever way to solve the problem, but unfortunately it is flawed. Electrons that wrap in the y direction were also exiting the system which is incorrect in this case. The only correct way to exit is to reach the cathode at the left end of the wire.

If you remove no-wrap commands the topology is no longer hard coded into the model so it's easier to test out your model on a different shape of world without a lot of extra coding (you may have to add a few extra checks to go from torus to box, this is explained more in-depth in the How to convert section.)

Note that though we've removed the no-wrap primitives from the dictionary they are still available for you to use; we did this so that old models don't have to be changed in order to run.

How to convert your model

When you first open up your model in 3.1 NetLogo will automatically change all cases of (`-screen-edge-x`) to `min-pxcor` and all cases of `screen-edge-x` to `max-pxcor` (and similarly for y) Though this is not directly related to the topology changes, you may also want to think about whether moving the origin off-center makes sense in your model at this time. Before

NetLogo 3.1 the world had to be symmetrical around the origin, thus, the world had to have an odd width and height. This is no longer true since you may use any min and max combinations you wish, given that the point (0,0) still exists in the world. If you are logically only modeling in one or two quadrants, or if it makes your code simpler to only use positive numbers you might want to consider changing your model. If you've modeled something that requires an even grid you'll certainly want to remove the programming hacks required to make that possible in the past.

Code Examples: Lattice Gas Automaton, Binomial Rabbits, Rugby

For NetLogo 3.1 we added new primitives which are essential if you change the topology, and quite convenient even if you don't. `random-pxcor`, `random-pycor`, `random-xcor`, and `random-ycor` report random values within the range between maximum and minimum (x and y). In older versions of NetLogo we often relied on wrapping to place turtles randomly across the world by writing `setxy random-float screen-size-x random-float screen-size-y`. However, if wrapping is not allowed in one direction or the other this no longer works (you get a runtime error for trying to place turtles outside the world). Regardless of topology, it is simpler and more straight forward to use `setxy random-xcor random-ycor` instead.

To convert a model to use a topology you must first decide what settings best describe the world. If the answer is not immediately obvious to you based on the real world, (a room is a box, a wire is a cylinder) there are a few clues that will help you. If anywhere in the code you are checking the bounds of the world or if some patches are not considered neighbors of the patches on the other side of the view it is likely that you are not using a torus. If you check bounds in both the x and y directions it's a box, in the x direction only, a horizontal cylinder, the y a vertical cylinder.

If you use no-wrap primitives you are probably not modeling a torus, however, be careful with this criterion if you use a mix of no-wrap and wrap primitives. It may be that you were using a no-wrap primitive for a visual element but the rest of the NetLogo world is still a torus.

After you've determined the topology and changed it by editing the view, you may have to make a few small changes to the code. If you've decided that the world is a torus you probably don't have to make any changes at all. If your model only uses patch neighbors and diffuse you probably will not need to make many changes.

If your model has turtles that move around your next step is to determine what happens to them when they reach the edge of the world. 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 simply 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 than 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.

By using a different topology you get diffuse for free (which was fairly difficult to do in the past). 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. If you had special code to handle diffuse then you can remove it. However, if you want the diffuse matter to still fall off the edges of the world as it would on an infinite plane you still need to clear the edges each step as in the Diffuse Off Edges Example.

Links

Links are an experimental part of NetLogo and primitives related to Links are subject to change. Because links are experimental, the names of the primitives begin with two underscores.

A link is a special turtle connecting two other turtles. The two turtles are called nodes. A link turtle's size is always equal to the distance between the two node turtles. Its heading is always equal to the heading from one node turtle to the other. Its location is always halfway between the two node turtles. When one of the nodes moves, the link turtle will automatically change its heading, size, and location.

There are two flavors of links, 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.

All link turtles must have a breed. This means you must define at least one breed in your model using the breed keyword. Link turtles can only be created using specific commands. The commands create-<breed>-with and create-<breeds>-with creates undirected links; the commands create-<breed>-to, create-<breeds>-to, create-<breed>-from, and create-<breeds>-from create directed links.

```
breed [ links link ]
breed [ nodes node ]
to setup
  ca
  create-custom-nodes 2 [ fd 10 ]
  ask node 0 [ __create-link-with node 1 [ ] ]
end
```

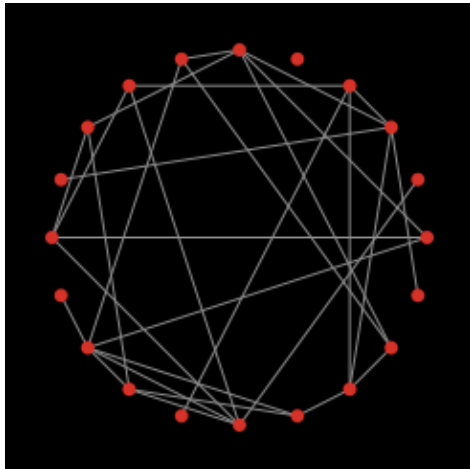
Once the first link of any breed has been created directed or undirected, all links of that breed must match; it's impossible to have two links of the same breed where one is directed and the other is undirected. A runtime error occurs if you try to do it. If all link turtles of a breed die, then you can create links of that breed that are different in flavor from the previous links.

In general, primitives that work with directed links have "in", "out", "to", and "from" in their names. Undirected ones either omit these or use "with".

There cannot be more than one undirected link of the same breed 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 between a pair if they are in opposite directions.

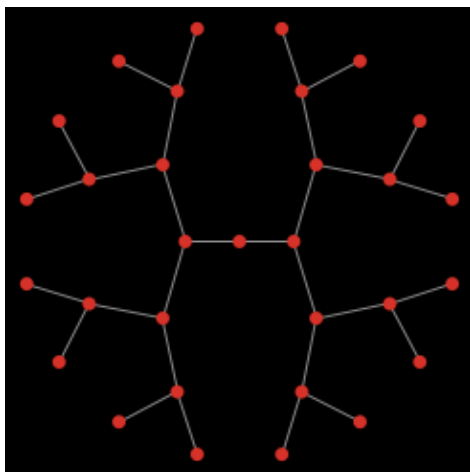
Layouts

As part of the experimental 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.

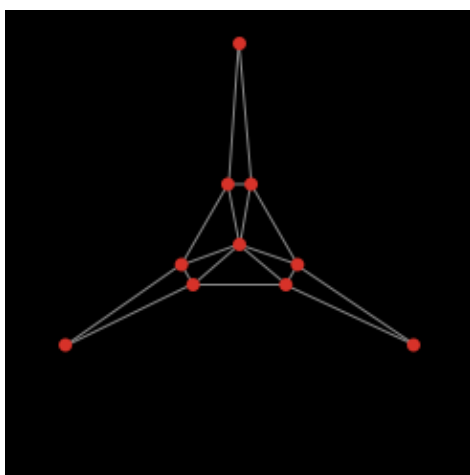


layout-radial is a good layout if you have something like a tree structure, though even if there are some cycles in the tree it will still work, though as there are more and more cycles it will probably not look as good. layout-radial takes a root agent to be the central node places it at (0,0) and arranges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

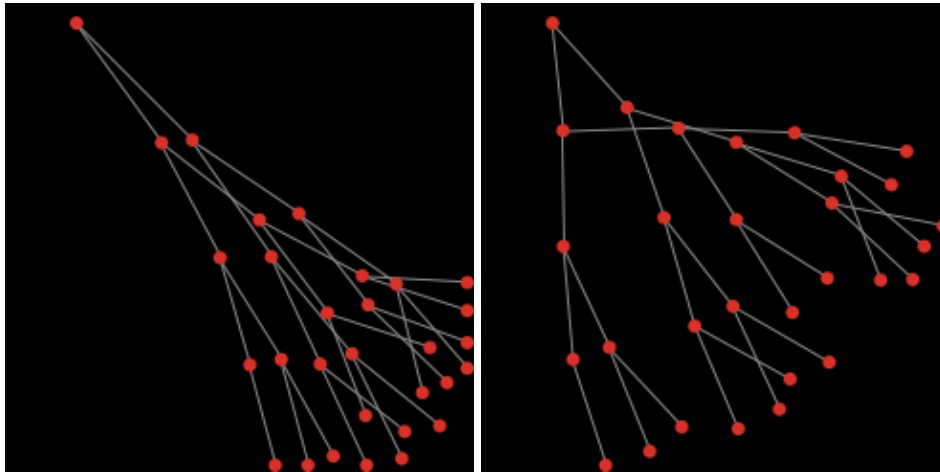
give more space to branches that are wider. `__layout-radial` also takes a breed as an input so you use one breed of links to layout the network and not another.



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 anchorset 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` and `__layout-magspring` are quite similar and are useful for many kinds of networks. The drawback is that they are relatively slow since they take many iterations to converge. In both layouts the links act as springs that pull the nodes they connect toward each other and the nodes repel each other. In the magnetic spring there is also a magnetic field pulling the nodes in a compass direction you choose. The strength of all of these forces are 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. The magnetic spring layout also has a boolean input, `bidirectional?`, which indicates whether the springs should push in both directions parallel to the magnetic field; if it is true the networks will be more evenly spaced.



Link Shape and Direction Indicators

The default shape of all link turtles is "link". You can still set their shape using `set shape "newshape"` or `set-default-shape linkbreed "newshape"`.

All directed links have a direction indicator shape, "link direction". This shape will be drawn over top of the link turtle near the destination node. It will have the same color as the link turtle. The shape will scale with the line thickness of the link turtle, see `__set-line-thickness`, with a minimum size of 1.

You may edit the "link" and "link direction" shapes, but they cannot be deleted. If you wish to have no direction indicators, remove all elements from the `link direction` shape.

Code Examples: Network Example , Giant Component, Small Worlds, Preferential Attachment

Tie

Tie is an experimental part of NetLogo and primitives related to Tie are subject to change. Because Tie is experimental, the names of the primitives begin with two underscores.

Tie connects two turtles so that the movement of the root turtle affects the location and heading of the leaf 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.

Note that the movements of the root affect the leaf, but not vice versa. The leaf is free to move and turn on its own, even while remaining tied, and the root is not affected.

The `tie` command connects a leaf turtle to a root turtle. The `untie` command removes the connection.

Code Example: Tie System Example model shows turtles tied together, including leaf turtles being the root of other leaf turtles.

Shapes Editor Guide

The Shapes Editor allows you to create and save turtle designs. NetLogo uses fully scalable and rotatable vector shapes, which means it lets you create designs by combining basic geometric elements, which can appear on-screen in any size or orientation.

Getting Started

To begin making shapes, choose **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 shapes from a library or from another model.

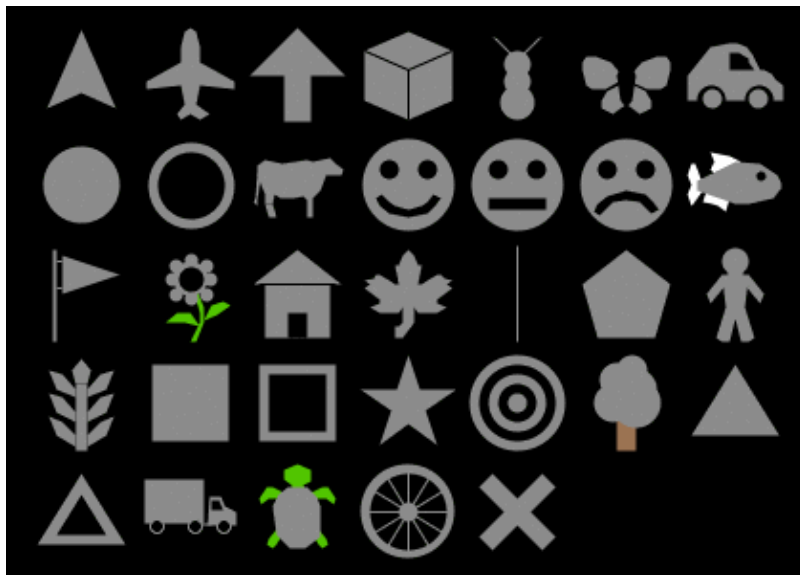
Importing Shapes

Every new model in NetLogo starts off containing a small core set of frequently used shapes. Many more 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 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, face happy, face neutral, face sad, fish
Third row: flag, flower, house, leaf, line, pentagon, person
Fourth row: plant, square, square 2, star, target, tree, triangle
Fifth row: triangle 2, truck, turtle, wheel, x

Shapes library

And here are the shapes in the shapes library (including all of the default shapes, too):



Creating and Editing 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.

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. For example, suppose you want to create 50 turtles with the shape "rabbit". Provided there is some shape called *rabbit* in this model, give this command to the observer in the command center:

```
O> crt 50
```

And then give these commands to the turtles to spread them out, then change their shape:

```
T> fd random 15
T> set shape "rabbit"
```

Voila! Rabbits! Note the use of double quotes around the shape name. Shape names are strings.

The `set-default-shape` command is also useful for assigning shapes to turtles.

BehaviorSpace Guide

This guide has three parts:

- **What is BehaviorSpace?**: A general description of the tool, including the ideas and principles behind it.
- **How It Works**: Walks you through how to use the tool and highlights its most commonly used features.
- **Advanced Usage**: How to use BehaviorSpace from the command line, or from your own Java code.

What is BehaviorSpace?

BehaviorSpace is a software tool integrated with NetLogo that allows you to perform experiments with models. It 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.

Why BehaviorSpace?

The need for this type of experiment is revealed by the following observations. Models often have many settings, each of which can take a range of values. Together they form what in mathematics is called a parameter space for the model, whose dimensions are the number of settings, and in which every point is a particular combination of values. Running a model with different settings (and sometimes even the same ones) can lead to drastically different behavior in the system being modeled. So, how are you to know which particular configuration of values, or types of configurations, will yield the kind of behavior you are interested in? This amounts to the question of where in its huge, multi-dimension parameter space does your model perform best?

For example, suppose you want speedy synchronization from the agents in the Fireflies model. The model has four sliders — number, cycle-length, flash-length and number-flashes — that have approximately 2000, 100, 10 and 3 possible values, respectively. That means there are $2000 * 100 * 10 * 3 = 600,000$ possible combinations of slider values! Trying combinations one at a time is hardly an efficient way to learn which one will evoke the speediest synchronization.

BehaviorSpace offers you a much better way to solve this problem. If you specify a subset of values from the ranges of each slider, it will run the model with each possible combination of those values and, during each model run, record the results. In doing so, it samples the model's parameter space — not exhaustively, but enough so that you will be able to see relationships form between different sliders and the behavior of the system. After all the runs are over, a dataset is generated which you can open in a different tool, such as a spreadsheet, database, or scientific visualization application, and explore.

By enabling you to explore the entire "space" of behaviors a model can exhibit, BehaviorSpace can be a powerful assistant to the modeler.

Historical Note

Old versions of NetLogo (prior to 2.0) included an earlier version of the BehaviorSpace tool. That version was much different. It wasn't nearly as flexible in the kinds of experiments it let you set up. But, it had facilities for display and analyzing experiment results that are missing from the current version. With the current version, it is assumed that you will use other software to analyze your results. We hope to re-add data display and analysis facilities to a future version of BehaviorSpace.

How It Works

To begin using BehaviorSpace, open your model, then choose the BehaviorSpace item on NetLogo's Tools menu.

Managing experiment setups

The dialog that opens lets you create, edit, duplicate, delete, and run experiment setups. Experiments are listed by name and how by model runs the experiment will consist of.

Experiment setups are considered part of a NetLogo model and are saved as part of the model.

To create a new experiment setup, press the "New" button.

Creating an experiment setup

In the new dialog that appears, you can specify the following information. Note that you don't always need to specify everything; some parts can be left blank, or left with their default values, depending on your needs.

Experiment name: If you have multiple experiments, giving them different names will help you keep them straight.

Vary variables as follows: This is where you specify which settings you want varied, and what values you want them to take. Variables can include sliders, switches, choosers, and any global variables in your model.

Variables can also include `max-pxcor`, `min-pxcor`, `max-pycor` and `min-pycor`, `world-width`, `world-height` and `random-seed`. These are not, strictly speaking, variables, but BehaviorSpace lets you vary them as if they were. Varying the world dimensions lets you explore the effect of world size upon your model. Since setting `world-width` and `world-height` does not necessarily define the bounds of the world how they are varied depends on the location of the origin. If the origin is centered, BehaviorSpace will keep it centered so the values `world-width` or `world-height` must be odd. If one of the bounds is at zero that bound will be kept at zero and the other bound will move, for example if you start with a world with `min-pxcor = 0` `max-pxcor = 10` and you vary `world-width` like this:

```
[ "world-width" [11 1 14]]
```

`min-pxcor` will stay at zero and `max-pxcor` will set to 11, 12, and 13 for each of the runs. If neither of these conditions are true, the origin is not centered, nor at the edge of the world you cannot vary `world-height` or `world-width` directly but you should vary `max-pxcor`, `max-pycor`, `min-pxcor` and `min-pycor` instead.

Varying `random-seed` lets you repeat runs by using a known seed for the NetLogo random number generator. Note that you're also free to use the `random-seed` command in your experiment's setup commands. For more information on random seeds, see the [Random Numbers](#) section of the Programmer's Guide.

You may specify values either by listing the values you want used, or by specifying that you want to try every value within a given range. For example, to give a slider named `number` every value from 100 to 1000 in increments of 50, you would enter:

```
[ "number" [100 50 1000]]
```

Or, to give it only the values of 100, 200, 400, and 800, you would enter:

```
[ "number" 100 200 400 800]
```

Be careful with the brackets here. Note that there are fewer square brackets in the second example. Including or not including this extra set of brackets is how you tell BehaviorSpace whether you are listing individual values, or specifying a range.

Also note that the double quotes around the variable names are required.

You can vary as many settings as you want, including just one, or none at all. Any settings that you do not vary will retain their current values. Not varying any settings is useful if you just want to do many runs with the current settings.

What order you list the variables in determines what order the runs will be done in. All values for a later variable will be tried before moving to the next value for an earlier variable. So for example if you vary both `x` and `y` from 1 to 3, and `x` is listed first, then the order of model runs will be: `x=1 y=1`, `x=1 y=2`, `x=1 y=3`, `x=2 y=1`, and so on.

Repetitions: Sometimes the behavior of a model can vary a lot from run to run even if the settings don't change, if the model uses run numbers. If you want to run the model more than once at each combination of settings, enter a higher number here than one.

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.

Measure runs at every tick: Normally NetLogo will measure model runs at every tick, using the reporters you entered in the previous box. If you're doing very long model runs, you might not want all that data. Uncheck this box if you only want to measure each run after it ends.

Setup commands: These commands will be used to begin each model run. Typically, you will enter the name of a procedure that sets up the model, typically `setup`. But it is also possible to include other commands as well.

Go commands: These commands will be run over and over again to advance to the model to the next "tick". Typically, this will be the name of a procedure, such as `go`, but you may include any commands you like.

Stop condition: This lets you do model runs of varying length, ending each run when a certain condition becomes true. For example, suppose you wanted each run to last until there were no more turtles. Then you would enter:

```
not any? turtles
```

If you want the length of runs to all be of a fixed length, just leave this blank.

The run may also stop because the go commands use the `stop` command, in the same way that `stop` can be used to stop a forever button. The `stop` command may be used directly in the go commands, or in a procedure called directly by the go commands. (The intent is that the same `go` procedure should work both in a button and in a BehaviorSpace experiment.) Note that the step in which `stop` is used is considered to have been aborted, so no results will be recorded for that step. Therefore, the stopping test should be at the beginning of the go commands or procedure, not at the end.

Final commands: These are any extra commands that you want run once, when the run ends. Usually this is left blank, but you might use it to call the `export-world` command or record the results of the run in some other way.

Time limit: This lets you set a fixed maximum length for each run. If you don't want to set any maximum, but want the length of the runs to be controlled by the stop condition instead, enter 0.

Running an experiment

When you're done setting up your experiment, press the "OK" button, followed by the "Run" button.

You will be prompted to select the formats you would like the data from your experiment saved in. Data is collected for each interval, run or tick, according to the setting of **Measure runs at every tick** option.

Table format lists each interval in a row, with each metric in a separate column. Table data is written to the output file as each run completes. Table format is suitable for automated processing of the data, such as importing into a database or a statistics package.

Spreadsheet format calculates the min, mean, max, and final values for each metric, and then lists each interval in a row, with each metric in a separate column. Spreadsheet data is more human-readable than Table data, especially if imported into a spreadsheet application.

(Note however that spreadsheet data is not written to the results file until the experiment finishes. Since spreadsheet data is stored in memory until the experiment is done, very large experiments could run out of memory. And if anything interrupts the experiment, such as a runtime error, running out of memory, or a crash or power outage, no results will be written. For long experiments, you may want to use both spreadsheet and table formats so that if something happens you'll at least get a table of partial results.)

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.

A dialog will appear, titled "Running Experiment". In this dialog, you'll see a progress report of how many runs have been completed so far and how much time has passed. If you entered any reporters for measuring the runs, and if you left the "Measure runs at every tick" box checked, then you'll see a plot of how they vary over the course of each run.

You can also watch the runs in the main NetLogo window. (If the "Running Experiment" dialog is in the way, just move it to a different place on the screen.) The view and plots will update as the model runs. If you don't need to see them update, then use the checkboxes in the "Running Experiment" dialog to turn the updating off. This will make the experiment go faster.

If you want to stop your experiment before it's finished, press the "Abort" button. But note that you'll lose any results that were generated up to that point.

When all the runs have finished, the experiment is complete.

Advanced usage

Running from the command line

It is possible to run BehaviorSpace experiments "headless", that is, from the command line, without any graphical user interface (GUI). This is useful for automating runs on a single machine or a

cluster of machines.

No Java programming is required. Experiment setups can be created in the GUI and then run later from the command line, or, if you prefer, you can create or edit experiment setups directly using XML.

It is easiest if you create your experiment setup ahead of time in the GUI, so it is saved as part of the model. To run an experiment setup saved in a model, here is an example command line:

```
java -server -Xms16M -Xmx512M -cp NetLogo.jar \
  org.nlogo.headless.HeadlessWorkspace \
  --model Fire.nlogo \
  --experiment experiment1
```

After the named experiment has run, the results are sent to standard output in spreadsheet format, as CSV. (To change this, see below.)

When running the HeadlessWorkspace class as an application, it forces the system property `java.awt.headless` to be true. This tells Java to run in headless mode, allowing NetLogo to run on machines when a graphical display is not available.

Note the use of the `-server` flag to tell Java to optimize performance for "server" type applications; we recommend this flag for best performance in most situations.

Note the use of `-Xmx` to specify a maximum heap size of 512 megabytes. If you don't specify a maximum heap size, you will get your VM's default size, which may be unusably small. (512 megabytes is an arbitrary size which should be more than large enough for most models; you can specify a different limit if you want.) Note also that `-Xms` is used to specify a larger-than-default initial heap size. This helps some models run faster by making garbage collection more efficient.

The `--model` argument is used to specify the model file you want to open.

The `--experiment` argument is used to specify the name of the experiment you want to run. (At the time you create an experiment setup in the GUI, you assign it a name.)

Here's another example that shows some additional, optional arguments:

```
java -server -Xms16M -Xmx512M -cp NetLogo.jar \
  org.nlogo.headless.HeadlessWorkspace \
  --model Fire.nlogo \
  --experiment experiment2 \
  --max-pxcor 100 \
  --min-pxcor -100 \
  --max-pycor 100 \
  --min-pycor -100 \
  --no-results
```

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.)

Note also the use of the optional `--no-results` argument to specify that no output is to be generated. This is useful if the experiment setup generates all the output you need by some other means, such as exporting world files or writing to a text file.

Yet another example:

```
java -server -Xms16M -Xmx512M -cp NetLogo.jar \
  org.nlogo.headless.HeadlessWorkspace \
  --model Fire.nlogo \
  --experiment experiment2 \
  --table table-output.csv \
  --spreadsheet spreadsheet-output.csv
```

The optional `--table <filename>` argument specifies that output should be generated in a table format and written to the given file as CSV data. If `-` is specified as the filename, then the output is sent to the standard system output stream. Table data is written as it is generated, with each complete run.

The optional `--spreadsheet <filename>` argument specified that spreadsheet output should be generated and written to the given file as CSV data. If `-` is specified as the filename, then the output is sent to the standard system output stream. Spreadsheet data is not written out until all runs in the experiment are finished.

Note that it is legal to specify both `--table` and `--spreadsheet`, and if you do, both kinds of output file will be generated.

The default output behavior, when no output formats are specified, is to send table output to the system standard output stream.

Here is one final example that shows how to run an experiment setup which is stored in a separate XML file, instead of in the model file:

```
java -server -Xms16M -Xmx512M -cp NetLogo.jar \
  org.nlogo.headless.HeadlessWorkspace \
  --model Fire.nlogo \
  --setup-file fire-setups.xml \
  --experiment experiment3
```

If the XML file contains more than one experiment setup, it is necessary to use the `--experiment` argument to specify the name of the setup to use.

The next section has information on how to create standalone experiment setup files using XML.

Setting up experiments in XML

We don't yet have detailed documentation on authoring experiment setups in XML, but if you already have some familiarity with XML, then the following pointers may be enough to get you started.

The structure of BehaviorSpace experiment setups in XML is determined by a Document Type Definition (DTD) file. The DTD is stored in NetLogo.jar, as `system/behaviorspace.dtd`. (JAR files are also zip files, so you can extract the DTD from the JAR using Java's "jar" utility or with any

program that understands zip format.)

The easiest way to learn what setups look like in XML, though, is to author a few of them in BehaviorSpace's GUI, save the model, and then examine the resulting .nlogo file in a text editor. The experiment setups are stored towards the end of the .nlogo file, in a section that begins and ends with a `experiments` tag. Example:

```
<experiments>
  <experiment name="experiment" repetitions="10" runMetricsEveryTick="true">
    <setup>setup</setup>
    <go>go</go>
    <exitCondition>not any? fires</exitCondition>
    <metric>burned-trees</metric>
    <enumeratedValueSet variable="density">
      <value value="40"/>
      <value value="0.1"/>
      <value value="70"/>
    </enumeratedValueSet>
  </experiment>
</experiments>
```

In this example, only one experiment setup is given, but you can put as many as you want between the beginning and ending `experiments` tags.

Between looking at the DTD, and looking at examples you create in the GUI, it will hopefully be apparent how to use the tags to specify different kind of experiments. The DTD specifies which tags are required and which are optional, which may be repeated and which may not, and so forth.

When XML for experiment setups is included in a model file, it does not begin with any XML headers, because not the whole file is XML, only part of it. If you keep experiment setups in their own file, separate from the model file, then the extension on the file should be .xml not .nlogo, and you'll need to begin the file with proper XML headers, as follows:

```
<?xml version="1.0" encoding="us-ascii"?>
<!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 `us-ascii`, such as `UTF-8`, but NetLogo doesn't support non-ASCII characters in most situations, so specifying a different encoding may be pointless.

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.

Conclusion

BehaviorSpace is still under development. We'd like to hear from you about what additional features would be useful to you in your work. Please write us at feedback@ccl.northwestern.edu.

HubNet Guide

This section of the User Manual introduces the HubNet system and includes instructions to set up and run a HubNet activity.

HubNet is a technology that lets you use NetLogo to run *participatory simulations* in the classroom. In a participatory simulation, a whole class takes part in enacting the behavior of a system as each student controls a part of the system by using an individual device, such as a networked computer or TI-83+ calculator.

For example, in the Gridlock simulation, each student controls a traffic light in a simulated city. The class as a whole tries to make traffic flow efficiently through the city. As the simulation runs, data is collected which can afterwards be analyzed on a computer or calculator.

For more information on participatory simulations and their learning potential, please visit the [Participatory Simulations Project web site](#).

Understanding HubNet

NetLogo

NetLogo is a programmable modeling environment. It comes with a large library of existing simulations, both participatory and traditional, that you can use and modify. Content areas include social science and economics, biology and medicine, physics and chemistry, and mathematics and computer science. You and your students can also use it to build your own simulations. For more about NetLogo, see the [NetLogo Users Manual](#).

In traditional NetLogo simulations, the simulation runs according to rules that the simulation author specifies. HubNet adds a new dimension to NetLogo by letting simulations run not just according to rules, but by direct human participation.

Since HubNet builds upon NetLogo, we recommend that before trying HubNet for the first time, you become familiar with the basics of NetLogo. To get started using NetLogo models, see [Tutorial #1: Running Models](#) in the NetLogo Users Manual.

HubNet Architecture

HubNet simulations are based on a client-server architecture. The activity leader uses the NetLogo application to run a HubNet activity. When NetLogo is running a HubNet activity, we refer to it as a HubNet server. Participants use a client application to log in and interact with the HubNet server.

There are two types of HubNet available. With [Computer HubNet](#), participants run the HubNet Client application on computers connected by a regular computer network. In [Calculator HubNet](#), created in conjunction with Texas Instruments, participants use TI-83+ graphing calculators as clients which communicate via the TI-Navigator system.

We hope to add support for other types of clients such as cell phones and PDA's (Personal Digital Assistants).

Computer HubNet

Activities

The following activities are available in the Models Library, in the Computer HubNet Activities folder. For many models, you will find a discussion of its educational goals and suggested ways to incorporate it into your classroom in the Participatory Simulations Guide on the [Participatory Simulations Project web site](#). More information can also be found in the Information Tab in each model.

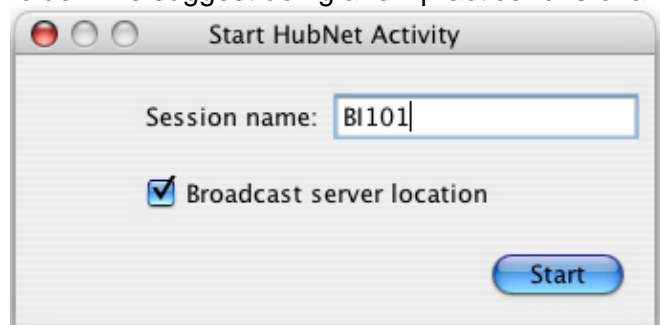
- Disease -- A disease spreads through the simulated population of students.
- Gridlock -- Students use traffic lights to control the flow of traffic through a city.
- Polling -- Ask students questions and plot their answers.
- Tragedy of the Commons -- Students work as farmers sharing a common resource.

Requirements

To use Computer HubNet, you need a networked computer with NetLogo installed for the activity leader, and a networked computer with NetLogo installed for each participant. 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 Computer Activities folder. We suggest doing a few practice runs of an activity before trying it in front of a class.



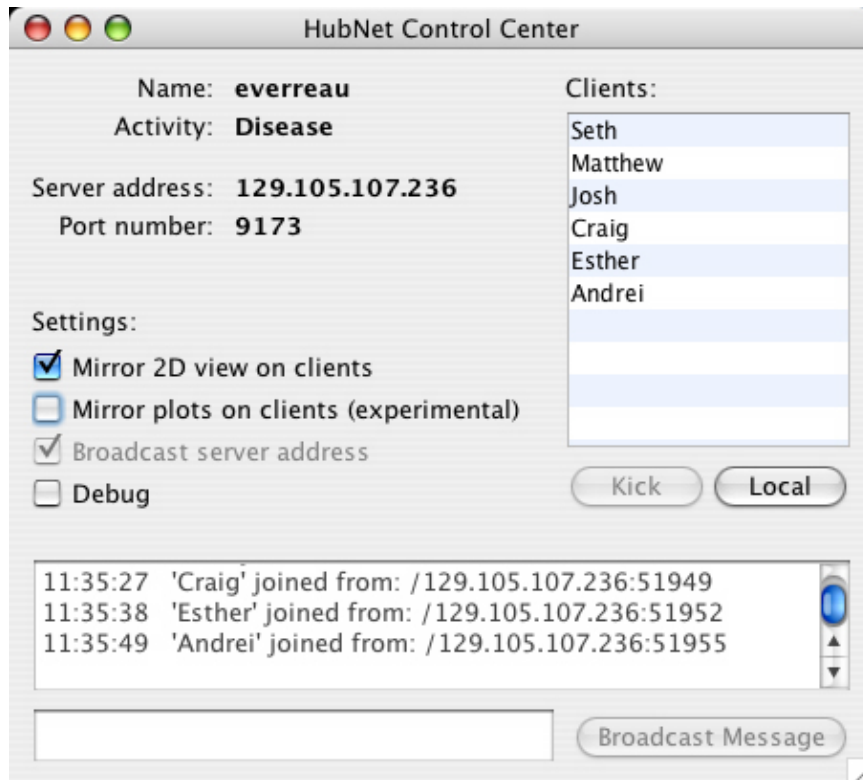
Open a Computer HubNet model. NetLogo will prompt you to enter the name of your new HubNet session. This is the name that participants will use to identify this activity. Enter a name and press Start.

NetLogo will open the HubNet Control Center, which lets you interact with the HubNet server.

In each activity, you'll see a box in the interface tab labeled "QuickStart Instructions". This contains step by step instructions to run the activity. Click the "Next>>>" button to advance to the next instruction.

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" 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" 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 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 name 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 grey.

- Verify that the "Mirror 2D view" checkbox in the HubNet Control Center is selected.
- Make sure that the display switch in the model is on.
- The view on the server must be exactly the same size as on the client. If you changed the size of the view on the HubNet server, you need to restore it to its original dimensions.

There is no view on the HubNet client.

Some activities don't have a view on the client.

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.

Please send us an email at feedback@ccl.northwestern.edu.

Known Limitations

If HubNet malfunctions, please send us an email at bugs@ccl.northwestern.edu.

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.
- Authoring new HubNet activities is more arcane and difficult than it should be.
- Computer HubNet has only been tested on LANs, and not on dial-up connections or WANs.

Calculator HubNet

Requirements

To use Calculator HubNet, you need:

- **A computer with an attached projector.** This computer will run NetLogo and project the simulation for class viewing.
- **A classroom set of Texas Instruments TI-83+ graphing calculators.**
- **The TI-Navigator calculator network from Texas Instruments.**

NOTE: Calculator HubNet works with a prototype version of the TI-Navigator system, and is not yet compatible with the commercially available version. To learn more about the TI-Navigator system, please visit the [Texas Instruments](#) web site.

We are actively working in partnership with Texas Instruments on integrating the new TI-Navigator with Calculator HubNet. We expect to release a new version in the near future.

For more information about Calculator HubNet, please refer to the Participatory Simulations Guide which can be found on the [Participatory Simulations Project web site](#).

Teacher workshops

For information on upcoming workshops and NetLogo and HubNet use in the classroom, please contact us at feedback@ccl.northwestern.edu.

HubNet Authoring Guide

To learn about authoring or modifying HubNet activities, see the [HubNet Authoring Guide](#).

Getting help

If you have any questions about Computer HubNet or Calculator HubNet, or need help getting started, please email us at feedback@ccl.northwestern.edu.

HubNet Authoring Guide

This explains how to use NetLogo to modify the existing HubNet activities or build your own, new HubNet activities.

- [General HubNet Information](#)
- [NetLogo Primitives](#)
 - ♦ [Setup](#)
 - ♦ [Data Extraction](#)
 - ♦ [Sending Data](#)
- [Calculator HubNet Information](#)
- [Computer HubNet Information](#)
 - ♦ [How To Make an Interface for a Client](#)
 - ♦ [View Updates on the Clients](#)
 - ♦ [Plot Updates on the Clients](#)
 - ♦ [Clicking in the View on Clients](#)
 - ♦ [Text Area for Input and Display](#)

General HubNet Information

If you are interested in more general information on what HubNet is or how to run HubNet activities, you should refer to the [HubNet Guide](#).

NetLogo Primitives

This section will introduce the set of primitives used to turn a NetLogo Model into a HubNet Activity. These commands allow you to send data to and receive data from the clients.

Setup

In order to make a NetLogo model into a HubNet Activity, it is necessary to first indicate whether the clients are computers or calculators and then establish a connection between the server (your computer) and the clients (the students' calculators or computers) using the following primitives:

hubnet-set-client-interface client-type client-info

If *client-type* is "COMPUTER", *client-info* is a list containing a string with the file name and path (relative to the model) to the file which will serve as the client's interface. This interface will be sent to any clients that log in.

```
hubnet-set-client-interface "COMPUTER" [ "clients/Disease client.nlogo" ]  
;; when clients log in, they will get the interface described in t  
;; Disease client.nlogo in the clients subdirectory of the model d
```

This primitive must be called before you use any other HubNet primitives including `hubnet-reset` so NetLogo knows which type of HubNet you are going to be using.

hubnet-reset

Starts up the HubNet system. HubNet must be started to use any of the other HubNet primitives with the exception of `hubnet-set-client-interface`. HubNet remains running as long as this model is open; it stops running when the model is closed or you quit NetLogo.

If you are using Computer HubNet, you will be prompted for a session name. This is an identifier to make servers discovered by the client uniquely identifiable.

These primitives are usually called from the `startup` procedure rather than `setup` of the NetLogo model since they should only be called **once** in a model.

Data extraction

During the activity you will be transferring data between the HubNet clients and the server. The following primitives allow you to extract data from the clients:

hubnet-message-waiting?

This looks for new information sent by the clients. It reports TRUE if there is new data, and FALSE if there is not.

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`. This will cause an error if there is no new data from the clients. So be sure to check for data with `hubnet-message-waiting?` before calling this.

hubnet-message-source

This reports the user name of the client that sent the data. This will cause an error if no data has been fetched. So be sure to fetch the data with `hubnet-fetch-message` before calling this.

hubnet-message-tag

This reports the tag that is associated with the data that was sent. For Calculator HubNet, this will report one of the variable names set with the `hubnet-set-client-interface` primitive. For Computer HubNet, this will report one of the Display Names of the interface elements in the client interface. (See [below](#) for more information about the Computer HubNet tags.) For both types of HubNet, this primitive will cause an error if no data has been fetched. So be sure to fetch the data with `hubnet-fetch-message` before calling this.

hubnet-message

This reports the data collected by `hubnet-fetch-message`. This will cause an error if no data has been fetched. So be sure to fetch the data with `hubnet-fetch-message` before calling this.

There are two additional data extraction primitives that are only used in Computer HubNet models.

hubnet-enter-message?

Reports true if a new computer client just entered the simulation. Reports false otherwise.

hubnet-exit-message?

Reports true if a new computer client just exited the simulation. Reports false otherwise.

For both `hubnet-enter-message?` and `hubnet-exit-message?`, `hubnet-message-source` will contain the user name of the client that just logged on or off. Also, if `hubnet-message` and `hubnet-message-tag` are used while `hubnet-enter-message?` or `hubnet-exit-message?` are true, a Runtime Error will be given.

Generally part of your go procedure will include checking for waiting messages and handling them.

```
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

```

Sending data

It is also possible to send data from NetLogo to the clients. For Calculator HubNet, NetLogo sends the data to the Navigator server, and then the calculators can then access it. For Computer HubNet, NetLogo is able to send the data directly to the clients.

The primitives for sending data to the server are:

hubnet-broadcast tag-name value

This broadcasts *value* from NetLogo to the variable, in the case of Calculator HubNet, or interface element, in the case of Computer HubNet, with the name *tag-name* to all the clients.

hubnet-broadcast-view

This broadcasts the current state of the 2D View in the NetLogo model to all the Computer HubNet Clients. It does nothing for Calculator HubNet.

hubnet-send list-of-strings tag-name value

hubnet-send string tag-name value

When using Calculator HubNet this primitive acts in exactly the same manner as *hubnet-broadcast*. For Computer HubNet, it has the following effects:

- ◊ When *string* is the first input, this sends *value* from NetLogo to the tag *tag-name* on the client that has *string* for a user name.
- ◊ When *list-of-strings* is the first input, 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*.
- ◊ All the information for the current state of the View is sent at this time, regardless of whether the clients' Views were already up to date.

Note: Sending a message to a non-existent client, using *hubnet-send*, generates a *hubnet-exit-message*.

hubnet-send-view string

hubnet-send-view list-of-strings

For Calculator HubNet, does nothing.

For Computer HubNet, it acts as follows:

- ◊ For a *string*, this sends the current state of the 2D View in the NetLogo model to the Computer HubNet Client with *string* for its user name.
- ◊ For a *list-of-strings*, this sends the current state of the 2D View in the NetLogo model to all the Computer HubNet clients that have a user name that is in the *list-of-strings*.
- ◊ All the information for the current state of the view is sent at this time, regardless of whether the clients' views were already up to date.

Note: Sending the View to a non-existent client, using *hubnet-send-view*, generates a *hubnet-exit-message*.

When using Calculator HubNet the `hubnet-send` and the `hubnet-broadcast` primitives, take a number, a string, a list of numbers, or a matrix (a list of lists) of numbers as the `value` input. When using Computer HubNet, you may send any kind of information with the exceptions of patches, turtles, and agentsets.

Here are some examples of using the two primitives to send various types of data that you can send:

data type	hubnet-broadcast example	hubnet-send example
number	<code>hubnet-broadcast "A" 3.14</code>	<code>hubnet-send "jimmy" "A" 3.14</code>
string	<code>hubnet-broadcast "STR1"</code> <code>"HI THERE"</code>	<code>hubnet-send ["12" "15"] "STR1"</code> <code>"HI THERE"</code>
list of numbers	<code>hubnet-broadcast "L2" [1 2 3]</code>	<code>hubnet-send</code> <code>hubnet-message-source "L2" [1 2 3]</code>
matrix of numbers	<code>hubnet-broadcast "[A]" [[1 2] [3 4]]</code>	<code>hubnet-send "susie" "[A]" [[1 2] [3 4]]</code>
list of strings (only for Computer HubNet)	<code>hubnet-broadcast</code> <code>"user-names" [{"jimmy"</code> <code>"susie"} [{"bob" "george"}]</code>	<code>hubnet-send "teacher"</code> <code>"user-names" [{"jimmy" "susie"}]</code> <code>["bob" "george"]]</code>

Examples

Study the models in the "HubNet Computer Activities" and the "HubNet Calculator Activities" sections of the Models Library to see how these primitives are used in practice in the Procedures window. Disease is a good one with which to start.

Calculator HubNet Information

The calculators are able to send and receive the following data types from NetLogo:

- Valid calculator lists, such as `L1` or `PLOTS`
- Valid calculator matrices, such as `[A]` or `[B]`
- Valid calculator strings, such as `Str1` or `Str5`
- Numbers, such as `A` or `B`

The length of the list of numbers that a calculator sends depends on what information you want to send to the NetLogo model. Further, how those numbers are interpreted by the model is also up to you.

For more information on writing the calculator program portion of a HubNet Activity, please [contact us](#).

Saving

The data sent by calculators or NetLogo is saved in the order that the server receives the data.

Computer HubNet Information

The following information is specific to Computer HubNet.

How To Make an Interface for a Client

Open a new model in NetLogo. Add any interface buttons, sliders, switches, monitors, plots, choosers, or text boxes that you want in the Interface Tab. For buttons and monitors, you only need to type a Display Name. Any code you write in the Code or Reporter sections will be ignored. The Display Name you give to the interface element is the tag that is returned by the `hubnet-message-tag` reporter in the NetLogo code.

For example, if in the Interface Tab of the client interface you had 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:

Be aware that this causes the restriction that you can only have **one** interface element with a specific name. Having more than one interface element with the same Display Name in the client interface will cause unpredictable behavior. For instance, if we had a monitor called Milk Supply and a plot named Milk Supply, when we send data to the client using the tag Milk Supply, the client will just pick either the plot or the monitor to give the data to.

If you wish to have a View in the client for a model, the view in the client and the one in the NetLogo model must have the same number of patches and the same patch size. If they do not, the view on the client will not display information sent by the server.

If you wish to make a client without a view in the client, you will have to hand edit the file after you have finished adding all the other interface elements in NetLogo. To do this, open the client file in a text editor such as Notepad on Windows, or TextEdit on Macs. You should see a file that starts with something similar to this:

```
; add model procedures here

@#$#@#$#@
GRAPHICS-WINDOW
321
10
636
325
17
17
9.0
1
10
0
0
```

```
CC-WINDOW
323
339
638
459
Command Center
```

You should remove all the text that is in the GRAPHICS-WINDOW section and then save the file. So that after you are done the beginning of the file should look similar to this:

```
; add model procedures here
```

```
@#$@#$#@
CC-WINDOW
323
339
638
459
Command Center
```

For more examples, study the models and interface files in the "HubNet Computer Activities" section of the Models Library. Disease.nlogo and Disease client.nlogo are good ones to start with.

View Updates on the Clients

Currently, there are two ways of sending the clients the View. The first way is done automatically by NetLogo and HubNet when 2D View mirroring is enabled and the client has a View in the interface. Whenever a patch or turtle is redrawn in the NetLogo View, it will be redrawn on **all** the clients. Actually, updates are accumulated and sent out periodically (about five times a second). This means that a lot of messages can be sent to the clients if a lot of turtles or patches are being redrawn. It is possible to reduce the number of messages sent to the clients, and thus possibly speed up the model, by making the View in the model not update. This can be done using the `no-display` and `display` primitives or by toggling the display on/off switch in the View Control Strip.

A second way of sending the clients the View is to use the `hubnet-broadcast-view` and `hubnet-send-view` primitives. `hubnet-broadcast-view` and `hubnet-send-view` both send the entire View to the clients instead of just the patches that need to be redrawn. This makes them less efficient, but for some models this feature is necessary. To send the View to the clients using this scheme, you must use the following NetLogo code:

```
hubnet-broadcast-view
```

to send to all the logged in clients.

To just send the View to a subset of all the clients use:

```
hubnet-send-view user-name-list
```

where *user-name-list* is either a single string or a list of strings of the user names of clients that you want to send it to.

If there is no View in the clients or if the Mirror View on Clients checkbox in the HubNet Control Center is not checked, then no view messages are sent to the clients.

NOTE: Since `hubnet-broadcast-view` and `hubnet-send-view` are experimental primitives, their behaviors may change in a future release.

Note: Some of the View features in NetLogo are not yet implemented on the HubNet clients such as View Wrapping and Observer Perspectives.

Plot Updates on the Clients

When 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.

If there is no plot with the exact same name in the clients or if the Mirror Plots on Clients checkbox in the HubNet Control Center is not checked, then no plot updates are sent to the clients.

Clicking in the View on Clients

If the View is included in the client, it is possible for the client to send locations in the View to NetLogo by clicking in the client's View. The tag reported by `hubnet-message-tag` for client clicks is the same as what is needed to send the View to a client, the string "View".

`hubnet-message` reports a two item list with the x coordinate being the first item and the y coordinate being the second item. So 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 ]
]
```

Text Area for Input and Display

A few models use an experimental interface element in the HubNet client that allows the modeler to display text on the client that can change throughout the run of the activity. Further, it can allow users to send text back to the server. If you are interested in using it in an activity, please [contact us](#) for further information.

Extensions Guide

NetLogo allows users to write new commands and reporters in Java and use them in their models. This section of the User Manual introduces this facility.

The first part discusses how to use an extension in your model once you have written one, or once someone has given you one.

The second part is intended for Java programmers interested in writing their own extensions.

Caution! The extensions facility is new in NetLogo 2.0.1 and is still in an early stage of development. Therefore it is considered "experimental". It is likely to continue to change and grow. If you write an extension now, it may need changes in order to continue to work in future NetLogo versions.

- [Using Extensions](#)
- [Writing Extensions](#)

The [NetLogo API Specification](#) contains further details.

Using Extensions

NetLogo extensions have names that end in ".jar" (short for "Java Archive").

To use an extension in a model, add the `__extensions` keyword at the beginning of the Procedures tab, before declaring any breeds or variables. (The keyword begins with two underscores to indicate that it is experimental. In a future NetLogo version, it may have a different name and syntax.)

`__extensions` takes one input, a list of strings. Each string contains the name of a NetLogo extension. For example:

```
__extensions [ "sound.jar" ]
```

NetLogo will look for extensions in two places: the directory that holds the model, and the NetLogo extensions folder.

So to install a NetLogo extension for use by any model, put the extension file (for example, "sound.jar") in the "extensions" directory inside the NetLogo directory. Or, you can just keep the extension in the same folder as the model that uses it.

You can also use extensions that are not installed in your NetLogo extensions folder by providing a path relative to the directory that contains the model, or an absolute path:

```
__extensions [ "lib/sound.jar" ]      ;; relative path
__extensions [ "../jars/sound.jar" ]  ;; relative path
__extensions [ "c:\\myfiles\\sound.jar" ] ;; absolute Windows path
__extensions [ "/Users/me/sound.jar" ]  ;; absolute Mac/Unix path
```

You may also use an extension which is stored on an Internet server instead of your local computer.

Just use the URL where you have stored the extension. For example:

```
__extensions [ "http://yourdomain.net/jars/sound.jar" ]
```

Using `__extensions` tells NetLogo to find and open the specified extension and makes the custom commands and reporters found in the extension available to the current model. You can use these commands and reporters just as if they were built-in NetLogo primitives.

To use more than one extension, list each extension separately. For example,

```
__extensions [ "sound.jar" "speech.jar" ]
```

Some extensions depend on additional files. Check the instructions that come with the extension to see if you need to keep any other files with the extension.

Applets

Models saved as applets (using "Save as Applet" on NetLogo's File menu) can make use of extensions. However, applets still cannot use extensions that require additional external jars. (We plan to fix this in a future release.)

Writing Extensions

We assume you have experience programming in Java.

Summary

A NetLogo extension is a JAR that contains:

- one or more classes that implement `org.nlogo.api.Primitive`,
- a main class that implements `org.nlogo.api.ClassManager`, and
- a NetLogo extension manifest file.

The manifest file must contain four tags:

- `Manifest-Version`, always 1.0
- `Extension-Name`, the name of the extension.
- `Class-Manager`, the fully-qualified name of a class implementing `org.nlogo.api.ClassManager`.
- `NetLogo-Version`, the version of NetLogo for which this JAR is intended. If a user opens the extension with a different version of NetLogo, a warning message is issued.

Tutorial

Let's write an extension that provides a single reporter called `first-n-integers`.

`first-n-integers` will take a single numeric input *n* and report a list of the integers 0 through *n* – 1. (Of course, you could easily do this just in NetLogo; it's only an example.)

1. Write primitives

A command performs an action; a reporter reports a value. To create a new command or reporter, create a class that implements the interface [org.nlogo.api.Command](#) or [org.nlogo.api.Reporter](#), which extend [org.nlogo.api.Primitive](#). In most cases, you can extend the abstract class [org.nlogo.api.DefaultReporter](#) or [org.nlogo.api.DefaultCommand](#).

`DefaultReporter` requires that we implement:

```
Object report (Argument args[], Context context)
    throws ExtensionException;
```

Since our reporter takes an argument, we also implement:

```
Syntax getSyntax();
```

Here's the implementation of our reporter, in a file called `IntegerList.java`:

```
import org.nlogo.api.*;

public class IntegerList extends DefaultReporter
{
    // take one number as input, report a list
    public Syntax getSyntax() {
        return Syntax.reporterSyntax(
            new int[] {Syntax.TYPE_NUMBER}, Syntax.TYPE_LIST
        );
    }

    public Object report(Argument args[], Context context)
        throws ExtensionException
    {
        // create a NetLogo list for the result
        LogoList list = new LogoList();

        // use typesafe helper method from
        // org.nlogo.api.Argument to access argument
        int n = args[0].getIntegerValue();

        if (n < 0) {
            // signals a NetLogo runtime error to the modeler
            throw new ExtensionException
                ("input must be positive");
        }

        // populate the list
        for (int i = 0; i < n; i++) {
            list.add(new Integer(i));
        }
        return list;
    }
}
```

Notice:

- To access arguments, use [org.nlogo.api.Argument](#)'s typesafe helper methods, such

as `getIntegerValue()`.

- Throw `org.nlogo.api.ExtensionException` to signal a NetLogo runtime error to the modeler.

A Command is just like a Reporter, except that reporters implement `Object report(...)` while commands implement `void perform(...)`.

2. Write a ClassManager

Each extension must include, in addition to any number of command and reporter classes, a class that implements the interface `org.nlogo.api.ClassManager`. The ClassManager tells NetLogo which primitives are part of this extension. In simple cases, extend the abstract class `org.nlogo.api.DefaultClassManager`, which provides empty implementations of the methods from ClassManager that you aren't likely to need.

Here's the class manager for our example extension, `SampleExtension.java`:

```
import org.nlogo.api.*;

public class SampleExtension extends DefaultClassManager {
    public void load(PrimitiveManager primitiveManager) {
        primitiveManager.addPrimitive
            ("first-n-integers", new IntegerList());
    }
}
```

`addPrimitive()` tells NetLogo that our reporter exists and what its name is.

3. Write a Manifest

The extension must also include a manifest. The manifest is a text file which tells NetLogo the name of the extension and the location of the ClassManager.

The manifest must contain three tags:

- `Extension-Name`, the name of the extension.
- `Class-Manager`, the fully-qualified name of a class implementing `org.nlogo.api.ClassManager`.
- `NetLogo-Version`, the version of NetLogo for which this JAR is intended. If a version mismatch is detected when a JAR is imported, a warning message will be issued, and the user will have the opportunity to cancel. If the user chooses to continue, NetLogo will attempt to import the JAR anyway, which of course may fail.

Here's a manifest for our example extension, `manifest.txt`:

```
Manifest-Version: 1.0
Extension-Name: example
Class-Manager: SampleExtension
NetLogo-Version: 3.1.3
```

The `NetLogo-Version` line should match the actual version of NetLogo you are using.

Make sure even the last line ends with a newline character.

4. Create a JAR

To create an extension JAR, first compile your classes as usual. Make sure `NetLogo.jar` (from the NetLogo distribution) is in your classpath. For example:

```
$ javac -classpath NetLogo.jar IntegerList.java SampleExtension.java
```

Then create a JAR containing the resulting class files and the manifest. For example:

```
$ jar cvfm example.jar manifest.txt IntegerList.class SampleExtension.class
```

For information about manifest files, JAR files and Java tools, see java.sun.com.

5. Use your extension in a model

To use our example extension, put the extension JAR in the NetLogo extensions folder, or in the same directory as the model that will use the extension. At the top of the Procedures tab write:

```
__extensions [ "example.jar" ]
```

Now you can use `first-n-integers` just like it was a built-in NetLogo reporter. For example, select the Interface tab and type in the Command Center:

```
observer> show first-n-integers 5
observer: [0 1 2 3 4]
```

Extension development tips

Debugging extensions

There are special NetLogo primitives to help you as you develop and debug your extension. Like the extensions facility itself, these are considered experimental and will be changed at a later date. (That's why they have underscores in their name.)

- `print __dump-extensions` prints information about loaded extensions
- `print __dump-extension-prims` prints information about loaded extension primitives
- `__reload-extensions` forces NetLogo to reload all extensions the next time you compile your model. Without this command, changes in your extension JAR will not take effect until you open a model or restart NetLogo.

Third party JARs

If your extension depends on code stored in a separate JAR, copy the extra JARs into the "extensions" directory of the NetLogo installation. Whenever an extension is imported, NetLogo makes all the JARs in this folder available to the extension.

If you plan to distribute your extension to other NetLogo users, make sure to provide installation instructions that describe which files should be copied to their extensions directory.

Conclusion

Don't forget to consult the [NetLogo API Specification](#) for full details on these classes, interfaces, and methods.

Note that there is no way for the modeler to get a list of commands and reporters provided by an extension, so it's important that you provide adequate documentation.

The extensions facility is considered experimental. This initial API doesn't include everything you might expect. Some facilities exist but are not yet documented. If you don't see a capability you want, please let us know. Do not hesitate to contact us at feedback@ccl.northwestern.edu with questions, as we may be able to find a workaround or provide additional guidance where our documentation is thin.

Hearing from users of this API will also allow us to appropriately focus our efforts for future releases. We are committed to making NetLogo flexible and extensible, and we very much welcome your feedback.

Controlling Guide

NetLogo can be invoked from another Java program and controlled by that program. For example, you might want to call NetLogo from a small program that does something simple like automate a series of model runs.

This section of the User Manual introduces this facility for Java programmers. We'll assume that you know the Java language and related tools and practices.

Note: The controlling facility is considered "experimental". It is likely to continue to change and grow. Code you write now that uses it may need changes in order to continue to work in future NetLogo versions.

- [Note on memory usage](#)
- [Example \(with GUI\)](#)
- [Example \(headless\)](#)
- [BehaviorSpace](#)
- [Other Options](#)
- [Conclusion](#)

The [NetLogo API Specification](#) contains further details.

Note on memory usage

In all of the examples below, when invoking Java to run them, you probably don't want to accept the default heap size settings. Most Java VM's have a very small initial heap size and a small maximum heap size too. When you run the NetLogo application, it uses an initial heap size of 16 megabytes and a maximum heap size of 512 megabytes. This is enough for most models. Here's how to specify the heap sizes on the command line:

```
java -server -Dsun.java2d.noddraw=true -Xms16M -Xmx512M ...
```

(Note that we recommend the `-server` flag for best performance in most situations. We also recommend setting the `noddraw` option to true when using the GUI as Direct Draw can conflict with OpenGL.)

Example (with GUI)

Here is a small but complete program that starts the full NetLogo application, opens a model, moves a slider, sets the random seed, runs the model for 50 ticks, and then prints a result:

```
import org.nlogo.app.App;
import java.awt.EventQueue;

public class Example1 {
    public static void main(String[] argv) {
        App.main(argv);
        try {
            EventQueue.invokeAndWait
                ( new Runnable()
                  { public void run() {
```

```

        try {
            App.app.open
                ("models/Sample Models/Earth Science/"
                 + "Fire.nlogo");
        }
        catch( java.io.IOException ex ) {
            ex.printStackTrace();
        }
    } } );

    App.app.command("set density 62");
    App.app.command("random-seed 0");
    App.app.command("setup");
    App.app.command("repeat 50 [ go ]");
    System.out.println
        (App.app.report("burned-trees"));
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}
}

```

In order to compile and run this, `NetLogo.jar` (from the NetLogo distribution) must be in the classpath.

Note the use of `EventQueue.invokeLater` to ensure that a method is called from the right thread. This is because most of the methods on the `App` class may only be called some certain threads. Most of the methods may *only* be called from the AWT event queue thread; but a few methods, such as `command()`, may only be called from threads *other* than the AWT event queue thread (such as, in this example, the main thread).

Rather than continuing to discuss this example in full detail, we refer you to the [NetLogo API Specification](#), which documents all of the ins and outs of the classes and methods used above. Additional methods are available as well.

Example (headless)

The example code in this case is very similar to the previous example, but with methods on an instance of the `HeadlessWorkspace` class substituted for static methods on `App`.

```

import org.nlogo.headless.HeadlessWorkspace;

public class Example2 {
    public static void main(String[] argv) {
        HeadlessWorkspace workspace =
            new HeadlessWorkspace() ;
        try {
            workspace.open
                ("models/Sample Models/Earth Science/"
                 + "Fire.nlogo");
            workspace.command("set density 62");
            workspace.command("random-seed 0");
            workspace.command("setup");
            workspace.command("repeat 50 [ go ]" );
            System.out.println
                (workspace.report("burned-trees"));
            workspace.dispose();
        }
    }
}

```

```

    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}

```

In order to compile and run this, either `NetLogo.jar` or `NetLogoLite.jar` (from the NetLogo distribution) must be in your classpath. (The latter jar is smaller, but is only capable of headless operation, not full GUI operation.) When running in a context that does not support a graphical display, the system property `java.awt.headless` must be true, to force Java to run in headless mode; `HeadlessWorkspace` automatically sets this property for you.

Since there is no GUI, NetLogo primitives which send output to the command center or output area now go to standard output instead. `export-world` can still be used to save the model's state. `export-view` works for writing an image file with a snapshot of the (otherwise invisible) 2D view. The `report()` method is useful for getting results out of the model and into your Java code.

You can make multiple instances of `HeadlessWorkspace` and they will operate independently on separate threads without interfering with each other.

When running headless, there are some restrictions:

- Plotting primitives are non-functional. (However, calling them will not produce any ill effects.)
- The `movie-*` primitives are not available; trying to use them will cause a Java exception.
- `user-*` primitives which query the user for input, such as `user-yes-or-no` will cause a Java exception.
- Sliders, switches, and choosers do not enforce constraints on the values they accept. For example, in the full NetLogo application, if you try to set a switch to a value other than true or false, the switch will not accept the new value. Similarly, a chooser enforces that its value is a valid choice, and a slider enforces that its value is permitted by its minimum, maximum, and increment. When running headless, none of these checks occur.

We plan to lift these restrictions in a future version of NetLogo.

The [NetLogo API Specification](#) contains further details.

BehaviorSpace

The Controlling API supports running BehaviorSpace experiments headless. (It does not support running them in BehaviorSpace's GUI, although you can write your own BehaviorSpace-like Java code to run your own BehaviorSpace-like experiments if you want.)

Note that it is definitely not necessary to use the API to do headless BehaviorSpace runs. Headless BehaviorSpace is supported directly from the command line with no Java programming at all required. See the [BehaviorSpace Guide](#) for instructions.

In most cases, the command line support will be enough, without needing to use the API. In some situations, though, you may want additional flexibility afforded by the API.

The `HeadlessWorkspace` has four methods for running experiments: three variants of

`runExperiment`, plus `runExperimentFromModel`.

`runExperimentFromModel` is used when the experiment setup is already stored in the model file.

The two forms of `runExperiment` that take `File` arguments are used when the experiment setup is stored in a standalone XML file, separate from the model file. If the file contains only one setup, you only need to pass in the `File` object. If the file contains multiple setups, you must also pass in a `String` object holding the experiment name.

The form of `runExperiment` that takes only a `String` argument (and an argument to specify the output format) is used to pass the XML for the experiment setup directly.

All of these methods take a `PrintWriter` as a destination for the results. If you just want to send them to standard output, you can pass `new java.io.PrintWriter(System.out)`.

The [BehaviorSpace Guide](#) explains how to specify experiment setups in XML.

The [NetLogo API Specification](#) contains further details on the `HeadlessWorkspace` class and its methods.

Other Options

When your program controls NetLogo using the `App` class, the entire NetLogo application is present, including tabs, menubar, and so forth. This arrangement is suitable for controlling or "scripting" a NetLogo model, but not ideal for embedding a NetLogo model in a larger application.

We also have a separate, similar API which allows embedding only parts of NetLogo, such as only the tabs (not the whole window), or only the contents of the Interface tab. At present, this additional API is not documented. If you are interested in using it, please contact us at feedback@ccl.northwestern.edu.

Conclusion

Don't forget to consult the [NetLogo API Specification](#) for full details on these classes and methods.

As mentioned before, the controlling facility is considered experimental. This initial API doesn't necessarily include everything you might expect. Some facilities exist, but are not yet documented. So if you don't see the capability you want, contact us; we may be able to help you do what you want. Please do not hesitate to contact us at feedback@ccl.northwestern.edu with questions, as we may be able to find a workaround or provide additional guidance where our documentation is thin.

GoGo Extension

What is the GoGo Board?

The GoGo Board extension lets you connect NetLogo to the physical world, using sensors, motors, light bulbs, LEDs, relays and other devices. The GoGo Extension for NetLogo provides primitives to communicate with a GoGo board via a serial interface.

A GoGo Board is an open source, easy-to-build, low cost, general purpose board especially designed to be used in educational projects. It was created by Arnan Sipitakiat at the MIT Media Lab. A GoGo Board has 8 sensor ports and 4 output ports, and also a connector for add-on boards (such as a display or a wireless communication module). Using the GoGo Board extension, NetLogo models can interact with the physical world in two ways. First, it can gather data from the environment, such as temperature, ambient light, or user input. This information can be used by the model to change or calibrate its behavior. Secondly, it can control output devices – NetLogo could control motors, toys, remote controlled cars, electrical appliances, light bulbs, and automated laboratory equipment.

How to get a GoGo Board?

The GoGo Board is not a commercial product, and thus cannot be bought at stores. To get a GoGo Board, you have to build one yourself or ask someone to do it for you. The board was especially designed to be easy and cheap to build, even if you don't have electronics skills. The main resource about the GoGo Board is the web site www.gogoboard.org, where you will find step-by-step instructions on how to buy components, design the printed circuit board, and assemble it. The GoGo Board mailing list is gogoboard@yahoogroups.com.

Installing the GoGo Extension

The GoGo Board needs to communicate with the computer in some way, and to do so it uses the serial port. The choice of this port instead of a USB port was motivated by the board's low cost principle: the components needed to build a USB compatible board would be more expensive. If your computer does not have a serial port, you need to purchase a USB-to-Serial adapter, which can be easily found in computer stores with prices ranging from US\$ 15 to US\$ 30 (if you have a Mac or Linux machine, make sure the adapter is compatible with your platform). To communicate with the GoGo Board through the serial port, the GoGo Extension uses Sun Microsystems' [Java Communications API](#). In each platform (PC, Mac, Linux), the procedures for installing the software needed to enable serial communication are slightly different.

Mac OS X

There is no official implementation of the Java Communications API for OS X, but the [RXTX](#) project provides an open-source implementation. You can [download](#) the RXTX installer for OS X from SourceForge. Be sure to follow all the directions in the installer to create lock file directories and make sure your user is in the appropriate groups to use the lock files.

There are several commercial implementations of the Java Communications API for OS X which have not been tested with the GoGo extension, but, in theory, should work. Please contact us if you

successfully use them, or run into problems trying.

Windows

Sun provides an implementation of the Java Communications API for Windows, which you can [download](#).

Once downloaded, extract the files into a temporary directory. Several files need to be copied into your Java Runtime Environment, or JRE, installation. If you are using the version of NetLogo which comes with its own Java VM, then your JRE installation is in the `jre` subdirectory of the NetLogo folder. Otherwise, it is in a directory like `c:\j2sdk1.4`. The files `comm.jar` and `javax.comm.properties` must be copied into the `lib` folder of the JRE installation. The files `win32comm.dll` must be copied to the `bin` folder of the JRE installation. The file `PlatformSpecific` has more detailed instructions.

Linux and others

There is no official implementation of the Java Communications API for Linux. You can use the [RXTX](#) implementation. Kevin Hester has written some [installation instructions](#).

Using the GoGo Extension

The GoGo Extensions comes preinstalled. To use the extension in your model, add a line to the top of your procedures tab:

```
__extensions [ "gogo.jar" ]
```

After loading the extension, see what ports are available by typing the following into the command center:

```
show gogo-ports
```

You can open the serial port the GoGo Board is connected to with the `gogo-open` command, and see if the board is responding with the `ping` reporter.

On Windows:

```
gogo-open "COM1"
show ping
```

On Linux:

```
gogo-open "/dev/ttyS01"
show ping
```

For more information on NetLogo extensions, see the [Extensions Guide](#).

Models saved as applets (using "Save as Applet" on NetLogo's File menu) cannot use the Gogo extension, since applets can't use extensions that require additional extra jars. (Also, unsigned applets aren't allowed to access external devices anyway.)

For examples that use the GoGo extension, see the GoGo section under Code Examples in NetLogo's Models Library.

Primitives

[gogo-open](#) [gogo-open](#) [gogo-open?](#) [gogo-ports](#) [output-port-coast](#) [output-port-off](#)
[output-port-reverse](#) [output-port-\[that|this\]way](#) [ping](#) [sensor](#) [set-output-port-power](#)
[talk-to-output-ports](#)

gogo-close

gogo-close

Close the connection to the GoGo Board.

See also [gogo-open](#) and [gogo-open?](#).

gogo-open

gogo-open *port-name*

Open a connection to the GoGo Board connected to serial port named *port-name*. See [gogo-ports](#) for more information about port names.

If the GoGo Board is not responding, or you attempt to open a port without a GoGo Board connected to it, an error will be generated.

Example:

```
gogo-open "COM1"
```

See also [gogo-open](#) and [gogo-close](#).

gogo-open?

gogo-open?

Reports true if there is a connection to a GoGo board open. Reports false otherwise.

gogo-ports

gogo-ports

Reports a list of serial port names which a GoGo Board **may** be connected to. On certain computers, you might get a list of two or three different serial ports. In that case, try to open each of them until the connection is successful.

output-port-coast

output-port-coast

Turns off the power of the active ports. When attached to motors, does not apply a braking force as output-port-off does. Therefore, the motor will gradually slow down before stopping completely. This will have the same effect as output-port-off on most output devices other than motors. The output-ports effected by this command are determined by the talk-to-output-ports command.

The following code will will turn on output port a for 1 second, and then stop the motor gradually:

```
talk-to-output-ports ["a"]
output-port-on
wait 1
output-port-coast
```

output-port-off

output-port-off

Turns off power to the output ports. If using motors, a braking force is applied. The output ports effected by this command are determined by the talk-to-output-ports command.

output-port-reverse

output-port-reverse

Reverses the direction of the output ports. The output ports effected by this command are determined by the talk-to-output-ports command.

output-port-[that/this]way

output-port-thatway

output-port-thisway

Apply power to the output port in a given direction. Output ports can be powered in two directions, arbitrarily called *thisway* and *thatway*. The output-ports effected by the command are determined by the talk-to-output-ports command. Note that this is different from output-port-reverse because *thisway* and *thatway* will always be the same direction provided the connector's polarity is the same.

talk-to-output-ports

talk-to-output-ports *output-portlist*

This command will set the corresponding output ports as active. They will be the ones affected by the commands such as output-port-on and output-port-off. The user can talk to one or multiple ports at the same time. Output ports are typically connected to motors, but you could also use bulbs, LEDs and relays. Output ports are identified by one letter names: "a", "b", "c", and "d".

Examples:

```
;; talk to all output-ports
talk-to-output-ports [ "a" "b" "c" "d" ]
;; will give power to all output-ports
output-port-on

;; talk to output-ports A and D
talk-to-output-ports [ "a" "d" ]
;; will turn off output-ports A and D.
;; The other output-ports will keep
;; their current state
output-port-off

talk-to-output-ports [ "c" "b" ]
;; turn off remaining output-ports
output-port-off
```

ping**ping**

Checks the status of GoGo board. This is mostly used to make sure the board is connected to the correct serial port. It reports true if the GoGo Board responds to a diagnostic message, and false otherwise.

Example:

```
show ping
```

sensor**sensor *sensor***

Reports the value of the sensor named *sensor* as a number. Sensors are named by numbers 1 to 8. Value ranges between 0–1023. 1023 is returned when there is no sensor attached to the port (highest resistance), or when the sensor is an open state. Zero is returned when the sensor is short circuited (no resistance).

Examples:

```
show sensor 1
;; will show the value of sensor 1

foreach [ 1 2 3 4 5 6 7 8 ]
  [show (word "Sensor " ? " = " sensor ?)]
;; will show the value of all sensors in the Command Center

if sensor 1 < 500 [ ask turtles [ fd 10 ]]
;; will move all turtles 10 steps forward if sensor 1's value is less than 500.

forever [if sensor 1 < 500 [ ask turtles [ fd 10 ] ] ]
;; will continuously check sensor 1's value and
;; move all turtles 10 steps forward every time
;; that the sensor value is less than 500.
```

set-output-port-power

set-output-port-power *power-level*

Sets the power level of the active output ports. *power-level* is a number between 0 (off) and 7 (full-power). The output-ports effected by those command are determined by the talk-to-output-ports command. Note that for many practical applications it is more efficient to use mechanical devices, such as gears and pulleys, to control the torque of motors.

Example:

```
talk-to-motors ["a" "b" "c" "d"]
set-motor-power 4
;; will lower the power of all output ports by half of the full power .
```

Sound Extension

The Sound Extension for NetLogo provides primitives to add sound to NetLogo models.

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.

Using the Sound Extension

The sound extension comes preinstalled. To use the extension in your model, add a line to the top of your procedures tab:

```
__extensions [ "sound.jar" ]
```

For more information on NetLogo extensions, see the Extensions Guide. Please note that the NetLogo extensions facility is under development and is still considered experimental, so the syntax is likely to change in a future version of NetLogo. Models saved as applets (using "Save as Applet" on NetLogo's File menu) cannot make use of extensions that require additional external jars. (We plan to fix this in a future release.)

For examples that use the sound extension, see the Sound section under Code Examples in the NetLogo Models Library.

Primitives

drums instruments play-drum play-note play-note-later start-note stop-note stop-instrument stop-music

drums

drums

Reports a list of the names of the 47 drums for use with "play-drum".

instruments

instruments

Reports a list of the names of the 128 instruments for use with "play-note", "play-note-later", "start-note" and "stop-note".

play-drum

play-drum *drum velocity*

Plays a drum.

```
play-drum "ACOUSTIC SNARE" 64
```

play-note

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
play-note "TRUMPET" 60 64 2
```

play-note-later

play-note-later *delay instrument keynumber velocity duration*

Waits for the specified delay before playing the note for a specified duration, in seconds. The agent does not wait for the note to finish before continuing to next command.

```
;; in one seconds play a trumpet at middle C for two seconds
play-note-later 1 "TRUMPET" 60 64 2
```

start-note

start-note *instrument keynumber velocity*

Starts a note.

The note will continue until "stop-note", "stop-instrument" or "stop-music" is called.

```
;; play a violin at middle C
start-note "VIOLIN" 60 64

;; play a C-major scale on a xylophone
foreach [60 62 64 65 67 69 71 72] [
  start-note "XYLOPHONE" ? 65
  wait 0.2
  stop-note "XYLOPHONE" ?
]
```


stop-note

stop-note *instrument keynumber*

Stops a note.

```
;; stop a violin note at middle C
stop-note "VIOLIN" 60
```

stop-instrument

stop-instrument *instrument*

Stops all notes of an instrument.

```
;; stop all cello notes
stop-instrument "CELLO"
```

stop-music

stop-music

Stops all notes.

Sound names

Drums

- | | |
|------------------------|--------------------|
| 35. Acoustic Bass Drum | 59. Ride Cymbal 2 |
| 36. Bass Drum 1 | 60. Hi Bongo |
| 37. Side Stick | 61. Low Bongo |
| 38. Acoustic Snare | 62. Mute Hi Conga |
| 39. Hand Clap | 63. Open Hi Conga |
| 40. Electric Snare | 64. Low Conga |
| 41. Low Floor Tom | 65. Hi Timbale |
| 42. Closed Hi Hat | 66. Low Timbale |
| 43. Hi Floor Tom | 67. Hi Agogo |
| 44. Pedal Hi Hat | 68. Low Agogo |
| 45. Low Tom | 69. Cabasa |
| 47. Open Hi Hat | 70. Maracas |
| 47. Low Mid Tom | 71. Short Whistle |
| 48. Hi Mid Tom | 72. Long Whistle |
| 49. Crash Cymbal 1 | 73. Short Guiro |
| 50. Hi Tom | 74. Long Guiro |
| 51. Ride Cymbal 1 | 75. Claves |
| 52. Chinese Cymbal | 76. Hi Wood Block |
| 53. Ride Bell | 77. Low Wood Block |
| 54. Tambourine | 78. Mute Cuica |
| 55. Splash Cymbal | 79. Open Cuica |
| 56. Cowbell | 80. Mute Triangle |
| 57. Crash Cymbal 2 | 81. Open Triangle |
| 58. Vibraslap | |

Instruments

Piano

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
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

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
103. Echoes
104. Sci-fi

Ethnic

105. Sitar
106. Banjo
107. Shamisen
108. Koto
109. Kalimba
110. Bag pipe
111. Fiddle
112. Shanai

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

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

FAQ (Frequently Asked Questions)

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. Please send comments, suggestions, and questions to feedback@ccl.northwestern.edu, and bug reports to bugs@ccl.northwestern.edu.

Questions

General

- Why is it called NetLogo?
- What programming language was NetLogo written in?
- How do I cite NetLogo in an academic publication?
- How do I cite a model from the Models Library in an academic publication?
- What license is NetLogo released under? Are there any legal restrictions on use, redistribution, etc.?
- Is the source code to NetLogo available?
- Do you offer any workshops or other training opportunities for NetLogo?
- What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?
- Has anyone built a model of <x>?
- Are NetLogo models runs scientifically reproducible?
- Are there any NetLogo textbooks?
- Is NetLogo available in a Spanish version, German version, (your language here) version, etc.?
- Is NetLogo compiled or interpreted?
- Will NetLogo and NetLogo 3D remain separate?

Downloading

- The download form doesn't work for me. Can I have a direct link to the software?
- Downloading NetLogo takes too long. Is it available any other way, such as on a CD?
- I downloaded and installed NetLogo but the Models Library has few or no models in it. How can I fix this?
- Can I have multiple versions of NetLogo installed at the same time?
- I'm on a UNIX system and I can't untar the download. Why?
- How do I install NetLogo on Windows 2003?

Applets

- I tried to run one of the applets on your site, but it didn't work. What should I do?
- Can I make my model available as an applet while keeping the code secret?
- Can a model saved as an applet use `import-world`, `file-open`, and other commands that read files?

Usage

- Can I run NetLogo from a CD?
- Why is NetLogo so much slower when I unplug my laptop?

- How do I change the number of patches?
- Can I use the mouse to "paint" in the view?
- How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?
- Can I import GIS data into NetLogo?
- My model runs slowly. How can I speed it up?
- I want to try HubNet. Can I?
- Can I run NetLogo from the command line, without the GUI?
- Can I have more than one model open at a time?
- Does NetLogo support multiple processors?
- Can I distribute NetLogo model runs across a cluster of computers?
- Can I use max-pxcor or max-pycor, etc., as the minimum or maximum of a slider?
- Can I change the choices in a chooser on the fly?
- Can I divide the code for my model up into several files?
- How do I show the legend in a plot?
- Why does my code have strange characters in it?

BehaviorSpace

- How do I gather data every n ticks?
- I'm varying a global variable I declared in the Procedures tab, but it doesn't work. Why?

Programming

- How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?
- How does the NetLogo language differ from other Logos?
- My model from NetLogo 3.0 or earlier doesn't work (or looks funny) in 3.1. Help!
- Why do I get a runtime error when I use setxy random world-width random world-height? It worked before.
- How do I take the negative of a number?
- My turtle moved forward 1, but it's still on the same patch. Why?
- patch-ahead 1 is reporting the same patch my turtle is already standing on. Why?
- How do I give my turtles "vision"?
- Can agents sense what's in the drawing layer?
- Does NetLogo have a command like StarLogo's "grab" command?
- I tried to put -at after the name of a variable, for example variable-at -1 0, but NetLogo won't let me. Why not?
- I'm getting numbers like 0.10000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?
- The documentation says that random-float 1.0 might return 0.0 but will never return 1.0. What if I want 1.0 to be included?
- How can I use different patch "neighborhoods" (circular, Von Neumann, Moore, etc.)?
- Can I connect turtles with lines, to indicate connections between them?
- How can I keep two turtles from occupying the same patch?
- How can I find out if a turtle is dead?
- How do I find out how much time has passed in my model?
- Does NetLogo have arrays?
- Does NetLogo have associative arrays or lookup tables?

- How can I convert an agentset to a list, or vice versa?
- How does NetLogo decide when to switch from agent to agent when running code?
- How do I stop foreach?
- How do I make an empty agentset?

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. It also refers to HubNet, the networked participatory simulation environment included in NetLogo.

What programming language was NetLogo written in?

NetLogo is written entirely in Java (version 1.4.1).

How do I cite NetLogo in an academic publication?

NetLogo itself: Wilensky, U. 1999. NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

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

How do I cite a model from the Models Library in a publication?

The proper citation is shown in the CREDITS AND REFERENCES section of each model's Info tab.

What license is NetLogo released under? Are there any legal restrictions on use, redistribution, etc.?

The license is given in the "Copyright" section of the NetLogo User Manual, as well as in the application's about box and the README file accompanying the download.

A quick summary of the license is that use is unrestricted, including commercial use, but there are some restrictions on redistribution and/or modification (unless you contact Uri Wilensky to arrange different terms).

We are in the process of reevaluating the language of the license in response to user feedback. In the future, we intend to send out a revised license.

Is the source code to NetLogo available?

At present, no. We are evaluating how best to distribute NetLogo when it is in a more mature state.

Making the source available is one possibility.

We do understand, however, that it is important that NetLogo not be a closed and non-extensible platform. That is not our intention for the product. So, for example, NetLogo includes APIs so that NetLogo can be controlled from external Java code and users can write new commands and reporters in Java. (See "Controlling" and "Extensions" in the User Manual.)

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 home page and on the netlogo-users group. If interested in this type of opportunity, please contact us at feedback@ccl.northwestern.edu.

What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?

The original StarLogo was developed at the MIT Media Lab in 1989–1990 and ran on a massively parallel supercomputer called the Connection Machine. A few years later (1994), a simulated parallel version was developed for the Macintosh computer. That version eventually became MacStarLogo. StarLogoT (1997), developed at the Center for Connected Learning and Computer-Based Modeling (CCL), is essentially an extended version of MacStarLogo with many additional features and capabilities.

Since then two multi-platform Java-based multi-agent Logos have been developed: NetLogo (from the CCL) and a Java-based version of StarLogo (from MIT).

The NetLogo language and environment differ in many respects from MIT StarLogo's. Both languages were inspired by the original StarLogo, but were redesigned in different ways. NetLogo's design was driven by the need to revise and expand the language so it is easier to use and more powerful, and by the need to support the HubNet architecture. NetLogo incorporates almost all of the extended functionality of our earlier StarLogoT, as well as a great many newer features.

Has anyone built a model of <x>?

The best place to ask this question is on the [NetLogo Users Group](#).

You should also check the Community Models section of our [Models Library](#) web page.

Are NetLogo models runs scientifically reproducible?

Yes. NetLogo's 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.
- 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. (Such models are rare. These two commands are common, but using them in a way that affects the outcome is not.)

- 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 to the possibility of random hardware failure, and also due to the possibility of human error in the design of: your model, NetLogo, your Java VM, your hardware, and so on.

Are there any NetLogo textbooks?

We at the CCL have hoped to write several NetLogo textbooks for quite some time. 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 people from the user community would like to collaborate on such a venture, please let us know. We would welcome it.

Is NetLogo available in a Spanish version, German version, (your language here) version, etc.?

At present, NetLogo is available only in English.

We plan to eventually make it possible for users to produce their own foreign-language "packs" for NetLogo and share them with each other. In order to do this, we need to separate all of the English text from the rest of the source code, so that is separately editable. We're not sure when this will happen.

Is NetLogo compiled or interpreted?

Short answer: interpreted, but we are working on a compiler.

Long answer: NetLogo does include a compiler, but the compiler does not produce native code, or even Java byte code. It produces a custom intermediate representation that can be interpreted more efficiently than the original code. However, we are working on a new compiler that will generate Java byte code. Once that is done, NetLogo will qualify as compiled, not interpreted. Since Java virtual machines have "just-in-time" compilers that in turn compile Java byte code all the way to native code, the new compiler should substantially improve the speed of NetLogo. We are not sure when the new compiler will be ready.

Will NetLogo and NetLogo 3D remain separate?

No. The split is temporary. Eventually a single unified version of NetLogo will support both 2D and 3D modeling. We will be sure to design the 3D world support in such a way that it doesn't get in the way when you are building 2D models.

Models built in NetLogo 3D Preview 1 may require some small changes in order to run in the eventual unified version.

Downloading

The download form doesn't work for me. Can I have a direct link to the software?

Please write us at bugs@ccl.northwestern.edu and we'll either fix the problem with the form, or provide you with an alternate method of downloading the software.

Downloading NetLogo takes too long. Is it available any other way, such as on a CD?

At present, no. If this is a problem for you, contact us at feedback@ccl.northwestern.edu.

I downloaded and installed NetLogo but the Models Library has few or no models in it. How can I fix this?

So far, users reporting this problem all used the "without VM" download option for Windows. Uninstall NetLogo and try the "with VM" download instead.

Even if the "with VM" download fixes it for you, please contact us at bugs@ccl.northwestern.edu so we can find out more details about your setup. We'd like to fix this in a future version, but to troubleshoot it we need help from users.

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 very 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 on Windows 2003 or Windows Server 2003?

On these operating systems, the NetLogo installer might not work unless you change some settings in the installer, as follows:

1. Locate the installer and right-click on it.
2. Select Properties
3. Select Compatibility tab
4. Check "Run this program in compatibility mode for:"
5. Select Windows XP
6. Click OK
7. Run the installer.

Applets

I tried to run one of the applets on your site, but it didn't work. What should I do?

Current versions of NetLogo require that your web browser support Java 1.4.1 or higher. Here's how to get the right Java:

- If you're on Windows 98 or newer, you need to download the Java browser plugin from http://www.java.com/en/download/windows_manual.jsp.
- If you're on Mac OS X, you need OS X 10.2.6 or higher. If you're on OS X 10.2, you also need Java 1.4.1 Update 1, which is available through Software Update. OS X 10.3 already has the right Java. You must also use a web browser that supports Java 1.4. Internet Explorer does not work; Safari does.
- If you're on Windows 95, MacOS 8, or MacOS 9, running models over the web is no longer supported; you must download the NetLogo 1.3.1 application and run the models that way instead.
- If you're on Linux or another Unix, you will need version 1.4.1 or higher of the Sun Java Runtime Environment. It is available for download at <http://www.java.com/>. Check your browser's home page for information about installing the Java plugin.

If you think you have the right browser and plugin, but it still doesn't work, check your browser's preferences to make sure that Java is enabled.

Can I make my model available as an applet while keeping the code secret?

No. In order for the applet to operate, the model file must be accessible also.

When you use "Save as applet" on the File menu, the HTML page generated contains a link where the user can download the model file. If you want, you can remove that link. Doing so will make it harder for the user to access the model file, but not impossible.

Can a model saved as an applet use `import-world`, `file-open`, and other commands that read files?

Yes, but only to read files that are stored in the same directory on your web server as the HTML and model files. Applets cannot read files on the user's computer, only the web server.

Usage

Can I run NetLogo from a CD?

Yes. NetLogo runs fine on a read-only file system.

Why is NetLogo so much slower when I unplug my 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 Sun, by editing the NetLogo.lax file, found in the NetLogo directory (under Program Files on your hard drive, unless you installed NetLogo in a different location). Edit this line:

```
lax.nl.java.option.additional=-Djava.ext.dirs= -server -Dsun.java2d.noddraw=true
```

and add `-Dsun.java2d.ddoffscreen=false` at the end of the last line.

You can see the details of the Java bug and vote for Sun to fix it [here](#).

How do I change how many patches there are?

A quick method is to use the three sets of black arrows in the upper left corner of the 2D view.

Another method is as follows. Select the 2D view by dragging a rectangle around it with the mouse. Click the "Edit" button in the Toolbar. A dialog will appear in which you may enter new values for "Screen Edge X" and "Screen Edge Y". (You can also right-click [Windows] or control-click [Mac] on the 2D view to edit it, or select it then double-click.)

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 use a special drawing model such as the Drawing Tool model by James Steiner which is available from <http://ccl.northwestern.edu/netlogo/models/community/>.

A third possibility is to create an image in another program and import it. See the answer to [Can I import a graphic into NetLogo?](#).

How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?

We have tested NetLogo with models that use hundreds of megabytes of RAM and they work fine. We haven't tested models that use gigabytes of RAM, though. Theoretically it should work, but you might hit some limits that are inherent in the underlying Java VM and/or operating system (either designed-in limits, or bugs).

The NetLogo engine has no fixed limits on size. On Macintosh and Windows operating systems, though, by default NetLogo ships with a 512 megabyte ceiling on how much total RAM it can use. (On other operating systems the ceiling is determined by your Java VM.)

Here's how to raise the limit if you need to:

- **Windows:** Edit this section of the "NetLogo.lax" file in the NetLogo folder:

```
# LAX.NL.JAVA.OPTION.JAVA.HEAP.SIZE.MAX
# -----
# allow the heap to get huge
lax.nl.java.option.java.heap.size.max=536870912
```

Note: this might not help on some Windows 98 or Windows ME systems.

- **Macintosh:** Edit the Contents/Info.plist file in the NetLogo application package. (You can reach this file by control-clicking the application in the Finder and choosing "Show Package Contents" from the popup menu.) The relevant section is this; the second number is the ceiling:

```
<key>VMOptions</key>
<string>-Xms16M -Xmx512M</string>
```

Note that (at least as of Mac OS X 10.3), the maximum possible heap size for any Java program is two gigabytes.

- **Other:** Java VMs from Sun let you set the ceiling on the command line as follows. If you are using a VM from a different vendor, the method may be different. Note that we recommend the `-server` flag for best performance in most situations. We also recommend setting the `noddraw` option to true when using the GUI as Direct Draw can conflict with OpenGL.

```
java -server -Dsun.java2d.noddraw=true -Xms16M -Xmx512M -jar NetLogo.jar
```

Can I import GIS data into NetLogo?

Yes, a number of users have constructed models using raster GIS data. (We don't know of users who have used vector GIS data, other than by converting it to raster data first.)

One simple way is to use `import-pcolors`, but that only works for importing maps that are images, not maps in other formats.

We do not have built-in support for reading common GIS formats. However, a number of our users are working with GIS data successfully using NetLogo code that reads GIS data using our file I/O primitives such as `file-open`.

It is also possible to use external software to convert GIS data into a format that is easier to read from NetLogo than the original format. This has been discussed on the [NetLogo Users Group](#) several times. We encourage users interested in using NetLogo for GIS applications to share their questions and experiences with the group.

My model runs slowly. How can I speed it up?

Here's some ways to make it run faster without changing the code:

- Edit the forever buttons in your model and turn off the "Force view update after each run" checkbox. This allows the view to skip frames, which may speed up models which are graphics-intensive. (See the Buttons section of the Programming Guide for a discussion of this.)
- Use the freeze switch in the view control strip, or the `no-display` command, to freeze the view temporarily. For example:

```
to go
  no-display
  ...
  ...
  display
end
```

If you use this technique, you should turn off the "Force view update" checkbox, since the `display` command already forces a view update.

- 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 these sizes are cached by NetLogo.

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. If you need help with this, if you contact us at feedback@ccl.northwestern.edu we may be able to help if you can send us your model or give us some idea of how it works. The members of the [NetLogo Users Group](#) may be able to help as well.

I want to try HubNet. Can I?

Yes. There are two types of HubNet available. With Computer HubNet, participants run the HubNet Client application on computers connected by a regular computer network. In Calculator HubNet, created in conjunction with [Texas Instruments](#), participants use TI-83+ graphing calculators and the [TI-Navigator](#) Classroom Learning System.

Note that Calculator HubNet works with a prototype version of the TI-Navigator system, and is not yet compatible with the commercially available TI-Navigator. We are actively working in partnership with Texas Instruments on integrating the new TI-Navigator with Calculator HubNet, which we expect to release in the near future.

For more information on HubNet, see the [HubNet Guide](#).

Can I run NetLogo from the command line, without the GUI?

Yes. The easiest way is to set up your model run as a BehaviorSpace experiment. No additional programming is required. See the BehaviorSpace section of the User Manual for details.

Another option is to use our Controlling API. Some light Java programming is required. See the "Controlling" section of the User Manual for details and sample 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. (We plan to change this in a future version.)

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 in the Finder, then open the copy. (The copy takes up only a very small amount of additional disk space.)

Does NetLogo take advantage of multiple processors?

Not for a single model run, no. The NetLogo engine is single threaded and we expect it to remain that way. We don't have any plans to make it possible to split a single model run across multiple computers.

You can take advantage of multiple processors to do multiple model runs concurrently, though, in either of two ways:

- By having multiple copies of NetLogo open, in separate Java virtual machines; see [this answer](#) for instructions.
- By using BehaviorSpace or the Controlling API to do model runs from the command line.

In a future version of NetLogo, we hope to improve the support for multiple processors as follows:

- Allow multiple models to be open simultaneously, each running in a different thread and hence on a different processor.
- Modify BehaviorSpace to optionally do multiple model runs in parallel in a configurable number of separate threads, so the runs would be spread across available processors.

Can I distribute NetLogo model runs across a cluster 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.

Numerous users are already using NetLogo on clusters. You can seek them out on the [NetLogo Users Group](#).

Can I use max-pxcor or max-pycor, etc., as the minimum or maximum of a slider?

At present, no. In a future version of NetLogo, we plan to support this.

Can I change the choices in a chooser on the fly?

At present, no. In a future version of NetLogo, we plan to support this.

Can I divide the code for my model up into several files?

At present, no. In a future version of NetLogo, we plan to support this.

How do I show the legend in a plot?

Click the word "Pens" in the upper right corner of the plot. The legend will include any pens that have the "Show in Legend" option checked when editing the plot. Clicking "Pens" again will hide the legend.

Why does my code have strange characters in it?

NetLogo only works in "en" locales. A locale is a setting which tells NetLogo which language you are using, as well as how to display dates and numbers. You may need to switch to to an English locale before launching NetLogo. This is usually done in the "Regional Settings" or "Internationalization" panel of the operating system.

In a future version of Netlogo we plan to support different languages and locales.

BehaviorSpace

How do I measure runs every n ticks?

Use `repeat` in your experiment's go commands, e.g.:

```
repeat 100 [ go ]
```

to measure the run after every 100 model steps. Essentially you are making one experiment step equal 100 model steps.

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

It's probably because your setup commands or setup procedure are using `ca` or `clear-all`, causing the values set by BehaviorSpace to be cleared. Try using the more specific clearing commands to clear only what you want cleared.

Programming

How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?

We don't have a document that specifically summarizes the differences between these two programs. If you have built models in StarLogoT before, then we suggest reading the [Programming Guide](#) section of this manual to learn about NetLogo, particularly the sections on "Ask" and "Agentsets". Looking at some of the sample models and code examples in the Models Library may help as well.

NetLogo 1.3.1 includes a StarLogoT model converter; you just open the model from the File menu and NetLogo will attempt to convert it. The converter doesn't do all that great a job though, so the result will very likely require additional changes before it will work. Note also that the model converter is no longer included in current versions of NetLogo, so if you have models you want to use it on, you will have to use NetLogo 1.3.1 to do the converting, then open the model in a current version.

If you need any help converting your StarLogo or StarLogoT model to NetLogo, please feel free to seek help on the [NetLogo Users Group](#). You may also ask us for help at feedback@ccl.northwestern.edu.

How does the NetLogo language differ from other Logos?

There is no standard definition of Logo; it is a loose family of languages. We believe that NetLogo shares enough syntax, vocabulary, and features 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 Procedures 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
<code>to square :x</code>	<code>to-report square [x]</code>
<code>output :x * :x</code>	<code>report x * x</code>
<code>end</code>	<code>end</code>

Deeper differences:

- NetLogo is 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 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. (NetLogo's `run` command is no help here.)
- As in most Logos, functions as values are not supported. Most Logos provide similar if less general functionality, though, by allowing passing and manipulation of fragments of source code in list form. NetLogo's capabilities in this area are presently limited. A few of our built-in special forms use UCBLogo-style "templates" to accomplish a similar purpose, for example, `sort-by [length ?1 < length ?2] string-list`. In some circumstances, using `run` and `runresult` instead is workable, but unlike most Logos they operate on strings, not lists.

Of course, the NetLogo language also contains many additional features not found in most other Logos, most importantly agents and agentsets.

My model from NetLogo 3.0 or earlier doesn't work (or looks funny) in 3.1. Help!

Here are short discussions of the issues that are likeliest to arise, with links to further information:

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. In NetLogo 3.1, if you don't want the view to wrap you must make it so the world doesn't wrap, using 3.1's 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.

If your model is behaving strangely or incorrectly, perhaps it's because in 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 in 3.1. 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.

Why do I get a runtime error when I use `setxy random world-width random world-height`? It worked before.

Many models made in NetLogo 3.0 or earlier use this code 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 are new 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.

The `random-xcor`, `random-ycor`, `random-pxcor`, and `random-pycor` primitives are new in NetLogo 3.1.

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?

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 a corner. 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.

A turtle is on a patch center when its `xcor` and `ycor` are multiples of 1.0.

patch-ahead 1 is reporting the same patch my turtle is already standing on. Why?

See previous answer. 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.

Does NetLogo have a command like StarLogo's "grab" command?

We don't have such a command. You can use the `without-interruption` primitive to arrange exclusive interaction between agents. For example:

```
turtles-own [mate]
to setup
  ask turtles [ set mate nobody ]
end
to find-mate ;; turtle procedure
  without-interruption
  [ if mate = nobody
    [ let candidate one-of other-turtles-here
      with [mate = nobody]
      if candidate != nobody
      [ set mate candidate
        set mate-of candidate self ] ] ]
end
```

Using `without-interruption` ensures that while a turtle is choosing a mate, all other agents are "frozen". This makes it impossible for two turtles to choose the same mate.

I tried to put `-at` after the name of a variable, for example `variable-at -1 0`, but NetLogo won't let me. Why not?

This syntax was supported by StarLogoT and some beta versions of NetLogo, but was removed from NetLogo 1.0. Instead, for a patch variable write e.g. `pcolor-of patch-at -1 0`, and for a turtle variable write e.g. `color-of one-of turtles-at -1 0`.

I'm getting numbers like `0.10000000004` and `0.79999999999` instead of `0.1` and `0.8`. Why?

See the "Math" section of the Programming Guide in the User Manual for a discussion of this issue.

The documentation says that `random-float 1.0` might return `0.0` but will never return `1.0`. What if I want `1.0` to be included?

It really doesn't matter. Even if `1.0` 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 `1.0`.

Nonetheless, if you feel it really must be possible to get `1.0`, you can use `precision` to round your answer to a certain number of decimal places. For example:

```
print precision (random-float 1.0) 10
0.2745173723
```

(If you use this method, note that `0.0` and `1.0` 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 places. Results between `0.0` and `0.5` get rounded to `0.0`, but results between `0.5` and `1.5` get rounded to `1.0`; the latter range is twice as large. So if you want `0.0`, `0.1`, `0.2`, ..., `0.9`, and `1.0` to all be equally likely, you must write `random 11 / 10.`)

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 [ ... ]
```

How do I find out how much time has passed in my model?

NetLogo does not automatically keep track of this. If you want to keep track of the passage of time, add a global variable to your model with a name like "clock" or "steps". In your setup procedure, set

the variable to 0. In your go procedure, increment the variable. Many of the models in the Models Library use this technique.

The reason NetLogo doesn't automatically keep track of this is that NetLogo is very flexible about letting you make buttons that do anything that you want them to. NetLogo has no one way of knowing which of your buttons should advance the clock and which shouldn't.

Does NetLogo have arrays?

What NetLogo calls "lists" are actually implemented internally as arrays, so they have some of the performance characteristics of arrays. For example, random access (using the `item` reporter) takes constant time. However, they're immutable arrays (they cannot be altered except by making a copy and altering the copy), so `replace-item` is linear time, not constant-time (because the whole array is copied).

For most purposes, the performance differences between lists and arrays doesn't matter; it only matters if you're dealing with very long lists.

In a future version of NetLogo we plan to change our lists to be ordinary singly linked lists as in other Logo (and Lisp) implementations. At the same time, we will also provide real, mutable arrays as a separate data type.

Does NetLogo have associative arrays or lookup tables?

No, but you can use lists to accomplish the same thing, though less efficiently. See:

- <http://groups.yahoo.com/group/netlogo-users/message/2344>
- <http://groups.yahoo.com/group/netlogo-users/message/2346>
- <http://groups.yahoo.com/group/netlogo-users/message/2354>

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.

Can I connect turtles with lines, to indicate connections between them?

Yes. See the Links section of the Programming Guide.

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.

If you want the list in a random order, here's how:

```
values-from <agentset> [self]
```

Because all operations on agentsets are in random order, the resulting list is in random order.

And here's how to convert a list of agents to an agentset:

```
turtles/patches with [member? self <list>]
```

Note however that this method is not efficient and may slow down your model. We hope to address this issue in a future version of NetLogo.

How does NetLogo decide when to switch from agent to agent when running code?

If you ask `turtles`, or ask a whole breed, the turtles are scheduled for execution in random order, because all operations on agentsets happen in random order.

Once scheduled, an agent's "turn" ends only once 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, or patch variable. (Setting a local variable doesn't count.)

To prolong an agent's "turn", use the `without-interruption` command. (The command blocks inside some commands, such as `cct` and `hatch`, have an implied `without-interruption` around them.)

NetLogo's scheduling mechanism 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 write your NetLogo code so that it does not depend on a particular scheduling mechanism. We make no guarantees that the scheduling algorithm will remain the same in future versions.

How do I stop foreach?

To stop a `foreach` from executing you need to define a separate procedure that contains only the `foreach`, for example:

```
to test
  foreach [ 1 2 3 ]
  [
    if ? = 2
    [ stop ]
    print ?
  ]
end
```

This code will only print the number 1. The `stop` returns from the current procedure so nothing after the `foreach` will be executed either. (If the procedure is a reporter procedure, use `report` instead of `stop`.)

How do I make an empty agentset?

The most straightforward way is:

```
n-of 0 turtles
```

or

```
n-of 0 patches
```

(Every agentset, even an empty one, is either a turtle set or a patch set.)

Primitives Dictionary

Alphabetical: [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [O](#) [P](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [?](#)

Categories: [Turtle](#) – [Patch](#) – [Agentset](#) – [Color](#) – [Control/Logic](#) – [World](#) – [Perspective](#)
[Input/Output](#) – [Files](#) – [List](#) – [String](#) – [Math](#) – [Plotting](#) – [Links](#) – [Movie](#) – [System](#) – [HubNet](#)

Special: [Variables](#) – [Keywords](#) – [Constants](#)

Categories of Primitives

This is an approximate grouping. Remember that a turtle-related primitive might still be called by patches or observers, and vice versa. To see which agent (turtles, patches, observer) can actually run each command, consult each individual entry in the dictionary.

Turtle-related

[back](#) (bk) [<breeds>-at](#) [<breeds>-here](#) [<breeds>-on](#) [can-move?](#) [clear-turtles](#) (ct) [create-<breeds>](#)
[create-custom-<breeds>](#) [create-custom-turtles](#) (cct) [create-turtles](#) (crt) [die](#) [distance](#) [distancexy](#)
[downhill](#) [downhill4](#) [dx](#) [dy](#) [face](#) [facexy](#) [forward](#) (fd) [hatch](#) [hatch-<breeds>](#) [hideturtle](#) (ht) [home](#) [inspect](#)
[is-<breed>?](#) [is-turtle?](#) [jump](#) [left](#) (lt) [myself](#) [nobody](#) [-of other-turtles-here](#) [other-<breeds>-here](#)
[patch-ahead](#) [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](#) [showturtle](#) (st) [sprout](#)
[sprout-<breeds>](#) [stamp](#) [stamp-erase](#) [subject](#) [subtract-headings](#) [tie](#) [towards](#) [towardsxy](#) [turtle](#)
[turtles](#) [turtles-at](#) [turtles-from](#) [turtles-here](#) [turtles-on](#) [turtles-own](#) [untie](#) [uphill](#) [value-from](#)

Patch-related primitives

[clear-patches](#) (cp) [diffuse](#) [diffuse4](#) [distance](#) [distancexy](#) [import-pcolors](#) [inspect](#) [is-patch?](#) [myself](#)
[neighbors](#) [neighbors4](#) [nobody](#) [nsum](#) [nsum4](#) [-of patch](#) [patch-at](#) [patch-ahead](#)
[patch-at-heading-and-distance](#) [patch-here](#) [patch-left-and-ahead](#) [patch-right-and-ahead](#)
[patches](#) [patches-from](#) [patches-own](#) [random-pxcor](#) [random-pycor](#) [self](#) [sprout](#) [sprout-<breeds>](#)
[subject](#) [value-from](#)

Agentset primitives

[any?](#) [ask](#) [at-points](#) [<breeds>-at](#) [<breeds>-here](#) [<breeds>-on](#) [count](#) [histogram-from](#) [in-cone](#)
[in-radius](#) [is-agent?](#) [is-agentset?](#) [is-patch-agentset?](#) [is-turtle-agentset?](#) [max-one-of](#) [min-one-of](#)
[n-of](#) [neighbors](#) [neighbors4](#) [one-of other-turtles-here](#) [other-<breeds>-here](#) [patches](#) [patches-from](#)
[sort](#) [sort-by](#) [turtles](#) [with](#) [with-max](#) [with-min](#) [turtles-at](#) [turtles-from](#) [turtles-here](#) [turtles-on](#)
[values-from](#)

Color primitives

[color](#) [extract-hsb](#) [extract-rgb](#) [hsb](#) [import-pcolors](#) [pcolor](#) [rgb](#) [scale-color](#) [shade-of?](#) [wrap-color](#)

Control flow and logic primitives

and carefully end error-message foreach if ifelse ifelse-value let loop map not or repeat report run
runresult : (semicolon) set stop startup to to-report wait while without-interruption xor

World primitives

clear-all (ca) clear-drawing (cd) clear-patches (cp) clear-turtles (ct) display import-drawing
import-pcolors no-display max-pxcor max-pycor min-pxcor min-pycor world-width world-height

Perspective primitives

follow follow-me reset-perspective (rp) ride ride-me subject watch watch-me

HubNet primitives

hubnet-broadcast hubnet-broadcast-view hubnet-enter-message? hubnet-exit-message?
hubnet-fetch-message hubnet-message hubnet-message-source hubnet-message-tag
hubnet-message-waiting? hubnet-reset hubnet-send hubnet-send-view
hubnet-set-client-interface

Input/output primitives

beep clear-output date-and-time export-view export-interface export-output export-plot
export-all-plots export-world import-drawing import-pcolors 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 primitives

file-at-end? file-close file-close-all file-delete file-exists? file-open file-print file-read
file-read-characters file-read-line file-show file-type file-write user-directory user-file
user-new-file

List primitives

but-first but-last empty? filter first foreach fput is-list? item last length list lput map member?
modes n-of n-values position one-of reduce remove remove-duplicates remove-item
replace-item reverse sentence shuffle sort sort-by sublist values-from

String primitives

Operators (+, <, >, =, !=, <=, >=) but-first but-last empty? first is-string? item last length member?
position remove remove-item read-from-string replace-item reverse substring word

Mathematical primitives

Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=) abs acos asin atan ceiling cos e exp floor int ln log max mean median min mod modes new-seed pi precision random random-exponential random-float random-gamma random-int-or-float random-normal random-poisson random-seed remainder round sin sqrt standard-deviation subtract-headings sum tan variance

Plotting primitives

autoplot? auto-plot-off auto-plot-on clear-all-plots clear-plot create-temporary-plot-pen export-plot export-all-plots histogram-from histogram-list plot plot-name plot-pen-down (ppd) plot-pen-reset plot-pen-up (ppu) plot-x-max plot-x-min plot-y-max plot-y-min plotxy ppd ppu set-current-plot set-current-plot-pen set-histogram-num-bars set-plot-pen-color set-plot-pen-interval set-plot-pen-mode set-plot-x-range set-plot-y-range

Link primitives

both-ends create-<breed>-from create-<breeds>-from create-<breed>-to create-<breeds>-to create-<breed>-with create-<breeds>-with end1 end2 in-<breed>-neighbor? in-<breed>-neighbors in-<breed>-from is-link? layout-circle layout-magspring layout-radial layout-spring layout-tutte <breed>-neighbor? <breed>-neighbors <breed>-with my-<breeds> my-in-<breeds> my-out-<breeds> other-end out-<breed>-neighbor? out-<breed>-neighbors out-<breed>-to remove-<breed>-from remove-<breeds>-from remove-<breed>-to remove-<breeds>-to remove-<breed>-with remove-<breeds>-with

Movie primitives

movie-cancel movie-close movie-grab-view movie-grab-interface movie-set-frame-rate movie-start movie-status

System primitives

netlogo-version

Built-In Variables

Turtles

breed color heading hidden? label label-color pen-mode pen-size shape size who xcor ycor

Patches

pcolor plabel plabel-color pxcor pycor

Other

?

Keywords

breed end globals patches-own to to-report turtles-own

Constants

Mathematical Constants

e = 2.718281828459045

pi = 3.141592653589793

Boolean Constants

false

true

Color Constants

black = 0.0

gray = 5.0

white = 9.9

red = 15.0

orange = 25.0

brown = 35.0

yellow = 45.0

green = 55.0

lime = 65.0

turquoise = 75.0

cyan = 85.0

sky = 95.0

blue = 105.0

violet = 115.0

magenta = 125.0

pink = 135.0

See the Colors section of the Programming Guide for more details.

A

abs

abs *number*

Reports the absolute value of *number*.

```
show abs -7
=> 7
show abs 5
=> 5
```

acos

acos *number*

Reports the arc cosine (inverse cosine) of the given number. The input must be in the range -1.0 to 1.0. The result is in degrees, and lies in the range 0.0 to 180.0.

and

condition1 and **condition2**

Reports true if both *condition1* and *condition2* are true.

Note that if *condition1* is false, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) and (pycor > 0)
  [ set pcolor blue ] ;; the upper-right quadrant of
                      ;; patches turn blue
```

any?

any? *agentset*

Reports true if the given agentset is non-empty, false otherwise.

Equivalent to "count *agentset* > 0", but 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 [nobody](#).

Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=)

All of these operators take two inputs, and all act as "infix operators" (going between the two inputs, as in standard mathematical use). NetLogo correctly supports order of operations for infix operators.

The operators work as follows: + is addition, * is multiplication, – is subtraction, / is division, ^ is exponentiation, < is less than, > is greater than, = is equal to, != is not equal to, <= is less than or equal, >= is greater than or equal.

Note that the subtraction operator (–) always takes two inputs unless you put parentheses around it, in which case it can take one input. For example, to take the negative of x, write (– x), with the parentheses.

All of the comparison operators also work on strings, and the addition operator (+) also functions as a string concatenation operator (see example below).

If you are not sure how NetLogo will interpret your code, you should insert parentheses.

```
show 5 * 6 + 6 / 3
=> 32
show 5 * (6 + 6) / 3
=> 20
show "tur" + "tle"
=> "turtle"
```

asin

asin *number*

Reports the arc sine (inverse sine) of the given number. The input must be in the range –1.0 to 1.0. The result is in degrees, and lies in the range –90.0 to 90.0.

ask

ask *agentset* [*commands*]

ask *agent* [*commands*]

The specified agent or agentset runs the given commands.

```
ask turtles [ fd 1 ]
;; all turtles move forward one step
ask patches [ set pcolor red ]
;; all patches turn red
ask turtle 4 [ rt 90 ]
;; only the turtle with id 4 turns right
```

Note: only the observer can ask all turtles or all patches. This prevents you from inadvertently having all turtles ask all turtles or all patches ask all patches, which is a common mistake to make if you're not careful about which agents will run the code you are writing.

Note: Only the agents that are in the agentset *at the time the ask begins* run the commands.

at-points

agentset at-points [[x1 y1] [x2 y2] ...]

Reports a subset of the given agentset that includes only the agents on the patches the given distances away from the calling agent. The distances are specified as a list of two-item lists, where the two items are the x and y offsets.

If the caller is the observer, then the points are measured relative to the origin, in other words, the points are taken as absolute patch coordinates.

If the caller is a turtle, the points are measured relative to the turtle's exact location, and not from the center of the patch under the turtle.

```
ask turtles at-points [[2 4] [1 2] [10 15]]
[ fd 1 ] ;; only the turtles on the patches at the
          ;; distances (2,4), (1,2) and (10,15),
          ;; relative to the caller, move
```

atan**atan x y**

Reports the arc tangent, in degrees (from 0 to 360), of x divided by y.

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.

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.)

```
show atan 1 -1
=> 135.0
show atan -1 1
=> 315.0
```

autoplot?**autoplot?**

Reports true if auto-plotting is on for the current plot, false otherwise.

auto-plot-off**auto-plot-on****auto-plot-off****auto-plot-on**

This pair of commands is used to control the NetLogo feature of auto-plotting in the current plot. Auto-plotting will automatically update the x and y axes of the plot whenever the current pen exceeds these boundaries. It is useful when wanting to display all plotted values in the current plot, regardless of the current plot ranges.

B

back

bk

back *number*



The turtle moves backward by *number* steps. (If *number* is negative, the turtle moves forward.)

Turtles using this primitive can move a maximum of one unit per time increment. So `bk 0.5` and `bk 1` both take one unit of time, but `bk 3` takes three.

If the turtle cannot move backward *number* steps because it is not permitted by the current topology the turtle will complete as many steps of 1 as it can and stop.

See also [forward](#), [jump](#), [can-move?](#).

beep

beep

Emits a beep. Note that the beep is emitted immediately, so several beep commands in succession will only produce a single audible sound.

Example:

```
beep                                ;; emits one beep

repeat 3 [ beep ]                  ;; emits 3 beeps at the exact same time,
                                   ;; so you only hear one sound

repeat 3 [ beep wait 0.1 ]          ;; produces 3 beeps in succession,
                                   ;; separated by 1/10th of a second
```

__both-ends

__both-ends



Reports the agentset of the 2 nodes connected by this link turtle.

```
;; Let a directed link turtle L be connected from N1 to N2.
ask L
[
  show __both-ends ;; displays an agentset consisting of N1 and N2.
]
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

breed

breed



This is a built-in turtle variable. It holds the agentset of all turtles of the same breed as this turtle. (For turtles that do not have any particular breed, this is the turtles agentset of all turtles.) You can set this variable to change a turtle's breed.

See also breed.

Example:

```
breed [cats cat]
breed [dogs dog]
;; turtle code:
if breed = cats [ show "meow!" ]
set breed dogs
show "woof!"
```

breed

breed [*<breeds>* *<breed>*]

This keyword, like the globals, turtles-own, and patches-own keywords, can only be used at the beginning of the Procedures 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
  ca
  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>](#), [create-custom-<breeds>](#), [<breeds>-at](#), [<breeds>-here](#).

but-first**bf****but-last****bl****but-first *list*****but-first *string*****but-last *list*****but-last *string***

When used on a list, **but-first** reports all of the list items of *list* except the first, and **but-last** reports all of the list items of *list* except the last.

On strings, **but-first** and **but-last** report a shorter string omitting the first or last character of the original string.

```
;; mylist is [2 4 6 5 8 12]
set mylist but-first mylist
;; mylist is now [4 6 5 8 12]
set mylist but-last mylist
;; mylist is now [4 6 8]
show but-first "string"
;; prints "tring"
show but-last "string"
;; prints "strin"
```

C**can-move?****can-move? *distance***

Reports true if the calling turtle can move *distance* in the direction it is facing without violating the topology; reports false otherwise.

It is equivalent to:

```
patch-ahead distance != nobody
```

carefully**carefully [*commands1*] [*commands2*]**

Runs *commands1*. If a runtime error occurs inside *commands1*, NetLogo won't stop and alert the user that an error occurred. It will suppress the error and run *commands2* instead.

The error-message reporter can be used in *commands2* to find out what error was suppressed in *commands1*. See [error-message](#).

Note: both sets of commands run without interruption (as with the without-interruption command).

```
carefully [ show 1 / 1 ] [ print error-message ]
=> 1
carefully [ show 1 / 0 ] [ print error-message ]
=> division by zero
```

ceiling

ceiling *number*

Reports the smallest integer greater than or equal to *number*.

```
show ceiling 4.5
=> 5
show ceiling -4.5
=> -4
```

clear-all

ca

clear-all



Resets all global variables to zero, and calls clear-turtles, clear-patches, clear-drawing, clear-all-plots, and clear-output.

clear-all-plots

clear-all-plots



Clears every plot in the model. See [clear-plot](#) for more information.

clear-drawing

cd

clear-drawing



Clears all lines and stamps drawn by turtles.

clear-output

clear-output

Clears all text from the model's output area, if it has one. Otherwise does nothing.

clear-patches**cp****clear-patches**

Clears the patches by resetting all patch variables to their default initial values, including setting their color to black.

clear-plot**clear-plot**

In the current plot only, resets all plot pens, deletes all temporary plot pens, resets the plot to its default values (for x range, y range, etc.), and resets all permanent plot pens to their default values. The default values for the plot and for the permanent plot pens are set in the plot Edit dialog, which is displayed when you edit the plot. If there are no plot pens after deleting all temporary pens, that is to say if there are no permanent plot pens, a default plot pen will be created with the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Name: "default"
- Interval: 1.0

See also [clear-all-plots](#).

clear-turtles**ct****clear-turtles**

Kills all turtles.

See also [die](#).

color

color

This is a built-in turtle variable. It holds the color of the turtle. You can set this variable to make the turtle change color.

See also [pcolor](#).

cos**cos *number***

Reports the cosine of the given angle. Assumes the angle is given in degrees.

```
show cos 180
=> -1.0
```

count**count *agentset***

Reports the number of agents in the given agentset.

```
show count turtles
;; prints the total number of turtles
show count patches with [pcolor = red]
;; prints the total number of red patches
```

__create-<breed>-to
__create-<breed>-from
__create-<breed>-with

__create-<breed>-to *agent* [*commands*]
__create-<breed>-from *agent* [*commands*]
__create-<breed>-with *agent* [*commands*]
__create-<breeds>-to *agentset* [*commands*]
__create-<breeds>-from *agentset* [*commands*]
__create-<breeds>-with *agentset* [*commands*]



Used for creating link turtles between agents. Links are special turtles. A link turtle's size is always equal to the distance between the two node turtles it links together. Its heading is always equal to the heading from one node turtle to the other. Its location is always halfway between the two node turtles. Their default shape is "link".

__create-<breed>-with creates an undirected link between the caller and *agent*.
__create-<breed>-to creates a link from the caller to *agent*. **__create-<breed>-from** creates a 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 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.

```
breed [links link] ;; breed for network links

to make-friend [ frd ] ;; node procedure
  ;; creates a friendship link between
  ;; the node on which this
  ;; procedure was called and the node "frd".
  __create-link-to frd
  [
    set color blue
  ]
end

to become-friendly ;; node procedure
  ;; makes this node link with an agentset of turtles
  __create-links-to friends
  []
end
```

This command is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

create-turtles

crt

create-<breeds>

create-turtles *number*

create-<breeds> *number*



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.

```
crt 100
ask turtles [ fd 10 ] ;; makes an evenly spaced circle
```

If the create-<breeds> form is used, the new turtles are created as members of the given breed.

create-custom-turtles

cct

create-custom-<breeds>

cct-<breeds>

create-custom-turtles *number* [*commands*]
create-custom-<breeds> *number* [*commands*]



Creates *number* new turtles (of the given breed, if specified). New turtles start at position (0, 0). New turtles are created with the 14 primary colors and have headings from 0 to 360, evenly spaced.

The new turtles immediately run *commands*. 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.)

```
breed [canaries canary]
breed [snakes snake]
to setup
  ca
  create-custom-canaries 50
    [ set color yellow ]
  create-custom-snakes 50
    [ set color green ]
end
```

Note: While the commands are running, no other agents are allowed to run any code (as with the `without-interruption` command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no display updates take place until the commands are done. This ensures that the new turtles are never drawn in the view until they are fully initialized.

create-temporary-plot-pen

create-temporary-plot-pen *string*

A new temporary plot pen with the given name is created in the current plot and set to be the current pen.

Few models will want to use this primitive, because all temporary pens disappear when `clear-plot` or `clear-all-plots` are called. The normal way to make a pen is to make a permanent pen in the plot's Edit dialog.

If a temporary pen with that name already exists in the current plot, no new pen is created, and the existing pen is set to be the current pen. If a permanent pen with that name already exists in the current plot, you get a runtime error.

The new temporary plot pen has the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Interval: 1.0

See: [clear-plot](#), [clear-all-plots](#), and [set-current-plot-pen](#).

D

date-and-time

date-and-time

Reports a string containing the current date and time. The format is shown below. All fields are fixed width, so they are always at the same locations in the string. The potential resolution of the clock is milliseconds. (Whether you get resolution that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.)

```
show date-and-time
=> "01:19:36.685 PM 19-Sep-2002"
```

die

die



The turtle dies.

```
if xcor > 20 [ die ]

;; all turtles with xcor greater than 20 die
```

See also: [ct](#)

diffuse

diffuse *patch-variable number*



Tells each patch to share (*number* * 100) percent of the value of *patch-variable* with 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 the remainder stays on the patch at the edge of the world.

Note that this is an observer command only, even though you might expect it to be a patch command. (The reason is that it acts on all the patches at once — patch commands act on individual patches.)

```
diffuse chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 8 patches. Thus,
;; each patch gets 1/8 of 50% of the chemical
;; from each neighboring patch.)
```

diffuse4

diffuse4 *patch-variable number*



Like `diffuse`, but only diffuses to the four neighboring patches (to the north, south, east, and west), not to the diagonal neighbors.

```
diffuse4 chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 4 patches. Thus,
;; each patch gets 1/4 of 50% of the chemical
;; from each neighboring patch.)
```

display

display

Causes the current view to be updated immediately.

Also undoes the effect of the `no-display` command, so that if display 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
```

There is exception to the "immediately" rule: if the command is used by an agent that is running "without interruption" (such as via the `without-interruption` command, inside a procedure defined using `to-report`, or inside a command such as `hatch`, `sprout`, or `cct`), then the view update takes place once the agent is done running without interruption.

Note that `display` and `no-display` operate independently of the switch in the view control strip that freezes the view.

See also [`no-display`](#).

distance

distance *agent*



Reports the distance from this agent to the given turtle or patch.

The distance to or a from a patch is measured from the center of the patch. Turtles and patches use the wrapped distance (around the edges of the view) if wrapping is allowed by the topology and the wrapped distance is shorter than the on-screen distance.

distancexy

distancexy *xcor ycor*



Reports the distance from this agent to the point (*xcor*, *ycor*).

The distance from a patch is measured from the center of the patch. Turtles and patches use the wrapped distance (around the edges of the view) if wrapping is allowed by the topology and the wrapped distance is shorter than the on-screen distance.

```
if (distancexy 0 0) > 10
  [ set color green ]
;; all turtles more than 10 units from
;; the center of the screen turn green.
```

downhill

downhill *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) in the direction of the minimum value of the variable *patch-variable*, of the patches in a one-patch radius of the turtle. (This could be as many as eight patches or as few as one patch, depending on the position of the turtle within its patch and the topology of the world.)

If there are multiple patches that have the same smallest value, a random one of those patches will be selected.

If the patch is located directly to the north, south, east, or west of the patch that the turtle is currently on, a multiple of 90 degrees is reported. However, if the patch is located to the northeast, northwest, southeast, or southwest of the patch that the turtle is currently on, the direction the turtle would need to reach the nearest corner of that patch is reported.

See also [downhill4](#), [uphill](#), [uphill4](#).

downhill4

downhill4 *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) as a multiple of 90 degrees in the direction of the minimum value of the variable *patch-variable*, of the four patches to the north, south, east, and west of the turtle. If there are multiple patches that have the same least value, a random patch from those patches will be selected.

See also [downhill](#), [uphill](#), [uphill4](#).

dx

dy

dx

dy



Reports the x-increment or y-increment (the amount by which the turtle's xcor or ycor would change) if the turtle were to take one step forward in its current heading.

Note: dx is simply the sine of the turtle's heading, and dy is simply the cosine. (If this is the reverse of what you expected, it's because in NetLogo a heading of 0 is north and 90 is east, which is the reverse of how angles are usually defined in geometry.)

Note: In earlier versions of NetLogo, these primitives were used in many situations where the new *patch-ahead* primitive is now more appropriate.

E

empty?

empty? *list*

empty? *string*

Reports true if the given list or string is empty, false otherwise.

Note: the empty list is written `[]`. The empty string is written `" "`.

end

end

Used to conclude a procedure. See [to](#) and [to-report](#).

__end1

__end1



Reports the first end of a link. In a directed link, the direction of the link is from end1 to the end2.

```
;; Let a directed link L be connected from N1 to N2.
ask L
[
  show __end1 ;; prints N1
]
```

Even though in an undirected link order of nodes is not important, end1 is deterministic and reports the same node on repeated calls by an undirected link.

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__end2

__end2



Reports the second end of a link. In a directed link, the direction of the link is from end1 to the end2.

```
;; Let a directed link L be connected from N1 to N2.
ask L
[
  show __end2 ;; prints N2
]
```

Even though in an undirected link order of nodes is not important, end2 is deterministic and reports the same node from repeated calls by an undirected link.

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

error-message

error-message

Reports a string describing the error that was suppressed by carefully.

This reporter can only be used in the second block of a carefully command.

See also [carefully](#).

every**every *number* [*commands*]**

Runs the given commands at most every *number* seconds.

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.

More technically, its exact behavior is as follows. When an agent reaches an "every", it checks a timer to see if the given amount of time has passed since the last time the same agent ran the commands in the "every" in the same context. If so, it runs the commands; otherwise they are skipped and execution continues.

Here, "in the same context" means during the same ask (or button press or command typed in the Command Center). So it doesn't make sense to write `ask turtles [every 0.5 [...]]`, because when the ask finishes the turtles will all discard their timers for the "every". The correct usage is shown below.

```
every 0.5 [ ask turtles [ fd 1 ] ]
;; twice a second the turtles will move forward 1
every 2 [ set index index + 1 ]
;; every 2 seconds index is incremented
```

See also [wait](#).

exp**exp *number***

Reports the value of e raised to the *number* power.

Note: This is the same as $e^{\textit{number}}$.

export-view**export-interface****export-output****export-plot****export-all-plots****export-world**

export-view *filename*

export-interface *filename*

export-output *filename*

export-plot *plotname filename*

export-all-plots *filename*

export-world *filename*

`export-view` writes the current contents of the current view to an external file given by the string *filename*. The file is saved in PNG (Portable Network Graphics) format, so it is recommended to supply a filename ending in ".png".

`export-interface` is similar, but for the whole interface tab.

`export-output` writes the contents of the model's output area to an external file given by the string *filename*. (If the model does not have a separate output area, the output portion of the Command Center is used.)

`export-plot` writes the x and y values of all points plotted by all the plot pens in the plot given by the string *plotname* to an external file given by the string *filename*. If a pen is in bar mode (mode 0) and the y value of the point plotted is greater than 0, the upper-left corner point of the bar will be exported. If the y value is less than 0, then the lower-left corner point of the bar will be exported.

`export-all-plots` writes every plot in the current model to an external file given by the string *filename*. Each plot is identical in format to the output of `export-plot`.

`export-world` writes the values of all variables, both built-in and user-defined, including all observer, turtle, and patch variables, the drawing, and the contents of the output area if one exists, to an external file given by the string *filename*. (The result file can be read back into NetLogo with the `import-world` primitive.)

`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 the file already exists, it is overwritten.

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
```

extract-hsb

extract-hsb *color*

Reports a list of three values in the range 0.0 to 1.0 representing the hue, saturation and brightness, respectively, of the given NetLogo *color* in the range 0 to 140.

```
show extract-hsb red
=> [0.0090 0.809 0.843]
show extract-hsb cyan
=> [0.5 0.571 0.769]
```

See also [hsb](#), [rgb](#), [extract-rgb](#).

extract-rgb

extract-rgb *color*

Reports a list of three values in the range 0.0 to 1.0 representing the levels of red, green, and blue, respectively, of the given NetLogo *color* in the range 0 to 140.

```
show extract-rgb red
=> [0.843 0.196 0.161]
show extract-rgb cyan
=> [0.329 0.769 0.769]
```

See also [rgb](#), [hsb](#), [extract-hsb](#).

F

face

face *agent*



Set the caller's heading towards *agent*.

If wrapping is allowed by the topology and the wrapped distance (around the edges of the view) is shorter than the on-screen distance, face will use the heading of the wrapped path.

If the caller and the agent are at the exact same position, the caller's heading won't change.

facexy

facexy *number number*



Set the caller's heading towards the point (x,y).

If wrapping is allowed by the topology and the wrapped distance (around the edges of the view) is shorter than the on-screen distance and wrapping is allowed, facexy will use the heading of the wrapped path.

If the caller is on the point (x,y), the caller's heading won't change.

file-at-end?

file-at-end?

Reports true when there are no more characters left to read in from the current file (that was opened previously with [file-open](#)). Otherwise, reports false.

```
file-open "myfile.txt"
print file-at-end?
=> false ;; Can still read in more characters
print file-read-line
=> This is the last line in file
print file-at-end
=> true ;; We reached the end of the file
```

See also [file-open](#), [file-close-all](#).

file-close

file-close

Closes a file that has been opened previously with [file-open](#).

Note that this and [file-close-all](#) are the only ways to restart to the beginning of an opened file or to switch between file modes.

If no file is open, does nothing.

See also [file-close-all](#), [file-open](#).

file-close-all

file-close-all

Closes all files (if any) that have been opened previously with [file-open](#).

See also [file-close](#), [file-open](#).

file-delete

file-delete *string*

Deletes the file specified as *string*

string must be an existing file with writable permission by the user. Also, the file cannot be open. Use the command [file-close](#) to close an opened file before deletion.

Note that the string can either be a file name or an absolute file path. If it is a file name, it looks in whatever the current directory is. This can be changed using the command [set-current-directory](#). It is defaulted to the model's directory.

file-exists?

file-exists? *string*

Reports true if *string* is the name of an existing file on the system. Otherwise it reports false.

Note that the string can either be a file name or an absolute file path. If it is a file name, it looks in whatever the current directory is. This can be changed using the command set-current-directory. It defaults to the model's directory.

file-open

file-open *string*

This command will interpret *string* as a path name to a file and open the file. You may then use the reporters file-read, file-read-line, and file-read-characters to read in from the file, or file-write, file-print, file-type, or file-show to write out to the file.

Note that you can only open a file for reading or writing but not both. The next file i/o primitive you use after this command dictates which mode the file is opened in. To switch modes, you need to close the file using file-close.

Also, the file must already exist if opening a file in reading mode.

When opening a file in writing mode, all new data will be appended to the end of the original file. If there is no original file, a new blank file will be created in its place. (You must have write permission in the file's directory.) (If you don't want to append, but want to replace the file's existing contents, use file-delete to delete it first, perhaps inside a carefully if you're not sure whether it already exists.)

Note that the string can either be a file name or an absolute file path. If it is a file name, it looks in whatever the current directory is. This can be changed using the command set-current-directory. It is defaulted to the model's directory.

```
file-open "myfile-in.txt"
print file-read-line
=> First line in file ;; File is in reading mode
file-open "C:\\NetLogo\\myfile-out.txt"
;; assuming Windows machine
file-print "Hello World" ;; File is in writing mode
```

See also file-close.

file-print

file-print *value*

Prints *value* to an opened file, followed by a carriage return.

The calling agent is *not* printed before the value, unlike file-show.

Note that this command is the file i/o equivalent of print, and file-open needs to be called before this command can be used.

See also file-show, file-type, and file-write.

file-read

file-read

This reporter will read in the next constant from the opened file and interpret it as if it had been typed in the Command Center. It reports the resulting value. The result may be a number, list, string, boolean, or the special value nobody.

Whitespace separates the constants. Each call to file-read will skip past both leading and trailing whitespace.

Note that strings need to have quotes around them. Use the command file-write to have quotes included.

Also note that the file-open command must be called before this reporter can be used, and there must be data remaining in the file. Use the reporter file-at-end? to determine if you are at the end of the file.

```
file-open "myfile.data"
print file-read + 5
;; Next value is the number 1
=> 6
print length file-read
;; Next value is the list [1 2 3 4]
=> 4
```

See also file-open and file-write.

file-read-characters

file-read-characters *number*

Reports the given *number* of characters from an opened file as a string. If there are fewer than that many characters left, it will report all of the remaining characters.

Note that it will return every character including newlines and spaces.

Also note that the file-open command must be called before this reporter can be used, and there must be data remaining in the file. Use the reporter file-at-end? to determine if you are at the end of the file.

```
file-open "myfile.txt"
print file-read-characters 8
;; Current line in file is "Hello World"
=> Hello Wo
```

See also file-open.

file-read-line

file-read-line

Reads the next line in the file and reports it as a string. It determines the end of the file by a carriage return, an end of file character or both in a row. It does not return the line terminator characters.

Also note that the file-open command must be called before this reporter can be used, and there must be data remaining in the file. Use the reporter file-at-end? to determine if you are at the end of the file.

```
file-open "myfile.txt"
print file-read-line
=> Hello World
```

See also file-open.

file-show

file-show *value*

Prints *value* to an opened file, preceded by the calling agent, and followed by a carriage return. (The calling agent is included to help you keep track of what agents are producing which lines of output.) Also, all strings have their quotes included similar to file-write.

Note that this command is the file i/o equivalent of show, and file-open needs to be called before this command can be used.

See also file-print, file-type, and file-write.

file-type

file-type *value*

Prints *value* to an opened file, *not* followed by a carriage return (unlike file-print and file-show). The lack of a carriage return allows you to print several values on the same line.

The calling agent is *not* printed before the value. unlike file-show.

Note that this command is the file i/o equivalent of type, and file-open needs to be called before this command can be used.

See also file-print, file-show, and file-write.

file-write

file-write *value*

This command will output *value*, which can be a number, string, list, boolean, or nobody to an opened file *not* followed by a carriage return (unlike [file-print](#) and [file-show](#)).

The calling agent is *not* printed before the value, unlike [file-show](#). Its output also includes quotes around strings and is prepended with a space. It will output the value in such a manner that [file-read](#) will be able to interpret it.

Note that this command is the file i/o equivalent of [write](#), and [file-open](#) needs to be called before this command can be used.

```
file-open "locations.txt"
ask turtles
  [ file-write xcor file-write ycor ]
```

See also [file-print](#), [file-show](#), and [file-type](#).

filter**filter [*reporter*] *list***

Reports a list containing only those items of *list* for which the boolean *reporter* is true -- in other words, the items satisfying the given condition.

In *reporter*, use [?](#) to refer to the current item of *list*.

```
show filter [? < 3] [1 3 2]
=> [1 2]
show filter [first ? != "t"] ["hi" "there" "everyone"]
=> ["hi" "everyone"]
```

See also [map](#), [reduce](#), [?](#).

first**first *list*****first *string***

On a list, reports the first (0th) item in the list.

On a string, reports a one-character string containing only the first character of the original string.

floor**floor *number***

Reports the largest integer less than or equal to *number*.

```
show floor 4.5
=> 4
```

```
show floor -4.5
=> -5
```

follow

follow *turtle*



Similar to *ride*, but, in the 3D view, the view is behind and above *turtle*.

See also [follow-me](#), [ride](#), [reset-perspective](#), [watch](#), [subject](#).

follow-me

follow-me



Asks the observer to follow the calling turtle.

See also [follow](#).

foreach

foreach *list* [*commands*]

(foreach *list1* ... *listn* [*commands*])

With a single list, runs *commands* for each item of *list*. In *commands*, use ? to refer to the current item of *list*.

```
foreach [1.1 2.2 2.6] [ show ? + " -> " + round ? ]
=> 1.1 -> 1
=> 2.2 -> 2
=> 2.6 -> 3
```

With multiple lists, runs *commands* 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. In *commands*, use ?1 through ?n to refer to the current item of each list.

Some examples make this clearer:

```
(foreach [1 2 3] [2 4 6]
  [ show "the sum is: " + (?1 + ?2) ])
=> "the sum is: 3"
=> "the sum is: 6"
=> "the sum is: 9"
(foreach list (turtle 1) (turtle 2) [3 4]
  [ ask ?1 [ fd ?2 ] ])
;; turtle 1 moves forward 3 patches
;; turtle 2 moves forward 4 patches
```

See also [map](#), [?](#).

forward

fd

forward *number*



The turtle moves forward by *number* steps. (If *number* is negative, the turtle moves backward.)

Turtles using this primitive can move a maximum of one unit per turn. So `fd 0.5` and `fd 1` both take one turn, but `fd 3` takes three.

If the turtle cannot move forward *number* steps because it is not permitted by the current topology the turtle will complete as many steps of 1 as it can, then stop.

See also [jump](#), [can-move?](#).

fput

fput *item list*

Adds *item* to the beginning of a list and reports the new list.

```
;; suppose mylist is [5 7 10]
set mylist fput 2 mylist
;; mylist is now [2 5 7 10]
```

G

globals

globals [*var1 var2 ...*]

This keyword, like the `breed`, `<breeds>-own`, `patches-own`, and `turtles-own` keywords, can only be used at the beginning of a program, before any function definitions. It defines new global variables. Global variables are "global" because they are accessible by all agents and can be used anywhere in a model.

Most often, `globals` is used to define variables or constants that need to be used in many parts of the program.

H

hatch

hatch—*<breeds>*

hatch *number* [*commands*]
hatch-<breeds> *number* [*commands*]



This turtle creates *number* new turtles, each identical to its parent, and asks the new turtles to run *commands*. You can use the commands to give 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 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.

Note: While the commands are running, no other agents are allowed to run any code (as with the without-interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no display updates take place until the commands are done. This ensures that the new turtles are never drawn in a partly initialized state.

```
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
```

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 variable. It holds a boolean (true or false) value indicating whether the turtle is currently hidden (i.e., invisible). You can set this variable to make a turtle disappear or reappear.

See also [hideturtle](#), [showturtle](#).

Example:

```
set hidden? not hidden?
```

```
;; if turtle was showing, it hides, and if it was hiding,
;; it reappears
```

hideturtle

ht

hideturtle



The turtle makes itself invisible.

Note: This command is equivalent to setting the turtle variable "hidden?" to true.

See also [show-turtle](#).

histogram-from

histogram-from *agentset* [*reporter*]

Draws a histogram showing the frequency distribution of the values reported when all agents in the agentset run the given reporter. The heights of the bars in the histogram represent the numbers of agents with values in those ranges.

Before the histogram is drawn, first any previous points drawn by the current plot pen are removed.

The reporter should report a numeric value. Any non-numeric values reported are ignored.

The histogram is drawn on the current plot using the current plot pen and pen color. Use `set-plot-x-range` to control the range of values to be histogrammed, and set the pen interval (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).

As of NetLogo 2.0.2, 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-from turtles [color]
;; draws a histogram showing how many turtles there are
;; of each color
```

Note: using this primitive amounts to the same thing as writing: `histogram-list values-from agentset [reporter]`, but is more efficient.

histogram-list

histogram-list *list*

Histograms the values in the given list, after first removing any previous points drawn by the current plot pen.

See [histogram-from](#), above, for more information.

home

home



The calling turtles moves to the origin (0,0). Equivalent to `setxy 0 0`.

hsb

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.

All three values should be in the range 0.0 to 1.0.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (It contains only certain discrete hues, and for each hue, either saturation or brightness may vary, but not both — at least one of the two is always 1.0.)

```
show hsb 0 0 0
=> 0.0 ;; (black)
show hsb 0.5 1.0 1.0
=> 85.0 ;; (cyan)
```

See also [extract-hsb](#), [rgb](#), [extract-rgb](#).

hubnet-broadcast

hubnet-broadcast tag-name value

This broadcasts *value* from NetLogo to the variable, in the case of Calculator HubNet, or interface element, in the case of Computer HubNet, with the name *tag-name* to the clients.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-broadcast-view

hubnet-broadcast-view

This broadcasts the current state of the 2D view in the NetLogo model to all the Computer HubNet Clients. It does nothing for Calculator HubNet.

Note: This is an experimental primitive and its behavior may change in a future version.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-enter-message?

hubnet-enter-message?

Reports true if a new computer client just entered the simulation. Reports false otherwise. hubnet-message-source will contain the user name of the client that just logged on.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-exit-message?

hubnet-exit-message?

Reports true if a computer client just exited the simulation. Reports false otherwise. hubnet-message-source will contain the user name of the client that just logged off.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-fetch-message

hubnet-fetch-message

If there is any new data sent by the clients, this retrieves the next piece of data, so that it can be accessed by hubnet-message, hubnet-message-source, and hubnet-message-tag. This will cause an error if there is no new data from the clients.

See the [HubNet Authoring Guide](#) for details.

hubnet-message

hubnet-message

Reports the message retrieved by hubnet-fetch-message.

See the [HubNet Authoring Guide](#) for details.

hubnet-message-source

hubnet-message-source

Reports the name of the client that sent the message retrieved by hubnet-fetch-message.

See the [HubNet Authoring Guide](#) for details.

hubnet-message-tag

hubnet-message-tag

Reports the tag that is associated with the data that was retrieved by [hubnet-fetch-message](#). For Calculator HubNet, this will report one of the variable names set with the [hubnet-set-client-interface](#) primitive. For Computer HubNet, this will report one of the Display Names of the interface elements in the client interface.

See the [HubNet Authoring Guide](#) for details.

hubnet-message-waiting?**hubnet-message-waiting?**

This looks for a new message sent by the clients. It reports true if there is one, and false if there is not.

See the [HubNet Authoring Guide](#) for details.

hubnet-reset**hubnet-reset**

Starts up the HubNet system. HubNet must be started to use any of the other hubnet primitives with the exception of [hubnet-set-client-interface](#).

See the [HubNet Authoring Guide](#) for details.

hubnet-send**hubnet-send *string tag-name value*****hubnet-send *list-of-strings tag-name value***

For Calculator HubNet, this primitive acts in exactly the same manner as [hubnet-broadcast](#). (We plan to change this in a future version of NetLogo.)

For Computer HubNet, it acts as follows:

For a *string*, this sends *value* from NetLogo to the tag *tag-name* on the client that has *string* for its user name.

For a *list-of-strings*, this sends *value* from NetLogo to the tag *tag-name* on all the clients that have a user name that is in the *list-of-strings*.

Sending a message to a non-existent client, using `hubnet-send`, generates a `hubnet-exit-message`.

See the [HubNet Authoring Guide](#) for details.

hubnet-send-view

hubnet-send-view *string*

hubnet-send-view *list-of-strings*

For Calculator HubNet, does nothing.

For Computer HubNet, it acts as follows:

For a *string*, this sends the current state of the 2D view in the NetLogo model to the Computer HubNet Client with *string* for its user name.

For a *list-of-strings*, this sends the current state of the view in the NetLogo model to all the Computer HubNet Clients that have a user name that is in the *list-of-strings*.

Sending the 2D view to a non-existent client, using `hubnet-send-view`, generates a `hubnet-exit-message`.

Note: This is an experimental primitive and its behavior may change in a future version.

See the [HubNet Authoring Guide](#) for details.

hubnet-set-client-interface

hubnet-set-client-interface *client-type client-info*

If *client-type* is "COMPUTER", *client-info* is a list containing a string with the file name and path (relative to the model) to the file which will serve as the client's interface. This interface will be sent to any clients that log in.

```
hubnet-set-client-interface
  "COMPUTER"
  ["clients/Disease client.nlogo"]
;; when clients log in, they will get the
;; interface described in the file
;; clients/Disease client.nlogo, relative to
;; the location of the model
```

Future versions of HubNet will support other client types. Even for Computer HubNet, the meaning of the second input to this command may change.

See the [HubNet Authoring Guide](#) for details.

I

if

if *condition* [*commands*]

Reporter must report a boolean (true or false) value.

If *condition* reports true, runs *commands*.

The reporter may report a different value for different agents, so some agents may run *commands* and others don't.

```
if xcor > 0[ set color blue ]
;; turtles on the right half of the view
;; turn blue
```

See also [ifelse](#), [ifelse-value](#).

ifelse**ifelse *reporter* [*commands1*] [*commands2*]**

Reporter must report a boolean (true or false) value.

If *reporter* reports true, runs *commands1*.

If *reporter* reports false, runs *commands2*.

The reporter may report a different value for different agents, so some agents may run *commands1* while others run *commands2*.

```
ask patches
[ ifelse pxcor > 0
  [ set pcolor blue ]
  [ set pcolor red ] ]
;; the left half of the view turns red and
;; the right half turns blue
```

See also [if](#), [ifelse-value](#).

ifelse-value**ifelse-value *reporter* [*reporter1*] [*reporter2*]**

Reporter must report a boolean (true or false) value.

If *reporter* reports true, the result is the value of *reporter1*.

If *reporter* reports false, the result is the value of *reporter2*.

This can be used when a conditional is needed in the context of a reporter, where commands (such as [ifelse](#)) are not allowed.

```
ask patches
[ set pcolor
```

```

ifelse-value (pxcor > 0)
  [ blue ]
  [ red ] ]
;; the left half of the view turns red and
;; the right half turns blue
show n-values 10 [ifelse-value (? < 5) [0] [1]]
=> [0 0 0 0 0 1 1 1 1 1]
show reduce [ifelse-value (?1 > ?2) [?1] [?2]]
  [1 3 2 5 3 8 3 2 1]
=> 8

```

See also [if](#), [ifelse](#).

import-drawing

import-drawing *filename*



Reads an image file into the drawing, scaling it to the size of the world, while retaining the original aspect ratio of the image. The image is centered in the drawing. The old drawing is not cleared first.

Agents cannot sense the drawing, so they cannot interact with or process images imported by `import-drawing`. If you need agents to sense an image, use [import-pcolors](#).

The following image file formats are supported: BMP, JPG, GIF, and PNG. If the image format supports transparency (alpha), that information will be imported as well.

import-pcolors

import-pcolors *filename*



Reads an image file, scales it to the same dimensions as the patch grid while maintaining the original aspect ratio of the image, and transfers the resulting pixel colors to the patches. The image is centered in the patch grid. The resulting patch colors may be distorted, since the NetLogo color space does not include all possible colors. (See the Color section of the Programming Guide.) `import-pcolors` may be slow for some images, particularly when you have many patches and a large image with many different colors.

Since `import-pcolors` sets the `pcolor` of patches, agents can sense the image. This is useful if agents need to analyze, process, or otherwise interact with the image. If you want to simply display a static backdrop, without color distortion, see [import-drawing](#).

The following image file formats are supported: BMP, JPG, GIF, and PNG. If the image format supports transparency (alpha), then all fully transparent pixels will be ignored. (Partially transparent pixels will be treated as opaque.)

import-world

import-world *filename*

Reads the values of all variables for a model, both built-in and user-defined, including all observer, turtle, and patch variables, from an external file named by the given string. The file should be in the format used by the export-world primitive.

Note that the functionality of this primitive is also directly available from NetLogo's File menu.

When using import-world, to avoid errors, perform these steps in the following order:

1. Open the model from which you created the export file.
2. Press the Setup button, to get the model in a state from which it can be run.
3. Import the file.
4. If you want, press Go button to continue running the model from the point where it left off.

If you wish to import a file from a location other than the model's location, you may include the full path to the file you wish to import. See export-world for an example.

in-cone**agentset in-cone *distance angle***

This reporter lets you give a turtle a "cone of vision" in front of itself. The cone is defined by the two inputs, the vision distance (radius) and the view angle. The view 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 including 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-<breed>-neighbor? *agent***

Reports true if there is a link going from *agent* to the caller.

```
to check-friendship-neighbors
  ca
  cct-turtles 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
]
end

```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__in-<breed>-neighbors

__in-<breed>-neighbors



Reports the agentset of all the node turtles that have links coming from them to the caller.

```

breed [links link] ;; breed for network links

to become-friendly ;; node procedure
  __create-links-to friends
  []
  show __out-link-neighbors ;; prints the agentset of friends
end

```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__in-<breed>-from

__in-<breed>-from *agent*



Report the link turtle from the *agent* to the caller. If no link exists then it reports nobody.

```

to find-friendship-with
  ca
  cct 2 []
  ask turtle 0
  [
    __create-link-to turtle 1 []
    __create-link-from turtle 1 []
    show __in-link-from turtle 1 ;; prints turtle 3
    show __out-link-to turtle 1 ;; prints turtle 2
    show __link-with turtle 1 ;; prints turtle 2
  ]
end

```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

in-radius

agentset in-radius *number*



Reports an agentset that includes only those agents from the original agentset whose distance from the caller is less than or equal to *number*, including the caller itself if wrapping is allowed by the topology members of the agentset within the radius using the wrapped distance will be included.

The distance to or a from a patch is measured from the center of the patch.

```
ask turtles
  [ ask patches in-radius 3
    [ set pcolor red ] ]
;; each turtle makes a red "splotch" around itself
```

inspect

inspect *agent*

Opens an agent monitor for the given agent (turtle or patch).

```
inspect patch 2 4
;; an agent monitor opens for that patch
inspect one-of sheep
;; an agent monitor opens for a random turtle from
;; the "sheep" breed
```

int

int *number*

Reports the integer part of number -- any fractional part is discarded.

```
show int 4.7
=> 4
show int -3.5
=> -3
```

is-agent?

is-agentset?

is-boolean?

is-<breed>?

__is-link?

is-list?

is-number?

is-patch?

is-patch-agentset?

is-string?**is-turtle?****is-turtle-agentset?****is-agent?** *value***is-agentset?** *value***is-boolean?** *value***is-<breed>?** *value***__is-link?** *value***is-list?** *value***is-number?** *value***is-patch?** *value***is-patch-agentset?** *value***is-string?** *value***is-turtle?** *value***is-turtle-agentset?** *value*

Reports true if *value* is of the given type, false otherwise.

item

item *index list***item** *index string*

On lists, reports the value of the item in the given list with the given index.

On strings, reports the character in the given string at the given index.

Note that the indices begin from 0, not 1. (The first item is item 0, the second item is item 1, and so on.)

```
;; suppose mylist is [2 4 6 8 10]
show item 2 mylist
=> 6
show item 3 "my-shoe"
=> "s"
```

The `__is-link?` reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

J

jump

jump *number*

Turtles move forward by *number* units all at once, without the number of turns it takes depending on the distance.

This command is useful for synchronizing turtle movements. A turtle takes 15 turns to do `fd 15`, but only one turn to do `jump 15`.

If the turtle cannot jump *number* units because it is not permitted by the current topology the turtle does not move at all.

Note: When turtles jump, they do not step on any of the patches along their path.

L

label

label



This is a built-in turtle variable. It may hold a value of any type. The turtle 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's label.

See also [label-color](#), [plabel](#), [plabel-color](#).

Example:

```
ask turtles [ set label who ]
;; all the turtles now are labeled with their
;; id numbers
ask turtles [ set label "" ]
;; all turtles now are not labeled
```

label-color

label-color



This is a built-in turtle variable. It holds a number greater than or equal to 0 and less than 140. This number determines what color the turtle's label appears in (if it has a label). You can set this variable to change the color of a turtle's label.

See also [label](#), [plabel](#), [plabel-color](#).

Example:

```
ask turtles [ set label-color red ]
;; all the turtles now have red labels
```

last

last *list*
last *string*

On a list, reports the last item in the list.

On a string, reports a one-character string containing only the last character of the original string.

__layout-circle

__layout-circle *agentset radius*
__layout-circle *list-of-turtles radius*

Arranges the given turtles in a circle centered on the patch at the center of the world with the given radius. (If the world has an even size the center of the circle is rounded down to the nearest patch.) The turtles point outwards.

If the first input is an agentset, the turtles are arranged in random order.

If the first input is a list, the turtles are arranged clockwise in the given order, starting at the top of the circle. (Any non-turtles in the list are ignored.)

```
;; in random order
__layout-circle turtles 10
;; in order by who number
__layout-circle sort turtles 10
;; in order by size
__layout-circle sort-by [size-of ?1 <size-of ?2] turtles 10
```

__layout-magspring

__layout-magspring *agentset anchor-agentset spring-constant spring-length*
repulsion-constant magnetic-field-strength magnetic-field-type bidirectional?

Very similar to __layout-spring, but with an added layer of complexity. The *agentset* of turtles attract and repel each other depending on the links between them, but there is also a magnetic field which the links try to align with.

anchor-agentset is a set of turtles whose positions remain fixed. (If no turtles are anchored down, then they will often all slide off to the side of the view!)

spring-constant is a measure of the "tautness" of the spring. (See __layout-spring)

spring-length is the "zero-force" length or the natural length of the springs. (See __layout-spring)

repulsion-constant is a measure of repulsion between the nodes. (See __layout-spring)

magnetic-field-strength is the force of the magnetic field. (Reasonable values range from 0.0 to 1.0, but 0.05 is a good default.)

magnetic-field-type is a number in the range from 0 to 10. Choices are listed in the table below.

<i>magnetic-field-type</i>	Description
NONE = 0	If no field is used, then this command works just like layout-spring.
NORTH = 1	Magnetic field runs toward the North
NORTHEAST = 2	Magnetic field runs toward the Northeast
EAST = 3	...
SOUTHEAST = 4	...
SOUTH = 5	...
SOUTHWEST = 6	...
WEST = 7	...
NORTHWEST = 8	...
POLAR = 9	Magnetic field runs outward at all angles from the origin.
CONCENTRIC = 10	Magnetic field runs clockwise around the origin in concentric circles.

If *bidirectional?* is true then links try to align with the magnetic field by pushing attached turtles both in the direction of the field, and in the opposite direction. Otherwise, the links just push in a single direction.

```

breed [links link]
breed[nodes node]
to make-a-tree
  set-default-shape nodes "circle"
  create-custom-nodes 5 []
  ask turtle 0
  [
    __create-link-with turtle 1 []
    __create-link-with turtle 2 []
  ]
  ask turtle 1
  [
    __create-link-with turtle 3 []
    __create-link-with turtle 4 []
  ]
  ; one turtle (the root of the tree) is our anchor-set
  let anchor-set (turtles with [ who = 0 ])
  ; layout with a fairly strong SOUTH magnetic field
  repeat 50 [ __layout-magspring
    nodes anchor-set 0.3 4 1.0 .50 5 false ]
end

```

This command is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__layout-radial

__layout-radial *agentset root-agent breed*

Arranges the agentset of turtles, *agentset*, in a radial tree layout, centered around *root-agent*.

Only links of the given *breed* will be used to determine the layout. (Useful if you have multiple link breeds in one model.)

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.

If *agentset* is an agentset of patches, an error is thrown.

```
breed [links link]

to make-a-tree
  set-default-shape turtles "circle"
  cct 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 (turtle 0) links
end
```

This command is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__layout-spring**__layout-spring *agentset spring-constant spring-length repulsion-constant***

Arranges the agentset of turtles, *agentset*, as if the links with which they are connected to each other were springs and the nodes were repelling each other.

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.

If *agentset* is an agentset of patches, an error is thrown.

The layout algorithm is based on the Fruchterman–Reingold layout algorithm. More information about this algorithm can be obtained [here](#).

```
breed [links link]
breed [nodes node]

to make-a-triangle
  set-default-shape nodes "circle"
  create-custom-nodes 3 []
  ask turtle 0
  [
    __create-links-with turtles with [ who-of self != 0 ]
    []
  ]
  ask turtle 1
  [
    __create-link-with turtle 2
    []
  ]
  repeat 30 [__layout-spring nodes 0.2 5 1] ;; lays the nodes in a triangle
end
```

This command is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__layout-tutte

__layout-tutte *agentset anchor-agentset radius*

The turtles in *anchor-agentset* are placed in a circle layout with the given *radius*. There should be at least 3 agents in the *anchor-agentset*.

The turtles in *agentset* (unless they are also members of *anchor-agentset*) 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-agentset* is to prevent all the turtles from collapsing down to one point.)

After a few iterations of this, the layout will stabilize.

If *agentset* is an agentset of patches, an error is thrown.

This layout is named after the mathematician William Thomas Tutte, who proposed it as a method for graph layout.

```
breed [links link ]

to make-a-tree
  set-default-shape turtles "circle"
  cct 6 []
```

```
ask turtle 0
[
  __create-link-with turtle 1 []
  __create-link-with turtle 2 []
  __create-link-with turtle 3 []
]
ask turtle 1
[
  __create-link-with turtle 4 []
  __create-link-with turtle 5 []
]
; place all the turtles with just one
; neighbor on the perimeter of a circle
; and then place the remaining turtles inside
; this circle, spread between their neighbors.
repeat 10 [ __layout-tutte turtles
            (turtles with [ count __link-neighbors = 1]) 12.0 ]
end
```

This command is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

left

It

left *number*



The turtle turns left by *number* degrees. (If *number* is negative, it turns right.)

length

length *list*

length *string*

Reports the number of items in the given list, or the number of characters in the given string.

let

let *variable value*

Creates a new local variable and gives it the given value. A local variable is one that exists only within the enclosing block of commands.

If you want to change the value afterwards, use [set](#).

Example:

```
let prey one-of sheep-here
if prey != nobody
[ ask prey [ die ] ]
```

list

list *value1 value2*
(list *value1 ... valuen***)**

Reports a list containing the given items. The items can be of any type, produced by any kind of reporter.

```
show list (random 10) (random 10)
=> [4 9] ;; or similar list
show (list 5)
=> [5]
show (list (random 10) 1 2 3 (random 10))
=> [4 1 2 3 9] ;; or similar list
```

ln

ln *number*

Reports the natural logarithm of *number*, that is, the logarithm to the base e (2.71828...).

See also [e](#), [log](#).

locals

locals [*Vax1 var2 ...*]

NOTE: This keyword should not be used in new models. Please use the [let](#) command instead. "locals" is included only for backwards compatibility with NetLogo version 2.0 and earlier. It will not necessarily continue to be supported in future versions of NetLogo.

Locals is a keyword used to declare "local" variables in a procedure, that is, variables that are usable only within that procedure. It must appear at the beginning of the procedure, before any commands.

See also [let](#).

log

log *number base*

Reports the logarithm of *number* in base *base*.

```
show log 64 2
=> 6
```

See also [ln](#).

loop

loop [*commands*]

Runs the list of commands forever, or until the current procedure exits through use of the stop command or the report command.

Note: In most circumstances, you should use a forever button in order to repeat something forever. The advantage of using a forever button is that the user can click the button to stop the loop.

lput

lput *value list*

Adds *value* to the end of a list and reports the new list.

```
;; suppose mylist is [2 7 10 "Bob"]
set mylist lput 42 mylist
;; mylist now is [2 7 10 "Bob" 42]
```

M

map

map [*reporter*] *list* **(map [*reporter*] *list1 ... list2*)**

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.

In *reporter*, use ? to refer to the current item of *list*.

```
show map [round ?] [1.1 2.2 2.7]
=> [1 2 3]
show map [? * ?] [1 2 3]
=> [1 4 9]
```

With multiple lists, the given reporter is run for each group of items from each list. So, it is run once for the first items, once for the second items, and so on. All the lists must be the same length.

In *reporter*, use ?1 through ?n to refer to the current item of each list.

Some examples make this clearer:

```
show (map [?1 + ?2] [1 2 3] [2 4 6])
=> [3 6 9]
show (map [?1 + ?2 = ?3] [1 2 3] [2 4 6] [3 5 9])
=> [true false true]
```

See also foreach, ?.

max

max *list*

Reports the maximum number value in the list. It ignores other types of items.

```
show max values-from turtles [xcor]
;; prints the x coordinate of the turtle which is
;; farthest right in the view
```

max-one-of

max-one-of *agentset* [*reporter*]

Reports the agent in the agentset that has the highest value for the given reporter. If there is a tie this command reports one random agent with the highest value. If you want all such agents, use `with-max` instead.

```
show max-one-of patches [count turtles-here]

;; prints the first patch with the most turtles on it
```

See also [with-max](#)

max-pxcor

max-pycor

max-pxcor

max-pycor

These reporters give the maximum x-coordinate and maximum y-coordinate, (respectively) for patches, which determines the size of the world.

Unlike in older versions of NetLogo the origin does not have to be at the center of the world. However, the maximum x- and y- coordinates must be greater than or equal to zero.

Note: You can set the size of the world only by editing the view -- these are reporters which cannot be set.

```
cct 100 [ setxy random-float max-pxcor
           random-float max-pycor ]
;; distributes 100 turtles randomly in the
;; first quadrant
```

See also [min-pxcor](#), [min-pycor](#), [world-width](#), and [world-height](#)

mean

mean *list*

Reports the statistical mean of the numeric items in the given list. Ignores non-numeric items. The mean is the average, i.e., the sum of the items divided by the total number of items.

```
show mean values-from turtles [xcor]
;; prints the average of all the turtles' x coordinates
```

median**median *list***

Reports the statistical median of the numeric items of the given list. Ignores non-numeric items. The median is the item that would be in the middle if all the items were arranged in order. (If two items would be in the middle, the median is the average of the two.)

```
show median values-from turtles [xcor]
;; prints the median of all the turtles' x coordinates
```

member?**member? *value list*****member? *string1 string2*****member? *agent agentset***

For a list, reports true if the given value appears in the given list, otherwise reports false.

For a string, reports true or false depending on whether *string1* appears anywhere inside *string2* as a substring.

For an agentset, reports true if the given agent is appears in the given agentset, otherwise reports false.

```
show member? 2 [1 2 3]
=> true
show member? 4 [1 2 3]
=> false
show member? "rin" "string"
=> true
show member? turtle 0 turtles
=> true
show member? turtle 0 patches
=> false
```

See also [position](#).

min**min *list***

Reports the minimum number value in the list. It ignores other types of items.

```
show min values-from turtles [xcor]
;; prints the lowest x-coordinate of all the turtles
```

min-one-of

min-one-of *agentset* [*reporter*]

Reports a random agent in the agentset that reports the lowest value for the given reporter. If there is a tie, this command reports one random agent that meets the condition. If you want all such agents use with-min instead.

```
show min-one-of turtles [xcor + ycor]
;; reports the first turtle with the smallest sum of
;; coordinates
```

See also [with-min](#)

min-pxcor

min-pycor

min-pxcor

min-pycor

These reporters give the minimum x-coordinate and minimum y-coordinate, (respectively) for patches, which determines the size of the world.

Unlike in older versions of NetLogo the origin does not have to be at the center of the world. However, the minimum x- and y- coordinates must be less than or equal to zero.

Note: You can set the size of the world only by editing the view — these are reporters which cannot be set.

```
cct 100 [ setxy random-float min-pxcor
              random-float min-pycor ]
;; distributes 100 turtles randomly in the
;; third quadrant
```

See also [max-pxcor](#), [max-pycor](#), [world-width](#), and [world-height](#)

mod

number1 mod *number2*

Reports *number1* modulo *number2*: that is, the residue of *number1* (mod *number2*). mod is equivalent to the following NetLogo code:

```
number1 - (floor (number1 / number2)) * number2
```

Note that mod is "infix", that is, it comes between its two inputs.

```
show 62 mod 5
```

```
=> 2
show -8 mod 3
=> 1
```

See also [remainder](#). `mod` and `remainder` behave the same for positive numbers, but differently for negative numbers.

modes

modes *list*

Reports a list of the most common item or items in *list*.

The input list may contain any NetLogo values.

If the input is an empty list, reports an empty list.

```
show modes [1 2 2 3 4]
=> [2]
show modes [1 2 2 3 3 4]
=> [2 3]
show modes [ [1 2 [3]] [1 2 [3]] [2 3 4] ]
=> [[1 2 [3]]
show modes values-from turtles [pxcor]
;; shows which columns of patches have the most
;; turtles on them
```

mouse-down?

mouse-down?

Reports true if the mouse button is down, false otherwise.

Note: If the mouse pointer is outside of the current view , `mouse-down?` will always report false.

mouse-inside?

mouse-inside?

Reports true if the mouse pointer is inside the current view, false otherwise.

mouse-xcor

mouse-ycor

mouse-xcor

mouse-ycor

Reports the x or y coordinate of the mouse in the 2D view. The value is in terms of turtle coordinates, so it is a floating-point number. 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?
  [ set pcolor-of patch-at mouse-xcor mouse-ycor red ]
```

movie-cancel

movie-cancel

Cancels the current movie.

movie-close

movie-close

Stops the recording of the current movie.

movie-grab-view movie-grab-interface

movie-grab-view movie-grab-interface

Adds an image of the current view or the interface panel to the current movie.

```
;; make a 20-step movie of the current view
setup
movie-start "out.mov"
repeat 20
  [ movie-grab-view
    go ]
movie-close
```

movie-set-frame-rate

movie-set-frame-rate *frame-rate*

Sets the frame rate of the current movie. The frame rate is measured in frames per second. (If you do not explicitly set the frame rate, it defaults to 15 frames per second.)

Must be called after movie-start, but before movie-grab-view or movie-grab-interface.

See also movie-status.

movie-start

movie-start *filename*

Creates a new movie. *filename* specifies a new QuickTime file where the movie will be saved, so it should end with ".mov".

See also [movie-grab-view](#), [movie-grab-interface](#), [movie-cancel](#), [movie-status](#), [movie-set-frame-rate](#), [movie-close](#).

movie-status**movie-status**

Reports a string describing the current movie.

```
print movie-status
=> No movie.
movie-start
print movie-status
=> 0 frames; Framerate = 15.0.
movie-grab-view
print movie-status
1 frames; Framerate = 15.0; Size = 315x315.
```

__my-<breeds>**__my-<breeds>**

Reports an agentset of all links coming in from other nodes to the caller.

```
breed [links link] ;; breed for network links

to become-friendly ;; node procedure
  __create-links-with other-turtles
  []
  show __my-links ;; prints the agentset of link turtles that connecting
                  ;; with this node.
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__my-in-<breeds>**__my-in-<breeds>**

Reports an agentset of all the directed links coming in from other nodes to the caller.

```
breed [links link] ;; breed for network links

to become-friendly ;; node procedure
  __create-links-to other-turtles
```



```
[ ]
show __my-in-links ;; prints the agentset of link turtles that connect
                    ;; to this node
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__my-out-<breeds>

__my-out-<breeds>



Reports an agentset of all the directed links going out from the caller to other nodes.

```
breed [links link] ;; breed for network links

to become-friendly ;; node procedure
  __create-links-to other-turtles
  [ ]
  show __my-out-links ;; prints the agentset of link turtles that connect
                      ;; from this node
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

myself

myself



"self" and "myself" are very different. "self" is simple; it means "me". "myself" means "the turtle or patch 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 or patch) 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, values-from, value-from, turtles-from, patches-from, histogram-from, with, min-one-of, and max-one-of.

```
ask turtles
[ ask patches in-radius 3
  [ set pcolor color-of myself ] ]
;; each turtle makes a colored "splotch" around itself
```

See the "Myself Example" code example for more examples.

See also [self](#).

N

n-of

n-of *size agentset*

From an agentset, reports an agentset of size *size* randomly chosen from the input set, with no repeats.

From a list, reports a list of size *size* randomly chosen from the input set, with no repeats. The items in the result appear in the same order that they appeared in the input list. (If you want them in random order, use `shuffle` on the result.)

It is an error for *size* to be greater than the size of the input.

```
ask n-of 50 patches [ set pcolor green ]
;; 50 randomly chosen patches turn green
```

See also [one-of](#).

n-values

n-values *size [reporter]*

Reports a list of length *size* containing values computed by repeatedly running *reporter*.

In *reporter*, use `?` to refer to the number of the item currently being computed, starting from zero.

```
show n-values 5 [1]
=> [1 1 1 1 1]
show n-values 5 [?]
=> [0 1 2 3 4]
show n-values 3 [turtle ?]
=> [(turtle 0) (turtle 1) (turtle 2)]
show n-values 5 [? * ?]
=> [0 1 4 9 16]
```

See also [reduce](#), [filter](#), [?](#).

neighbors neighbors4

neighbors neighbors4



Reports an agentset containing the 8 surrounding patches (neighbors) or 4 surrounding patches (neighbors4).

```
show sum values-from neighbors [count turtles-here]
;; prints the total number of turtles on the eight
```

```

;; patches around the calling turtle or patch
ask neighbors4 [ set pcolor red ]
;; turns the four neighboring patches red

```

__<breed>-neighbors

__<breed>-neighbors



Reports the agentset of all the node turtles that have links coming from them to the caller, or coming from the caller to them.

```

breed [links link] ;; breed for network links

to become-friendly ;; node procedure
  __create-links-to friends
  []
  show __link-neighbors ;; prints the agentset of agents who are
                        ;; "friends" with, or whom it is "friends" with
end

```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__<breed>-neighbor?

__<breed>-neighbor? *agent*



Reports true if there is a link going from *agent* to the caller, or from the caller to *agent*.

```

to check-friendship-neighbors
  ca
  cct-turtles 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
    show __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
    show __link-neighbor? turtle 0    ;; prints true
  ]
end

```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

netlogo-version

netlogo-version

Reports a string containing the version number of the NetLogo you are running.

```
show netlogo-version  
=> "3.1.3"
```

new-seed

new-seed

Reports a number suitable for seeding the random number generator.

The numbers reported by new-seed are based on the current date and time in milliseconds and lie in the range -2147483648 to 2147483647.

new-seed never reports the same number twice in succession. (This is accomplished by waiting a millisecond if the seed for the current millisecond was already used.)

See also [random-seed](#).

no-display

no-display

Turns off all updates to the current view until the display command is issued. This has two major uses.

One, you can control when the user sees view updates. You might want to change lots of things on the view behind the user's back, so to speak, then make them visible to the user all at once.

Two, your model will run faster when view updating is off, so if you're in a hurry, this command will let you get results faster. (Note that normally you don't need to use no-display for this, since you can also use the on/off switch in view control strip to freeze the view.)

Note that display and no-display operate independently of the switch in the view control strip that freezes the view.

See also [display](#).

nobody

nobody

This is a special value which some primitives such as turtle, one-of, max-one-of, etc. report to indicate that no agent was found. Also, when a turtle dies, it becomes equal to nobody.

Note: Empty agentsets are not equal to nobody. If you want to test for an empty agentset, use any?. You only get nobody back in situations where you were expecting a single agent, not a whole agentset.

```
set other one-of other-turtles-here
if other != nobody
  [ set color-of other red ]
```

not

not *boolean*

Reports true if *boolean* is false, otherwise reports false.

```
if not (color = blue) [ fd 10 ]
;; all non-blue turtles move forward 10 steps
```

nsum

nsum4

nsum *patch-variable*

nsum4 *patch-variable*



For each patch, reports the sum of the values of *patch-variable* in the 8 surrounding patches (nsum) or 4 surrounding patches (nsum4).

Note that nsum/nsum4 are equivalent to the combination of the sum, values-from, and neighbors/neighbors4 primitives:

```
sum values-from neighbors [var]
  ;; does the same thing as "nsum var"
sum values-from neighbors4 [var]
  ;; does the same thing as "nsum4 var"
```

Therefore nsum and nsum4 are included as separate primitives mainly for backwards compatibility with older versions of NetLogo, which did not have the neighbors and neighbors4 primitives.

See also neighbors, neighbors4.

O

-of

VARIABLE-of *agent*

Reports the value of the *VARIABLE* of the given agent. Can also be used to set the value of the variable.

```
show pxcor-of one-of patches
;; prints the value of a random patch's pxcor variable
```

```
set color-of one-of turtles red
;; a randomly chosen turtle turns red
ask turtles [ set pcolor-of (patch-at -1 0) red ]
;; each turtle turns the patch on its left red
```

one-of

one-of *agentset*
one-of *list*

From an agentset, reports a random agent. If the agentset is empty, reports nobody.

From a list, reports a random list item. It is an error for the list to be empty.

```
ask one-of patches [ set pcolor green ]
;; a random patch turns green
set pcolor-of one-of patches green
;; another way to say the same thing
ask patches with [any? turtles-here]
[ show one-of turtles-here ]
;; for each patch containing turtles, prints one of
;; those turtles

;; suppose mylist is [1 2 3 4 5 6]
show one-of mylist
;; prints a value randomly chosen from the list
```

See also n-of.

or

boolean1* or *boolean2

Reports true if either *boolean1* or *boolean2*, or both, is true.

Note that if *condition1* is true, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) or (pycor > 0) [ set pcolor red ]
;; patches turn red except in lower-left quadrant
```

other-turtles-here

other-<*breeds*>-here

other-turtles-here
other-<*breeds*>-here



Reports an agentset consisting of all turtles on the calling turtle's patch (*not* including the caller itself). If a breed is specified, only turtles with the given breed are included.

```
;; suppose I am one of 10 turtles on the same patch
show count other-turtles-here
=> 9
```

Example using breeds:

```
breed [cats cat]
breed [dogs dog]
show count other-dogs-here
;; prints the number of dogs (that are not me) on my patch
```

See also [turtles-here](#).

__other-end

__other-end



Given a link and one of it's ends, this reports the other end of that link. First we "Ask" one of the ends and then "ask" the link. In the inner block "other-end" can be used as an idiom for the other-end of the link besides the end "ask"-ed earlier. The syntax is as shown below.

Suppose a link L connects nodes N1 and N2.

```
ask N1
[
  ask L
  [
    show __other-end ;; prints N2
  ]
]

ask N2
[
  ask L
  [
    show __other-end ;; prints N1
  ]
]
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__out-<breed>-neighbor?

__out-<breed>-neighbor? *agent*



Reports true if there is a link going from the caller to *agent*.

```
to check-friendship-neighbors
  ca
  cct-turtles 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
  ]
end
```

```
ask turtle 1
[
  show __in-link-neighbor? turtle 0 ;; prints true
  show __out-link-neighbor? turtle 0 ;; prints false
]
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

out-<breed>-neighbors

out-<breed>-neighbors



Reports the agentset of all the node turtles that have links coming from the caller to them.

```
breed [links link] ;; breed for network links

to become-friendly ;; node procedure
  __create-links-to friends
  []
  show __out-link-neighbors ;; prints the agentset of friends
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

__out-<breed>-to

__out-<breed>-to agent



Reports the link turtle from the call to the *agent*. If no link exists then it reports nobody.

```
to find-friendship-with
  ca
  cct 2 []
  ask turtle 0
  [
    __create-link-to turtle 1 []
    __create-link-from turtle 1 []
    show __in-link-from turtle 1 ;; prints turtle 3
    show __out-link-to turtle 1 ;; prints turtle 2
    show __link-with turtle 1 ;; prints turtle 2
  ]
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

output-print

output-show

output-type

output-write

output-print *value*
 output-show *value*
 output-type *value*
 output-write *value*

These commands are the same as the print, show, type, and write commands except that *value* is printed in the model's output area, instead of in the Command Center. (If the model does not have a separate output area, then the Command Center is used.)

P

patch

patch *pxcor pycor*

Given two integers, reports the single patch with the given *pxcor* and *pycor*. (The coordinates are the actual coordinates; they are not computed relative to the calling agent, as with patch-at.) *pxcor* and *pycor* must be integers.

```
ask (patch 3 -4) [ set pcolor green ]
;; patch with pxcor of 3 and pycor of -4 turns green
```

See also patch-at.

patch-ahead

patch-ahead *distance*



Reports the single patch that is the given distance "ahead" of the calling turtle, that is, along the turtle's current heading. Reports nobody if the patch does not exist because it is outside the world.

```
set pcolor-of (patch-ahead 1) green
;; turns the patch 1 in front of the calling turtle
;;   green; note that this might be the same patch
;;   the turtle is standing on
```

See also patch-at, patch-left-and-ahead, patch-right-and-ahead, patch-at-heading-and-distance.

patch-at

patch-at *dx dy*

Reports the single patch at (dx, dy) from the caller, that is, dx patches east and dy patches north of the caller. (If the caller is the observer, the given offsets are computed from the origin.) Reports nobody if the patch does not exist because it is outside the world.

```
ask patch-at 1 -1 [ set pcolor green ]
;; if caller is the observer, turn the patch
;;   at (1, -1) green
;; if caller is a turtle or patch, turns the
;;   patch just southeast of the caller green
```

See also [patch](#), [patch-ahead](#), [patch-left-and-ahead](#), [patch-right-and-ahead](#), [patch-at-heading-and-distance](#).

patch-at-heading-and-distance

patch-at-heading-and-distance *heading distance*



patch-at-heading-and-distance reports the single patch that is the given distance from the calling turtle or patch, along the given absolute heading. (In contrast to patch-left-and-ahead and patch-right-and-ahead, the calling turtle's current heading is not taken into account.) Reports nobody if the patch does not exist because it is outside the world.

```
set pcolor-of (patch-at-heading-and-distance -90 1) green
;; turns the patch 1 to the west of the calling patch
;;   green
```

See also [patch](#), [patch-at](#), [patch-left-and-ahead](#), [patch-right-and-ahead](#).

patch-here

patch-here



patch-here reports the patch under the turtle.

Note that this reporter isn't available to a patch because a patch can just say "self".

patch-left-and-ahead

patch-right-and-ahead

patch-left-and-ahead *angle distance*

patch-right-and-ahead *angle distance*



Reports the single patch that is the given distance from the calling 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.)

```
set pcolor-of (patch-right-and-ahead 30 1) green
;; the calling 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](#).

patches

patches

Reports the agentset consisting of all patches.

patches-from

patches-from *agentset* [*reporter*]

Reports a patch agentset made by gathering together all the patches reported by *reporter* for each agent in *agentset*.

For each agent, the reporter must report a patch agentset, a single patch, or nobody.

```
patches-from turtles [patch-here]
;; reports the set of all patches with turtles on them;
;; if there are many more patches than turtles, this will
;; run much faster than "patches with [any? turtles-here]"
```

See also [turtles-from](#).

patches-own

patches-own [*var1 var2 ...*]

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.

See also [color](#).

pen-down

pd

pen-erase

pe

pen-up

pu

pen-down

pen-erase

pen-up



The turtle changes modes between drawing lines, removing lines or neither. The lines will always be displayed on top of the patches and below the turtles. To change the color of the pen set the color of the turtle using `set color`.

Note: When a turtle's pen is down, all movement commands cause lines to be drawn, including `jump` and `setxy`.

Note: These commands are equivalent to setting the turtle variable "pen-mode" to "down", "up", and "erase".

Note: On Windows drawing and erasing a line might not erase every pixel.

pen-mode



This is a built-in turtle variable. It holds the state of the turtle's pen. You set the variable to draw lines, erase lines or stop either of these actions. Possible values are "up", "down", and "erase".

pen-size



This is a built-in turtle variable. It holds the width of the line, in pixels, that the turtle will draw (or erase) when the pen is down (or erasing).

plabel

plabel



This is a built-in patch variable. It may hold a value of any type. The patch appears in the view with the given value "attached" to it as text. You can set this variable to add, change, or remove a patch's label.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [plabel-color](#), [label](#), [label-color](#).

plabel-color

plabel-color



This is a built-in patch variable. It holds a number greater than or equal to 0 and less than 140. This number determines what color the patch's label appears in (if it has a label). You can set this variable to change the color of a patch's label.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [plabel](#), [label](#), [label-color](#).

plot

plot *number*

Increments the x-value of the plot pen by plot-pen-interval, then plots a point at the updated x-value and a y-value of *number*. (The first time the command is used on a plot, the point plotted has an x-value of 0.)

plot-name

plot-name

Reports the name of the current plot (a string).

plot-pen-down**ppd****plot-pen-up****ppu****plot-pen-down****plot-pen-up**

Puts down (or up) the current plot-pen, so that it draws (or doesn't). (By default, all pens are down initially.)

plot-pen-reset**plot-pen-reset**

Clears everything the current plot pen has drawn, moves it to (0,0), and puts it down. If the pen is a permanent pen, the color and mode are reset to the default values from the plot Edit dialog.

plotxy**plotxy** *number1 number2*

Moves the current plot pen to the point with coordinates (*number1*, *number2*). If the pen is down, a line, bar, or point will be drawn (depending on the pen's mode).

plot-x-min**plot-x-max****plot-y-min****plot-y-max****plot-x-min****plot-x-max****plot-y-min****plot-y-max**

Reports the minimum or maximum value on the x or y axis of the current plot.

These values can be set with the commands `set-plot-x-range` and `set-plot-y-range`. (Their default values are set from the plot Edit dialog.)

position**position** *item list***position** *string1 string2*

On a list, reports the first position of *item* in *list*, or false if it does not appear.

On strings, reports the position of the first appearance *string1* as a substring of *string2*, or false if it does not appear.

Note: The positions are numbered beginning with 0, not with 1.

```
;; suppose mylist is [2 7 4 7 "Bob"]
show position 7 mylist
=> 1
show position 10 mylist
=> false
show position "rin" "string"
=> 2
```

See also [member?](#).

precision

precision *number places*

Reports *number* rounded to *places* decimal places.

If *places* is negative, the rounding takes place to the left of the decimal point.

```
show precision 1.23456789 3
=> 1.235
show precision 3834 -3
=> 4000
```

print

print *value*

Prints *value* in the Command Center, followed by a carriage return.

The calling agent is *not* printed before the value, unlike [show](#).

See also [show](#), [type](#), and [write](#).

See also [output-print](#).

pxcor

pycor

pxcor

pycor



These are built-in patch variables. They hold the x and y coordinate of the patch. They are always integers. You cannot set these variables, because patches don't move.

pxcor is greater than or equal to min-pxcor and less than or equal to max-pxcor; similarly for pycor and min-pycor and max-pycor.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [xcor](#), [ycor](#).

R

random

random *number*

If *number* is positive, reports a random integer greater than or equal to 0, but strictly less than *number*.

If *number* is negative, reports a random integer less than or equal to 0, but strictly greater than *number*.

If *number* is zero, the result is always 0 as well.

Note: In versions of NetLogo prior to version 2.0, this primitive reported a floating point number if given a floating point 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 3.0
;; prints 0, 1, or 2
show random 3.5
;; prints 0, 1, 2, or 3
```

See also [random-float](#).

random-float

random-float *number*

If *number* is positive, reports a random floating point number greater than or equal to 0.0 but strictly less than *number*.

If *number* is negative, reports a random floating point number less than or equal to 0.0, but strictly greater than *number*.

If *number* is zero, the result is always 0.0.

```
show random-float 3
;; prints a number at least 0.0 but less than 3.0,
;; for example 2.589444906014774
show random-float 2.5
```



```
;; prints a number at least 0.0 but less than 2.5,
;; for example 1.0897423196760796
```

random-exponential

random-gamma

random-normal

random-poisson

random-exponential *mean*

random-gamma *alpha lambda*

random-normal *mean standard-deviation*

random-poisson *mean*

Reports an accordingly distributed random number with the *mean* and, in the case of the normal distribution, the *standard-deviation*.

random-exponential reports an exponentially distributed random floating point number.

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: $\alpha = \text{mean} * \text{mean} / \text{variance}$; $\lambda = 1 / (\text{variance} / \text{mean})$.)

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-int-or-float

random-int-or-float *number*

NOTE: This primitive should not be used in new models. It is included only for backwards compatibility with NetLogo 1.x. It will not necessarily continue to be supported in future versions of NetLogo.

When a NetLogo 1.x model is read into NetLogo 2.0 or higher, all uses of the "random" primitive are automatically converted to "random-int-or-float" instead, because the meaning of "random" has changed. It used to sometimes return an integer and sometimes a floating point number; now it always reports an integer. This primitive mimics the old behavior, as follows:

If *number* is positive, reports a random number greater than or equal to 0 but strictly less than *number*.

If *number* is negative, the number reported is less than or equal to 0, but strictly greater than *number*.

If *number* is zero, the result is always zero as well.

If *number* is an integer, reports a random integer.

If *number* is floating point (has a decimal point), reports a floating point number.

```
show random-int-or-float 3
;; prints 0, 1, or 2
show random-int-or-float 5.0
;; prints a number at least 0.0 but less than 5.0,
;; for example 4.686596634174661
```

random-pxcor random-pycor

random-pxcor random-pycor

Reports a random integer ranging from min-pxcor (or -y) to max-pxcor (or -y) inclusive.

```
ask turtles [
  ;; move each turtle to the center of a random patch
  setxy random-pxcor random-pycor
]
```

See also [random-xcor](#), [random-ycor](#).

random-seed

random-seed *number*

Sets the seed of the pseudo-random number generator to the integer part of *number*. The seed may be any integer in the range supported by NetLogo (−2147483648 to 2147483647).

See the [Random Numbers](#) section of the Programming Guide for more details.

```
random-seed 47823
show random 100
=> 57
show random 100
=> 91
random-seed 47823
show random 100
=> 57
show random 100
=> 91
```

random-xcor random-ycor

random-xcor random-ycor

Reports a random floating point number from the allowable range of turtle coordinates along the given axis, x or y.

Turtle coordinates range from min-pxcor – 0.5 (inclusive) to max-pxcor + 0.5 (exclusive) horizontally; vertically, substitute –y for –x.

```
ask turtles [
  ;; move each turtle to a random point
  setxy random-xcor random-ycor
]
```

See also [random-pxcor](#), [random-pycor](#).

read-from-string

read-from-string *string*

Interprets the given string as if it had been typed in the Command Center, and reports the resulting value. The result may be a number, list, string, or boolean value, or the special value "nobody".

Useful in conjunction with the [user-input](#) primitive for converting the user's input into usable form.

```
show read-from-string "3" + read-from-string "5"
=> 8
show length read-from-string "[1 2 3]"
=> 3
crt read-from-string user-input "Make how many turtles?"
;; the number of turtles input by the user
;; are created
```

reduce

reduce [*reporter*] *list*

Reduces a list from left to right using *reporter*, resulting in a single value. This means, for example, that `reduce [?1 + ?2] [1 2 3 4]` is equivalent to $((1 + 2) + 3) + 4$. If *list* has a single item, that item is reported. It is an error to reduce an empty list.

In *reporter*, use ?1 and ?2 to refer to the two objects being combined.

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] [1 2 3]
=> 6
```

```
show reduce [?1 - ?2] [1 2 3]
=> -4
show reduce [?2 - ?1] [1 2 3]
=> 2
show reduce [?1] [1 2 3]
=> 1
show reduce [?2] [1 2 3]
=> 3
show reduce [sentence ?1 ?2] [[1 2] [3 [4]] 5]
=> [1 2 3 [4] 5]
show reduce [fput ?2 ?1] (fput [] [1 2 3 4 5])
=> [5 4 3 2 1]
```

Here are some more useful examples:

```
;; find the longest string in a list
to-report longest-string [strings]
  report reduce
    [ifelse-value (length ?1 >= length ?2) [?1] [?2]]
    strings
end

show longest-string ["hi" "there" "!"]
=> "there"

;; count the number of occurrences of an item in a list
to-report occurrences [x xs]
  report reduce
    [ifelse-value (?2 = x) [?1 + 1] [?1]] (fput 0 xs)
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 eval-polynomial [coeffs x]
  report reduce [(x * ?1) + ?2] coeffs
end

;; evaluate 3x^2 + 2x + 1 at x = 4
show eval-polynomial [3 2 1] 4
=> 57
```

remainder

remainder *number1 number2*

Reports the remainder when *number1* is divided by *number2*. This is equivalent to the following NetLogo code:

```
number1 - (int (number1 / number2)) * number2

show remainder 62 5
=> 2
show remainder -8 3
=> -2
```

See also [mod](#). `mod` and `remainder` behave the same for positive numbers, but differently for

negative numbers.

remove

remove *item list*

remove *string1 string2*

For a list, reports a copy of *list* with all instances of *item* removed.

For strings, reports a copy of *string2* with all the appearances of *string1* as a substring removed.

```
set mylist [2 7 4 7 "Bob"]
set mylist remove 7 mylist
;; mylist is now [2 4 "Bob"]
show remove "na" "banana"
=> "ba"
```

remove-duplicates

remove-duplicates *list*

Reports a copy of *list* with all duplicate items removed. The first of each item remains in place.

```
set mylist [2 7 4 7 "Bob" 7]
set mylist remove-duplicates mylist
;; mylist is now [2 7 4 "Bob"]
```

remove-item

remove-item *index list*

remove-item *index string*

For a list, reports a copy of *list* with the item at the given index removed.

For strings, reports a copy of *string2* with the character at the given index removed.

Note that the indices begin from 0, not 1. (The first item is item 0, the second item is item 1, and so on.)

```
set mylist [2 7 4 7 "Bob"]
set mylist remove-item 2 mylist
;; mylist is now [2 7 7 "Bob"]
show remove-item 3 "banana"
=> "banna"
```

__remove-<breed>-from

__remove-<breed>-to

__remove-<breed>-with

__remove-<breed>-from *agent*
__remove-<breed>-to *agent*
__remove-<breed>-with *agent*
__remove-<breeds>-from *agentset*
__remove-<breeds>-to *agentset*
__remove-<breeds>-with *agentset*



Used for removing link turtles between node turtles.

__remove-<breed>-to removes the link turtle from the caller to *agent*. **__remove-<breed>-to** removes all agents in *agentset*. **__remove-<breed>-from** removes the link turtle from *agent* or all agents in *agentset* to the caller.

__remove-<breed>-with ignores direction and removes link turtles going in either direction.

__remove-<breed>-to and **__remove-<breed>-from** throw an error if the caller is not linked to *agent* or one or more agents in *agentset* in the direction that they try to remove in.

__remove-<breed>-with throws an error only if there are no links of links breed in either direction.

```
breed [links link] ;; breed for network links
```

```
to reduce-friends ;; node procedure
  __remove-link-to one-of __link-neighbors
end
to become-nasty ;; node procedure
  __remove-links-to __link-neighbors ;; breaks any friendships this node had
end
```

This command is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

repeat

repeat *number* [*commands*]

Runs *commands* *number* times.

```
pd repeat 36 [ fd 1 rt 10 ]
;; the turtle draws a circle
```

replace-item

replace-item *index list value*

replace-item *index string1 string2*

On a list, replaces an item in that list. *index* is the index of the item to be replaced, starting with 0. (The 6th item in a list would have an index of 5.) Note that "replace-item" is used in conjunction with "set" to change a list.

Likewise for a string, but the given character of *string1* removed and the contents of *string2* spliced in instead.

```
show replace-item 2 [2 7 4 5] 15
=> [2 7 15 5]
show replace-item 1 "sat" "lo"
=> "slot"
```

report

report *value*

Immediately exits from the current `to-report` procedure and reports *value* as the result of that procedure. `report` and `to-report` are always used in conjunction with each other. See [to-report](#) for a discussion of how to use them.

reset-perspective

rp

reset-perspective

The observer stops watching, following, or riding any turtles (or patches). (If it wasn't watching, following, or riding anybody, nothing happens.) In the 3D view, the observer also returns to its default position (above the origin, looking straight down).

See also [follow](#), [ride](#), [watch](#).

reset-timer

reset-timer

Resets the global clock to zero. See also [timer](#).

reverse

reverse *list*

reverse *string*

Reports a reversed copy of the given list or string.

```
show mylist
;; mylist is [2 7 4 "Bob"]
set mylist reverse mylist
;; mylist now is ["Bob" 4 7 2]
show reverse "string"
=> "gnirts"
```

rgb

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 0.0 to 1.0.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (See [hsb](#) for a description of what parts of the HSB color space NetLogo colors cover; this is difficult to characterize in RGB terms.)

```
show rgb 0 0 0
=> 0.0 ;; black
show rgb 0 1.0 1.0
=> 85.0 ;; cyan
```

See also [extract-rgb](#), [hsb](#), and [extract-hsb](#).

ride

ride *turtle*



Set the perspective to *turtle*.

Every time *turtle* moves the observer also moves. Thus, in the 2D View the turtle will stay at the center of the view. In the 3D view it is as if looking through the eyes of the turtle. If the turtle dies, the view will return to the default position.

See also [reset-perspective](#), [watch](#), [follow](#), [subject](#).

ride-me

ride-me



Asks the observer to ride the calling turtle.

See also [ride](#).

right

rt

right *number*



The turtle turns right by *number* degrees. (If *number* is negative, it turns left.)

round

round *number*

Reports the integer nearest to *number*.

If the decimal portion of *number* is exactly .5, the number is rounded in the **positive** direction.

Note that rounding in the positive direction is not always how rounding is done in other software programs. (In particular, it does not match the behavior of StarLogoT, which always rounded numbers ending in 0.5 to the nearest even integer.) The rationale for this behavior is that it matches how turtle coordinates relate to patch coordinates in NetLogo. For example, if a turtle's xcor is -4.5, then it is on the boundary between a patch whose pxcor is -4 and a patch whose pxcor is -5, but the turtle must be considered to be in one patch or the other, so the turtle is considered to be in the patch whose pxcor is -4, because we round towards the positive numbers.

```
show round 4.2
=> 4
show round 4.5
=> 5
show round -4.5
=> -4
```

run

run *string*

This agent interprets the given string as a sequence of one or more NetLogo commands and runs them.

The code runs in the agent's current context, which means it has access to the values of local variables, "myself", and so on.

See also [runresult](#).

runresult

runresult *string*

This agent interprets the given string as a NetLogo reporter and runs it, reporting the result obtained.

The code runs in the agent's current context, which means it has access to the values of local variables, "myself", and so on.

See also [run](#).

S

scale-color

scale-color *color number range1 range2*

Reports a shade of *color* proportional to *number*.

If *range1* is less than *range2*, then the larger the number, the lighter the shade of *color*. But if *range2* is less than *range1*, the color scaling is inverted.

If *number* is less than *range1*, then the darkest shade of *color* is chosen.

If *number* is greater than *range2*, then the lightest shade of *color* is chosen.

Note: for *color* shade is irrelevant, e.g. green and green + 2 are equivalent, and the same spectrum of colors will be used.

```
ask turtles [ set color scale-color red age 0 50 ]
;; colors each turtle a shade of red proportional
;; to its value for the age variable
```

self

self



Reports this turtle or patch.

"self" and "myself" are very different. "self" is simple; it means "me". "myself" means "the turtle or patch who asked me to do what I'm doing right now."

```
ask turtles with [self != myself]
[ die ]
;; this turtle kills all other turtles
```

See also [myself](#).

; (semicolon)

; *comments*

After a semicolon, the rest of the line is ignored. This is useful for adding "comments" to your code — text that explains the code to human readers. Extra semicolons can be added for visual effect.

NetLogo's Edit menu has items that let you comment or uncomment whole sections of code.

sentence

se

sentence *value1 value2*
(sentence *value1 ... valuen***)**

Makes a list out of the values. If any value is a list, its items are included in the result directly, rather than being included as a sublist. Examples make this clearer:

```
show sentence 1 2
=> [1 2]
show sentence [1 2] 3
```

```
=> [1 2 3]
show sentence 1 [2 3]
=> [1 2 3]
show sentence [1 2] [3 4]
=> [1 2 3 4]
show sentence [[1 2]] [[3 4]]
=> [[1 2] [3 4]]
show (sentence [1 2] 3 [4 5] (3 + 3) 7)
=> [1 2 3 4 5 6 7]
```

set

set *variable value*

Sets *variable* to the given value.

Variable can be any of the following:

- An global variable declared using "globals"
 - The global variable associated with a slider, switch, or chooser
 - A variable belonging to the calling agent
 - If the calling agent is a turtle, a variable belonging to the patch under the turtle.
 - An expression of the form *VARIABLE-of agent*
 - A local variable created by the let command
-

set-current-directory

set-current-directory *string*

Sets the current directory that is used by the primitives file-delete, file-exists?, and file-open.

The current directory is not used if the above commands are given an absolute file path. This is defaulted to the user's home directory for new models, and is changed to the model's directory when a model is opened.

Note that in Windows file paths the backslash needs to be escaped within a string by using another backslash "C:\\"

The change is temporary and is not saved with the model.

Note: in applets, this command has no effect, since applets are only allowed to read files from the same directory on the server where the model is stored.

```
set-current-directory "C:\\NetLogo"
;; Assume it is a Windows Machine
file-open "myfile.txt"
;; Opens file "C:\\NetLogo\\myfile.txt"
```

set-current-plot

set-current-plot *plotname*

Sets the current plot to the plot with the given name (a string). Subsequent plotting commands will affect the current plot.

set-current-plot-pen

set-current-plot-pen *penname*

The current plot's current pen is set to the pen named *penname* (a string). If no such pen exists in the current plot, a runtime error occurs.

set-default-shape

set-default-shape turtles *string*

set-default-shape *breed* *string*



Specifies a default initial shape for all turtles, or for a particular breed. When a turtle is created, or it changes breeds, its shape is set to the given shape.

This command doesn't affect existing turtles, only turtles you create afterwards.

The specified breed must be either turtles or a breed defined by the breed keyword, and the specified 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 individual turtle's shape later; turtles don't have to be stuck with their breed's default shape.

```
create-turtles 1 ;; new turtle's shape is "default"
create-cats 1    ;; new turtle's shape is "default"
```

```
set-default-shape turtles "circle"
create-turtles 1 ;; new turtle's shape is "circle"
create-cats 1    ;; new turtle's shape is "circle"
```

```
set-default-shape cats "cat"
set-default-shape dogs "dog"
create-cats 1 ;; new turtle's shape is "cat"
ask cats [ set breed dogs ]
  ;; all cats become dogs, and automatically
  ;; change their shape to "dog"
```

See also shape.

set-histogram-num-bars

set-histogram-num-bars *integer*

Set the current plot pen's plot interval so that, given the current x range for the plot, there would be *integer* number of bars drawn if the histogram-from or histogram-list commands were called.

See also [histogram-from](#).

__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 reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details. (Note that the command can be used with non-link turtles too, though.)

set-plot-pen-color

set-plot-pen-color *number*

Sets the color of the current plot pen to *number*.

set-plot-pen-interval

set-plot-pen-interval *number*

Tells the current plot pen to move a distance of *number* in the x direction during each use of the plot command. (The plot pen interval also affects the behavior of the histogram-from and histogram-list commands.)

set-plot-pen-mode

set-plot-pen-mode *number*

Sets the mode the current plot pen draws in to *number*. The allowed plot pen modes are:

- 0 (line mode) the plot pen draws a line connecting two points together.

- 1 (bar mode): the plot pen draws a bar of width `plot-pen-interval` with the point plotted as the upper (or lower, if you are plotting a negative number) left corner of the bar.
- 2 (point mode): the plot pen draws a point at the point plotted. Points are not connected.

The default mode for new pens is 0 (line mode).

set-plot-x-range

set-plot-y-range

set-plot-x-range *min max*

set-plot-y-range *min max*

Sets the minimum and maximum values of the x or y axis of the current plot.

The change is temporary and is not saved with the model. When the plot is cleared, the ranges will revert to their default values as set in the plot's Edit dialog.

setxy

setxy *x y*



The turtle sets its x-coordinate to *x* and its y-coordinate to *y*.

Equivalent to `set xcor x set ycor y`, except it happens in one time step instead of two.

If *x* or *y* is outside the world, NetLogo will throw a runtime error.

```
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
```

shade-of?

shade-of? *color1 color2*

Reports true if both colors are shades of one another, false otherwise.

```
show shade-of? blue red
=> false
show shade-of? blue (blue + 1)
=> true
show shade-of? gray white
=> true
```

shape

shape



This is a built-in turtle variable. It holds a string that is the name of the turtle's current shape. You can set this variable to change a turtle's shape. New turtles 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 Shapes Editor
```

See also [set-default-shape](#), [shapes](#).

shapes

shapes

Reports a list of strings containing all of the turtle shapes in the model.

New shapes can be created, or imported from the shapes library or from other models, in the [Shapes Editor](#).

```
show shapes
=> ["default" "airplane" "arrow" "box" "bug" ...
ask turtles [ set shape one-of shapes ]
```

show

show *value*

Prints *value* in the Command Center, preceded by the calling agent, and followed by a carriage return. (The calling agent is included to help you keep track of what agents are producing which lines of output.) Also, all strings have their quotes included similar to [write](#).

See also [print](#), [type](#), and [write](#).

See also [output-show](#).

show-turtle

st

showturtle



The turtle becomes visible again.

Note: This command is equivalent to setting the turtle variable "hidden?" to false.

See also [hideturtle](#).

shuffle

shuffle *list*

Reports a new list containing the same items as the input list, but in randomized order.

```
show shuffle [1 2 3 4 5]
=> [5 2 4 1 3]
show shuffle [1 2 3 4 5]
=> [1 3 5 2 4]
```

sin

sin *number*

Reports the sine of the given angle. Assumes angle is given in degrees.

```
show sin 270
=> -1.0
```

size

size



This is a built-in turtle variable. It holds a number that is the turtle's apparent size. The default size is 1.0, which means that the turtle is the same size as a patch. You can set this variable to change a turtle's size.

sort

sort *list*

sort *agentset*

If the input is a list, reports a list containing the same items as the input list, but in ascending order. If there is at least one number in the list, the list is sorted in numerically ascending order and any non-numeric items of the input list are discarded. If there are no numbers, but at least one string in the list, the list is sorted in alphabetically ascending order and any non-string items are discarded.

If the input is an agentset or a list of agents, reports a list (never an agentset) of agents. If the agents are turtles, they are listed in ascending order by who number. If the agents are patches, they are listed left-to-right, top-to-bottom.

```
show sort [3 1 4 2]
=> [1 2 3 4]
let n 0
foreach sort patches [
```



```
ask ? [
  set plabel n
  set n n + 1
]
]
;; patches are labeled with numbers in left-to-right,
;; top-to-bottom order
```

sort-by

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 (true or false) *reporter*.

In *reporter*, use ?1 and ?2 to refer to the two objects being compared. *reporter* should be true if ?1 comes strictly before ?2 in the desired sort order, and false otherwise.

If the input is an agentset or a list of agents, reports a list (never an agentset) of agents.

```
show sort-by [?1 < ?2] [3 1 4 2]
=> [1 2 3 4]
show sort-by [?1 > ?2] [3 1 4 2]
=> [4 3 2 1]
show sort-by [length ?1 < length ?2] ["zzz" "z" "zz"]
=> ["z" "zz" "zzz"]
foreach sort-by [size-of ?1 < size-of ?2] turtles
  [ ask ? [ do-something ] ]
;; turtles run "do-something" one at a time, in
;; ascending order by size
```

sprout

sprout-*<breeds>*

sprout *number* [*commands*]

sprout-*<breeds>* *number* [*commands*]



Creates *number* new turtles on the current patch. The new turtles have random colors and orientations, and they immediately run *commands*. This is useful for giving the new turtles different colors, headings, or whatever. (The new turtles are created all at once then run one at a time, in random order.)

If the **sprout-*<breeds>*** form is used, the new turtles are created as members of the given breed.

```
sprout 1 [ set color red ]
sprout-sheep 1 [ set color black ]
```

Note: While the commands are running, no other agents are allowed to run any code (as with the without-interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no display updates take place until the commands are done. This ensures that the new turtles are never drawn in the view until they are fully initialized.

sqrt

sqrt *number*

Reports the square root of *number*.

stamp

stamp



The calling turtle leaves an image of its shape in the drawing at its current location.

Note: The shapes made by stamp may not be pixel-for-pixel identical from computer to computer.

stamp-erase

stamp-erase



The calling turtle removes any pixels below it in the drawing inside the bounds of its shape.

Note: The shapes made by stamp-erase may not be pixel-for-pixel identical from computer to computer.

standard-deviation

standard-deviation *list*

Reports the unbiased statistical standard deviation of a *list* of numbers. Ignores other types of items.

```
show standard-deviation [1 2 3 4 5 6]
=> 1.8708286933869707
show standard-deviation values-from turtles [energy]
;; prints the standard deviation of the variable "energy"
;; from all the turtles
```

startup

startup



User-defined procedure which, if it exists, will be called when a model is first loaded.

```
to startup
  setup
end
```

stop

stop

The calling agent exits immediately from the enclosing procedure, ask, or ask-like construct (cct, hatch, sprout). Only the current procedure stops, not all execution for the agent.

Note: stop can be used to stop a forever button. 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.)

subject

subject

Reports the turtle (or patch) that the observer is currently watching, following, or riding. Reports nobody if there is no such turtle (or patch).

See also watch, follow, ride.

sublist

substring

sublist *list position1 position2*
substring *string position1 position2*

Reports just a section of the given list or string, ranging between the first position (inclusive) and the second position (exclusive).

Note: The positions are numbered beginning with 0, not with 1.

```
show sublist [99 88 77 66] 1 3
=> [88 77]
show substring "turtle" 1 4
=> "urt"
```

subtract-headings

subtract-headings heading1 heading2

Computes the difference between the given headings, that is, the number of degrees in the smallest angle by which heading2 could be rotated to produce heading1. A positive answer means a clockwise rotation, a negative answer counterclockwise. The result is always in the range -180 to 180, but is never exactly -180.

Note that simply subtracting the two headings using the - (minus) operator wouldn't work. Just subtracting corresponds to always rotating clockwise from heading2 to heading1; but sometimes the counterclockwise rotation is shorter. For example, the difference between 5 degrees and 355 degrees is 10 degrees, not -350 degrees.

```
show subtract-headings 80 60
=> 20
show subtract-headings 60 80
=> -20
show subtract-headings 5 355
=> 10
show subtract-headings 355 5
=> -10
show subtract-headings 180 0
=> 180
show subtract-headings 0 180
=> 180
```

sum

sum *list*

Reports the sum of the items in the list.

```
show sum values-from turtles [energy]
;; prints the total of the variable "energy"
;; from all the turtles
```

T

tan

tan *number*

Reports the tangent of the given angle. Assumes the angle is given in degrees.

tie

tie *leaf-turtle root-turtle*

Connects *leaf-turtle* to *root-turtle* so that the movement of the *root-turtle* affects the location and heading of the *leaf-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.

See also untie

This command is part of the experimental Tie support in NetLogo. See the Tie section of the Programming Guide for more details.

timer

timer

Reports how many seconds have passed since the command `reset-timer` was last run (or since NetLogo started). The potential resolution of the clock is milliseconds. (Whether you get resolution that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.)

to

to *procedure-name*

to *procedure-name* [*input1 input2 ...*]

Used to begin a command procedure.

```
to setup
  ca
  crt 500
end

to circle [radius]
  cct 100 [ fd radius ]
end
```

to-report

to-report *procedure-name*

to-report *procedure-name* [*input1 input2 ...*]

Used to begin a reporter procedure.

The body of the procedure should use `report` to report a value for the procedure. See [report](#).

```
to-report average [a b]
  report (a + b) / 2
end

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

to-report first-turtle?
  report who = 0 ;; reports true or false
end
```

towards

towards agent

Reports the heading from this agent to the given agent.

If wrapping is allowed by the topology and the wrapped distance (around the edges of the view) is shorter than the on-screen distance, towards will report the heading of 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.

towardsxy**towardsxy x y**

Reports the heading from the turtle or patch towards the point (x,y).

If wrapping is allowed by the topology and the wrapped distance (around the edges of the view) is shorter than the on-screen distance, towardsxy will report the heading of the wrapped path.

Note: asking for the heading to the point the agent is already standing on will cause a runtime error.

turtle**turtle *number***

Reports the turtle with the given ID number, or nobody if there is no such turtle. *number* must be an integer.

```
set color-of turtle 5 red
;; turtle with id number 5 turns red
ask turtle 5 [ set color red ]
;; another way to do the same thing
```

turtles**turtles**

Reports the agentset consisting of all turtles.

```
show count turtles
;; prints the number of turtles
```

turtles-at**<breeds>-at**

turtles-at *dx dy* **<breeds>-at *dx dy***

Reports an agentset containing the turtles on the patch (*dx*, *dy*) from the caller (including the caller itself if it's a turtle). If the caller is the observer, *dx* and *dy* are calculated from the origin (0,0).

```
;; suppose I have 40 turtles at the origin
show count turtles-at 0 0
=> 40
```

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-custom-dogs 5 [ setxy 2 3 ]
show count dogs-at 2 3
=> 5
```

turtles-from

turtles-from *agentset* [*reporter*]

Reports a turtle agentset made by gathering together all the turtles reported by *reporter* for each agent in *agentset*.

For each agent, the reporter must report a turtle agentset, a single turtle, or nobody.

```
turtles-from patches [one-of turtles-here]
;; reports a turtle set containing one turtle from
;; each patch (that has any turtles on it)
turtles-from neighbors [turtles-here]
;; if run by a turtle or patch, reports the set of
;; all turtles on the neighboring eight patches; note that
;; this could be written more concisely using turtles-on,
;; like this:
;;   turtles-on neighbors
```

See also [patches-from](#), [turtles-on](#).

turtles-here **<breed>-here**

turtles-here **<breeds>-here**

Reports an agentset containing all the turtles on the caller's patch (including the caller itself if it's a turtle).

```
ca
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
```

See also [other-turtles-here](#).

turtles-on

<breeds>-on

turtles-on agent

turtles-on agentset

<breeds>-on agent

<breeds>-on agentset

Reports an agentset containing all the turtles that are on the given patch or patches, or standing on the same patch as the given turtle or turtles.

```
ask turtles [
  if not any? turtles-on patch-ahead 1
    [ fd 1 ]
]
ask turtles [
  if not any? turtles-on neighbors [
    die-of-loneliness
  ]
]
```

If the name of a breed is substituted for "turtles", then only turtles of that breed are included.

See also [turtles-from](#).

turtles-own

<breeds>-own

turtles-own [var1 var2 ...]

<breeds>-own [var1 var2 ...]

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 breed may list the same variable.)

```
breed [cats cat ]
breed [dogs dog]
breed [hamsters hamster]
turtles-own [eyes legs] ;; applies to all breeds
```



```
cats-own [fur kittens]
hamsters-own [fur cage]
dogs-own [hair puppies]
```

See also [globals](#), [patches-own](#), [breed](#), [<breeds>-own](#).

type

type *value*

Prints *value* in the Command Center, *not* followed by a carriage return (unlike [print](#) and [show](#)). The lack of a carriage return allows you to print several values on the same line.

The calling agent is *not* printed before the value. unlike [show](#).

```
type 3 type " " print 4
=> 3 4
```

See also [print](#), [show](#), and [write](#).

See also [output-type](#).

U

[__](#) **untie**

untie *leaf-turtle root-turtle*

Disconnects *leaf-turtle* from *root-turtle* if they were previously tied together.

See also [__tie](#)

This command is part of the experimental Tie support in NetLogo. See the [Tie](#) section of the Programming Guide for more details.

uphill

uphill *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) in the direction of the maximum value of the variable *patch-variable*, of the patches in a one-patch radius of the turtle. (This could be as many as eight or as few as five patches, depending on the position of the turtle within its patch.)

If there are multiple patches that have the same greatest value, a random one of those patches will be selected.

If the patch is located directly to the north, south, east, or west of the patch that the turtle is currently on, a multiple of 90 degrees is reported. However, if the patch is located to the northeast, northwest, southeast, or southwest of the patch that the turtle is currently on, the direction the turtle would need

to reach the nearest corner of that patch is reported.

See also [uphill4](#), [downhill](#), [downhill4](#).

uphill4

uphill4 *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) as a multiple of 90 degrees in the direction of the maximum value of the variable *patch-variable*, of the four patches to the north, south, east, and west of the turtle. If there are multiple patches that have the same greatest value, a random patch from those patches will be selected.

See also [uphill](#), [downhill](#), [downhill4](#).

user-directory

user-directory

Opens a dialog that allows the user to choose an existing directory on the system.

It reports a string with the absolute path or false if the user cancels.

```
set-current-directory user-directory
;; Assumes the user will choose a directory
```

user-file

user-file

Opens a dialog that allows the user to choose an existing file on the system.

It reports a string with the absolute file path or false if the user cancels.

```
file-open user-file
;; Assumes the user will choose a file
```

user-new-file

user-new-file

Opens a dialog that allows the user to choose a location and name of a new file to be created.

It reports a string with the absolute file path or false if the user cancels.

```
file-open user-new-file
;; Assumes the user will choose a file
```

Note that this reporter doesn't actually create the file; normally you would create the file using `file-open`, as in the example.

user-input

user-input *value*

Reports the string that a user types into an entry field in a dialog with title *value*.

value may be of any type, but is typically a string.

```
show user-input "What is your name?"
```

user-message

user-message *value*

Opens a dialog with *value* displayed as the message.

value may be of any type, but is typically a string.

```
user-message "There are " + count turtles + " turtles."
```

user-one-of

user-one-of *value list-of-choices*

Opens a dialog with *value* displayed as the message and *list-of-choices* displayed as a popup menu for the user to select from.

Reports the item in *list-of-choices* selected by the user.

value may be of any type, but is typically a string.

```
if "yes" = user-one-of? "Set up the model?" [ "yes" "no" ]
  [ setup ]
```

user-yes-or-no?

user-yes-or-no? *value*

Reports true or false based on the user's response to *value*.

value may be of any type, but is typically a string.

```
if user-yes-or-no? "Set up the model?"
  [ setup ]
```

V

value-from

value-from *agent* [*reporter*]

Reports the value of the reporter for the given agent (turtle or patch).

```
show value-from (turtle 5) [who * who]
=> 25
show value-from (patch 0 0) [count turtles in-radius 3]
;; prints the number of turtles located within a
;; three-patch radius of the origin
```

values-from

values-from *agentset* [*reporter*]

Reports a list that contains the value of the reporter for each agent in the agentset.

```
ca
crt 4
show values-from turtles [who]
=> [0 1 2 3]
show values-from turtles [who * who]
=> [0 1 4 9]
```

variance

variance *list*

Reports the sample variance of a *list* of numbers. Ignores other types of items.

The sample variance is the sum of the squares of the deviations of the numbers from their mean, divided by one less than the number of numbers in the list.

```
show variance [2 7 4 3 5]
=> 3.7
```

W

wait

wait *number*

Wait the given number of seconds. (You can use floating-point numbers to 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 ]
```

See also [every](#).

watch

watch *agent*



Puts a spotlight on *agent*. In the 3D view the observer will also turn to face the subject.

See also [follow](#), [subject](#), [reset-perspective](#), [watch-me](#).

watch-me

watch-me



Asks the observer to watch the calling agent.

See also [watch](#).

while

while [*reporter*] [*commands*]

If *reporter* reports false, exit the loop. Otherwise run *commands* and repeat.

The reporter may have different values for different agents, so some agents may run *commands* a different number of times than other agents.

```
while [any? other-turtles-here]
  [ fd 1 ]
;; turtle moves until it finds a patch that has
;; no other turtles on it
```

who

who



This is a built-in turtle variable. It holds the turtle's id number (an integer greater than or equal to zero). You cannot set this variable; a turtle's id number never changes.

When NetLogo starts, or after you use the [clear-all](#) or [clear-turtles](#) commands, new turtles are created with ids in order, starting at 0. If a turtle dies, though, a new turtle may eventually be assigned the same id number that was used by the dead turtle.

Example:

```
show values-from (turtles with [color = red]) [who]
;; prints a list of the id numbers of all red turtles
```

```
;; in the Command Center
ca
cct 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 id number. See also [turtle](#).

with

agentset with [reporter]

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a boolean reporter. Reports a new agentset containing only those agents that reported true -- in other words, the agents satisfying the given condition.

```
show count patches with [pcolor = red]
;; prints the number of red patches
```

__<breed>-with

__<breed>-with *agent*



Report the link turtle from the *agent* to the caller, or from the caller to the *agent*. If no link exists then it reports nobody.

```
to find-friendship-with
  ca
  cct 2 []
  ask turtle 0
  [
    __create-link-to turtle 1 []
    __create-link-from turtle 1 []
    show __in-link-with turtle 1 ;; prints turtle 3
    show __out-link-with turtle 1 ;; prints turtle 2
    show __link-with turtle 1    ;; prints turtle 2
  ]
end
```

This reporter is part of the experimental Links support in NetLogo. See the [Links](#) section of the Programming Guide for more details.

with-max

agentset with-max [reporter]

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a reporter. Reports a new agentset containing all agents reporting the maximum value of the given reporter.

```
show count patches with-max [pxcor]
;; prints the number of patches on the right edge
```

See also [max-one-of](#)

with-min

agentset with-min [reporter]

Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a reporter. Reports a new agentset containing only those agents that have the minimum value of the given reporter.

```
show count patches with-min [pycor]
;; prints the number of patches on the bottom edge
```

See also [min-one-of](#)

without-interruption

without-interruption [commands]

The agent runs all the commands in the block without allowing other agents to "interrupt". That is, other agents are put "on hold" and do not run any commands until the commands in the block are finished.

```
crt 5
ask turtles
  [ without-interruption
    [ type 1 fd 1 type 2 ] ]
=> 1212121212
;; because each turtle will output 1 and move,
;; then output 2. however:
ask turtles
  [ type 1 fd 1 type 2 ]
=> 1111122222
;; because each turtle will output 1 and move,
;; then output 2
```

word

word value1 value2

(word value1 ... valuen)

Concatenates the inputs together and reports the result as a string.

```
show word "tur" "tle"
=> "turtle"
word "a" 6
=> "a6"
set directory "c:\\foo\\fish\\"
show word directory "bar.txt"
=> "c:\\foo\\fish\\bar.txt"
show word [1 54 8] "fishy"
```

```
=> "[1 54 8]fishy"
show (word "a" "b" "c" 1 23)
=> "abc123"
```

world-width

world-height

world-width

world-height

These reporters give the total width and height of the NetLogo world.

The width and height of the world is the same as $\text{max-p(x/y)cor} - \text{min-p(x/y)cor} + 1$.

See also [max-pxcor](#), [max-pycor](#), [min-pxcor](#), and [min-pycor](#)

wrap-color

wrap-color *number*

wrap-color checks whether *number* is in the NetLogo color range of 0 to 140 (not including 140 itself). If it is not, wrap-color "wraps" the numeric input to the 0 to 140 range.

The wrapping is done by repeatedly adding or subtracting 140 from the given number until it is in the 0 to 140 range. (This is the same wrapping that is done automatically if you assign an out-of-range number to the color turtle variable or pcolor patch variable.)

```
show wrap-color 150
=> 10
show wrap-color -10
=> 130
```

write

write *value*

This command will output *value*, which can be a number, string, list, boolean, or nobody to the Command Center *not* followed by a carriage return (unlike [print](#) and [show](#)).

The calling agent is *not* printed before the value, unlike [show](#). Its output will also includes quotes around strings and is prepended with a space.

```
write "hello world"
=> "hello world"
```

See also [print](#), [show](#), and [type](#).

See also [output-write](#).

X

xcor

xcor



This is a built-in turtle variable. It holds the current x coordinate of the turtle. 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 (min-pxcor - 0.5) and strictly less than (max-pxcor + 0.5).

See also [setxy](#), [ycor](#), [pxcor](#), [pycor](#).

xor

boolean1 xor boolean2

Reports true if either *boolean1* or *boolean2* is true, but not when both are true.

```
if (pxcor > 0) xor (pycor > 0)
  [ set pcolor blue ]
;; upper-left and lower-right quadrants turn blue
```

Y

ycor

ycor



This is a built-in turtle variable. It holds the current y coordinate of the turtle. 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 (min-pycor - 0.5) and strictly less than (max-pycor + 0.5).

See also [setxy](#), [xcor](#), [pxcor](#), [pycor](#).

?

?

?, ?1, ?2, ...

These are special local variables. They hold the current inputs to a reporter or command block for certain primitives (for example, the current item of a list being visited by [foreach](#) or [map](#)).

? is always equivalent to ?1.

You may not set these variables, and you may not use them except with certain primitives, currently foreach, map, reduce, filter, sort-by, and n-values. See those entries for example usage.