NetLogo 2.0.1 User Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

What is NetLogo?

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It 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 MacOS, Windows, Linux, et al
  - ♦ Models can be saved as applets to be embedded in web pages
- Language:
  - ♦ Fully programmable
  - ♦ Simple language structure
  - ♦ Language is Logo dialect extended to support agents and concurrency
  - ♦ Unlimited numbers of agents and variables
  - ♦ Many built−in primitives
  - ♦ Integer and double precision floating point math
  - ♦ Runs are exactly reproducible cross−platform
- Environment:
  - ♦ Graphics display supports turtle shapes and sizes, exact turtle positions, and turtle and patch labels
  - ♦ Interface builder w/ buttons, sliders, switches, choices, monitors, text boxes

- "Control strip" including speed slider
- Powerful and flexible plotting system
- Info area for annotating your model
- HubNet: participatory simulations using networked devices
- Agent monitors for inspecting and controlling agents
- BehaviorSpace tool used to collect data from multiple runs of a model
- Export and import functions (export data, save and restore state of model)

Copyright Information

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.

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 2.0.1 (May 7, 2004)

- system:
  - our bundled Java VM on Windows is now Sun 1.4.2_04 (was 1.4.2_02)
- features:
  - new, experimental "extensions" API lets users write new commands and reporters in Java; see new "Extensions" section of User Manual
  - previously unpubished "controlling" API lets users control NetLogo from external Java code (such as for automating multiple runs); see new "Controlling" section of User Manual
- content:
  - new social science models: Scatter, El Farol, Minority Game
  - new earth science model: Erosion
  - new code examples: Hex Cells Example, Hex Turtles Example, Patch Clusters Example
  - improved models: Koch Curve (revamped), Binomial Rabbits (bugfix), Ant Lines (bugfix), Shepherds (better code), Prob Graphs Basic (simplified interface), Random Basic (usability improvements), Random Balls (bugfix)
  - minor updates to the User Manual
- engine fixes:
  - fixed bug where recompiling the model undid the effects of `set-default-shape`
  - fixed the `file-read` command so it skips trailing whitespace, which makes it more convenient to use in conjunction with `file-at-end?`
  - fixed bug where using the mouse primitives could freeze NetLogo under certain unusual circumstances
  - code of the form `<breed> in-radius ...` now runs much faster
- interface fixes:
  - fixed bug where turtle monitors were sometimes blank
  - fixed bug where unfreezing the display from the graphics control strip sometimes caused a graphics update even when the `no-display` command was in effect
  - fixed bug where saving the model would close files opened by `file-open`
  - fixed shapes editor bug where changes to shapes weren't always immediately visible in the graphics window
  - fixed shapes editor bug where renaming a shape would cause the shape to be duplicated
  - fixed Windows-only bug where code sometimes disappeared from buttons containing multiple lines of code
- computer HubNet changes:
  - improved activities: Tragedy of the Commons (bugfix), Gridlock Alternate (bugfix, improvements), Sampler (bugfix)
  - server discovery is now off by default in the client, since we suspected it of causing problems on some configurations
  - fixed bug where the list of servers contained duplicates

- fixed assorted bugs with plots displaying incorrectly
- fixed bugs that caused the clients' graphics to sometimes be out of sync with the server

Version 2.0.0 (December 10, 2003)

- system:
  - NetLogo now requires Java 1.4.1 or higher; on non−Macintosh systems, 1.4.2 is preferred
  - NetLogo now fully supports Mac OS X (not beta anymore)
  - Mac OS X 10.2 users are strongly encouraged to get Java 1.4.1 Update 1 through Software Update
  - NetLogo no longer supports Windows 95, MacOS 8, or MacOS 9 (however, we will continue to support NetLogo 1.3.x, which works on those systems)
  - our user interface now uses Java's Swing toolkit (and not the older AWT toolkit)
  - increased overall reliability
  - NetLogo now functions more smoothly in many respects on Linux (and probably other Unix systems)
  - our recommended Java VM on Windows is now Sun 1.4.2 (instead of IBM 1.1.8); we offer an installer that includes this VM
- content:
  - new biology models: Fur, Sunflower
  - new physics model: Turbulence
  - new chemistry model: B−Z Reaction
  - new computer science models: CA Continuous, CA Stochastic
  - new social science model: Gridlock (non−HubNet version)
  - new ProbLab model: Random Basic
  - new code examples One Turtle Per Patch Example, Look Ahead Example, Grouping Turtles Example, User Interaction Example, Neighborhoods Example, Partners Example, Random Seed Example
  - models improved: Plant Growth (bugfix), Fireflies (interface changes), Decay (made flashing optional), Fire (new turtle−based version is much faster), Percolation (much faster), Ants (bugfix), Rugby (bugfix, improved code), Segregation (bugfix, improved code)
  - models improved and promoted out of "unverified": Sand (bugfix), Wandering Letters (overhauled), Lattice Gas Automaton, Dining Philosophers, Turing Machine 2D, Division, CA Continuous, CA Stochastic, Vector Fields, Random Basic, Prob Graphs Basic, Stochastic Patchwork
  - new sections in Programming Guide on "Random Numbers", "Buttons", and "Math"
  - restored missing punctuation marks in PDF version of User Manual
- features:
  - the "Exact Turtle Positions" and "Turtle Sizes" settings have been merged into a single option, and they are now much faster, more reliable, and flicker−free; so much so, in fact, that they are now on by default; turtle and patch labels are much faster now too
  - a full of suite of primitives for reading and writing external files is now included; see the File I/O section of the Programming Guide in the User Manual for details
  - "strict math" mode is now always on, so model results are reproducible cross−platform

- new "Export Graphics" and "Export Interface" menu items and `export-graphics` and `export-interface` primitives let you save the contents of the graphics window, or the whole interface tab, to disk as a bitmap (in PNG format), also enabling the creation of movies of model runs
- the old BehaviorSpace tool has been scrapped; a replacement version, still under development, is included; it does not yet have all the functionality of the old tool, but is already useful, and already has the following advantages over the old version:
  - you can vary any global variables, not just sliders, but also switches and choices, and any variable declared in the procedures tab, and the values they range over can be any values, not just numbers anymore
  - you can enter arbitrary code for "setup" and "go" now; you're not tied to using buttons that exist in your model
  - you can vary variables in any order you want, rather than BehaviorSpace choosing the order for you
  - you can collect any result type you want, not just numbers anymore
- added new "Update screen each iteration" checkbox for forever buttons, for controlling whether a screen update is forced each time through the button
- new primitive `set-current-directory` lets you set the folder that file operations take place in

- user interface improvements:
  - improved look and feel throughout the application
  - mouse scroll wheels now work
  - models library dialog can be navigated with the keyboard
  - dialog boxes support cut, copy, and paste (using keyboard shortcuts)
  - the command center and the Procedures and Errors tabs now use the exact same syntax highlighting editor, resolving many small issues and inconsistencies in behavior
  - added a "Go to User Community Models web page" button to the Models Library dialog
  - fixed longstanding Windows bug where the output area of the command center could only hold 25K; there is now no known limit
  - the graphics window and plots are flicker-free now
  - plots containing very large numbers of points are faster now
  - you can now press the Enter or Return key to close dialog boxes
  - when turtle shapes are off, if the patch size is at least 5 then turtles are now drawn slightly smaller than the patches, so if a turtle is in the center of its patch, you can still see the patch color around its edges
  - agent monitors now report syntax errors to you instead of silently ignoring them
  - graphics control strip icons have tooltips now
  - improved the automatic sizing and positioning of the main NetLogo window so that available screen real estate is more fully utilized
  - changed the keyboard shortcuts for zooming, indenting, and commenting to more standard choices
  - fixed bug where if you made changes to your code in the Errors tab and then saved your model without switching tabs first, your changes wouldn't be saved
- language changes:
  - changes in random number generation:
    - NetLogo now uses the research-grade Mersenne Twister algorithm to generate all random events
    - added new `random-gamma` primitive for generating gamma-distributed random numbers

  - ◊ added new `random-float` primitive that always returns a floating point number
  - ◊ changed `random` primitive so it always returns an integer, not a floating point number
  - ◊ added new `random-int-or-float` primitive that behaves the way `random` used to; when opening an old NetLogo 1.x model, all uses of `random` will automatically be changed to `random-int-or-float`, but we suggest you edit the model and change every occurrence to either `random` or `random-float`, as appropriate
- ♦ `atan 0 0` is now a runtime error, and so is using `towards` (and `towardsxy` and so on) to ask for the heading from a point to that same point
- ♦ we now support unary minus if you put parentheses around it, so for example you can now write `(- x)`
- ♦ added new `n-values` reporter for conveniently constructing lists by repeatedly running a reporter
- ♦ added new reporters `patch-ahead`, `patch-right-and-ahead`, `patch-left-and-ahead`, and `patch-at-heading-and-distance` (helpful for giving turtles "vision", among other uses)
- ♦ added new `shuffle` reporter for shuffling a list
- ♦ added new `modes` reporter for finding the most frequently appearing items in a list
- ♦ added new `remove-item` primitive for removing an item from a list (or string) at a specified position
- ♦ added new `turtles-on` primitive for collecting the set of turtles (or turtles of a particular breed) standing on a patch or set of patches
- ♦ added new `ifelse-value` reporter (lets you put conditionals in reporters; this is the same as what StarLogoT called `ifelse-report`)
- ♦ added new `in-radius-nowrap` reporter (like `in-radius`, but doesn't wrap around screen edges)
- ♦ the `display` primitive now has the additional function of forcing an immediate screen update, since the graphics window in NetLogo 2.0 sometimes skips frames
- ♦ new primitives `turtles-from` and `patches-from` provide a powerful new way to construct agentsets
- ♦ `random-one-of` and `random-n-of` now work on lists too, not just agentsets
- ♦ the `show` primitive now puts double quotes around strings, so you can distinguish them from other values; there is also a new `write` primitive that behaves the same way, but doesn't show the agent
- ♦ `butfirst` and `butlast` now cause a runtime error if given an empty list as input
- ♦ the color constants (`red`, `blue`, and so on) are now floating point numbers, not integers, since turtle and patch colors are always floating point
- ♦ the `display` and `no-display` commands may now be used by turtles and patches as well as by the observer
- ♦ primitives such as `import-world` and `hubnet-set-client-interface` now interpret relative pathnames as relative to the location of the model, not relative to the location of the application
- ♦ old models will automatically be updated to reflect the following changes when opened in the new version:
  - ◊ dropped support for `pc` as an alias for `pcolor`
  - ◊ renamed `any` to `any?` and `user-yes-or-no` to `user-yes-or-no?`
  - ◊ renamed `histogram` to `histogram-from`
  - ◊ dropped support for `set-plot-pen` as an alias for `create-temporary-plot-pen`

- engine fixes:
  - fixed bug where using `report` or `stop` inside a `repeat` or `foreach` loop could cause incorrect results or a Java exception
  - fixed bug where some primitives (including `value-from`, `-of`, `distance`, `towards`, and `inspect`) did not deal properly with dead turtles
  - fixed bug where `every` didn't work as expected when used inside an explicit `without-interruption`
  - fixed issue where using `without-interruption` always caused an agent's turn to end after the code inside ran
  - fixed bug where under certain very obscure circumstances dead turtles could continue to execute code for a short time after dying
  - fixed obscure bug where if a turtle agentset was stored in a variable and in a turtle in the set died, then under certain circumstances, code that subsequently tried to use the agentset could cause a Java exception
  - fixed bug where using `hatch` with a negative number could cause a Java exception (now it just does nothing)
  - fixed bug where `min` and `max` didn't signal a runtime error in every case if given an list with no numbers in it
  - fixed `towards` (and `towards-nowrap`, `towardsxy`, and `towardsxy-nowrap`) to return numbers in the range [0,360), not [-90,270) as they previously did
  - fixed bug where numbers in sliders displayed in scientific notation could be shown with an incorrect exponent
  - fixed minor issue where `remove` primitive evaluated its arguments right-to-left instead of left-to-right
- computer HubNet improvements:
  - computer HubNet is no longer alpha or beta, but a normal release
  - improved reliability
  - greatly improved graphics window mirroring features and performance; this is now no longer considered an "experimental" feature
  - improved activity: Tragedy of the Commons (now out of unverified)
  - new unverified activities: Polling Advanced, Gridlock Alternate, Beer Game, Beer Game Alternate 1, Beer Game Alternate 2
  - server now asks for a name for the computer that is running the activity; this can be the facilitator's name or some other name; it is displayed on the clients in the server menu
  - server menu on clients now only lists servers that are running a compatible version of NetLogo, and the version is also checked when connecting
  - you can now export data from a plot in a client (by right or control clicking)
  - new feature: text input widgets in clients enables clients to input text or NetLogo code into the simulation

Version 1.3.1 (September 2003)

- system:
  - on Mac OS X systems running Java 1.4.1 Update 1, don't give spurious warning that the wrong version of Java is installed
- documentation:
  - fixed some broken links in the User Manual
- engine fixes:

    - fixed bug in the `at-points` reporter that could cause an agentset with duplicates in it to be returned
    - fixed obscure bug in the `die` command that caused some models (such as Gas Chromatography) to behave incorrectly
    - fixed bug where correct code using the `reduce` primitive would sometimes fail to compile
    - fixed bug in "Import World" where if the import file contained the empty string, the built-in variable would be set to 0.0 without notifying the user (this bug should only have affected users who edited a world file using another application)
- interface fixes:
    - compiler now gives a clearer error message if you try to use a non-ASCII character in an identifier; non-ASCII characters are now permissible in some other contexts (such as in strings) where they weren't in NetLogo 1.3
    - fixed obscure bug where occasionally models were saved with turtle shapes in a format that couldn't be loaded on certain Java virtual machines
    - fixed bug where hitting the close box in the dialog that opens when creating a new widget in the interface tab could cause a Java exception

Version 1.3 (June 2003)

- graphics window control strip
- choice widgets
- "Export World" now inter-operates properly with Excel and other programs
- Edit menu now has items for commenting/indenting
- redesigned interface toolbar
- strict math mode so results are identical on all platforms (requires Java 1.3 or higher)
- new primitives: `run/runresult`, `map/foreach/filter/reduce`, `sort-by`, `self`, `asin/acos`
- some primitives such as `list` now accept a variable number of inputs
- many improvements to usability and reliability of computer HubNet

Version 1.2 (March 2003)

- models run faster now
- beta-level support for running as a native application on Mac OS X
- running the full authoring environment as an applet in a web browser is no longer supported (saving individual models as applets is still supported, however)
- 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
- many new primitives and other language improvements
- display of coordinates when mousing over plots

Version 1.1 (July 2002)

- models run faster now
- "Save as Applet" lets you embed your model in any web page
- printer support
- Procedures menu

- scrollable Interface tab
- contextual menus in Interface tab
- improved agent monitors
- experimental "Turtle Sizes" and "Exact Turtle Positions" options
- many new primitives
- improved HubNet support, improved activities

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. 1.4.2 or later is preferred.

Windows 95 is no longer supported by the current version of NetLogo. Windows 95 users should use NetLogo 1.3 instead. We plan to continue to support NetLogo 1.3.

Mac OS X

- OS X version 10.2.6 or later
- 128 MB RAM (256 MB RAM 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.

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 instead. We plan to continue to support NetLogo 1.3.

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 1.4.2 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.

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

Windows−only bugs

- The "User Manual" item on the Help menu does not work on every machine (Windows 98 is more likely to be affected, newer Windows versions less so)

Macintosh−only bugs

- On OS X 10.2, the "User Manual" item on the Help menu will sometimes launch a web browser other than your default browser

Linux/UNIX−only bugs

- User Manual always opens in Netscape, not your default browser (suggested workaround: bookmark the file docs/index.html in your favorite browser)
- We have discovered a problem where the "exp" reporter sometimes returns a slightly different answer (differing only in the last decimal place) for the same input; 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; we have determined that the problem is a bug in the Sun's Java VM, and not in NetLogo itself; we have reported the problem to Sun, and are awaiting a response; we hope that only the "exp" reporter is affected, but we can't be entirely certain of this; we do not know if the problem is specific to Linux, or can occur on Solaris and other Unixes as well

Known issues with computer HubNet

See the HubNet Guide for a list of known issues with computer HubNet.

Planned features

This is only a partial list of features we plan to add. Periodically, new versions of NetLogo will be available from our web site.

- "Export World" and "Import World" save and restore the contents of plots, too
- Declaring constants in the procedures tab
- Models detect and react to individual keystrokes
- "Let" command for introducing a new local variable anywhere in the code
- Add arrays as a distinct data type from lists

- Optional randomized agent scheduler
- Ability to have multiple models open
- Making a QuickTime movie of your model
- Parenthesis and bracket matching in the editor

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`
- `beep`
- `readlist`
- `import-turtles`, `import-patches`, `import-turtles-and-patches` (note that NetLogo adds `import-world`, though)
- miscellaneous seldom-used plotting reporters such as `plot-pencolor`, `pp-plotlist`, `pp-plotpointlist`, `ppinterval`, `ppxcor`, `ppycor`, etc.
- `bit`, `bitand`, `bitneg`, `bitor`, `bitset`, `bitstring`, `bitxor`, `make-bitarray`, `rotate-left`, `rotate-right`, `shift-left`, `shift-right`
- `close-movie`, `open-movie/setup-movie`, `movie-snapshot`, `snapshot`
- `load-pict`, `save-pict`
- `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.

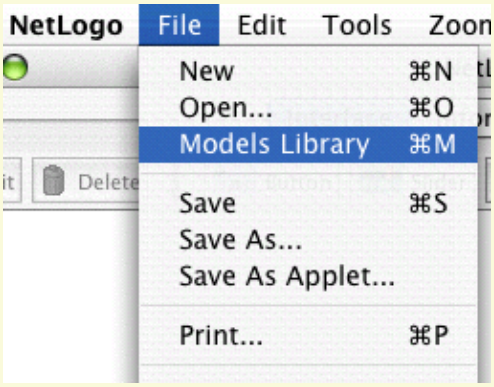
Is there any type of pattern to this kind of grouping?

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:

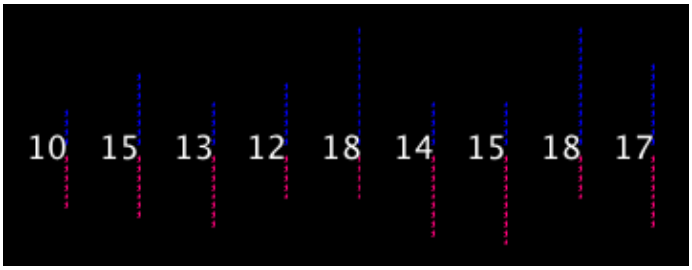
1. Start NetLogo.
2. Choose "Models Library" from the File menu.



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 Graphics Window, 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 Graphics Window 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.

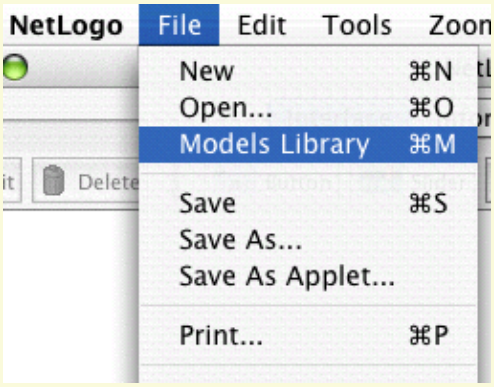
Throughout all of the tutorials, we'll be asking you to make predictions about what the effects of making changes to the models will be. Keep in mind that the effects are often surprising. We think these surprises are exciting and provide excellent opportunities for learning.

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 Graphics Window 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 Graphics Window?

- Wait until the "setup" button pops back up (in other words, turns blue again).
- 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. (Note: If you do not wait for the button to pop back up before pressing any other buttons, the model might get confused, and you may get an error message.)

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.

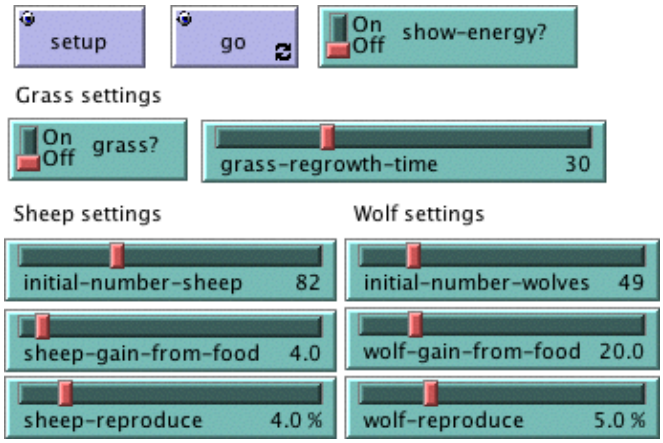
- If you like, experiment with the "setup" and "go" buttons in the Wolf Sheep Predation model.

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 then 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 choice. 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

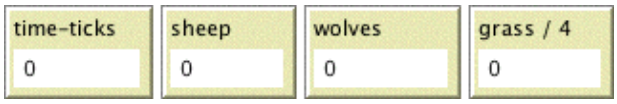
The plot in Wolf Sheep Predation contains three lines: sheep, wolves, and grass / 4. (The grass count is divided by four so it doesn't make the graph too tall.) The lines show what's happening in the model over time. To see which line is which, click on "Pens" in the upper right corner of the plot window 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 by 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 graph 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 Graphics Window

If you look at the graphics window, you'll see a strip of controls along the top edge. The control strip lets you control various aspects of the graphics window.

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

The size of the Graphics Window is determined by three separate settings: Screen Edge X, Screen Edge Y, and Patch Size. Let's take a look at what happens when we change the size of the Graphics Window in the "Wolf Sheep Predation" model.

- Experiment with the three sets of black arrows on the left of the control strip.

What happens the first time you press one of them?

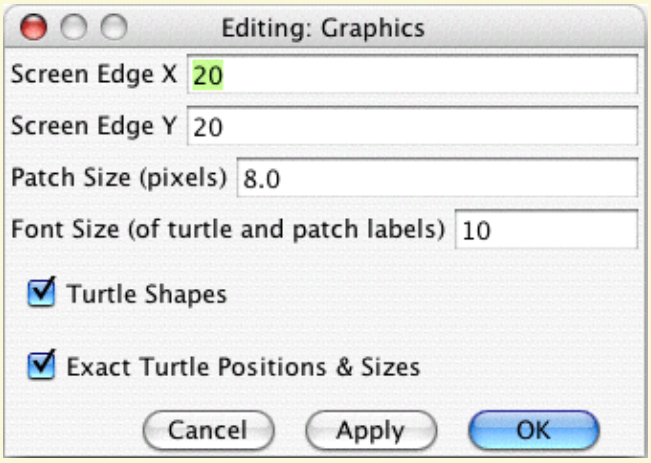
What happens after that? Try all three sets of arrows.

The arrows give you a convenient way of changing the number of patches in the world. NetLogo can't change the number of patches without starting the model over from the beginning, so that's why it warns you the first time you press an arrow.

There are more graphics window settings than there's room for in the control strip. The "More..." button lets you get to the rest of the settings.

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

A dialog box will open containing all the settings for the Graphics Window:



What are the current settings for Screen Edge X, Screen Edge Y, 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 Graphics Window.

You will notice that the pointer turns into a crosshair.

- Hold down the mouse button and drag the crosshair over the Graphics Window.

The Graphics Window 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 Graphics Window.
- Unselect the graphics window by clicking anywhere in the white background of the Interface tab.
- Press the "More..." button again and look at the settings.

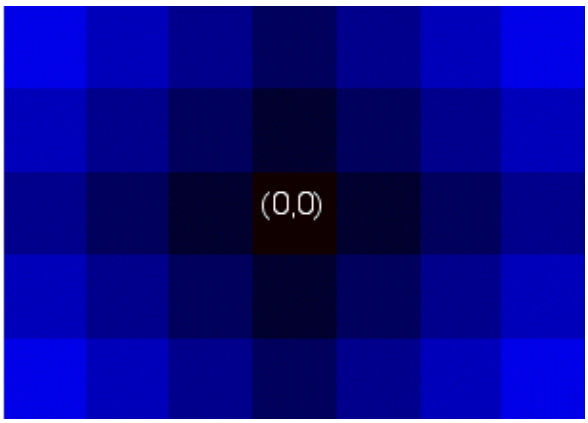
What numbers changed?

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.

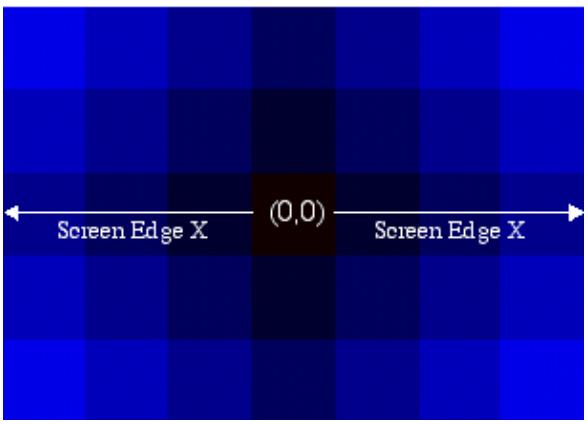
Think of the patches as being like square tiles in a room with a tile floor. Exactly in the middle of the room is a tile labeled (0,0); meaning that if the room was divided in half one way and then the other way, these two dividing lines would intersect on this tile. We now have a coordinate system that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

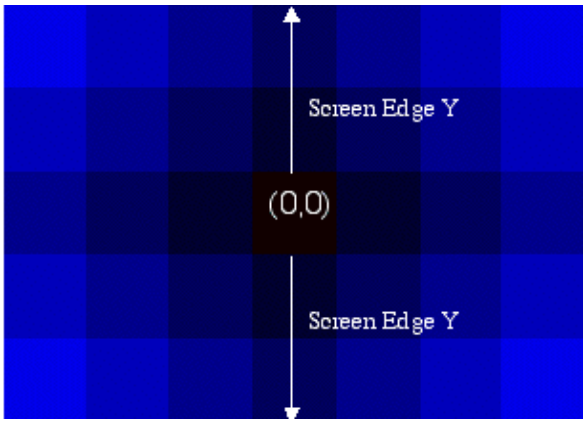


How many tiles away is the (0,0) tile from the right side of the room?

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

In NetLogo, the distance the middle tile is from the right or left edge of the room this is called Screen Edge X. And the distance the middle tile is from the top and bottom edges is called Screen Edge Y:





In these diagrams, Screen Edge X is 3 and Screen Edge Y 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 Screen Edge X and Screen Edge Y.

- Using the Edit dialog that is still open, change Screen Edge X to 30 and Screen Edge Y value to 10.

What happened to the shape of the Graphics Window?

- Press the "setup" button.

Now you can see the new patches you have created.

- Edit the Graphics Window again.
- Change the patch size to 20 and press "OK".

What happened to the size of the Graphics Window? Did its shape change?

Editing the Graphics window also lets you change three other settings: the font size of labels, whether the Graphics Window uses shapes, and whether turtles are drawn in their exact positions or whether they "snap" to fixed grid positions. Feel free to experiment with these 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 140 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.)

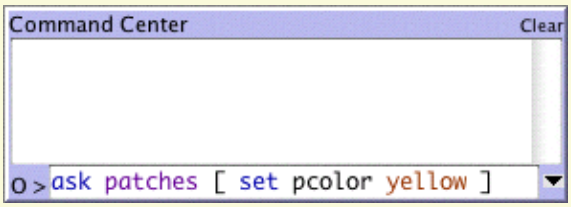
To make these simple changes we will be using the Command Center.

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 Graphics Window?

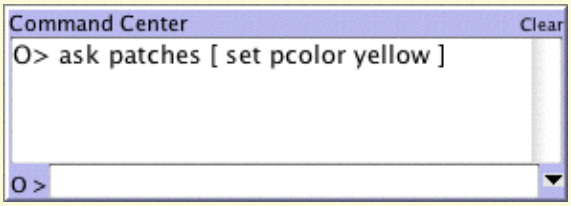
You may have noticed the background of the Graphics Window has turned all yellow and the street has disappeared.

Why didn't the cars turn yellow too?

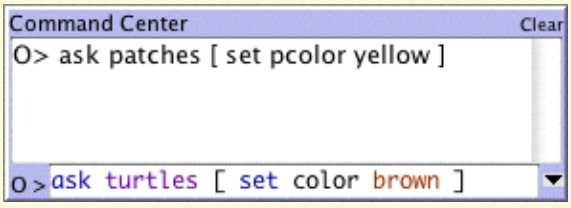
Looking back at the command that was written, we asked only the patches to change their color. In this model, the cars are represented by a different kind of agent, called "turtles". Therefore, the cars did not received these instructions and thus did not change.

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:

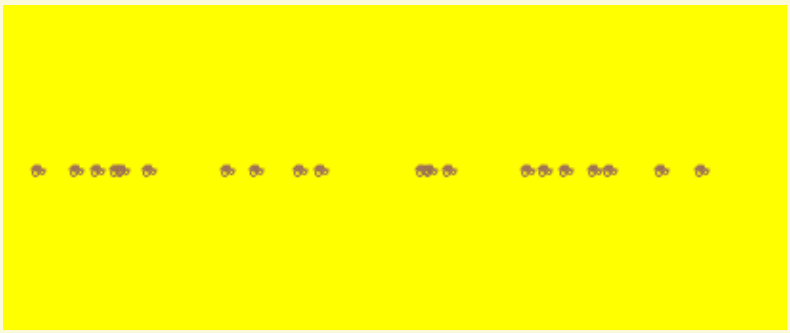


- Type in the white box at the bottom of the Command Center the text shown below:



Was the result what you expected?

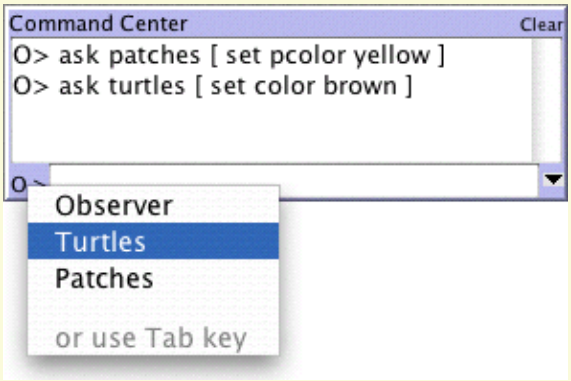
Your Graphics Window should have a yellow background with a line of brown cars in the middle of the window:



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 "O>" 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 "P>" in the bottom left corner.
- Type `set pcolor white` and press return.

What does the Graphics Window 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 (O>`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 (P>`set pcolor white`), you only have to give the command itself.

- Press "setup".

What happened?

Why did the Graphic Window 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:

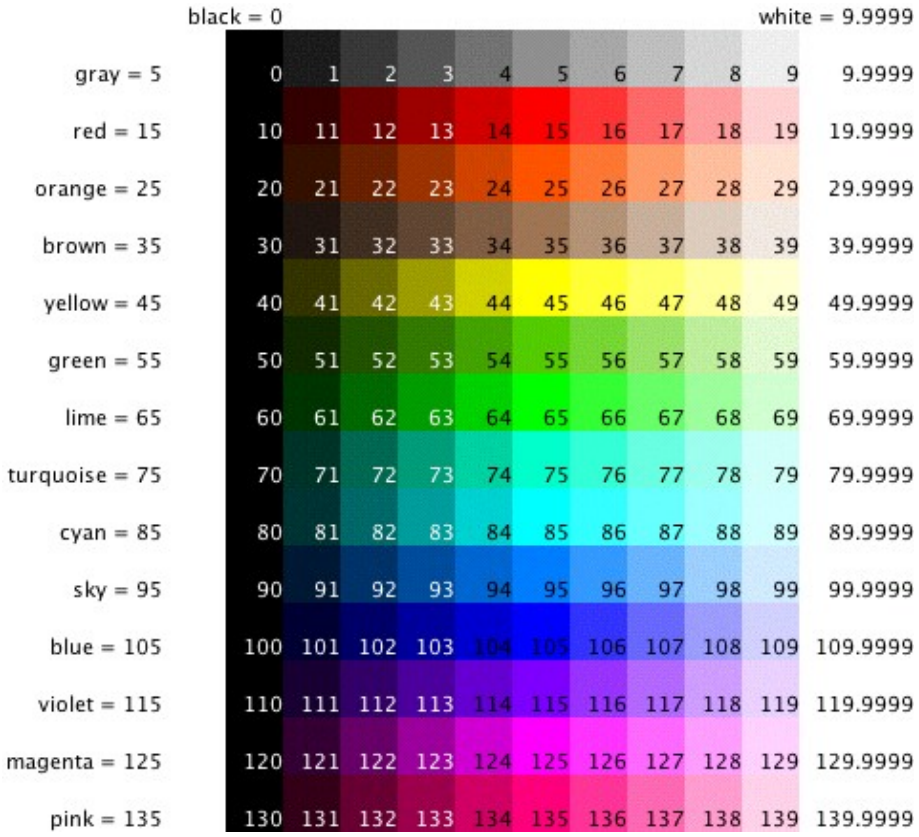
- Type `set pcolor red` instead and press return.

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.

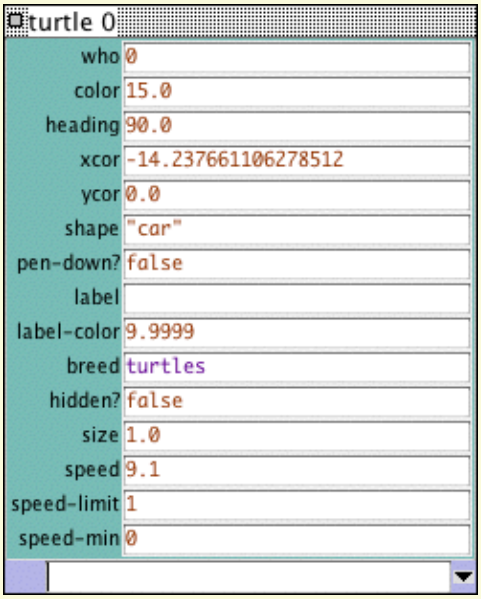
You can use this technique on any of the colors listed in the chart.

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:



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 Graphics Window?

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.

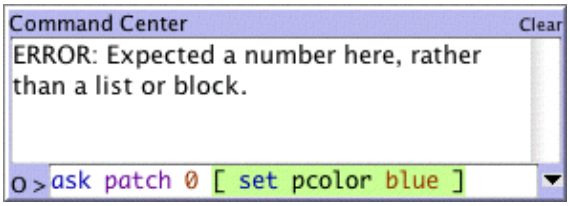
- In the Command Center, select "Observer" from the popup menu (or use the tab key).
- Type `ask turtle 0 [set color blue]` and press return.

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?

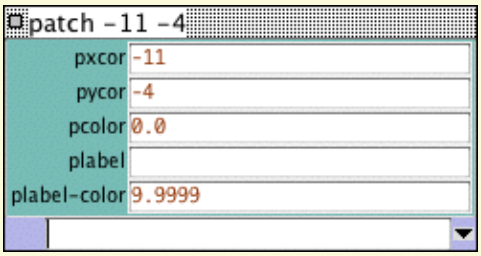
If you try to have the observer `ask patch 0 [set pcolor blue]`, you'll get an error message:



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.

- Type `ask patch −11 −4 [set pcolor blue]` and press return.

What are the two words in this command that "tip you off" that we are addressing a patch?

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

In Tutorial #2, you learned how to use command centers 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. This tutorial leads you through the process of building a complete model, built up stage by stage, with every step explained along the way.

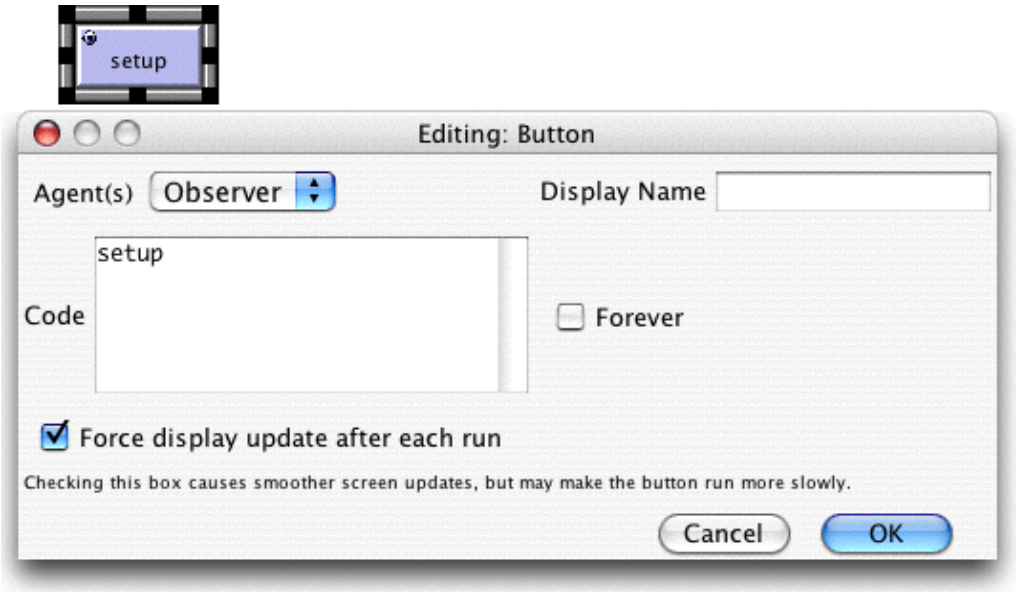
You've already been exposed to the three types of agents you can give commands to in NetLogo: turtles, patches, and the observer. As you start to write your own procedures, it'll be helpful to keep in mind how people usually think of these three different kinds of agents. The turtles and patches usually don't use information about the whole world. They mostly use information about what's close to them. The observer, on the other hand, typically uses and accesses the whole world. Also, while patches can't move and often represent some sort of environment, turtles can move around in the world.

Setup and Go

To start a new model, select "New" from the the File menu. Then begin making your model by creating a once−button called 'setup'.

Here's how to make the button:

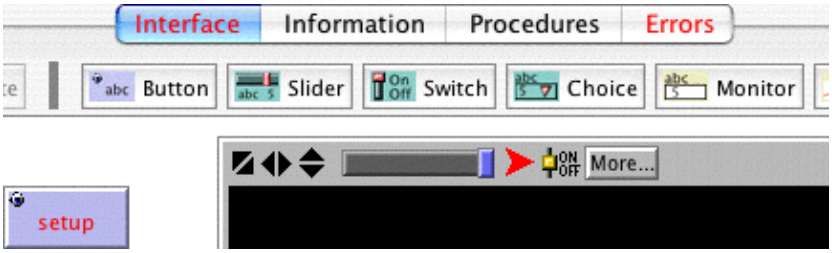
1. Click on the button icon in the Toolbar
2. Click where you want the button to be in the empty white area of the Interface tab
3. When the dialog box for editing the properties of the button opens, type `setup` in the box labeled "Code"



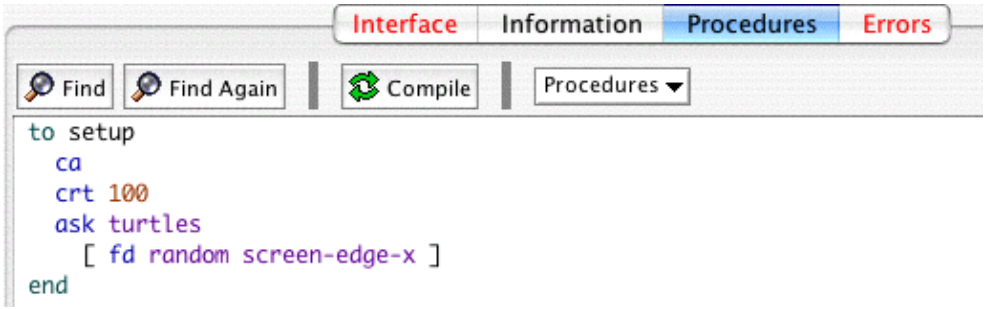
4. Press "OK" to dismiss the dialog box

Now you have a button called 'setup'. It will execute the procedure 'setup' when pressed, which once we define it, will do just that −− set up the NetLogo world.

At this point, both the new button and the Errors tab have turned red. That's because there is no procedure called 'setup'! If you want to see the actual error message, switch to the Errors tab:



Now switch to the Procedures Tab and create the 'setup' procedure shown below. Notice that the lines are indented different amounts. A lot of people find it very helpful to indent their code in a way at least similar to how it's done here. It helps them keep track of where they're at inside of a procedure and makes what they write easier for others to read as well.



One line at a time:

to setup begins defining a procedure named "setup".

ca is short for **clear−all** (you can also spell it out if you want). This command will blank out the screen, initialize any variables you might have to 0, and remove all turtles. Basically, it wipes the slate clean for a new run of the project.

crt 100 will then create 100 turtles. (**crt** is short for **create−turtles**.) If the turtles didn't move after this command is given, each of these turtles would begin on the center patch (at location 0,0). You would only see what looks like one turtle on the screen; they'd all be on top of each other −− lots of turtles can share the same patch. Only the last turtle to arrive on the patch would be visible. Each of these newly−created turtles has its own color, its own heading. All of the turtles are evenly distributed around the circle.

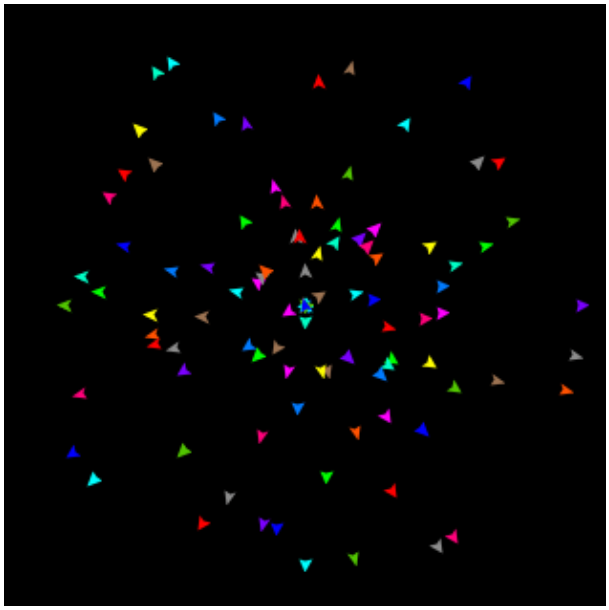
ask turtles [...] tells each turtle to execute, independently, the instructions inside the brackets. Note that **crt** is not inside the brackets. If the agent (observer, turtle, or patch) is not specified using **ask**, the observer runs it. Here the observer runs the ask, but the turtles run the commands inside the ask.

fd (random screen−edge−x) is a command that also uses "reporters". Reporters, as opposed to commands, are instructions that report a result. Each turtle will first run the reporter **random screen−edge−x** which will report a random integer at least 0 but less than 'screen−edge−x' (the dimension from the center to the edge of the screen along the x−axis). It then takes this number, and goes **fd** (short for **forward**) that number of steps, in the direction of its heading. The steps are

the same size as the patches.

end completes the definition of the "setup" procedure.

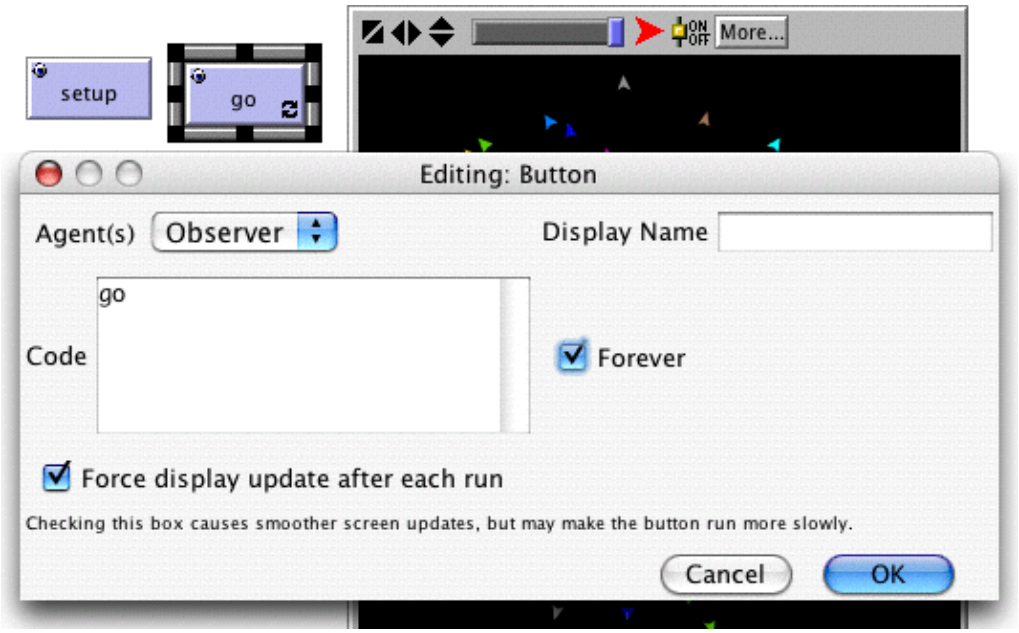
When you're done typing in the code, switch to the Interface tab and press your 'setup' button . You will see the turtles quickly spread out in a rough cluster:



Notice the density distribution of the turtles on the Graphics Window. Press 'setup' a couple more times, and watch how the turtles' arrangement changes. Keep in mind that some turtles may be right on top of each other.

Can you think of other ways to randomly distribute the turtles over the screen? Note that if a turtle moves off the screen, it "wraps", that is, comes in the other side.

Make a forever−button called 'go'. Again, begin by creating a button, but this time check the "forever" checkbox in the edit dialog.



Then add its 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 **fd** is? No, it's a procedure that you're about to write, right after the **go** procedure:

```
to move-turtles
  ask turtles [
    set heading (random 360)
    fd 1
  ]
end
```

Be careful of the spacing around the "-". 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 one word.

Line by line:

ask turtles [*commands*] says that each turtle should execute the commands in the brackets.

set heading (random 360) is another command that uses a reporter. First, each turtle picks a random integer between 0 and 359 (**random** doesn't include the number you give it as a possible result). Then the turtle sets its heading to the number it picked. Heading is measured in degrees, clockwise around the circle, starting with 0 degrees at twelve o'clock (north).

fd 1: Each turtle moves forward one step in the new direction it just set its heading to.

Why couldn't we have just written that in ***go***? We could, 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 could include many other things you want to have happen

as the model runs, such as calculating something or plotting the results. Each of these sub−procedures could have its own name.

The 'go' button you made in the Interface tab is a forever−button, meaning that it will continually execute its code 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 turtles stop in their tracks.

We suggest you start experimenting with other turtle commands. You might try typing **T> pendown** into the Command Center and then pressing go. Another thing to try is changing **set heading** (**random 360**) to **rt** (**random 360**) inside of ***move−turtles***. ("rt" is short for "right turn".) Also, you can try changing **set heading** (**random 360**) to **lt** (**random 45**) inside of ***move−turtles***. Type commands into the Command Center (like **set color***red*), or add them to ***setup***, ***go***, or ***move−turtles***. Note that when you enter commands in the Command Center, you must choose **T>**, **P>**, or **O>** in the popup menu on the left, depending on which agents are going to execute the commands. You can also use the tab key, which you might find more convenient than using the popup menu. **T>*commands*** is identical to **O> ask turtles [*commands*]**, and **P>*commands*** is identical to **O> ask patches [*commands*]**.

Play around. It's easy and the results are immediate and visible −− one of NetLogo's many strengths. Regardless, the tutorial project continues...

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:

```
patches−own [elevation]

to setup
  ca
  setup−patches
  setup−turtles
end
```

The line at the top, **patches−own [elevation]** declares that we have a variable for the patches, called ***elevation*.** Our 'setup−patches' procedure that we haven't defined yet will then use this variable. We also still need to define 'setup−turtles' as well, but, for now, here's how to define **setup−patches**:

```
to setup−patches
  ask patches
    [ set elevation (random 10000) ]
  diffuse elevation 1
  ask patches
    [ set pcolor scale−color green elevation 1000 9000 ]
end
```

The **setup−patches** procedure sets the elevation and color of every patch. First, each patch picks a random integer between 0 and 9999 and sets its *elevation* variable to that number.

We then use an observer primitive, **diffuse**, that smooths out the distribution of this variable over the neighboring patches. Remember that primitives are built in procedures in NetLogo, as opposed to procedures that you define.

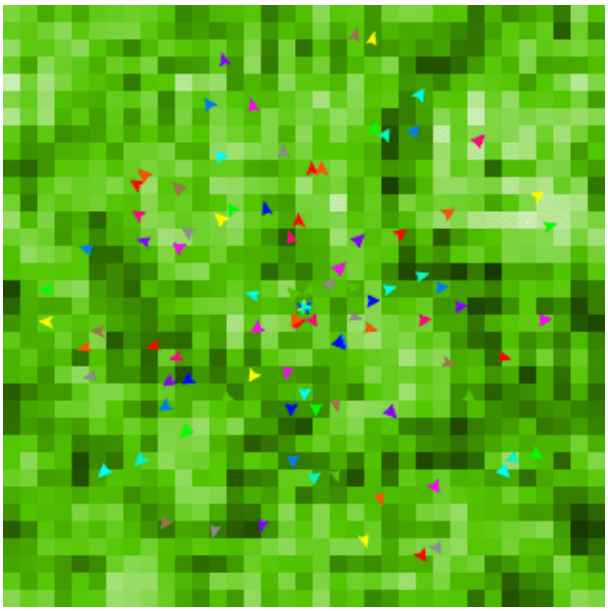
Scale−color is a reporter that uses the different values of *elevation* to assign colors to the patches. In this case, we're assigning different shades of green to all the patches. (Don't worry about the numbers given to **diffuse** and **scale−color** just yet...) The larger *elevation* is, the lighter the shade of green. Low values of *elevation* will result in darker shades.

The only part remaining in our new 'setup' that is still undefined is **setup−turtles**:

```
to setup-turtles
  crt 100
  ask turtles
    [ fd (random screen-edge-x) ]
end
```

Setup−turtles is exactly what we were doing in the old **setup** procedure.

After typing all of this in, press the 'setup' button back in the Interface tab. Voila! A lush NetLogo landscape complete with turtles and green patches appears. After seeing the new 'setup' work a few times, you may find it helpful to read through the procedure definitions again.



Here's a way for you to see what **diffuse** does. Return to the Procedures tab, and use a semicolon to 'deactivate' the diffuse command like this:

```
;diffuse elevation 1
```

Semicolons are very useful in writing procedures. They can be used as above to save you from having to delete code to try something else out and then having to rewrite them. Also, they can be used to add some explanatory text to your procedures. A lot of people like to do this to make their procedures more readable to others. Notice that all the text to the right of a semicolon becomes gray.

Press 'setup' again −− looks different, doesn't it? This is because, as mentioned above, **diffuse** has each patch share its value of *elevation* with all its neighbors, by having every patch reset its value of *elevation* to a new value that depends on the value of *elevation* all around it. For further explanation of how diffuse works, go to the Primitives Dictionary if you'd like. Also, it may help to toy with the values being passed to it and see what happens.

We're now prepared to create some kind of dialog between the turtles and the patches. In fact, we even have an idea for a project here. Notice that we called the patch variable 'elevation', and that our landscape sort of looks topographical? We're going to have our turtles do what is called 'hill−climbing', where every turtle seeks to find the highest elevation it can.

In order to do this, we will learn how to write more complex instructions. Go to the Command Center, and type **O> show max values−from patches [elevation]** and **show min values−from patches [elevation]**. These two reporters will, respectively, search over all the patches to return to you the highest elevation and the lowest. These commands work like this (you can read about them in the NetLogo Primitives Dictionary):

Look up 'values−from' in the dictionary. It shows "values−from AGENTSET [expression]" and says it returns a list. In this case, it looks at the expression (elevation) for each agent in the agentset (patches) and returns all of these as a list of elevations.

Look up 'min' in the dictionary. It shows "min *list*" and says it's a reporter. So it takes the list of elevations and reports the smallest value.

'Show' displays this value in the command center.

We will use these reporters −− **max values−from patches [elevation]** and **min values−from patches [elevation]** −− in our model.

Just in case we end up needing the highest and lowest elevation in several places in our procedures, let's make a shortcut. We'll do a little extra work now so that if we need these values later, we'll have a shortcut to use. First, at the top of your code (right after the 'patches−own' declaration), declare two global variables as such:

```
globals [highest ;; the highest patch elevation
         lowest] ;; the lowest patch elevation
```

(Notice the use of semicolons here. Although the names of the global variables are descriptive, the semicolons allow us to describe the variables even more.)

Global variables can be used by all the agents in the model. In particular, patches can use `highest` and `lowest` in the ***setup−patches*** procedure. We need to store the highest and lowest elevations in these global variables once, and then everyone will have quick access to them after that. Write:

```
to setup−patches
  ask patches
    [ set elevation (random 10000) ]
  diffuse elevation 1
  ask patches
    [ set pcolor scale−color green elevation 1000 9000 ]
  set highest max values−from patches [elevation]
  set lowest min values−from patches [elevation]
```

```
  ask patches [
    if (elevation > (highest − 100))
      [set pcolor white]
    if (elevation <(lowest + 100))
      [set pcolor black] ]
end
```

Now we have saved the highest and lowest points in our terrain and displayed them graphically.

Look at the last two commands, the **if** commands. Each patch, when it runs these commands, compares its own value of *elevation* to our global variables *highest* and *lowest*. If the comparison reports 'true', the patch executes the commands inside the brackets. In this case, the patch changes its color. If the comparison reports 'false', the patch skips over the commands inside the brackets.

These **if**s cause all patches whose value of *elevation* is NEAR to the highest (within about 1% for our values) change their color to white, and all patches whose values are NEAR to the lowest become black. We want this so that they'll be easier to see. You can make a couple of quick changes here if you wish −− they won't affect the rest of the model. For example, instead of saying 'set pcolor white' and 'set pcolor black', you can say 'set pcolor blue' and 'set pcolor red' (or whatever other colors you may wish). Also, you can change the range of 'highest peaks' and 'lowest peaks' by changing the number 100 to some other number.

After this, 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.) Name one of them `highest` and the other one `lowest`. The reporters you'll want in each of them happen to be `highest` and `lowest` as well. (If you want to learn more about reporters, you can look them up in the NetLogo Programming Guide). Now every time you click 'setup' and redistribute the values of *elevation*, you'll know exactly what the highest and lowest elevations are, and where they can be found.

An Uphill Algorithm

Okay. Finally we're ready to start hill−climbing. To rehash: we've got some turtles randomly spread out from the origin; and we've got a landscape of patches, whose primary attribute is their *elevation*. Lastly, we have two kinds of tools to help us understand the patch landscape: each patch has a color, depending on its value of *elevation*, and we have a pair of monitors telling us what the highest peak and lowest valley are. What we need now is for the turtles to wander around, each trying to get to the patch that has the highest elevation.

Let's try a simple algorithm first. We'll assume three things: 1), that the turtles cannot see ahead farther than just one patch; 2), that each turtle can move only one square each turn; and 3), that turtles are blissfully ignorant of each other. Before, we had a procedure ***move−turtles*** like this:

```
to move−turtles
  ask turtles [
    set heading (random 360)
    fd 1
```

```
  ]
end
```

But now we don't want them to move randomly about. We want each turtle to look at the *elevation* of each patch directly around it, and move to the patch with the highest elevation. If none of the patches around it have a higher elevation than the patch it is on, it'll stay put. This new procedure should replace 'move−turtles' inside of 'go'. Type in the following code and run it once or twice:

```
;; each turtle goes to the highest elevation in a radius of one
to move−to−local−max
  ask turtles [
    set heading uphill elevation
    if ( elevation−of patch−ahead 1 > elevation )
    [ fd 1 ]
  ]
end
```

Now that you've seen the uphill algorithm work in the model, let's go through the new primitives involved. (If you haven't run the model yet since writing 'move−to−local−max', give it a try.) There are three new primitives here: '**uphill**', '**−of**', and '**patch−ahead**'. 'uphill elevation' finds the heading to the patch with the highest value of *elevation* in the patches in a one−patch radius of the turtle. Then through the use of the command 'set heading', the turtle sets its heading to that direction. 'elevation−of patch−ahead 1' has each turtle look at the variable *elevation* in the patch on which the turtle would be if it went forward 1. If the test reports true, the turtle moves itself forward 1. (The test is necessary because if the turtle is already on the peak, we don't want it to move off it!)

Go ahead and type that in, but before you test it out by pressing the 'go' button, ask yourself this question: what do you think will happen? Try and predict how a turtle will move, where it will go, and how long it'll take to get there. When you're all set, press the button and see for yourself.

Surprised? Try to understand why the turtles converge to their peaks so quickly. Maybe you don't believe the algorithm we've chosen works 'correctly'. There's a simple procedure you can make to test it. write a procedure ***recolor−patches*** so that it says:

```
to recolor−patches
  ask patches
  [
    set elevation pycor
    set pcolor scale−color green elevation
                (0 − screen−edge−y) screen−edge−y
  ]
end
```

Press 'setup'. The model looks the same as it did before because ***recolor−patches*** hasn't been run yet. Instead of making a button that calls your testing procedure, let's do something different. Type O>`recolor−patches` into the command center, the procedure gets called. Now, when you press 'go', see that the turtles all head for the highest elevation −− the top of the screen.

Another common tool to see what's going on is to write **T> pd** in the Command Center. Then each turtle traces its path with its color. This will show you where the turtle has been.

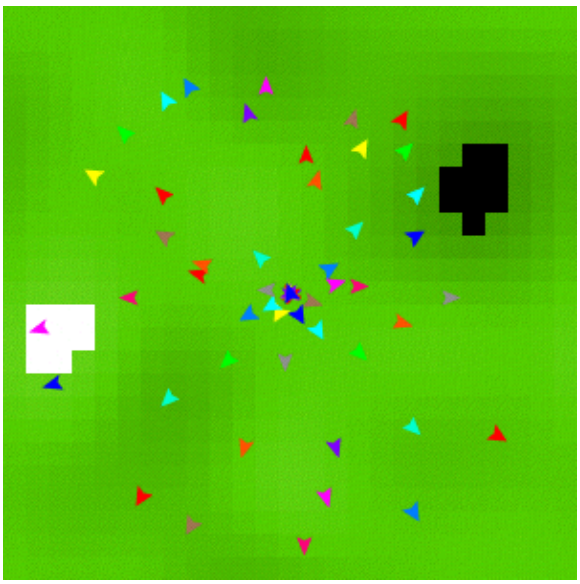
Our turtles rapidly arrive at local maxima in our landscape. Local maxima and minima abound in a randomly generated landscape like this one. Our goal is to still get the turtles to find an 'optimal maximum', which is one of the white patches.

Part of the problem is that our terrain is terribly lumpy. Every patch picked a random elevation, and then we diffused these values one time. This really doesn't give us a continuous spread of elevation across the graphics window, as you might have noticed. We can correct this problem to an arbitrary degree by diffusing more times. Replace the line:

```
diffuse elevation 1
```

with:

```
repeat 5 [ diffuse elevation 1 ]
```



The **repeat** command is another way for NetLogo to loop (besides making a forever button, which you already know how to do). **Repeat** takes a number (here, 5) and some commands (here, the **diffuse** command), and executes the commands that number of times (here, five times). Try it out, and look at the landscape (i.e. press 'setup' and see what you think). Then, press 'go' and watch the turtles' behavior. (Remember that the lighter the patch, the greater the elevation.)

Obviously, fewer peaks make for an improvement in the turtles' performance. On the other hand, maybe you feel like this is cheating −− the turtles really aren't doing any better, it's just that their problem was made easier. True enough. If you call **repeat** with an even higher number (20 or so), you'll end up with only a handful of peaks, as the values become more evenly distributed with every successive call. (Watch the values in the monitors.)

In order to specify how 'smooth' you want your world to be, let's make it easier to try different values. Maybe one time you'll want the turtles to try and 'solve a hard world', and maybe another time you'll just want to look at an easy landscape. So we'll make a global variable named "smoothness". Create a slider in the Interface tab and call it "smoothness" in the editing box. The minimum can be 0, and the maximum can be 25 or so. Then change your code to:

```
repeat smoothness [ diffuse elevation 1 ]
```

Experiment with the turtles' performance in different terrains.

We still haven't even begun to solve the problem of getting all the turtles to the highest elevation, though. So far we've just been getting the turtles to the highest point that's near them. If a turtle starts off in one corner of the world on a hill and there's a mountain in a different corner, the turtle will never find the mountain. To find the mountain, the turtle would have to go down off the hill first, but in our model, turtles only move up. Notice that the individual turtles don't use 'highest' anywhere. The turtles just look at elevations close to them and go the highest point they can see.

Before trying something else, it'd be nice if we could have some other, more precise method for evaluating the turtles' performance. Fortunately, NetLogo allows us to plot data as we go along.

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 **go** to call the new procedure we're about to add:

```
to go
  move-to-local-max
  do-plots
end
```

Now add the new procedure. What we're plotting is the number of turtles who've reached our 'peak-zone' (within 1% of the highest elevation) at some given time.

```
to do-plots
   set-current-plot "Turtles at Peaks"
   plot count turtles with
     [ elevation >= (highest - 100) ]
end
```

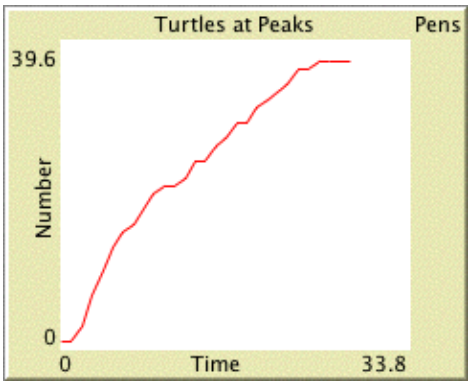
Note that we use the **plot** primitive to add the next point to a plot, but before doing that, we need to tell NetLogo which plot we want, since later our model might have more than one plot.

Thus we're plotting the number of turtles within 100 units of our maximum elevation at some given point in time. The **plot** command moves the current plot pen to the point that has x- coordinate equal to 1 greater than the old x- coordinate and y-coordinate equal to the value given in the plot command (in this case, the number of turtles whose elevation is within 100 of highest). Then the *plot* command draws a line from the current position of the plot pen to the last point it was on.

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

Note that when you create the plot you can set the minimum and maximum values on the x and y axes, and the color of the default plot pen (pick any color you like). 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.

Now reset the project and run it again. You can now watch the plot be created as the model is running. If you notice that your plot doesn't look exactly like the picture below, try to think about why it doesn't look the same. If you think it's because 'go' remains pressed until you manually unpress it, we'll fix that problem by the end of the tutorial. Remember that we kept "Autoplot?" on. This allows the plot to readjust itself when it runs out of room.



You might try running the model several times under different settings (i.e. different values of *smoothness*) and watch how fast the plot converges to some value, and what fraction of the turtles make it to the top. You may want to even try the same settings several times.

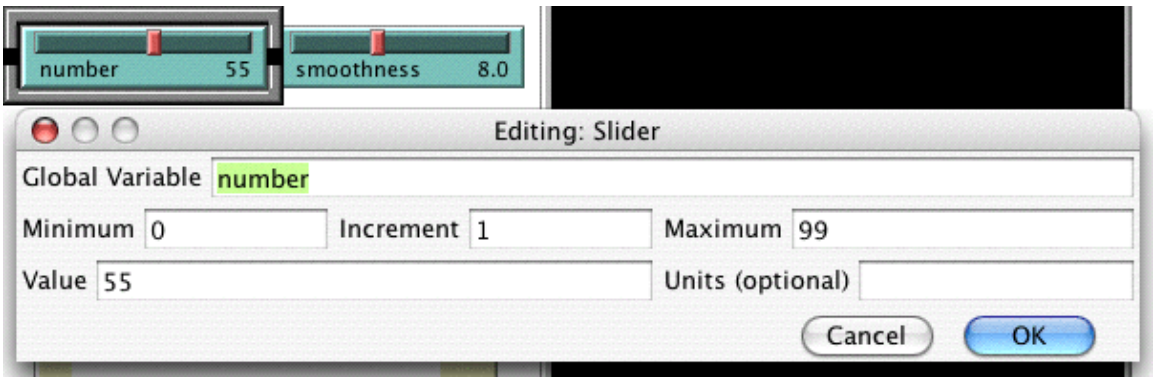
Some More Details

There are a few quirks you may already have noticed. Here are some quick changes you can make.

First, we have a green landscape −− a naturally green turtle is going to be hard to see. In the ask turtles block in 'setup−turtles', you can say:

```
if (shade-of? green color)
  [ set color red ]
```

Second, instead of always using 100 turtles, you can have a variable number of turtles. Make a slider variable (say, 'number'):



Then, inside of **setup−turtles**, instead of 'crt 100', you can type:

```
crt number
```

How does using more or fewer turtles affect the success value displayed by the plot?

Third, when all the turtles have found their local maxima, wouldn't it be nice for the model to stop? This requires a few lines of code.

- Add a global variable *turtles−moved?* to the "globals" list:

```
globals [
  highest         ;; maximum patch elevation
  lowest          ;; minimum patch elevation
  turtles-moved?  ;; so we know when to stop the model
]
```

- At the end of the **go** procedure, add a test to see if any turtles have moved.

```
to go
  set turtles-moved? false
  move-to-local-max
  do-plots
  if (not turtles-moved?)
  [ stop ]
end
```

- In **move-to-local-max** if a turtle moves, set *turtles-moved?* to true.

```
to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-ahead 1 > elevation )
    [
      fd 1
      set turtles-moved? true
    ]
  ]
end
```

Finally, what rules can you think of that would help turtles escape from lower peaks and all get to the highest ones? Try writing them.

What's Next?

So now you have a nice framework for exploring this problem of hill-climbing, using all sorts of NetLogo modeling features: buttons, sliders, monitors, plots, and the graphics window. You've even written a quick procedure to give the turtles something to do. And 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 NetLogo Programming Guide.

Also, You can continue with this model if you'd like, experimenting with different variables and algorithms to see what works the best (what makes the most turtles reach the peaks).

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 pleasant just to watch patches and turtles forming patterns, or whatever. Hopefully you will have learned a few things, both in terms of syntax and general methodology for model- building. The entire code 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".

```
patches-own [ elevation ]      ;; elevation of the patch

globals [
  highest         ;; maximum patch elevation
  lowest          ;; minimum patch elevation
  turtles-moved?  ;; so we know when to stop the model
]

;; We also have two slider variables, 'number' and
;; 'smoothness'. 'number' determines the number of
;; turtles, and 'smoothness' determines how erratic
;; terrain becomes during diffusion of 'elevation'.

;; resets everything
to setup
  ca
  setup-patches
  setup-turtles
end

;; creates a random landscape of patch elevations
to setup-patches
  ask patches [set elevation (random 10000) ]
  repeat smoothness [diffuse elevation 1 ]
  ask patches
    [ set pcolor scale-color green elevation 1000 9000 ]

  set highest max values-from patches [elevation]
  set lowest min values-from patches [elevation]
  ask patches [
    if (elevation > (highest - 100))
      [set pcolor white]
    if (elevation <(lowest + 100))
      [set pcolor black]
  ]
end

;; initializes the turtles
to setup-turtles
  crt number
  ask turtles [
    if (shade-of? green color) [ set color red ]
    fd (random screen-edge-x)
  ]
end

;; RUN-TIME PROCEDURES
;; main program control
to go
  set turtles-moved? false
  move-to-local-max
  do-plots
  if (not turtles-moved?)
  [ stop ]
end
```

```
;; each turtle goes to the highest elevation in a radius of one
to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-ahead 1 > elevation )
    [
      fd 1
      set turtles-moved? true
    ]
  ]
end

to do-plots
  set-current-plot "Turtles at Peaks"
  plot count turtles with
    [ elevation >= (highest - 100) ]
end
```

Interface Guide

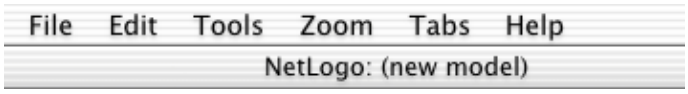
This section of the manual walks you through every element of the NetLogo interface in order and explains its function.

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

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

Menus

On 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.

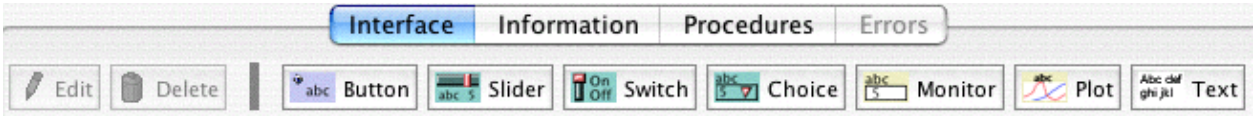
Chart: NetLogo Menus

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 and the current state of all turtles and patches 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 Graphics	Save a picture of the graphics window to a file (in PNG format).

	Export Output	Save the contents of the output section of the command center to a file.
	Import World	Load a file that was saved by Export World.
	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.
	Find	Finds a word or sequence of characters within the Information, Procedures, or Errors tab.
	Find Again	Find the next occurrence of the word or sequence you last used Find with.
	Shift Left / Shift Right	Used in the Procedures and Errors tabs to change the indentation level of code.
	Comment / Uncomment	Used in the Procedures and Errors tabs 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 Graphics Window; see the Graphics Window 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 Graphics Window; see the Graphics Window section below.)
	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.
	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 4. On Window, it's Control 1 through Control 4.)
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.

Main Window

At the top of NetLogo's main window are four tabs labeled "Interface", "Information", "Procedures", and "Errors". 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 Graphics Window, where the turtles and patches appear, and the Command Center, which allows you to issue NetLogo commands.

Interface Toolbar

The toolbar 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.

Working With Interface Elements

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.

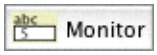
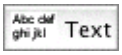
Moving: Select the interface element, then drag it with your mouse to its new location. If you hold down the shift key while dragging, the element will move only straight up and down or straight left and right.

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 *once−only* buttons or *forever* 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.
	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 observe what happens in the model.
	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.
	Choices let the user pick a value for a global variable from a list of choices, presented in a drop down menu.
	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.
	Plots are real−time graphs of data the model is generating.
	A Text Box lets you create text labels in the Interface tab.

Graphics Window

The Graphics Window initially appears as a large black square on the Interface tab. This is the graphical world of NetLogo's turtles and patches.

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

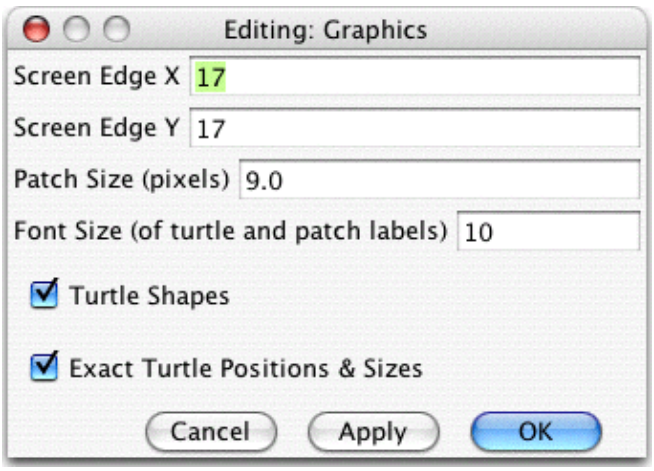
The Graphics Window provides an easy way to open a turtle monitor or patch monitor. Just control−click (Macintosh) or right−click (other systems) on the turtle or patch you want to inspect, and choose "inspect turtle ..." or "inspect patch ..." from the popup menu. (Turtle and patch monitors can also be opened from the Tools menu or by using the `inspect` command.)

There are a number of settings associated with the Graphics Window. There are two ways of changing the settings: by using the control strip along the top edge of the Graphics Window, or by editing the Graphics Window, as described in the "Working With Interface Elements" section above. (Or, pressing the "More..." button in the control strip is an alternate and quicker way.)

The controls in the control strip work as follows:

- The three sets of black arrows let you change the size of the world.
- 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 graphics window don't change until you unfreeze it by flipping the switch again. Most models run much faster when the graphics window is frozen.

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



To change the size of the Graphics Window adjust the "Patch Size" setting, which is measured in pixels. This does not change the number of patches, only how large the patches appear on the screen. To change the number of patches, alter the "Screen Edge X" and "Screen Edge Y" settings. (Note that changing the numbers of patches requires rebuilding the NetLogo world; you will lose all turtles and the values of all variables.)

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

In most NetLogo models, turtles are visible at their exact locations, and may vary in size. If you turn off the "Exact Turtle Positions" checkbox, then:

- Every turtle is drawn at the same size
- Every turtle is drawn as if it were standing on the center of its patch

- Only the top turtle on a patch is visible

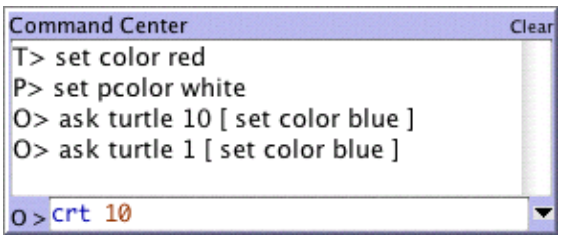
This makes a model appear is it would have in earlier versions of NetLogo, which did not support turtle sizes and exact turtle positions. This checkbox does not change behavior of a model, only its visual appearance.

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 turtles, patches, and the observer.) 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 closer look at the design of the Command Center.



You will notice there is a large display box, an agent popup menu (O>), a "clear" button, and the history popup menu (with the little black triangle). The top large display box temporarily stores all of the commands that are entered into the Command Center. This area is strictly for reference; commands cannot be accessed or changed from this box. To clear this box, click "clear" in the top right corner.

The smaller text box, below the large box, is where commands are entered. On the left of this box is the agent popup menu, and on the right is the history popup menu.

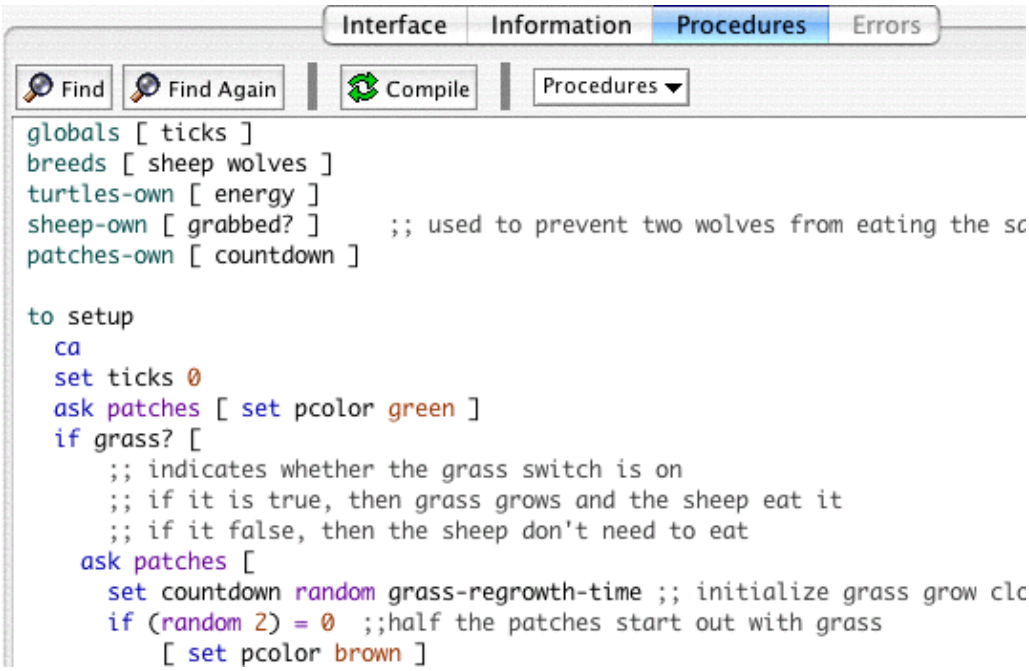
The agent popup menu allows you to select either observer, turtles, or patches. This is an easy way to assign an agent to a command and it is the same as writing `ask turtles [ ... ]`. Note: a quicker way to change between observer, turtles, and patches is to use the tab key on your keyboard.

The history popup menu lists all of the commands entered that can be accessed and used again. The up and down arrow keys on your keyboard will retrieve that last command that was written.

Note that pressing the "clear" function clears only the large display box and not the history. To clear the history section, choose "clear history", found at the top of its popup menu.

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, press the "Compile" button. If there are any syntax errors, the Errors tab will come to the front of the screen and turn red. 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 compiled and any errors will be shown, so if you switch tabs, pressing the Compile button first isn't necessary.

To find a fragment of code in the procedures, click on the magnifying glass on the Procedures Toolbar. Then enter the text you are looking for, and hit the "Find" button. The "Find Again" button finds the next location of the word or fragment of code throughout the procedure.

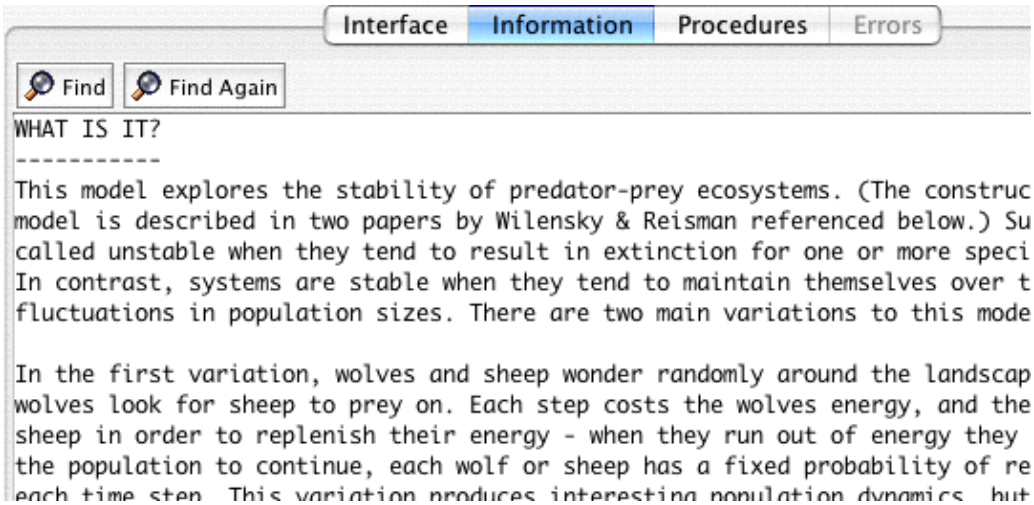
To find a particular procedure definition in your code, use the "Procedures" popup menu in the Procedures Toolbar. The menu lists all procedures in alphabetical order.

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

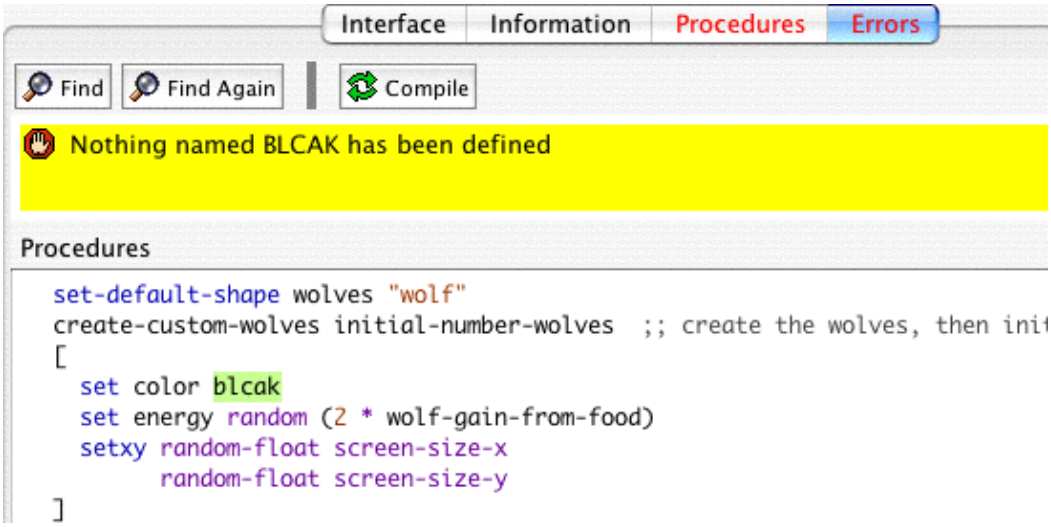
The Information tab provides an introduction to the model and an explanation of how to use it, things to explore, possible extensions, and NetLogo features. It is very helpful when you're first exploring a model.



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.

Errors Tab

The Errors tab provides a place for errors to be highlighted and brought to your attention. If there are any errors in your code, the "Errors" tab will turn red and come to the front with an explanation of what caused the error. When there are no errors, the Tab is disabled.



Switching tabs, or pressing the "Compile" button in the toolbar, causes your code to be checked for errors.

Programming Guide

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

(Note: If you are already familiar with StarLogo or StarLogoT, then the material in the first four sections may already be familiar to you.)

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

Agents

The NetLogo world is made up of agents. Agents are beings that can follow instructions. 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 in the center of the world has coordinates (0, 0). 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 `screen-edge-x` and `screen-edge-y`. When NetLogo starts up, both `screen-edge-x` and `screen-edge-y` are 17. 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 Graphics window.)

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 isn't bounded, but "wraps" −− so when a turtle moves past the edge of the world, it disappears and reappears on the opposite edge. 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.

Procedures

In NetLogo, commands and reporters tell agents what to do. **Commands** are actions for the agents to carry out. **Reporters** carry out some operation and report a result either to a command or another reporter.

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 screen
  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.

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 by 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 ]
```

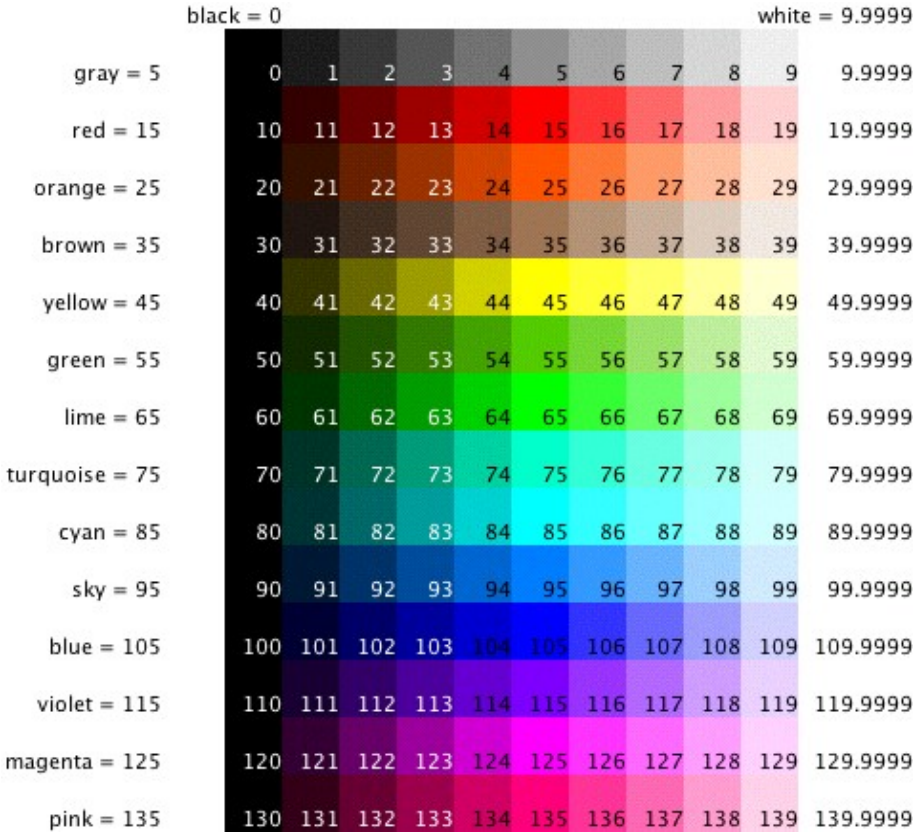
causes every turtle to make the patch it is standing on red. (Because patch variables are shared by turtles in this way, you can't have a turtle variable and a patch variable with the same name.)

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 random-one-of turtles-here yellow ]
;; on every patch, a random turtle turns yellow
```

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, 10, 20, and so on are all black. 9.9999, 19.9999, 29.9999 and so on are all white.

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

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 is available, and you can either decrease the brightness (darken) or decrease the saturation (lighten) of those hues to get your desired color, but you may not decrease both the brightness and saturation.

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-hsb` 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 screen 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 make 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.

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 screen
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 `random-one-of`. For example, we can make a randomly chosen turtle turn green:

  ```
  set color-of random-one-of turtles green
  ```

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

  ```
  ask random-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

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 `breeds` keyword, at the top of your model, before any procedures:

```
breeds [wolves sheep]
```

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), `sheep-here`, and `sheep-at`.

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 random-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:

```
breeds [mice frogs]
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 the user presses 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 the user presses the button again to stop it. If the user stops 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 may 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.)

Buttons and display updates

When you edit a button, there is a checkbox called "Force display 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 care more about smooth animation than raw speed, you should uncheck the box, but if you care more about smooth animation, you should leave it checked. 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 graphics window. 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 1/20 of a second, and then redraws the world, 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 the user 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 screen 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 the user presses 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−calculation ]
```

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

But if you write it this way, they are:

```
ask turtles [ fd random 10 ]
ask turtles [ do−calculation ]
```

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−calculation" at the same time.

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

```
fd random 10 ,
do−calculation ,
```

Procedures (advanced)

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 size]
   pd
   repeat num−sides
     [ fd size
       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
```

Procedures with local variables

A local variables is defined and used only in the context of a particular procedure. To add a local variable to your procedure, use the `locals` keyword. It must come at the beginning of your procedure. For example:

```
to swap−colors [turtle1 turtle2]
   locals [temp]
   set temp color−of turtle1
   set (color−of turtle1) (color−of turtle2)
   set (color−of turtle2) temp
end
```

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 of all 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 contains 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 lists, you can use the `list` reporter with more than two inputs, but in order to do so, you must enclose the entire call in parentheses, like this:

```
(list 1 2 3 4 5)
```

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 the 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, only one command changes a list -- `set`. This is used in conjunction with reporters. For example, to change the third item of a list to 10, you could use the following code:

```
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.

Math

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

Integers have no fractional part and may range from −2147483648 to 2147483647 (−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.

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 to the user.

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 "pseudorandom". (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, pseudorandom 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 pseudorandom 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.

Code Example: Random Seed Example

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).

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, random−one−of, and so on) aren't the only ones that use pseudorandom numbers. Some other primitives also make random choices. For example, the `sprout` command creates turtles with random colors and headings, and the `downhill` reporter chooses a random patch when there's a tie. These random choices are governed by the random seed as well, so models runs can be reproducible.

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 may 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 (carriage return)
- `\t` = tab
- `\"` = double quote
- `\\` = backslash

Turtle shapes

In StarLogoT, turtle shapes were bitmaps. They all had a single fixed size and could only rotate in 45 degree increments.

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.

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.

Use the Shapes Editor to create your own turtle shapes. For more information, see the Shapes Editor section of this manual.

Code Examples: Breeds and Shapes Example, Shape Animation Example

File I/O

In NetLogo, there are a set of primitives that give the user 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 can be read into NetLogo. On the other hand, writing allows data to flow out of NetLogo and into a file.

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

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

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

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.)

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
```

Letting the user choose

It should be noted also that the user-choose-directory, user-choose-file, and user-choose-new-file primitives are useful when you want the user to choose a file or directory for your code to operate on.

Shapes Editor Guide

The Shapes Editor allows you to create and save turtle designs. NetLogo uses fully scalable and rotatable vector graphics, which means it lets you create designs by combining basic geometric figures, 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 create a new shape, edit, copy, or delete an existing shape, or import shapes from other models.

Creating and Editing Shapes

Pressing the **New** button, will cause a new shape to be created and the editing window will appear, allowing you to name and make the shape. The shape you make will appear in the main drawing area and in the three smaller preview areas found on the left side of the editing window.

The preview areas show your shape at different sizes as it might appear within your model, as well as how it looks while rotating. The rotatable feature can be turned off if you want a shape that does not rotate as the turtle's heading changes.

There are four drawing tools (select from four button icons), each of which can be filled with color or not, and a background grid to guide you. You can move and size these basic shapes with the mouse pointer as a drawing tool.

The basic geometric figure created last will fall on top. Geometric figures can be removed one at a time with the **Remove Last** button. The **Remove Last** and **Remove All** buttons are the only way of correcting mistakes −− you can't go back to move or alter a geometric figure after it's been drawn. If you make a mistake drawing a geometric figure, to fix it, remove it and then redraw it.

Geometric figures that use the *key color* (selected from a drop−down menu −− the default is gray) will change according to the value of each turtle's *color* variable in your model. Figures filled with any other color (selected from the palette at the top) will stay that color regardless of each turtle's *color*. For example, you could create cars that always have yellow headlights and black wheels but have bodies of different colors by drawing the bodies in the *key color*.

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 all the detail. Simple, bold shapes are best.

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 Shapes Editor's list along with the "default" shape.

If you want to use a shape from another model in this model, you must first import the shape into this model. Press the **Import** button to select a NetLogo model from which to import one or more shapes. Once you have chosen a model, a list of that model's shapes will appear. Choose as many of these shapes as you like to import into the original model and press the **Import** button to import

the shapes.

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:

```
crt 50
```

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

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

Voila! Rabbits! Note the use of double quotes around the shape name.

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

BehaviorSpace Guide

This guide is broken up into three parts:

- **BehaviorSpace: Old and New**: An explanation of how BehaviorSpace changed between NetLogo 1.3 and 2.0.
- **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.

BehaviorSpace: Old and New

For NetLogo 2.0, the old BehaviorSpace tool from NetLogo 1.x was replaced with a new, rewritten version which, though functional, is still under development.

The new version does not yet include all the functionality of the old one. The biggest difference is that it does not include any of the data analysis capabilities the old version had. Instead, it is assumed that you will use other software (such as a spreadsheet program or scientific visualization tool) to analyze your results.

However, the new version already has the following advantages over the old one:

- you can vary any global variables, not just sliders, but also switches and choices, and any variable declared in the procedures tab, and the values they range over can be any values, not just numbers anymore
- you can enter arbitrary code for "setup" and "go" now; you're not tied to using buttons that exist in your model
- you can vary variables in any order you want, rather than BehaviorSpace choosing the order for you
- you can collect any result type you want, not just numbers anymore

The documentation for the the new version is still sketchy; if you have questions, please write feedback@ccl.northwestern.edu.

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.

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 or scientific visualization application, and explore.

The idea behind BehaviorSpace is that the way to truly understand a model is to run it multiple times with different parameter (slider) settings in order to see the whole range of behaviors the system is capable of producing. Only then is it possible to investigate when and why certain behaviors arise. Isolated trials are insufficient for this purpose because you have no reason to assume that the model will always demonstrate the particular behaviors you see. It's like eating in one restaurant in New York and then claiming you've seen all that the city has to offer. By enabling you to explore the entire parameter space of a model, and thus its entire space of behaviors, BehaviorSpace can be a powerful tool for model understanding.

How It Works

To begin using BehaviorSpace, open your model, then choose the BehaviorSpace item on NetLogo's Tools menu. A small window will appear containing "Edit Experiment Setup" and "Run Experiment" buttons.

Press "Edit Experiment Setup" to begin setting up your experiment.

Setting up an experiment

In the dialog that appears, you'll need to specify the following information:

Vary variables as follows: This is where you specify which settings you want varied, and what values you want them to take. Settings can be sliders, switches, choices, or any global variable in your model. 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 steps 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 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.

You can vary as many settings as you want, but you must vary at least one.

Measure runs using this reporter: This is where you specify what data you want to collect from each run. You must enter a NetLogo reporter that reports the value you want. For example, if you wanted to record how the population of turtles rose and fell during each run, you would enter:

```
count turtles
```

You may only enter one reporter, but it is still possible to collect multiple measurements, by storing the measurements in a list. For example, if you wanted to record the populations of three different breeds of turtles, you could enter:

```
(list (count frogs) (count mice) (count birds))
```

Set up model with these commands: This is where you enter the commands that 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.

Step model with these commands: This is where you enter the commands that will be run to advance to the model to the next "step". Typically, this will be the name of a procedure, such as `go`, but you may include any commands you like.

Stop after this many steps: This lets you set a maximum length for each run. If you don't want to set any maximum, but want the length of runs to be controlled by the next setting setting instead, enter 0.

Stop if this reporter becomes true: This lets you do model runs of varying length, ending each model 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, enter `false` here.

Running an experiment

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

Experiment" button.

A window will appear, titled "Running Experiment". In this window, you'll see a progress report of how many runs have been completed so far and how much time has passed. If the reporter you entered for "Measure runs using this reporter" reports a number, then you'll see a plot of how that number varies over the course of each run.

You can watch your model run in the main NetLogo window. (If the "Running Experiment" window is in the way, just move it to a different place on the screen.) The graphics window 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" window to turn the updating off. This will make your model run faster, so you'll get your final results sooner.

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. BehaviorSpace will then prompt you for the name of a file to save the results to. The default name is "behaviors.csv". You can change this 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.

If you open the results file in a program that displays it in tabular format, you'll see that each individual model run occupies a single column. At the top of the column, you'll see summary information about the run, such as what the settings were for that run and how many ticks the run lasted. If you used a numeric reporter to measure your runs, then you'll see the min, max, and mean values of that reporter during the run. Finally, in the "all run data" section, you'll see what the value of your reporter was at every time step during the entire run.

Conclusion

That concludes the tour of the features of BehaviorSpace. The tool is still under development, so we'd like to hear from you about what what additional features would be useful to you in your work. Please write us at feedback@ccl.northwestern.edu. lope of the data always remains the same. The higher the weight selected in the **relative weight menu**, the less this menu selection matters, because that means slope matters increasingly more than error. Error is determined by attempting to fit the data to either a line or a curve, depending on which is chosen, and then finding the Least−Square Error involved in doing so.

- Fitting data to a line means finding a function of the form y = ax + b that best approximates the data.
- Fitting data to an exponential curve means finding a function of the form y = ae^(bx), where e~=2.71828, that best approximates the data.
- If 'exponential' is selected and any run has both positive and negative data values, then a warning dialog is brought up saying that the data cannot be fit to an exponential curve, and 'linear' is automatically selected −− this occurs because exponential curves of this form can be either above the x−axis (when a is positive) or below the x−axis (when a is negative), but not both.

Slope value menu: Determines whether it is "better" for the data to be increasing (the more

positive the slope the better), decreasing (the more negative the better), or constant (the closer to 0 the better)

Relative weight menu: Determines the relative importance of the slope of the data compared to how well it fits a line or exponential curve (see **slope type menu**). If 60 is chosen, then 60% of the data's rank comes from its slope (how much it is increasing, decreasing or constant) and 40% comes from line and curve−fitting error. So, a value of 100 means only slope is considered, while a value or 0 means only the error is considered.

- If you do not care about seeing strictly linear or exponential behavior in the run data, you should use high relative weight values, and otherwise use low ones.

Fitness landscape: Consists of a grid of squares, each of which represents the model run whose slider values were the current values of the constant sliders below and the ith and jth values of the two sliders on the grid's axes, where 'i' is the horizontal index of the square and 'j' is the vertical. The entire grid thus represents *all* runs in which the constant sliders had their current values (the current values being the ones displayed on the **set of slider components held constant** below the grid). This grid earns the name fitness landscape because the squares of which it is composed each has a fitness value, expressed by the color of the square, enabling you to see a landscape of colors denoting regions of high and low fitness −− that is, sets of neighboring near−black and near−blue squares. The landscape represents a 2−dimensional plane in the multi−dimensional parameter space of the model, since it accounts for all possible values of two sliders, while the rest are held constant. By changing the values of any constant sliders you are shifting that 2−D plane through the parameter space, and by selecting new sliders to be the axes, you are selecting a new plane entirely (that is, a plane through different dimensions).

- As you move your mouse over different squares in the grid, the data from those runs will appear in the **behavior plot** to the left, in addition to the line or curve that best fits the data if the **add best−fit−line checkbox** is selected.
- As your mouse passes over a square in the landscape, information about the model run that square represents appears above the landscape. This information consists of the value that each of the two selected sliders had during that run, and the overall fitness of that run.
- The fitness of a run is an estimation of how "good" the run was according to the criteria you select in the menus listed above. When the **slope checkbox** is selected, fitness derives from the slope of the data as well as how well it fits a linear or exponential curve. You can control the relative importance of these factors with the **relative weight menu**. When the **point statistic checkbox** is selected, the value that you see above the landscape is not the fitness of the run, but the value of the chosen statistic. For instance, the minimum value of the run is shown when "minimum value" is selected in the **point statistic type menu**. This value is *not* necessarily the fitness of the run, however, since the "goodness" of a particular value changes when you make a new selection in the **point statistic value menu**.
- The grid has a maximum height and width, which is reached when there are more than 5 squares in either direction, and cannot be resized.
- Squares turn gray when there is insufficient information to construct a grid −− this happens when either axis has not been assigned a slider (one axis is sufficient if a model only has one slider).

Set of sliders components held constant: These sliders consist of all those not present as axes of the **fitness landscape**. Since all the model runs represented on the landscape have the same value for these sliders −− namely, the one currently selected on the slider components −− they are considered held constant for the landscape. By changing the value of a single one of these sliders

the whole landscape changes, because it now represents a whole new set of runs. If you changed a slider from 10 to 12, say, then all the runs in which that slider had the value 10 are replaced by those in which it had the value 12.

- Changing the value of these sliders does *not* affect the value of the sliders on the main interface.
- The only values the sliders can take are those that they actually had in the experiment.

Behavior plot: Displays the data from individual runs as you move the mouse over their corresponding squares on the **fitness landscape**. It's x−axis is time clicks and its y−axis is behavior −− that is, the value reported by the **behavior reporter** entered in the setup window.

- The best fit line or curve can be added along with the data if the **add best−fit−line checkbox** is selected.
- Run data can be continually collected in the plot and superimposed on each other by selecting the **superimpose plots checkbox**.
- Only 14 plot pen colors are used, so if more than 14 data sets are superimposed at a single time, the system will begin reusing colors for the new pens.
- Automatically scales to show all data.

Add best−fit−line checkbox: Adds the line or exponential curve that best fits the data to the **behavior plot**, depending on whether 'linear' or 'exponential' was selected in the **slope type menu**.

Superimpose plots checkbox: Stops old data from being removed from the **behavior plot** when new data is added, resulting in increasing sets of data being superimposed on each other. Unselected, the checkbox causes all data to be removed.

Export Plot button: Saves the data currently being displayed in the **behavior plot** to a file.

Export Behavior Data button: Saves the behavior data gathered during all of the model runs to a file. The data saved for each model run includes: the slider settings for that run; how many time ticks the run lasted; the minimum, maximum, average, and final value for the behavior reporter; and the value of the behavior reporter at each time tick during the run. (Note that the fitnesses are not exported, only the raw behavior data. In a future version of BehaviorSpace the fitness data will be exported as well.)

Note: The export buttons create files 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. −−>

HubNet Guide

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 TI−83+ calculator or a networked computer.

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

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

About HubNet

HubNet Types

Currently, there are two types of HubNet available. One, called Calculator HubNet, was created in conjunction with Texas Instruments and makes use of the TI−Navigator system. This type of HubNet uses TI−83+ graphing calculators as the clients, the devices used to control a portion of a NetLogo model.

The second type of HubNet is Computer HubNet. This is adapted from the calculator version and uses laptop or desktop networked computers as the clients.

In the future, we hope to add more types of HubNet that support other types of clients such as PDA's (Personal Digital Assistants).

What do I need to get started?

Calculator HubNet

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

> **NOTE:** TI−Navigator recently became commercially available. To learn more about the TI−Navigator system, visit Texas Instruments' site. However, HubNet does not work with this version of TI−Navigator; it might only work with a future release version of TI−Navigator.

Computer HubNet

- **A networked computer with NetLogo installed and, optionally, an attached projector for the leader.** This computer will run NetLogo and act as the server for the HubNet Clients. The projector, if attached, will project the simulation for class viewing.

  > **NOTE:** By default, the NetLogo Graphics Window and plots are not mirrored to the HubNet clients. For instructions on how to make a HubNet model mirror the NetLogo Graphics Window or plots to clients, please refer to the Computer HubNet portion of the Running an activity section.

- **A networked computer with NetLogo installed for each participant.** Be aware that Computer HubNet has only been tested on LANs, and never via dial−up connections or WANs. It performance is unknown using these types of networks.

First−time NetLogo user?

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, if you choose.

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 should be familiar with the basics of NetLogo.

Teacher workshops

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

Getting Started With HubNet

Using NetLogo

We recommend that you become familiar with NetLogo itself before using the HubNet technology.

You can become familiar with NetLogo by trying out some of the models in the Models Library. Open the Models Library from the File menu in NetLogo. Then click on a model that you want to try and press the Open button. The Information tab in each of the models gives background information and instructions.

Other sections of the NetLogo User Manual may be helpful when learning NetLogo. We suggest that beginning users focus on the section Tutorial #1: Running Models.

If you have any questions about NetLogo, feel free to E−mail us. You can reach us at feedback@ccl.northwestern.edu.

HubNet Activities

Below are the current HubNet activities that are fully developed. Be aware that some of these models have only been implemented for one type of HubNet. For many models, you will find its educational goals and suggested ways to incorporate them into your classroom in the Participatory Simulations Guide which can be found on the Participatory Simulations Project web site.

- Disease −− A disease spreads through the simulated population of students.
- Elevators −− Student−controlled elevators demonstrate the relation of velocity and position.
- Function Activity −− Students experience the concept of a function by forming themselves into graphs.
- Gridlock −− Students use traffic lights to control the flow of traffic through a city.
- People Molecules −− Using CBR's (Calculator Based Range−finders), students use their bodies to represent gas molecules.
- Polling −− Ask students questions and plot their answers.
- Regression −− As students move on−screen, they see the best−fit line of their positions.

NOTE: In addition to the discussion of learning goals and classroom techniques, these materials also contain step by step instructions and screen shots. As of December 2003, many are out of date and no longer match the actual activities in many respects. We are working on updating them. In the meantime, please use these materials for the discussions, but for step by step instructions, rely instead on the QuickStart Instructions built into the activities (see next section).

Running an activity

You'll find the HubNet activities in NetLogo's Models Library, under the HubNet Calculator Activities and HubNet Computer Activities folders.

In each of the activities, you'll see a box on the screen labeled "QuickStart Instructions". It contains step by step instructions on how to run that activity. Click the "Next>>>" button to advance to the next instruction.

We suggest doing a few practice runs of the activity before trying it in front of an actual class.

If you have any questions about running the activities, feel free to E−mail us. You can reach us at feedback@ccl.northwestern.edu.

Calculator HubNet

When you open the first Calculator HubNet activity for each session of NetLogo, you will be prompted by the TI−Navigator Login dialog. This prompts you to enter information (such as User Id or Password) that is necessary for connecting to the TI−Navigator system and running the HubNet activity. If you don't actually want to run the model, just press the Cancel button.

For more information about how to log in to the calculators and other details of using Calculator HubNet, please refer to the Participatory Simulations Guide which can be found on the Participatory Simulations Project web site.

Computer HubNet

Setting up a Computer Activity

Opening a Computer HubNet model will cause NetLogo to start a server which will allow people to join the activity. A dialog will prompt you to enter a unique name that will help participants identify the activity you are running. This name will appear if the server is discovered on the clients. While this is not necessary, entering a name is recommended since it can help reduce confusion over which activity participants should enter. You should then follow the instructions for the particular model found in the QuickStart Instructions monitor. Most models will require you to press a forever button, often called GO. You, as the leader, should notify everyone that they may join.

For them to join, you should give them the IP address for the computer that is running NetLogo. The IP address of the server can be found in the HubNet Control Center, which can be opened by choosing the HubNet Control Center option in NetLogo's Tools menu. Every user will have to enter this IP address to be able to enter the activity. On some systems, the HubNet Client will automatically detect all the Computer HubNet activities currently being run. On these systems, instead of typing in the IP address, everyone can select the activity from a list. Each item in the list should contain the unique name you entered in the dialog when you first opened the HubNet model followed by the name of the activity followed by the IP address of the computer running NetLogo.

Every participant will also have to enter a unique user name. (If a participant types in a user name that someone else is already using, they will be asked to choose a different name.)

Joining a Computer Activity

Once the leader has informed you that you can join, you should open up the HubNet Client application. This will prompt you for the IP address of the computer running NetLogo. Enter the IP address that the leader gave you. Alternately, if your network supports the "server discovery" feature, you may instead choose the activity from the list shown.

Regardless of how you choose the Computer HubNet server, you will also have to enter a User Name. Please enter one. The leader will give directions on how to choose one. If no one else is using the one you enter, NetLogo will send you the client interface and you can start playing. If you didn't choose a unique User Name, your client application will display a message indicating this.

You should keep trying new User Names until you are able to enter. A good method for choosing a unique User Name is to use your first and last name together. For example, if your name is Joe Smith, you might want to choose the User Name, joesmith.

HubNet Control Center

One feature of Computer HubNet is the HubNet Control Center. You can open this using the HubNet Control Center option in the Tools Menu when a Computer HubNet Activity is open and the server is running. The Control Center shows you useful information, such as the IP address of the computer, who is logged in, etc. It also allows you to disconnect clients from the activity and send out instructions or messages for all the clients to read.

In addition to these features, the HubNet Control Center allows you to control whether debugging output is on or off. Debugging output is useful when creating your own HubNet activity or to keep track of which participant sent what data when. Any output is sent to the log.txt file on Windows machines, or the Console to Macs. The Control Center also allows you to control whether the NetLogo Graphics Window and plots are mirrored on the clients. These options are useful if there are a lot of changes to the Graphics Window or plots or if you just don't wish the clients to see what is happening on the clients. By default, these options are turned off since they are not suited for all models.

HubNet Authoring Guide

If you wish to learn more about authoring or modifying HubNet activities, you should look at the HubNet Authoring Guide.

Computer HubNet Troubleshooting Tips

Here are some things that have been known to go wrong and ways of fixing or working around them.

I have tried quitting a HubNet Client, but it just sits there and won't quit no matter how many times I try.

You will have to force the client to quit. On Macs, force quit the application (Command+Option+Esc). On Windows, open the Windows Task Manager (Ctrl+Alt+Delete), select Hubnet and press 'End Task'. Then you should inform the leader of the simulation to use the HubNet Control Center to kick your client out of the simulation.

We believe that we have resolved all instances of these problems with the exception of the client trying to quit when it has lost its network connection. If you encounter this problem and still have a network connection, please let us know.

When I open a Computer HubNet Activity, I got a dialog that said:

```
Could not start the Computer HubNet server. Another program may be using the port that is needed.
```

Another program (perhaps another copy of NetLogo) is using a port or some other resource that HubNet needs. Check to see if you have another copy of NetLogo open that is running Computer HubNet. If you do, quit it and then try reopening the model. If you don't, you can try quitting other

programs and reopening the model until you are able to start the Computer HubNet server successfully.

Somehow two or more HubNet Clients are controlling the same agents in the simulation and they aren't supposed to. What can I do to fix this problem?

Remove all of the clients from the simulation by either by having the people exit the activity or by using the Kick Client feature in the HubNet Control Center to forcibly remove them. If you wish them to still be in the simulation, they all should log back in with different user names.

I had run and then stopped a HubNet model. I changed windows or minimized NetLogo or did something else to obscure or redraw the Graphics Window. When I came back to the NetLogo, the Graphics Window was all gray. What happened? Is all my data lost?

Most likely what has happened is that the Graphics Window is not longer being updated because the `no-display` primitive was called, or the freeze/unfreeze display button in the Graphics Window Control Strip is pressed. Try using the `display` primitive in the NetLogo Command Center or toggling the freeze/unfreeze display button.

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.

We have had reports that if a computer goes to sleep while running Computer HubNet (either the HubNet Client or the server), sometimes things do not work after the computer wakes up. Please send us the error message along with the situation that caused it. However, we suggest that people keep their computers from sleeping while running an activity.

In my HubNet client, I see the same server displayed multiple times in the list of available servers. Which one should I pick?

These all refer to the same server. Pick any of them to select that server.

The clients' Graphics Windows are not mirroring the NetLogo Graphics Window properly. What is the problem?

There are a few things that could be wrong.

- Be sure that the clients actually have a Graphics Window. The clients won't mirror the model's Graphics Window, if they don't have one themselves.
- Ensure that Graphics Window mirroring is on, by verifying that the Graphics Mirroring mirroring option is checked in the HubNet Control Center.
- Make sure that the display is on in the model. If the display isn't on, the client Graphics Windows will not be updated.
- Verify that the size of the Graphics Window in the NetLogo model and the Graphics Window on the clients are not exactly the same, the Computer HubNet clients will not mirror the model's Graphics Window.

If you verified that none of the above conditions apply, and mirroring is still not working for you, please contact us. We may be able to help you solve the problem, or it might be a bug.

If your situation does not fall into one of the above problem descriptions, please send us a bug report.

Known Computer HubNet Issues

If HubNet malfunctions, please send us a bug report. See the "Contact Us" section for instructions.

Known bugs (all systems)

- 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.
- Increasing the number of clients can increase the response time for clients.
- There are situations that require the clients to force quit the client application. For example:
    - ♦ If the network connection is lost by the clients or the server, even just for a few minutes, the clients and/or the server might need to be forcibly quit.
- 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 never via dial−up connections or WANs.
- Sometimes servers displayed in the list of servers in the HubNet client may be repeated. When this occurs, it doesn't matter which one you choose.

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

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

In the model, the modeler uses a set of commands to set the model up to use a type of HubNet, extract data from and send data to the Navigator system or the computer clients. Below you can find explanations of each of the NetLogo primitives used to do these tasks.

Setup

In order to make a NetLogo model into a HubNet one, it is necessary to establish a connection with the Navigator server in the case of Calculator HubNet and to start a HubNet server in the case of Computer HubNet. For Calculator HubNet, it is also necessary for NetLogo to tell the Navigator server what variables to send to NetLogo. For all forms of HubNet, it is also necessary to inform NetLogo what the client interface is. All these tasks are done with the following primitives:

`hubnet-reset`

This 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 this is the first time called for this NetLogo session and you are using Calculator HubNet, the TI-Navigator Login dialog appears prompting you to input the appropriate information to be able to log into the Navigator system. Once you press the Login button, NetLogo will attempt to log you into the TI-Navigator system and start HubNet. If you successfully log into the Navigator system, you will not be prompted by this dialog again as long as this session of NetLogo remains open.

If you are using Computer HubNet, you will be prompted by a dialog asking for a unique name for your computer. This is an optional identifier to help make servers discovered on the client more different and unique. If you don't wish to enter a name, just press the Cancel button.

hubnet−set−client−interface client−type client−info

If *client−type* is "TI−83+", *client−info* is a list containing two items. The first item is a string containing the name of the activity to enable on the TI Navigator web site. `hubnet−set−client−interface "TI−83+"` notifies the user to enable this activity. The second item is a list of the tags for which to check. The tag list sets which variables NetLogo expects from the calculators. NetLogo will only check for these variables and will ignore all others. Currently, the valid types that NetLogo will be able to receive from the calculator are the following:

- ◊ 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`

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 "TI−83+" [ "AAA − Gridlock 1.3" ["L1" "LOCS"] ]
;; notifies the user to enable the activity AAA − Gridlock 1.3 and looks for the
;; calculator lists L1 and LOCS on the Navigator server

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

This primitive must be called before you try to use HubNet calling the `hubnet−reset` reporter so NetLogo can know which type of HubNet you are going to be using.

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

Data extraction

The data extraction primitives are:

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.

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.

Note: Since NetLogo must send the data to the Navigator server in Calculator HubNet, it is not currently possible to send data from NetLogo directly to only an individual calculator. However, once the server has the data, any connected calculator can grab it. This is done using the calculators' communication facilities, rather than through NetLogo.

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-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*.

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

For both of these primitives, when using Calculator HubNet, *value* may be a number, a string, a list of numbers, or a matrix (a list of lists) of numbers. For 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	`hubnet-broadcast "A" 3.14`	`hubnet-send "jimmy" "A" 3.14`
string	`hubnet-broadcast "STR1" "HI THERE"`	`hubnet-send ["12" "15"] "STR1" "HI THERE"`
list of numbers	`hubnet-broadcast "L2" [1 2 3]`	`hubnet-send hubnet-message-source "L2" [1 2 3]`
matrix of numbers	`hubnet-broadcast "[A]" [[1 2] [3 4]]`	`hubnet-send "suzy" "[A]" [[1 2] [3 4]]`
list of strings (only for Computer HubNet)	`hubnet-broadcast "user-names" [["jimmy" "suzy"] ["bob" "george"]]`	`hubnet-send "teacher" "user-names" [["jimmy" "suzy"] ["bob" "george"]]`

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 calculator sends and receives data by storing a set of parameters in the string `Str0`. Depending upon what type of data you are trying to send or receive, `Str0` will have different values. For instance, if the modeler wanted to create and send a list of numbers in the list `L1`, it would be done as follows. Set the value of the list to some numbers (in this case, 20, `A`, and `B` where `A` and `B` are number variables that are set previously in the calculator code). Then write:

```
{20,A,B}->L1
"1 L1"->Str0
Asm(prgmSENDVAR)
```

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.

You can also receive data from the NetLogo model. To do this, use the following calculator code:

```
"4 Str6 1"->Str0
Asm(prgmGETVAR)
```

Let's take a look at how the values of the string `Str0` are set. The first input in the string represents the type of variable that you are trying to get. Since we are trying to get a string (`Str6`), we give the first input the value 4. (See below for the values of legal data types.) The second input is the variable in which you would like to save the data received. In this case, we want to save the data to the variable `Str6`. The third input tells how you would like to save this data into this variable. (See below for values of valid commands.) It should be noted that for sending a variable, the command defaults to 0, i.e. no command.

data type	associated value
number	0
list of numbers	1
matrix of numbers	2
string	4

command number	command explanation
0	No Command
1	Collate (Lists into a matrix, reals into a list, append strings)
2	Teacher Variable
4	Append Lists

You should note that you must always save the information into the variable `Str0` when you are sending or receiving information from the calculators. You can't use any other variable.

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, choices, 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 funny things to happen. 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.

Be aware that if you wish to have a Graphics Window in the client for a model, the Graphics Window 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 Graphics Window on the client will not display information sent by the server.

If you wish to make a client without a Graphics Window 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.

Graphics Window Updates on the Clients

Currently, there are two ways of sending the clients the Graphics Window. The first way is done automatically by NetLogo and HubNet when Graphics Window mirroring is enabled. Whenever a patch or turtle is redrawn in the NetLogo Graphics Window, it will be redrawn on **all** the clients. 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 using the `no-display` and `display` primitives.

A second way of sending the clients the Graphics Window is to use the `hubnet-broadcast` and `hubnet-send` primitives. `hubnet-broadcast` and `hubnet-send` both send the entire Graphics Window to the clients instead of just the patches that need to be redrawn. This makes them less efficient, but sometimes better for a particular model. To send the Graphics Window to the clients using this scheme, you must use the following NetLogo code:

```
hubnet-broadcast "Graphics Window" "Graphics Window"
```

to send to all the logged in clients.

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

```
hubnet-send user-name-list "Graphics Window" "Graphics Window"
```

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.

It should be mentioned that if there is no Graphics Window in the clients or if the Mirror Graphics Window on Clients checkbox in the HubNet Control Center is not checked, then no graphics updates are sent to the clients.

> **NOTE:** Since Computer HubNet is still in the process of being developed, the way that graphics updates take place, including the syntax of sending the NetLogo Graphics Window using the `hubnet-broadcast` and `hubnet-send` primitives, may change in a future release.

Plot Updates on the Clients

Plots on the clients are updated in the following way. If a change occurs on a NetLogo plot 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.

It should be mentioned that 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 Graphics Window on Clients

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

The NetLogo API Specification contains further details.

Using Extensions

An extension (which includes one or more custom commands and reporters) is stored in a "JAR" file (short for "Java Archive"). For a model to use an extension, it must declare what JARs it uses and where the JARs are stored.

For this purpose, the NetLogo language includes a keyword, `__extensions`, which may appear only 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 a pathname to a JAR file. For example:

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

In this example, no directory name is given, only a pathname, so NetLogo will look for the JAR in the same directory that holds the model. You may also use pathnames relative to that directory, or absolute pathnames:

```
__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 JAR. 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 JAR 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 pathname seperately. For example,

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

Third party JARs

Some extensions depend on code stored in a separate JAR. (Many Java libraries are distributed as JAR files).

If your model uses an extension that requires extra JARs in addition to the extension itself, copy the extra JARs into the "extensions" folder of your NetLogo installation. (If the folder doesn't exist, create it.) Whenever an extension is imported, NetLogo will make all the JARs in this folder available to the extension.

Extension authors are responsible for providing third−party JARs and for documenting the installation requirements for the NetLogo user.

Applets

Models saved as applets (using "Save as Applet" on NetLogo's File menu) cannot make use of extensions. (We plan to fix this in a future release.)

Writing Extensions

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 1 through *n*. (Of course, you could easily do this just in NetLogo; it's only an example.) We'll assume you have experience programming in Java.

Writing 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`.

The `Reporter` interface requires that we implement the following methods:

```
Reporter newInstance (String name);
Object report (Argument args[], Context context)
  throws ExtensionException;
```

`Reporter` extends `org.nlogo.api.Primitive`, which specifies these additional methods:

```
String getAgentClassString();
Syntax getSyntax();
```

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

```
import org.nlogo.agent.LogoList;
import org.nlogo.api.Argument;
import org.nlogo.api.Context;
import org.nlogo.api.ExtensionException;
import org.nlogo.api.Reporter;
import org.nlogo.api.Syntax;

public class IntegerList implements Reporter
{
    public Reporter newInstance(String name) {
        return new IntegerList();
    }
    public String getAgentClassString() {
        return "OTP";
    }
    public Syntax getSyntax() {
        return Syntax.reporterSyntax
          (new int[] {Syntax.TYPE_NUMBER},
           Syntax.TYPE_LIST);
    }
    public Object report(Argument args[], Context context)
        throws ExtensionException
    {
        LogoList list = new LogoList();
        int n = args[0].getIntegerValue();
        if (n < 0) {
            throw new ExtensionException
              ("input must be positive");
        }
        for (int i = 1; i <= n; i++) {
            list.add(new Integer(i));
        }
        return list;
    }
}
```

Some things to notice:

- NetLogo will not call the constructor directly; instead, it will call `Reporter.newInstance(...)`.
- 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(...)`.

Writing 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, use 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, `ExampleManager.java`:

```
import org.nlogo.api.DefaultClassManager;
import org.nlogo.api.PrimitiveManager;

public class ExampleManager 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.

Writing 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`:

```
Extension-Name: example
Class-Manager: ExampleManager
NetLogo-Version: 2.0.1
```

Creating a NetLogo Extension 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 ExampleManager.java
```

Then create a JAR containing the resulting class files and the manifest. For example:

```
$ jar cvfm example.jar manifest.txt IntegerList.class ExampleManager.class
```

For information about manifest files, JAR files and Java tools, please see java.sun.com.

Using your extension

To use our example extension write, at the top of the Procedures tab:

```
__extensions [ "example.jar" ]  ;; assumes JAR, model in same directory
```

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:

```
O> show first-n-integers 5
observer: [1 2 3 4 5]
```

Extension development hints

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.

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.

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

The NetLogo API Specification contains further details.

An Example

Here is a small but complete program that starts NetLogo, opens a model, moves a slider, and runs the model for a few seconds:

```
import org.nlogo.app.App;
import org.nlogo.compiler.CompilerException;
import java.awt.EventQueue;

public class Example {
    public static void main(String[] argv) {
        App.main(argv);
        try {
            EventQueue.invokeAndWait
                ( new Runnable()
                    { public void run() {
                        App.app.open
                            ("models/Sample Models/Earth Science/"
                             + "Fire.nlogo");
                    } } );
            App.app.command("set density 62");
            App.app.command("setup");
            for(int i=0; i
```

In order to compile and run this, `NetLogo.jar` (from the NetLogo distribution) must be in the classpath.

Note the use of `EventQueue.invokeAndWait` 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 `commmand()`, 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.

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.

We are also working on making it possible to run NetLogo "headless", that is with no GUI at all, but at present this is not possible.

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 you 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.

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.

General

- **What language is 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 are 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 written a model of <x>?**
- **Are NetLogo models runs scientifically reproducible?**

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?**

Applets

- **I tried to run one of the applets on your site, but it didn't work. What should I do?**

Usage

- **How do I change the number of patches?**
- **How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?**
- **Can I import a graphic into NetLogo?**
- **My model runs slowly. How can I speed it up?**
- **I want to try HubNet. Can I?**
- **Can I run a NetLogo model from the command line? Can I run it without a GUI?**
- **Can I have more than one model open at a time?**
- **Can I copy or save a picture of the graphics window?**
- **Can I make a movie of my model?**

Programming

- **How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?**

General

What language is 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/ps/. Center for Connected Learning and Computer−Based Modeling, Northwestern University. Evanston, IL.

How do I cite a model from the Models Library in an academic publication?

Wilensky, U. (year). Name of Model. URL of model. Center for Connected Learning and Computer−Based Modeling, Northwestern University. Evanston, IL.

To determine the URL for a model, visit our web−based version of the Models Library and click on the name of the model. An example model URL is: http://ccl.northwestern.edu/netlogo/models/PepperedMoths.

To determine the year, open the model from the NetLogo application and look in the copyright information at the bottom of the Procedures tab.

What license is NetLogo released under? Are there are 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.

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. For example, NetLogo 2.0.1 now 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 also incorporates almost all of the extended functionality of our earlier StarLogoT.

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

StarLogoT	NetLogo	Features
X	X	Fully programmable
X	X	Language is Logo dialect extended to support agents and concurrency
X	X	Interface builder w/ buttons, sliders, switches, choices, monitors, and text boxes
X	X	Info area for annotating your model
X	X	Powerful and flexible plotting system
X	X	Agent Monitors for inspecting agents
X	X	Export and import model function (save and restore state of model)
	X	Cross−platform: runs on MacOS, Windows, Linux, et al
	X	Models can be saved as applets to be embedded in a web page

	X	Unlimited numbers of agents and variables
	X	Double precision arithmetic
	X	Simplified language structure
	X	"Agentsets" make many programming tasks easier
	X	Syntax−highlighting code editor
	X	Rotatable and scalable vector shapes for turtles
	X	Exact on−screen turtle positioning
	X	Redesigned user interface
	X	Text labels for turtles and patches
	X	Many new primitives
	X	API for user extensions
	X	API for controlling NetLogo from outside
	X	BehaviorSpace: a tool used to collect data from multiple runs of a model
	X	HubNet: participatory simulations using networked devices

Has anyone written 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 2.0 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, 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.

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 it 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)".

Applets

I tried to run one of the applets on your site, but it didn't work. What should I do?

The applets on the CCL web site have been upgraded to use NetLogo 2.0, which requires that your web browser support Java 1.4.1. Here's how to get the right Java:

- 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 application and run the models that way instead.
- If you're on Windows 98 or newer, you need to download the Java browser plug−in from http://java.sun.com/getjava/download.html.

- 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 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.

Usage

How do I change how many patches there are?

A quick method is to use the three pairs of black arrows in the upper left corner of the graphics window.

Another method is as follows. Select the Graphics Window 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 Graphics Window to edit it, or select it then double−click.)

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
```

- **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>-XX:+PrintJavaStackAtFatalState −Xms16M −Xmx512M</string>
```

- **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.

```
java −Xmx512M −jar NetLogo.jar
```

Can I import a graphic into NetLogo?

At present, this capability is not built in. We plan to add it in a future version.

In the meantime, though, there are some possible solutions. One is to convert the image data into a format that `import-world` can understand, then import it that way. Another is to read and translate the image data yourself using `file-open` and `file-read-characters`. (You may wish to first use another program to convert the image into a format that is easier to read that way, such as PBM.)

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 display update after each run" checkbox. This allows the graphics window to skip frames, which may speed up models which are display-intensive. (See the Buttons section of the Programming Guide for a discussion of this.)
- Use the display switch in the graphics control strip, or the `no-display` command, to turn graphics off temporarily. For example:

```
to go
  no-display
  ...
  ...
  display
end
```

- 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.

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?

Currently, there are two implementations of HubNet: "calculator HubNet" and "computer HubNet".

"Calculator HubNet" uses Texas Instruments calculators and requires TI's calculator network system, called Navigator. Unfortunately, the version of Navigator which is currently commercially available does not support HubNet. In the future, we hope that calculator HubNet will be supported on a commercially available version of Navigator.

"Computer HubNet" uses networked computers and does not require calculators or Navigator. Computer HubNet is included in the NetLogo download and can be used without additional materials. Please refer to the HubNet Guide for more information on computer HubNet.

Can I run a NetLogo model from the command line? Can I run it without a GUI?

At present there is no way to run without the GUI. We plan to add this in a future version. (On X11−based systems such as Linux, though, one possible workaround is to use X11's "virtual frame buffer" feature, which provides a virtual display for NetLogo to run on.)

To open a model on startup, you can just pass the −−open flag followed the pathname of the model on the command line as an argument to the executable. Examples:

- Windows:

```
"C:\Program Files\NetLogo 2.0\NetLogo 2.0.exe"
  −−open "C:\myfiles\mymodel.nlogo"
```

- Mac OS X:

```
"/Applications/NetLogo 2.0/NetLogo 2.0.app/Contents/MacOS/NetLogo"
  −−open "/Users/me/mymodel.nlogo"
```

- Linux et al:

```
cd netlogo−2.0
java −jar NetLogo.jar −−open "/home/me/mymodel.nlogo"
```

If you want the model to actually start running automatically once opened, you have two options.

The easiest option is to add a procedure to the Procedures tab called startup. For example:

```
to startup
    setup
    repeat 100 [ go ]
    export−world (word get−date−and−time ".csv")
  end
```

You can put any code you want in the startup procedure.

If you need to do something more sophisticated, or if you don't want to have to add code to your model, then a more advanced option is to use NetLogo's Java API for "controlling" NetLogo from external Java code. See the "Controlling" section of the NetLogo User Manual for details. Note that some light Java programming is required in order to take advantage of this API.

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 Macs, you'll need to duplicate the application in the Finder, then open the copy. (This will use only a very small amount of additional disk space, since most of NetLogo is actually in the NetLogo.jar file, which is stored outside the application "bundle.")

Can I save the contents of the graphics window? Of the interface tab

Yes, using "Export Graphics" on the File menu, or by right−clicking (on Mac, control−clicking) the graphics window, or using the `export−graphics` command.

You can also use "Export Interface" or the `export-interface` command to save an image of the entire interface tab.

Can I make a movie of my model?

NetLogo itself does not have this capability presently, but there are many third-party software packages that let you capture movies from any running application.

Another option would be to use the `export-graphics` or `export-interface` commands to save a series of snapshots of the model, then use some other tool to convert the snapshots into a movie file. This should produce smaller files than the other technique.

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 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 NetLogo 2.0, so if you have models you want to use it on, you will have to use NetLogo 1.3 to do the conversion, then open the model in 2.0.

If you need any help converting your StarLogo or StarLogoT model to NetLogo, please feel free to get in touch with us at feedback@ccl.northwestern.edu. The NetLogo Users Group is also a good resource for getting help from other users.

The NetLogo world is a torus, that is, the edges of the screen are connected to each other, so turtles and patches "wrap around". Can I use a different world topology: bounded, infinite plane, sphere, etc.?

Torus is the only topology directly supported by NetLogo, but you can often simulate a different topology without too much extra effort.

If you want the world to be a bounded rectangle, you may need to add some code to your model to enforce this. Often a helpful technique is to turn the edge patches a different color, so turtles can easily detect when they "hit" the edge. Also, there are "no-wrap" versions of primitives such as "distance" and "towards"; these should help.

If you want your turtles to move over an infinite plane, you can simulate this by having the turtles keep track of their position on the infinite plane, then hide the turtle when it goes "out of bounds". The Random Walk 360 model in the Models Library shows you how to code this.

Simulating a spherical or other topology might be difficult; we haven't seen a model that does this. (If you have one, please send it in!)

Does NetLogo have a command like StarLogo's "grab" command?

We don't have such a command, although we plan to add one -- or perhaps several! In the meantime, though, 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
  locals [candidate]
  without-interruption
    [ if mate = nobody
        [ set candidate random-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 0.799999999999 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.

How can I keep two turtles from occupying the same patch?

How to best accomplish this depends somewhat on the details of how your model works. The following techniques may be helpful:

- A turtle can test whether there are other turtles on the patch it is standing on with `any? other-turtles-here`.
- A turtle can test whether there are other turtles on the patch it is facing with `any? turtles-on patch-ahead 1`. This checks the patch that the turtle would land on if it executed `fd 1`. Note however that if the turtle's heading isn't a multiple of 90, then "fd 1" will not necessarily take the turtle to a new patch -- it might only move from one corner of a patch to the opposite corner of the same patch.

Sometimes code that looks like it should prevent two turtles from ever being on the same patch can fail because of unexpected interactions between the turtles as they all execute the code

concurrently. In these situations, appropriate use of `without-interruption` can be helpful.

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.

If you want a Moore or Von Neumann neighborhood of a different radius, or a different kind of neighborhood altogether, you can define it yourself, using the `at-points` primitive and/or other techniques. If the neighborhoods do not change over time, then the most efficient way to use them is to compute the neighborhoods only once, ahead of time, and store them in agentsets. See this URL for a discussion and example code: http://groups.yahoo.com/group/netlogo-users/message/377.

Can I connect turtles with lines, to indicate connections between them?

Yes. The usual technique is to create a new breed of turtle, whose shape is a line, and position and size that turtle appropriately so that it appears to connect the two turtles you want to connect.

We plan to support this more directly in a future version of NetLogo.

How can I convert an agentset to a list, or vice versa?

Here's how to convert an agentset to a list of agents:

```
values-from <agentset> [self]
```

And here's how to convert a list of agents to an agentset:

```
turtles/patches with [member? self <list>]
```

For a discussion of the whole issue of agentsets versus lists of agents, see:

- http://groups.yahoo.com/group/netlogo-users/message/652
- http://groups.yahoo.com/group/netlogo-users/message/655
- http://groups.yahoo.com/group/netlogo-users/message/656

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 ascending order by ID number. If you `ask patches`, the patches are scheduled for execution by row: left to right within each row, and starting with the top row.

If you `ask` a different agentset besides the set of all turtles or patches or a breed, then the execution order will vary according to how the agentset was constructed. The execution order is chosen deterministically and reproducibly, though, and will remain the same if you `ask` the same agentset multiple times.

In a future version of NetLogo, we plan to add an option for randomized scheduling.

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.

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.

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 − Display
HubNet − Input/Output − Files − List − String − Math − Plotting

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) *BREED*−at *BREED*−here *BREED*−on clear−turtles (ct) create−*BREED* create−custom−*BREED* create−custom−turtles (cct) create−turtles (crt) die distance distance−nowrap distancexy distancexy−nowrap downhill downhill4 dx dy forward (fd) hatch hideturtle (ht) home inspect is−turtle? jump left (lt) myself no−label nobody −of other−turtles−here other−*BREED*−here patch−ahead patch−here patch−left−and−ahead patch−right−and−ahead pen−down (pd) pen−up (pu) right (rt) self set−default−shape setxy showturtle (st) sprout stamp towards towards−nowrap towardsxy towardsxy−nowrap turtle turtles turtles−at turtles−from turtles−here turtles−on turtles−own uphill value−from

Patch−related primitives

clear−patches (cp) diffuse diffuse4 distance distance−nowrap distancexy distancexy−nowrap inspect is−patch? myself neighbors neighbors4 no−label 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 self sprout value−from

Agentset primitives

any? ask at−points *BREED*−at *BREED*−here *BREED*−on count histogram−from in−radius in−radius−nowrap is−agent? is−agentset? is−patch−agentset? is−turtle−agentset? max−one−of min−one−of neighbors neighbors4 one−of other−turtles−here other−*BREED*−here patches patches−from random−n−of random−one−of turtles with turtles−at turtles−from turtles−here turtles−on values−from

Color primitives

extract−hsb extract−rgb hsb rgb scale−color shade−of? wrap−color

Control flow and logic primitives

and end foreach if ifelse ifelse−value locals loop map not or repeat report run runresult ; (semicolon) stop startup to to−report wait while without−interruption xor

Display primitives

clear−all (ca) clear−graphics (cg) clear−patches (cp) clear−turtles (ct) display no−display no−label screen−edge−x screen−edge−y screen−size−x screen−size−y

HubNet primitives

hubnet−broadcast 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−set−client−interface

Input/output primitives

clear−output export−graphics export−interface export−output export−plot export−all−plots export−world get−date−and−time import−world mouse−down? mouse−xcor mouse−ycor print read−from−string reset−timer set−current−directory show timer type user−choice user−choose−directory user−choose−file user−choose−new−file user−input user−message 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−choose−directory user−choose−file user−choose−new−file

List primitives

but−first but−last empty? filter first foreach fput is−list? item last length list lput map member? modes n−values position random−n−of random−one−of reduce remove remove−duplicates remove−item replace−item reverse sentence shuffle sort sort−by 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 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 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

Built−In Variables

Turtles

breed color heading hidden? label label−color pen−down? shape size who xcor ycor

Patches

pcolor plabel plabel−color pxcor pycor

Other

?

Keywords

breeds end globals locals patches−own to to−report turtles−own

Constants

Mathematical Constants

e = 2.718281828459045
pi = 3.141592653589793

Boolean Constants

false
true

Color Constants

The allowable range of values for colors is 0 up to but not including 140. Each color ranges from black to white over a scale of ten. Thus the color red goes from black (10) to dark red (11) to red (15) to light red (19) to white (19.9999). The scale is discontinuous; 19.9999 is white, but 20.0 is black.

The available color names are listed below. (See also the rgb and hsb primitives.)

black = 0
gray = 5
white = 9.9999
red = 15
orange = 25
brown = 35
yellow = 45
green = 55
lime = 65
turquoise = 75
cyan = 85
sky = 95
blue = 105
violet = 115
magenta = 125
pink = 135

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!" ]
```

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, then write (− x).

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*]**

Takes a list of commands that will be run by the specified agent or agentset.

```
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
```

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.

See also forward, jump.

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 breeds.

Example:

```
breeds [cats dogs]
;; turtle code:
if breed = cats [ show "meow!" ]
set breed dogs
show "woof!"
```

breeds

breeds [*breed1 breed2* ...]

This keyword, like the globals, turtles−own, and patches−own keywords, can only be used at the beginning of a program, before any function definitions. It defines breeds and their associated agentsets.

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.

```
breeds [ mice frogs ]
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
```

See also globals, patches−own, turtles−own, *<BREED>*−own, create−*<BREED>*, create−custom−*<BREED>*, *<BREED>*−at, *<BREED>*−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

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

Kills all turtles, resets all global variables to zero, and calls clear-patches and clear-all-plots.

clear-all-plots

clear-all-plots

Clears every plot in the model. See clear-plot for more information.

clear-graphics
cg

clear-graphics

Kills all turtles and clears all patches. Combines the effect of clear-turtles and clear-patches.

clear-output
cc

clear-output

Clears all text from the output portion of the Command Center.

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−turtles
crt
create−<*BREED*>

create−turtles *number*
create−<*BREED*> *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−<*BREED*> form is used, the new turtles are created as members of the given breed.

---

create−custom−turtles
cct
create−custom−<*BREED*>
cct−<*BREED*>

create−custom−turtles *number* [*commands*]
create−custom−<*BREED*> *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.

```
breeds [canaries snakes]
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 screen updates take place until the commands are done. This ensures that the new turtles are never drawn on-screen 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 the 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

---

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.

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 graphics window 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 screen updates, so that fewer total updates take place, so that models run faster. This command lets you force a display 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 display update takes place once the agent is done running without interruption.

Note that display and no−display operate independently of the switch in the graphics window control strip that freezes the display.

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.

Unlike "distance−nowrap", turtles and patches use the wrapped distance (around the edges of the screen) if that distance is shorter than the on−screen distance.

---

distance−nowrap

distance−nowrap *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.

Unlike "distance", this always reports the on−screen distance, never a distance that would require wrapping around the edges of the screen.

---

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.

Unlike "distancexy−nowrap", the wrapped distance (around the edges of the screen) is used if that 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.
```

distancexy−nowrap

distancexy−nowrap *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.

Unlike "distancexy", this always reports the on−screen distance, never a distance that would require wrapping around the edges of the screen.

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 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 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.

---

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 ^ *number*.

---

export−graphics
export−interface
export−output
export−plot
export−all−plots
export−world

export−graphics *filename*
export−interface *filename*
export−output *filename*
export−plot *plotname filename*
export−all−plots *filename*
export−world *filename*

export−graphics writes the current contents of the graphics window 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, not just the graphics window.

export−output writes the contents of the output portion of the Command Center to an external file given by the string *filename*.

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, 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
=> [0.0 1.0 1.0]
show extract−hsb cyan
```

```
=> [0.5 1.0 1.0]
```

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
=> [1.0 0.0 0.0]
show extract−rgb cyan
=> [0.0 1.0 1.0]
```

See also rgb, hsb, extract−hsb.

F

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 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 exist when 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 (The user needs to have writable permission in the file's directory).

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 will 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
```

---

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 time increment. So `fd 0.5` and `fd 1` both take one unit of time, but `fd 3` takes three.

See also jump.

---

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

---

get−date−and−time

get−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 get−date−and−time
=> "01:19:36.685 PM 19−Sep−2002"
```

---

globals

globals [*var1 var2* ...]

This keyword, like the breeds, <*BREED*>−own, patches−own, and turtles−own keywords, can only be used at the beginning of a program, before any function definitions. It defines new global variables. Global variables are "global" because they are accessible by all agents and can be used anywhere in a model.

Most often, globals is used to define variables or constants that need to be used in many parts of the program.

H

hatch

hatch *number* [*commands*]

Each turtle creates *number* new turtles, each identical to itself, and tells the new turtles to run *commands*. This is useful for giving the new turtles different colors, headings, breeds, or whatever.

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 screen updates take place until the commands are done. This ensures that the new turtles are never drawn on−screen in an only partly initialized state.

```
hatch 1 [ lt 45 fd 1 ]
;; each turtle creates one new turtle,
;; and the child turns and moves away
```

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 showturtle.

histogram−from

histogram−from *agentset* [*reporter*]

Removes points drawn by the current plot pen, then draws a histogram of the values reported when all agents in the agentset run the given reporter. (It 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).

```
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*

Removes points drawn by the current plot pen, then draws a histogram of the values in the given list.

See histogram−from, above, for more information.

home

home

Moves the turtle to the origin. 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−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−set−client−interface

hubnet−set−client−interface *client−type client−info*

If *client−type* is "TI−83+", *client−info* is a list containing two items. The first item is a string containing the name of the activity to enable on the TI Navigator web site. `hubnet−set−client−interface` "TI−83+" notifies the user to enable this activity. The second item is a list of the tags for which to check. The tag list sets which variables NetLogo expects from the calculators. NetLogo will only check for these variables and will ignore all others.

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
  "TI−83+"
  ["AAA − Gridlock 1.3" ["L1" "LOCS"]]
;; notifies the user to enable the activity
;; AAA − Gridlock 1.3 and looks for the calculator
;; lists L1 and LOCS on the Navigator server
```

```
hubnet-set-client-interface
  "COMPUTER"
  ["clients/Disease client.nlogo"]
;; when clients log in, they will will get the
;; interface described in the file
;; clients/Disease client.nlogo, relative to
;; the location of the model
```

Future versions of HubNet may support other client types, and/or change the meaning of the second input to this command.

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 screen
;; turn blue
```

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 screen turns red and
;; the right half turns blue
```

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 screen 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
```

---

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−radius
in−radius−nowrap

agentset* in−radius *number
agentset* in−radius−nowrap *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*.

The distance to or a from a patch is measured from the center of the patch.

in−radius allows its distance measurements to wrap around the edge of the screen; in−radius−nowrap does not.

```
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 random-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−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−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"
```

J

---

jump

jump ***number***


Turtles move forward by *number* units all at once, without the amount of time passing depending on the distance.

This command is useful for synchronizing turtle movements. The command forward 15 takes 15 times longer to run than forward 1, but jump 15 runs in the same amount of time as forward 1.

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 graphics window 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 no−label, 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 no-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 no−label, 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.

left
lt

left *number*


The turtle turns left by *number* degrees. (If *number* is negative, it turns right.)

---

length

length *list*
length *string*

Reports the number of items in the given list, or the number of characters in the given string.

---

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 ...*]

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.

A local variable may not have the same name as an existing observer, turtle, or patch variable.

See to and to−report.

```
to hunt  ;; turtle procedure
  locals [prey]
  set prey random-one-of other-turtles-here
  if prey != nobody
    [ eat prey ]
end
```

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 on the screen
```

---

max-one-of

max-one-of *agentset* [*reporter*]

Reports the agent in the agentset that has the highest value for the given reporter.

```
show max-one-of patches [count turtles-here]

;; prints the patch with the most turtles on it
```

---

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*

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.

```
show member? 2 [1 2 3]
=> true
show member? 4 [1 2 3]
=> false
show member? "rin" "string"
=> true
```

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 the agent in the agentset that reports the lowest value for the given reporter.

```
show min−one−of turtles [xcor + ycor]
;; reports the turtle with the smallest sum of
;; its coordinates
```

mod

number1* mod *number2

Reports *number1* modulo *number2*: that is, the residue of *number1* (mod *number2*). mod is is equivalent to the following NetLogo code:

```
number1 - (floor (number1 / number2)) * number2
```

Note that mod is "infix", that is, it comes between its two inputs.

```
show 62 mod 5
=> 2
show -8 mod 3
=> 1
```

See also remainder. mod and remainder behave the same for positive numbers, but differently for negative numbers.

---

modes

modes *list*

Reports a list of the most common item or items in *list*.

The input list may contain any NetLogo values.

If the input is an empty list, reports an empty list.

```
show modes [1 2 2 3 4]
=> [2]
show modes [1 2 2 3 3 4]
=> [2 3]
show modes [ [1 2 [3]] [1 2 [3]] [2 3 4] ]
=> [[1 2 [3]]
show modes 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 NetLogo graphics window, mouse-down? will always report false.

---

mouse−xcor
mouse−ycor

mouse−xcor
mouse−ycor

Reports the x or y coordinate of the mouse in the Graphics Window. 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 NetLogo graphics window, 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 ]
```

---

myself

myself


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−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
```

---

no-display

no-display

Turns off all updating of the graphics window until the display command is issued. This has two major uses.

One, you can control when the user sees screen updates. You might want to change lots of things on the screen 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 graphics 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 graphics window control strip to freeze the display.)

Note that display and no-display operate independently of the switch in the graphics window control strip that freezes the display.

See also display.

---

no-label

no-label

This is a special value used to remove labels from turtles and patches.

When you set a turtle's label to no−label, or a patch's plabel to no−label, then a label will no longer be drawn on top of the turtle or patch.

See also label, label−color, plabel, plabel−color.

nobody

nobody

This is a special value which some primitives such as turtle, random−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?.

```
set other random-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.

```
set color−of random−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*

If given a turtle agentset, reports the turtle in the set with the lowest numbered ID.

If given a patch agentset, reports the patch in the set with the highest pycor and, if a tie−breaker is needed, with the lowest pxcor.

If the agentset is empty, reports nobody.

See also random−one−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−*BREED*−here

other−turtles−here
other−*BREED*−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:

```
breeds [cats dogs]
show count other-dogs-here
;; prints the number of dogs (that are not me) on my patch
```

See also turtles-here.

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.

```
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.)

```
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.)

```
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.

(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, breeds, <*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, breeds, <*BREED*>−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−up
pu

pen−down
pen−up


The turtle its pen down (or up), so that it draws (leaves a trail) when they move (or doesn't).

Turtles draw by changing the color of the patches underneath them to their own color. To change the color of the turtle's pen (and the color of the turtle itself), use `set color`.

Note: When a turtle's pen is down, only the commands forward and back cause drawing.

Note: Theses commands are equivalent to setting the turtle variable "pen−down?" to true or false.

See also pen−down?.

---

pen−down?

pen−down?


This is a built−in turtle variable. It holds a boolean (true or false) value indicating whether the turtle's pen is currently down. You can set this variable to put a turtle's pen down or pick it back up again.

See also pen−down, pen−up.

---

plabel

plabel


This is a built−in patch variable. It may hold a value of any type. The patch appears in the graphics window 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 no−label, 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 no−label, 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.

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 (− screen−edge−x) and less than or equal to screen−edge−x; similarly for pycor and screen−edge−y.

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 = mean * mean / variance; lambda = 1 / (variance / mean).)

random−normal reports a normally distributed random floating point number.

random−poisson reports a Poisson−distributed random integer.

```
show random−exponential 2
;; prints an exponentially distributed random floating
;; point number with a mean of 2
show random−normal 10.1 5.2
;; prints a normally distributed random floating point
;; number with a mean of 10.1 and a standard deviation
;; of 5.2
show random−poisson 3.4
;; prints a Poisson−distributed random integer with a
;; mean of 3.4
```

random−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 returns 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−n−of

random−n−of *size agentset*

From an agentset, reports an agentset of size *size* randomly chosen from the input set.

From an list, reports a list of size *size* randomly chosen from the input set. 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 random-n-of 50 patches [ set pcolor green ]
;; 50 randomly chosen patches turn green
```

See also random-one-of.

random-one-of

random-one-of *agentset*
random-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 random-one-of patches [ set pcolor green ]
;; a random patch turns green
set pcolor-of random-one-of patches green
;; another way to say the same thing
ask patches with [any? turtles-here]
  [ show random-one-of turtles-here ]
;; for each patch containing turtles, prints one of
;; those turtles

;; suppose mylist is [1 2 3 4 5 6]
show random-one-of mylist
;; prints a value randomly chosen from the list
```

See also one-of and random-n-of.

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
```

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"
```

---

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−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.

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
```

screen−edge−x
screen−edge−y

screen−edge−x
screen−edge−y

These reporters give the maximum x−coordinate and maximum y−coordinate (respectively) of the Graphics Window.

screen−edge−x and −y are the "half−width" and "half−height" of the NetLogo world −− the distances from the origin to the edges. screen−size is the same as ((2 * screen−edge) + 1).

Note: You can set the size of the Graphics Window only by editing it −− these are reporters which cannot be set.

```
cct 100 [ setxy (random−float screen−edge−x)
                     (random−float screen−edge−y) ]
;; distributes 100 turtles randomly in the
;; first quadrant
```

screen−size−x
screen−size−y

screen−size−x
screen−size−y

These reporters give the total width and height of the NetLogo world.

Screen−size is the same as ((2 * screen−edge) + 1).

---

self

self


Reports this turtle or patch.

```
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.

---

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 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 choice
- 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*

---

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, it shape is set to the given shape.

The specified breed must be either turtles or a breed defined by the breeds 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−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.

```
setxy 0 0
;; turtle moves to the middle of the center 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.

See also set−default−shape.

---

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.

---

showturtle
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 in the graphics window. The default size for a turtle 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.

All turtles appear the same size in the graphics window unless the "Turtle Sizes" checkbox in the graphics window edit dialog is checked. If that checkbox is not checked, you can still use this variable, but it will not have any visible effect.

Note: the "Turtle Sizes" feature is currently considered experimental. It may cause your model to run much more slowly, and it may cause display anomalies.

sort

sort *list*

Reports a new 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.

sort−by

sort−by [*reporter*] *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.

```
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"]
```

sprout

sprout *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, breeds, or whatever.

```
sprout 1 [ set color red ]
```

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 screen updates take place until the commands are done. This ensures that the new turtles are never drawn on-screen until they are fully initialized.

sqrt

sqrt *number*

Reports the square root of *number*.

stamp

stamp *color*

Sets the color of the patch under the turtle to the given color.

```
repeat 30 [ stamp yellow fd 3 rt 6 ]
;; the turtle records its arched path -- contrast to the
;; effect of "pen-down"
```

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.)

substring

substring *string position1 position2*

Reports just a section of the given string, ranging between the given positions.

Note: The positions are numbered beginning with 0, not with 1.

```
show substring "turtle" 1 4
=> "urt"
```

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.

---

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−nowrap

towards *agent*
towards−nowrap *agent*

Reports the heading from this agent to the given agent.

If the wrapped distance (around the edges of the screen) is shorter than the on−screen distance, towards will report the heading of the wrapped path. towards−nowrap never uses 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−nowrap

towardsxy *x y*
towardsxy−nowrap *x y*

Reports the heading from the turtle or patch towards the point (*x*,*y*).

If the wrapped distance (around the edges of the screen) is shorter than the on−screen distance, towardsxy will report the heading of the wrapped path. towardsxy−nowrap never uses 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
BREED−at

turtles−at *dx dy*
BREED*−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.

```
breeds [cats dogs]
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 [random−one−of turtles−here]
  ;; reports a turtleset 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
***BREED*−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.

```
breeds [cats dogs]
create-cats 5
create-dogs 1
ask dogs [ show count cats-here ]
=> 5
```

See also other-turtles-here.

---

turtles-on
BREED-on

turtles-on *agent*
turtles-on *agentset*
BREED*-on *agent
BREED*-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
BREED-own

turtles-own [*var1 var2* ...]
***BREED*-own [*var1 var2* ...]**

The turtles-own keyword, like the globals, breed, <*BREED*>-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.)

```
breeds [cats dogs hamsters]
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, breeds, <*BREED*>-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.

U

---

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, downhill.

---

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−choice

user−choice *value list−of−choices*

Opens a dialog with *value* displayed as the message and a button corresponding to each item in *list−of−choices*.

Reports the item in *list−of−choices* that is associated with the button the user presses.

value may be of any type, but is typically a string.

```
if "yes" = (user−choice
                   "Set up the model?"
                   ["no" "yes"])
  [ setup ]
```

user−choose−directory

user−choose−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−choose−directory
;; Assumes the user will choose a directory
```

user−choose−file

user−choose−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−choose−file
;; Assumes the user will choose a file
```

user−choose−new−file

user−choose−new−file

Opens a dialog that allows the user to choose a new file on the system.

It reports a string with the absolute file path or false if the user cancels.

Note that no file is ever created or overwritten with this reporter.

```
file-open user-choose-new-file
;; Assumes the user will choose a file
```

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-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.

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
```

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 execute 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"
```

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.

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 (− screen−edge−x) and strictly less than screen−edge−x.

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 (− screen−edge−y) and strictly less than screen−edge−y.

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.