The Time extension provides NetLogo with three kinds of capabilities for models that use discrete-event simulation or represent time explicitly. The package provides tools for common date and time operations, discrete event scheduling, and using time-series input data.

Quickstart

Include the extension in your NetLogo model (at the top):

```
extensions [time]
```

Date/Time Utilities

Create a global date/time and initialize in the setup procedure:

```
globals[dt]
to setup
  set dt time:create "2000/01/01 10:00"
end
```

From the console, execute setup and then print a formatted version of your date/time to the console:

```
setup
print time:show dt "EEEE, MMMM d, yyyy"
;; prints "Sunday, January 2, 2000"
```

Print the hour of the day, the day of the week, and the day of the year:

```
print time:get "hour" dt        ;; prints 10
print time:get "dayofweek" dt   ;; prints 6
print time:get "dayofyear" dt   ;; prints 1
```

Add 3 days to your date/time and print the date/time object to the screen:

```
set dt time:plus dt 3 "days"
print dt
```

Compare your date/time to some other date/time:

```
ifelse time:is-after? dt (time:create "2000-01-01 12:00") [print "yes"][print "no"]
```

Time Series Tool

NOTE: The time series tool is not currently included in the extension. However, all of the functionality (plus some) has been reproduced in NetLogo code in the time-series.nls file in this repo which you can download by clicking that link and then right clicking the "raw" button on the upper right of the file and selecting

"download linked file". You might have to remove a '.txt' file extension so that the file ends with the '.nls' extension. You can then include that file with the _____includes___ primitive.

Download this example time series file and place in the same directory as your NetLogo model. Here are the first 10 lines of the file:

```
; meta data at the top of the file
; is skipped when preceded by
; a semi-colon
timestamp,flow,temperature
2000-01-01 00:00:00,1000,10
2000-01-01 01:00:00,1010,11
2000-01-01 03:00:00,1030,13
2000-01-01 04:00:00,1040,14
2000-01-01 05:00:00,1050,15
...
...
```

Create a global to store a LogoTimeSeries object. In your setup procedure, load the data from the CSV file:

```
globals[time-series]

set time-series ts-load "time-series-data.csv"
```

Create a LogoTime and use it to extract the value from the "flow" column that is nearest in time to that object:

```
let current-time time:create "2000-01-01 01:20:00"

let current-flow ts-get time-series current-time "flow"

;; By default, the nearest record in the time series is retrieved (in this case 1010),
;; you can alternatively require an exact match or do linear interpolation.
```

Discrete Event Scheduler

Create a few turtles and schedule them to go forward at tick 10, then schedule one of them to also go forward at tick 5.

```
create-turtles 5

time:schedule-event turtles ([ [] -> fd 1]) 10
time:schedule-event (turtle 1) ([ [] -> fd 1]) 5
```

Execute the discrete event schedule (all events will be executed in order of time):

```
time:go

;; turtle 1 will go foward at tick 5,
;; then all 5 turtles will go forward at tick 10
```

back to top

What is it?

This package contains the NetLogo **time extension**, which provides NetLogo with three kinds of capabilities for models that use discrete-event simulation or represent time explicitly. The package provides tools for common date and time operations, discrete event scheduling, and using time-series input data.

Dates and Times

The extension provides tools for representing time explicitly, especially by linking NetLogo's ticks to a specific time interval. It allows users to do things such as starting a simulation on 1 January of 2010 and end on 31 December 2015, have each tick represent 6 hours, and check whether the current simulation date is between 1 and 15 March.

This extension is powered by the Java Time Library, which has very sophisticated and comprehensive date/time facilities. A subset of these capabilities have been extended to NetLogo. The **time extension** makes it easy to convert string representations of dates and date/times to a **LogoTime** object which can then be used to do many common time manipulations such as incrementing the time by some amount (e.g. add 3.5 days to 2001-02-22 10:00 to get 2001-02-25 22:00).

Time Series Utilities

Modelers commonly need to use time series data in NetLogo. The **time extension** no longer provides time series functionality (plus some), but the same functionality is included the time-series.nls file in this repo which you can download by clicking that link and then right clicking the "raw" button on the upper right of the file and selecting "download linked file" (you might have to remove a '.txt' file extension so that the file ends with the '.nls' extension). You can then include that file with the ______includes___ primitive. The time-series.nls file includes convenient procedures for handling time series data. With a single command, you can load an entire time series data set from a text file. The first column in that text file holds dates or datetimes. The remaining columns can be numeric or string values. You then access the data by time and by column heading, akin to saying "get the flow from May 8, 2008".

Users can also create and record a time series of events within their model, access that series during simulations, and export it to a file for analysis. For example, a market model could create a time series object into which is recorded the date and time, trader, price, and size of each trade. The time series utilities let model code get (for example) the mean price over the previous day or week, and save all the trades to a file at the end of a run.

Discrete Event Scheduling

Note: Formerly this capability was published as the **Dynamic Scheduler Extension**, but that extension has been merged into the **time extension** in order to integrate the functionality of both.

The **time extension** enables a different approach to scheduling actions in NetLogo. Traditionally, a NetLogo modeler puts a series of actions or procedure calls into the "go" procedure, which is executed once each tick. Sometimes it is more natural or more efficient to instead say "have agent X execute procedure Y at time Z". This is what discrete event scheduling (also know as "dynamic scheduling"") enables. Discrete event simulation has a long history and extensive literature, and this extension makes it much easier to use in NetLogo.

When is discrete event scheduling useful? Discrete event scheduling is most useful for models where agents spend a lot of time sitting idle despite knowing when they need to act next. Sometimes in a NetLogo model, you end up testing a certain condition or set of conditions for every agent on every tick (usually in the form of an "ask"), just waiting for the time to be ripe.... this can get cumbersome and expensive. In some models, you might know in advance exactly when a particular agent needs to act. Dynamic scheduling cuts out all of those superfluous tests. The action is performed only when needed, with no condition testing and very little overhead.

For example, if an agent is a state machine and spends most of the time in the state "at rest" and has a predictable schedule that knows that the agent should transition to the state "awake" at tick 105, then using a dynamic scheduler allows you to avoid code that looks like: "if ticks = 105

$$do - something$$

", which has to be evaluated every tick!

A second common use of discrete event scheduling is when it is important to keep track of exactly when events occur in continuous time, so the simplifying assumption that all events happen only at regular ticks

is not appropriate. One classic example is queuing models (e.g., how long customers have to stand in line for a bank teller), which use a continuous random number distribution (e.g., an exponential distribution) to determine when the next agent enters the queue.

back to top

Installation

The Time extension comes bundled with NetLogo. Simply include `extensions [time]` at the top of the model and it will be loaded. New versions of the extension are available through the Netlogo extensions manager as they are released.

Alternatively, download the latest version of the extension (Note that this extension is compiled for NetLogo 6.1 and may not work for earlier versions of NetLogo) Unzip the archive and rename the directory to "time". Move the renamed directory to the "extensions" directory inside your NetLogo application folder (i.e. [NETLOGO]/extensions/). Or you can place the time directory under the same directory holding the NetLogo model in which you want to use this extension.

For more information on NetLogo extensions: http://ccl.northwestern.edu/netlogo/docs/extensions.html

back to top

Examples

See the example models in the time extension's subfolder "examples" for thorough demonstrations of usage.

Data Types

The **time extension** introduces some new data types (more detail about these is provided in the behavior section):

- **LogoTime** - A LogoTime object stores a time stamp; it can track a full date and time, or just a date (with no associated time).
- **LogoEvent** - A LogoEvent encapsulates a who, a what, and a when. It allows you to define, for example, that you want turtle 7 to execute the go-forward procedure at tick 10. When scheduling an event using the **time extension** you pass the who, what, and when as arguments (e.g. "time:schedule-event (turtle 1) fd 5").
- **Discrete Event Schedule** - A discrete event schedule is a sorted list of LogoEvents that is maintained by this extension and manages the dispatch (execution) of those events. Users do not need to manipulate or manage this schedule directly, but it is useful to understand that it stores and executes LogoEvents when the "time:go" or "time:go-until" commands are issued. As the schedule is executed, the **time extension** automatically updates the NetLogo ticks to match the current event in the schedule.

The time-series.nls file in this repo provides one more datatype. * **LogoTimeSeries** - A LogoTimeSeries object stores a table of data indexed by LogoTime. The time series can be read in from a file or recorded by the code during a simulation. It is currently implemented in NetLogo code using a list of lists.

back to top

Behavior

The **time extension** has the following notable behavior:

- **LogoTimes can store DATETIMEs, DATEs, or DAYs** - A LogoTime is a flexible data structure that will represent your time data as one of three varieties depending on how you create the LogoTime object. A LogoTime can be a DATETIME, a DATE, or a DAY:

- A DATETIME is a fully specified instant in time, with precision down to a millisecond (e.g. January 2, 2000 at 3:04am and 5.678 seconds).
- A DATE is a fully specified day in time but lacks any information about the time of day (e.g. January 2, 2000).
- A DAY is a generic date that does not specify a year (e.g. January 2).

The behavior of the **time extension** primitives depend on which variety of LogoTime you are storing. For example, the difference between two DATETIMES will have millisecond resolution, while the difference between two DATES or two DAYS will only have resolution to the nearest whole day.

As another example, a DAY representing 01/01 is always considered to be before 12/31. Because there's no wrapping around for DAYs, they are only useful if your entire model occurs within one year and doesn't pass from December to January. If you need to wrap, use a DATE and pick a year for your model, even if there's no basis in reality for that year. (DAY LogoTime variables are useful, however, as parameters defining the date when something happens every simulated year. You can, for example, use time extension commands to say that a species' breeding season starts on one DAY—15 May—and ends on another—25 June—each year.)

- **You create LogoTime objects by passing a string** - The time:create primitive was designed to both follow the standard used by the Java time library, and to make date time parsing more convenient by allowing a wider range of delimiters and formats. For example, the following are all valid DATETIME strings:

  - "2000-01-02T03:04:05.678"
  - "2000-01-02T3:04:05.678"
  - "2000-01-02 03:04:05"
  - "2000-01-02 3:04:05"
  - "2000-01-02 03:04"
  - "2000-01-02 3:04"
  - "2000-01-02 03"
  - "2000-01-02 3"
  - "2000/01/02 03:04:05.678"
  - "2000-1-02 03:04:05.678"
  - "2000-01-2 03:04:05.678"
  - "2000-1-2 03:04:05.678"

  The following are all valid DATE strings:

  - "2000-01-02"
  - "2000-01-2"
  - "2000-1-02"
  - "2000/1/02"

  The following are all valid DAY strings:

  - "01-02"
  - "01-2"
  - "1-02"
  - "1/2"

  Note that if you do not include a time in your string, the **time extension** will assume you want a DATE. If you want a DATETIME that happens to be at midnight, specify the time as zeros: "2000-01-02 00:00".

- **Time extension recognizes "period types"** - In order to make it easy to specify a time period like "2 days" or "4 weeks", the **time extension** will accept strings to specify a period type. The following is a table of the period types and strings that **time** recognizes (note: any of these period type strings can be pluralized and are case **in**sensitive):

PERIOD TYPE	Valid string specifiers
YEAR	“year”
MONTH	“month”
WEEK	“week”
DAY	“day”, “dayofmonth”, “dom”
DAYOFYEAR	“dayofyear”, “doy”, “julianday”, “jday”
DAYOFWEEK	“dayofweek”, “dow”, “weekday”, “wday”
HOUR	“hour”
MINUTE	“minute”
SECOND	“second”
MILLI	“milli”

- **Time extension has millisecond resolution** - This is a fundamental feature of the Java Time library and cannot be changed. Milliseconds are represented as fractions of a second, truncated to the third decimal place to prevent parsing issues with the Java Time library. This truncation can create artifacts; for example if time:plus is used to add a value that is less than one whole millisecond, it does nothing.
- **Daylight savings time is ignored** - All times are treated as local, or “zoneless”, and daylight savings time (DST) is ignored. It is assumed that most NetLogo users don’t need to convert times between time zones or be able to follow the rules of DST for any particular locale. Instead, users are much more likely to need the ability to load a time series and perform date and time operations without worrying about when DST starts and whether an hour of their time series will get skipped in the spring or repeated in the fall. It should be noted that the Time library definitely can handle DST for most locales on Earth, but that capability is not extended to NetLogo here and won’t be unless by popular demand.
- **Leap days are included** - While we simplify things by excluding time zones and DST, leap days are kept to allow users to reliably use real-world time series in their NetLogo model. LogoTime variable type DAY assumes a leap year, so DAY variables can include 2/29. (If you add one day to a DAY with value of 28 February, the result is 29 February, not 1 March.)
- **LogoTimes are mutable when anchored** - If you anchor a LogoTime (using the *time:anchor-to-ticks* primitive) you end up with a variable whose value changes as the value of Netlogo ticks changes. If you have an anchored variable called “anchored-time” and you assign it to another variable “set new-time anchored-time”, your new variable will *also be mutable* and change with ticks. If what you want is a snapshot of the anchored-time that doesn’t change, then use the time:copy primitive: “set new-time time:copy anchored-time”.
- **Decimal versus whole number time periods** - In this extension, decimal values can be used by the *plus* and *anchor-to-ticks* primitives for seconds, minutes, hours, days, and weeks (milliseconds can’t be fractional because they are the base unit of time). These units are treated as *durations* because they can unambiguously be converted from a decimal number to a whole number of milliseconds. But there is ambiguity in how many milliseconds there are in 1 month or 1 year, so month and year increments are treated as *periods* which are by definition whole number valued. So if you use the *time:plus* primitive to add 1 month to the date “2012-02-02”, you will get “2012-03-02”; and if you add another month you get “2012-04-02” even though February and March have different numbers of days. If you try to use a fractional number of months or years, it will be rounded to the nearest integer and then added. If you want to increment a time variable by one and a half 365-day years, then just increment by 1.5 * 365 days instead of 1.5 years.
- **LogoEvents are dispatched in order, and ties go to the first created** - If multiple LogoEvents are scheduled for the exact same time, they are dispatched (executed) in the order in which they were added to the discrete event schedule.
- **LogoEvents can be created for an agentset** - When an agentset is scheduled to perform an anonymous command (before NetLogo 6.0 these were called tasks) at the same time, the individual

agents execute the procedure in a non-random order, which is different from *ask* which shuffles the agents. Not shuffling the agents may save some execution time. To shuffle the order, use the *add-shuffled* primitive, which will execute the actions in random order with low overhead.

- **LogoEvents won't break if an agent dies** - If an agent is scheduled to perform an anonymous command in the future but dies before the event is dispatched, the event will be silently skipped.
- **LogoEvents can be scheduled to occur at a LogoTime** - LogoTimes are acceptable alternatives to specifying tick numbers for when events should occur. However, for this to work the discrete event schedule must be "anchored" to a reference time so it knows a relationship between ticks and time. See *time:anchor-schedule** below for an example of anchoring.

back to top

Format

The time extension utilizes rules and standards, provided by the Java 8 time library, to parse and accept valid dates. The extension provides default formats for quickly creating time objects, but includes an option for specifying a custom format. In addition, the extension adheres to the ISO 8601 standard, following the Java 8 STRICT format and 24 hour clock format. The same formats are used for both input and output.

Date Format

Date formats are encoded strings with unique characters that represent various units of time and their position (date and time). Each character is meant to be placed in a contiguous group to indicate its expected location and type. If a user provides a date string that does not follow the parsing format or bounds, then an exception will be thrown. With create primitives, the selected default format can create a date-time, date, and day object depending on the string provided. The set of default formats are below. The default delimiter between months, days and years is '/', between hours, minutes and seconds is ":" and between seconds and milliseconds is ".". Using any other delimiter requires using a custom formatter (see User Defined Formatting).

- **DateTime Default Formatter**

```
MM/dd/yyyy HH:mm:ss.SSS
```

For date-time formatters, the format specifies all 7 units (month,day,year,etc) will be available for parsing and generating a date-time object.

- **Date Default Formatter**

```
MM/dd/yyyy
```

For date formatters, the month, day, and year need to be specified to obtain a date object.

- **Day Default Formatter**

```
MM/dd
```

For day formatters, the month and day need to be specified to obtain a day object. Day objects are based on the year 2000, which contains a leap day, which could lead to miscalculations if only 28 days are expected in February.

Supported Format Characters

For supported format characters ('H','m',etc), there are three main modes for parsing: shorthand, sized, and full. Each mode controls the number of acceptable charcters for parsing digits.

- **Shorthand** For all units, except the millisecond and year fields, "shorthand" formats use a single unique character meant to accept 1 or 2 digits for the corresponding unit. The short-hand case is a lenient option for values that can fluctuate values between 1 or 2 digits. An example:

```
M/d
```

- **Sized** For year and millisecond, the number of characters for a specific unit of time determines the number of acceptable numbers. For example, a format of "MM/dd/yyyy HH:mm:ss.S" assumes that that the input/output has tenths of seconds while "MM/dd/yyyy HH:mm:ss.SSS" assumes that the input/output has miliseconds.

```
MM/dd/yyyy HH:mm:ss.SSS
MM/dd/yyyy HH:mm:ss.S
```

- **Full** For all units of time, the maximum number of representable formatting characters can be used to provide a verbose and accurrate representation of the time objects. If the string does not follow the format, an exception will be thrown. Example:

```
MM/dd/yyyy HH:mm:ss.SSS
```

Month	Day	Year	Hour	Minute	Second	Millisecond
`MM` or `M`	`dd` or `d`	`yyyy` or `yy`	`HH` or `H`	`mm` or `m`	`ss` or `s`	`SSS` or `SS` or `S`

NOTE: generally 'yyyy' (or 'uuuu') should be used for years and not 'YYYY' which is a "week based year." This means that a date at the very end of one year, e.g. 12/31/2000 might be given the year 2001 if 'YYYY' is used, because it is counted as part of the first week of 2001.

Date-time Bounds

Since not all dates are representable within the time extension, a strict formatting is enforced to minimize the effects of invalid input. Dates are bounded by the Gregorian calendar while the time is bounded with their respective unit of time.

Month	Day	Year	Hour	Minute	Second	Millisecond
1 - 12	1 - 31 or 30 or 29 or 28	1000-9999	0-23	0-59	0-59	0 - 999

User Defined Formatting

The time extension supports user-defined formatters for brevity and control. User defined formatters allow for reordering format characters for units of time and using alternative delimiters from the defaults.

```
M:yyyy:d
MM/d/yyyy HH:mm
```

back to top

Primitives

Date/Time Utilities

time:create

time:create time-string

Reports a LogoTime created by parsing the *time-string* argument. A LogoTime is a custom data type included with this extension, used to store time in the form of a DATETIME, a DATE, or a DAY. All other primitives associated with this extension take one or more LogoTimes as as an argument. See the "Behavior" section above for more information on the behavior of LogoTime objects. The `time:create` primitive raises an error if time-string does not represent a real date or time, e.g., if the day exceeds the number of days in the month or the month is not between 1 and 12. Hours must have values 0-23, and minutes and seconds must be 0-59. Seconds with milliseconds must be between 0.000 and 59.999.

```
;; Create a datetime, a date, and a day
let t-datetime time:create "2000-01-02 03:04:05.678"
let t-date time:create "2000/01/02"
let t-day time:create "01-02"
```

---

time:create-with-format

time:create-with-format time-string format-string

Like `time:create`, but parses the *time-string* argument using the *format-string* argument as the format specifier.

```
;; Create a datetime, a date, and a day using American convention for dates: Month/Day/Year
let t-datetime time:create-with-format "01-02-2000 03:04:05.678" "MM-dd-yyyy HH:mm:ss.SSS"
let t-date time:create-with-format "01/02/2000" "MM/dd/yyyy"
let t-day time:create-with-format "01-02" "MM-dd"
```

See the following link for a full description of the available format options:

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

---

time:show

time:show logotime string-format

Reports a string containing the *logotime* formatted according the *string-format* argument.

```
let t-datetime time:create "2000-01-02 03:04:05.678"

print time:show t-datetime "EEEE, MMMM d, yyyy"
;; prints "Sunday, January 2, 2000"

print time:show t-datetime "yyyy-MM-dd HH:mm"
;; prints "2000-01-02 03:04"
```

See the following link for a full description of the available format options:

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

Note In the case for DATE and DAY variables, time:get returns default values for unsupported values. Day has a default year of 2000 to calculate for leap years.

```
let t-date (time:create "2000-01-01")
let t-day (time:create "01-01")

print time:show t-date "HH"
print time:show t-day "HH"
;; print 00

print time:show t-date "mm"
print time:show t-day "mm"
;; print 00

print time:show t-date "ss"
print time:show t-day "ss"
;; print 00

print time:show t-date "SSS"
```

```
print time:show t-day "SSS"
;; print 000

print time:show t-day "yyyy"
;; print 2000
```

---

time:get

time:get period-type-string logotime

Retrieves the numeric value from the *logotime* argument corresponding to the *period-type-string* argument. For DATETIME variables, all period types are valid; for DATEs, only period types of a day or higher are valid; for DAYs, the only valid period types are "day" and "month". For accessing seconds, seconds are not rounded to the nearest integer, but truncated. When time:get is used with a sub-day period type ("hour", "minute") and a logotime that is a DATE or DAY instead of a DATETIME, it returns zero. Getting the year of a DAY variable returns the default of 2000.

```
let t-datetime (time:create "2000-02-02 03:04:05.678")

print time:get "year" t-datetime
;;prints "2000"

print time:get "month" t-datetime
;;prints "1"

print time:get "day" t-datetime
;;prints "2"

print time:get "dayofyear" t-datetime
;;prints "33"

print time:get "hour" t-datetime
;;prints "3"

print time:get "second" t-datetime
;;prints "5"

print time:get "milli" t-datetime
;;prints "678"
```

Note In the case for Day and Date time formats, time:get returns default values for unsupported values. Day has a default year of 2000 to calculate for leap years.

```
let t-date (time:create "2000-01-01")
let t-day (time:create "01-01")

print time:get "hour" t-date
print time:get "hour" t-day
;; print 0

 print time:get "minute" t-date
 print time:get "minute" t-day
;; print 0

print time:get "second" t-date
```

```
print time:get "second" t-day
;; print 0

print time:get "milli" t-date
print time:get "milli" t-day
;; print 0

print time:get "year" t-day
;; print 2000
```

---

time:plus

time:plus logotime number period-type-string

Reports a LogoTime resulting from the addition of some time period to the *logotime* argument. The time period to be added is specified by the *number* and *period-type-string* arguments. Valid period types are YEAR, MONTH, WEEK, DAY, DAYOFYEAR, HOUR, MINUTE, SECOND, and MILLI. To subtract time, use a negative value of number. When applying additions to LogoTimes, addition is applied through converting the format to datetime, adding the values, and reconverting LogoTime to its respective format. This can lead to incorrectly applying time updates and lost information.

```
let t-datetime (time:create "2000-01-02 03:04:05.678")

;; Add some period to the datetime
print time:plus t-datetime 10 "milli"
;; prints "{{time:logotime 2000-01-02 03:04:06.688}}"

print time:plus t-datetime 1.0 "seconds"
;; prints "{{time:logotime 2000-01-02 03:04:06.678}}"

print time:plus t-datetime 1.0 "minutes"
;; prints "{{time:logotime 2000-01-02 03:05:05.678}}"

print time:plus t-datetime (60.0 * 24) "minutes"
;; prints "{{time:logotime 2000-01-03 03:04:05.678}}"

print time:plus t-datetime 1 "week"
;; prints "{{time:logotime 2000-01-09 03:04:05.678}}"

print time:plus t-datetime 1.0 "weeks"
;; prints "{{time:logotime 2000-01-09 03:04:05.678}}"

print time:plus t-datetime 1.0 "months"
;; note that decimal months or years are rounded to the nearest whole number
;; prints "{{time:logotime 2000-02-02 03:04:05.678}}"

print time:plus t-datetime 1.0 "years"
;; prints "{{time:logotime 2001-01-02 03:04:05.678}}"
```

Addition with Decimals

- Week, Day, DayOfYear, Hour, Minute, and Second all support decimal addition so you can add half a day, a quarter of a second, or another subvalue. You cannot add fractions of milliseconds because LogoTime variables do not have sub-millisecond resolution. You can add fractions of days (or hours, minutes, etc.) to LogoTime variables of type DATE and DAY, but the result will be truncated to an

integer day. This causes some information to be lost. For example:

```
print time:plus (time:create "2000-01-01 00:00:00.000") 2.3 "day" => "2000-01-03 07:11:59.999"
print time:plus (time:create "2000-01-01") 2.3 "day" => "2000-01-03"

print time:plus (time:create "2000-01-04 00:00:00.000") -2.3 "day" => "2000-01-01 2000-01-01 16:48:00.0
print time:plus (time:create "2000-01-04") -2.3 "day" => "2000-01-01"
```

- Year and Month both round the provided number to the nearest integer. There is no decimial addition support for year and month.

Addition with Negative Decimal Numbers

- Year, Month, Week, Day, DayOfYear, Hour, Minute, Second, and Millisecond all support subtraction through the use of negative numbers. Subtracting with whole numbers works as expected, but when subtracting with decimals the results may produce unexpected behavior. When subtracting with DateTime, DateTime objects maintain accurate times without truncation or rounding. With DATE and DAY variables, when decimals are subtracted two conversions are applied. First the DATE/DAY is converted to a DATETIME starting at midnight (00:00:00), tthen the subraction iis made, and the resulting DATETIME is converted back the original DATE or DAY type by setting the hours, minutes, etc. to zero. One example is with subtracting an hour from a DATE. Subtracting one hour from a DATE variable equal to 01/02/2000 would generate a date of 01/01/2000 because of the default (00:00:00) conversion and the truncation after applying the addition.

  "2000-01-02" (convert) => "2000-01-02 00:00:00.000" (subtract hour) => "2000-01-01 23:00:00.000" (application) => "2000-01-01" (restore type)

How Rounding, Truncating and Converting works

- Week, Day, DayOfYear, Hour, Minute, Second, and Millisecond: convert user provided time/unit to milliseconds as a long integer => apply addition => convert to initial format (DateTime, Date, or Day) => Remove unsupported units (Date, Day)
- Year and Month: round user provided time/value to the nearest integer => apply addition => convert to initial foramt => Remove Unsupported units (Day)

Carryover

- Week, Day, DayOfYear, Hour, Minute, Second, and Millisecond: All time types allow carrying over values from their subunits. This means that if 24 hours (or 100,000 seconds) are added to a DATE or DAY, result is the next day. If 23 hours are added to a DATE or DAY, its value will not change thanks to truncation. DATETIME variables can have carryover with decimal addition since a DATETIME can represent decimal values.
- Year and Month numbers are rounded to the nearest integer before being added.

---

time:is-before? time:is-after? time:is-equal? time:is-between?

time:is-before? logotime1 logotime2 time:is-after? logotime1 logotime2 time:is-equal? logotime1 logotime2 time:is-between? logotime1 logotime2 logotime3

Note: Prior versions of the time extension included these primitives without the "?" at the end. If you have used an older version of the time extension, you will need to update your code accordingly.

Reports a boolean for the test of whether *logotime1* is before/after/equal-to *logotime2*. The is-between? primitive returns true if *logotime1* is between *logotime2* and *logotime3*. All LogoTime arguments must be of the same variety (DATETIME, DATE, or DAY).

```
print time:is-before? (time:create "2000-01-02") (time:create "2000-01-03")
;;prints "true"
```

```
print time:is-before? (time:create "2000-01-03") (time:create "2000-01-02")
;;prints "false"

print time:is-after?  (time:create "2000-01-03") (time:create "2000-01-02")
;;prints "true"

print time:is-equal?  (time:create "2000-01-02") (time:create "2000-01-02")
;;prints "true"

print time:is-equal?  (time:create "2000-01-02") (time:create "2000-01-03")
;;prints "false"

print time:is-between? (time:create "2000-03-08")  (time:create "1999-12-02") (time:create "2000-05-03")
;;prints "true"
```

---

time:difference-between

time:difference-between logotime1 logotime2 period-type-string

Reports the amount of time between *logotime1* and *logotime2* in units of *period-type-string*. Note that if the period type is YEAR or MONTH, then the reported value will be a whole number based soley on the month and year components of the LogoTimes. If *logotime2* is smaller (earlier than) *logotime1*, the reported value will be negative.

This primitive is useful for recording the elapsed time between model events because (unlike time:get) it reports the total number of time units, including fractions of units. For example, if *start-time* is a LogoTime variable for the time a simulation starts and *end-time* is when the simulation stops, then use *show time:difference-between start-time end-time "days"* to see how many days were simulated.

** Note **

For period type of "MONTH", difference-between reports the difference in the month number (1-12) of logotime1 and logotime2, not the total number of months between the two dates, even if the two times are DATETIME or DATE values. This difference is calculated as the number of times the day number (1-31) of logotime1 occurs between the two dates. For example, if *logotime1* is 2000-01-02 then difference-between reports 0 months for *logotime2* = 2000-02-01, 1 month for 2000-02-02, and -1 month for logotime2 = 1999-12-01. However, the value reported is never greater than 11 or less than -11; it reverts to zero when logotime2 reaches one year from logotime1 and then is 1 when logotime2 is 13 months after logotime1, etc.

For "YEAR", difference-between reports the number [integer] of years there are between two logotime instances. The difference is calculated through matching the day and month while truncating values around that date. This should apply for leap years, as well. For example, if logotime1 is 2000-01-02 then `time:difference-between` reports 0 years for *logotime2* = 2001-01-01, 1 year for 2000-01-02, and 5 years for 2005-01-02. The total number of months between two dates can therefore be calculated as 12 times the value of `time:difference-between` in years plus the value of `time:difference-between` in months.

```
print time:difference-between (time:create "2000-01-02 00:00") (time:create "2000-02-02 00:00") "days"
;;prints "31"

print time:difference-between (time:create "2000-01-02") (time:create "2001-02-02") "days"
;;prints "397"

print time:difference-between (time:create "01-02") (time:create "01-01") "hours"
;;prints "-24"

print time:difference-between (time:create "2000-01-02") (time:create "2000-02-15") "months"
```

```
;;prints "1"
```

---

time:anchor-to-ticks

time:anchor-to-ticks logotime number period-type

Reports a new LogoTime object which is "anchored" to the native time tracking mechanism in NetLogo (i.e the value of *ticks*). Once anchored, this LogoTime object will always hold the value of the current time as tracked by *ticks*. Any of the three varieties of LogoTime can be achored to the tick. The time value of the *logotime* argument is assumed to be the time at tick zero. The *number* and *period-type* arguments describe the time represented by one tick (e.g. a tick can be worth 1 day or 2 hours or 90 seconds, etc.)

Note: *time:anchor-to-ticks* is a one-way coupling. Changes to the value of *ticks* (e.g. when using the *tick* or *tick-advance* commands) will automatically update the anchored LogoTime, but changing the value of the LogoTime variable will not update the value of *ticks*; in fact it is a bad idea for anything other than ticks to change an anchored variable. However, see the discrete event scheduling capability and the *time:anchor-schedule* command to influence the value of *ticks* through the use of LogoTimes.

```
set tick-datetime time:anchor-to-ticks (time:create "2000-01-02 03:04:05.678") 1 "hour"
set tick-date time:anchor-to-ticks (time:create "2000-01-02") 2 "days"
set tick-day time:anchor-to-ticks (time:create "01-02") 3 "months"

reset-ticks
tick
print (word "tick " ticks)  ;; prints "tick 1"
print (word "tick-datetime " tick-datetime)  ;; prints "tick-dateime {{time:logotime 2000-01-02 04:04:05
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-04}}"
print (word "tick-day " tick-day)  ;; prints "tick-day {{time:logotime 04-02}}"


tick
print (word "tick " ticks)  ;; prints "tick 2"
print (word "tick-datetime " tick-datetime)  ;; prints "tick-dateime {{time:logotime 2000-01-02 05:04:05
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-06}}""
print (word "tick-day " tick-day)  ;; prints "tick-day {{time:logotime 07-02}}""
```

back to top

---

time:copy

time:copy logotime

Returns a new LogoTime object that holds the same date/time as the *logotime* argument. The copy will not be anchored regardless of the argument, making this the recommended way to store a snapshot of an anchored LogoTime.

```
set tick-date time:anchor-to-ticks (time:create "2000-01-02") 2 "days"
reset-ticks
tick
print (word "tick " ticks)  ;; prints "tick 1"
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-04}}"

set store-date time:copy tick-date

tick
print (word "tick " ticks)  ;; prints "tick 1"
```

```
print (word "tick-date " tick-date)  ;; prints "tick-date {{time:logotime 2000-01-06}}"
print (word "store-date " store-date)  ;; prints "store-date {{time:logotime 2000-01-04}}"
```

Time Series Tool

NOTE: The time series tool is not currently included in the extension. However, all of the functionality (plus some) has been reproduced in NetLogo code. It is in the time-series.nls file in this repo which you can download by clicking that link and then right clicking the "raw" button on the upper right of the file and selecting "download linked file" (you might have to remove a '.txt' file extension so that the file ends with the '.nls' extension). You can then include that file with the _____includes___ primitive. Since it is not part of the extension, these "primitives" don't have the "time:" prefix. Also, the NetLogo file must have the csv extension for the time-series.nls functionality to work.

ts-create

ts-create column-name-list

Reports a new, empty LogoTimeSeries. The number of data columns and their names are defined by the number and values of *column-name-list* parameter, which must be a list of strings. The first column, which contains dates or times, is created automatically. The words "ALL" and "LOGOTIME" (in any combination of lower- or upper-case) cannot be used as column names because they have special meanings in the ts-get and ts-get-range primitives.

```
let turtle-move-times (ts-create ["turtle-show" "new-xcor" "new-ycor"])
```

---

ts-add-row *NOTE*: This is the only time-series functionality in time-series.nls that is not backwards compatible with the old time extension. Since time-series.nls implements time series with native NetLogo lists, they are immutable. So, ts-add-row returns a new timeseries, it does not mutate the old one. This means that variables holding the time series must be re-assigned.

ts-add-row logotimeseries row-list

Returns a new LogoTimeSeries with row-list added to *logotimeseries*. This primitive is how data are added to a LogoTimeSeries. The *row-list* should be a list containing a LogoTime as the first element and the rest of the data corresponding to the number of columns in the LogoTimeSeries object. Columns are either numeric or string valued.

```
;; A turtle records the time and destination each time it moves
;; model-time is a DATETIME variable anchored to ticks.
set turtle-move-times ts-add-row turtle-move-times (sentence model-time who xcor ycor)
```

---

ts-get

ts-get logotimeseries logotime column-name

Reports the value from the *column-name* column of the *logotimeseries* in the row matching *logotime*. If there is not an exact match with *logotime*, the row with the nearest date/time will be used. If there are multiple rows with the same logotime, only one will be returned. For LogoTimeSeries containing multiple rows with the same logotime value, using ts-get-range to return a list of all times within a range is recommended instead of *ts-get*. If "ALL" or "all" is specified as the column name, then the entire row, including the logotime, is returned as a list.

```
print ts-get ts (time:create "2000-01-01 10:00:00") "flow"
;; prints the value from the flow column in the row containing a time stamp of 2000-01-01 10:00:00
```

---

ts-get-interp

ts-get-interp logotimeseries logotime column-name

Behaves like ts-get, except that if there is not an exact match with the *logotime*, then the value returned is a linear interpolation between the two values with date/times nearest (just before and just after) *logotime*. This command will throw an exception if the values in the column are strings instead of numeric.

```
print ts-get-interp ts (time:create "2000-01-01 10:30:00") "flow"
```

---

ts-get-exact

ts-get-exact logotimeseries logotime column-name

Behaves like ts-get, except that if there is not an exact match with the date/time stamp, then an exception is thrown. If there are multiple rows with the same logotime, only one will be returned. In such a case, usinig `ts-get-range` to return a list of all times within a range is recommended instead.

```
print ts-get-exact ts (time:create "2000-01-01 10:30:00") "flow"
```

---

ts-get-range

ts-get-range logotimeseries logotime1 logotime2 column-name

Reports a list of all of the values from the *column-name* column of the *logotimeseries* in the rows between *logotime1* and *logotime2* (inclusively). If "ALL" or "all" is specified as the column name, then a list of lists is reported, with one sub-list for each column in *logotimeseries*, including the date/time column. If "LOGOTIME" or "logotime" is specified as the column name, then the date/time column is returned.

If in the event *logotime1* is after *logotime2*, `ts-get-range` will reverse their values, so it does not matter which of these inputs is smaller.

```
print ts-get-range time-series time:create "2000-01-02 12:30:00" time:create "2000-01-03 00:30:00" "all
```

---

ts-has-repeat-times? *ts-has-repeat-times? logotimeseries*

Reports whether there are any repeated times in the logotimeseries (i.e. two rows with the same time). This can be helpful for determining if it is appropriate to use ts-get or ts-get-range.

---

ts-has-repeat-of-time? *ts-has-repeat-of-time? logotimeries logotime*
Reports whether *logotime* appears more than once in *logotimeseries*.

---

ts-load

ts-load filepath

Loads time series data from a text input file (comma or tab separated) and reports a new LogoTimeSeries object that contains the data.

```
let ts ts-load "time-series-data.csv"
```

Each input file and LogoTimeSeries object can contain one or more variables, which are accessed by the column names provided on the first line of the file. The first line of the file must therefore start with the date/time column name, followed by the names of the variables (other columns) in the file. Do not use "all" or "ALL," or "logotime" or "LOGOTIME" for a column name as thse are reserved keywords (see `ts-get` and `ts-get-range`).

The first column of the file must be timestamps that can be parsed by this extension (see the behavior section for acceptable string formats). Finally, if the timestamps do not appear in chronological order in the text file, they will be automatically sorted into order when loaded.

Comment lines can appear at the start of input file, but nowhere else in it. Comment lines must start with a semicolon.

The following is an example of hourly river flow and water temperature data that is formatted correctly:

```
; Flow and temperature data for Big Muddy River
timestamp,flow,temperature
2000-01-01 00:00:00,1000,10
2000-01-01 01:00:00,1010,11
2000-01-01 03:00:00,1030,13
```

back to top

---

ts-load-with-format

ts-load filepath format-string

Identical to ts-load except that the first column is parsed based on the *format-string* specifier.

```
let ts ts-load "time-series-data-custom-date-format.csv" "dd-MM-yyyy HH:mm:ss"
```

See Behavior and Format above concerning format-string options.

---

ts-write

ts-write logotimeseries filepath

Writes the time series data to a text file in CSV (comma-separated) format.

```
ts-write ts "time-series-output.csv"
```

The column names will be written as the header line, for example:

```
timestamp,flow,temperature
2000-01-01 00:00:00,1000,10
2000-01-01 01:00:00,1010,11
2000-01-01 03:00:00,1030,13
```

back to top

Discrete Event Scheduler

time:anchor-schedule

time:anchor-schedule logotime number period-type

Anchors the discrete event schedule to the native time tracking mechanism in NetLogo (i.e the value of *ticks*). Once anchored, LogoTimes can be used for discrete event scheduling (e.g. schedule agent 3 to perform some anonymous command on June 10, 2013). The value of the *logotime* argument is assumed to be the time at tick zero. The *number* and *period-type* arguments describe the length of one tick (e.g. a tick can represent 1 day, 2 hours, 90 seconds, etc.)

```
time:anchor-schedule time:create "2013-05-30" 1 "hour"
```

---

time:schedule-event

time:schedule-event agent anonymous-command tick-or-time time:schedule-event agentset anonymous-command tick-or-time time:schedule-event "observer" anonymous-command tick-or-time

Add an event to the discrete event schedule. The order in which events are added to the schedule is not important; they will be dispatched in order of the times specified as the last argument of this command. An *agent*, an *agentset*, or the string "observer" can be passed as the first argument along with an *anonymous command* as the second. The anonymous command is executed by the agent(s) or the observer at *tick-or-time* (either a number indicating the tick or a LogoTime), which is a time greater than or equal to the present moment (>= *ticks*).*.

If *tick-or-time* is a LogoTime, then the discrete event schedule must be anchored (see time:anchor-schedule). If tick-or-time is in the past (less than the current tick/time), a run-time error is raised. (The *is-after?* primitive can be used to defend against this error: add an event to the schedule only if its scheduled time is after the current time.)

Once an event has been added to the discrete event schedule, there is no way to remove or cancel it.

```
time:schedule-event turtles [ [] -> go-forward ] 1.0
time:schedule-event turtles [ [] -> fd 1 ] 1.0
time:schedule-event "observer" [ [] -> print "hello world" ] 1.0
```

---

time:schedule-event-shuffled

time:schedule-event-shuffled agentset anonymous-command tick-or-time

Add an event to the discrete event schedule and shuffle the agentset during execution. This is identical to *time:schedule-event* but the individuals in the agentset execute the action in randomized order.

```
time:schedule-event-shuffled turtles [ [] -> go-forward ] 1.0
```

---

time:schedule-repeating-event time:schedule-repeating-event-with-period

time:schedule-repeating-event agent anonymous-command tick-or-time interval-number time:schedule-repeating-event agentset anonymous-command tick-or-time-number interval-number time:schedule-repeating-event "observer" anonymous-command tick-or-time interval-number time:schedule-repeating-event-with-period agent anonymous-command tick-or-time period-duration period-type-string time:schedule-repeating-event-with-period agentset anonymous-command tick-or-time-number period-duration period-type-string time:schedule-repeating-event-with-period "observer" anonymous-command tick-or-time period-duration period-type-string

Add a repeating event to the discrete event schedule. This primitive behaves almost identically to *time:schedule-event* except that after the event is dispatched it is immediately rescheduled *interval-number* ticks into the future using the same *agent* (or *agentset*) and *anonymous-command*. If the schedule is anchored (see time:anchor-schedule), then *time:schedule-repeating-event-with-period* can be used to expressed the repeat interval as a period (e.g. 1 "day" or 2.5 "hours"). Warning: repeating events can cause an infinite loop to occur if you execute the schedule with time:go. To avoid infinite loops, use time:go-until.

```
time:schedule-repeating-event turtles [ [] -> go-forward ] 2.5 1.0
time:schedule-repeating-event-with-period turtles [ [] -> go-forward ] 2.5 1.0 "hours"
```

---

time:schedule-repeating-event-shuffled time:schedule-repeating-event-shuffled-with-period

time:schedule-repeating-event-shuffled agentset anonymous-command tick-or-time-number interval-number time:schedule-repeating-event-shuffled-with-period agentset anonymous-command tick-or-time-number interval-number

Add a repeating event to the discrete event schedule and shuffle the agentset during execution. This is identical to *time:schedule-repeating-event* but the individuals in the agentset execute the action in randomized order. If the schedule is anchored (see time:anchor-schedule), then *time:schedule-repeating-event-shuffled-with-period* can be used to expressed the repeat interval as a period (e.g. 1 "day" or 2.5 "hours"). Warning: repeating events can cause an infinite loop to occur if you execute the schedule with time:go. To avoid infinite loops, use time:go-until.

```
time:schedule-repeating-event-shuffled turtles [ [] -> go-forward ] 2.5 1.0
time:schedule-repeating-event-shuffled-with-period turtles [ [] -> go-forward ] 2.5 1.0 "month"
```

---

time:clear-schedule

time:clear-schedule

Clear all events from the discrete event schedule.

```
time:clear-schedule
```

---

time:go

time:go

Dispatch (execute) all of the events in the discrete event schedule. When each event is executed, NetLogo's tick counter (and any LogoTime variables anchored to ticks) is updated to that event's time. Note that this command will continue to dispatch events until the discrete event schedule is empty. If repeating events are in the discrete event schedule or if procedures in the schedule end up scheduling new events, it's possible for this to become an infinite loop.

```
time:go
```

---

time:go-until

time:go-until halt-tick-or-time

Dispatch all of the events in the discrete event schedule that are scheduled for times up until *halt-tick-or-time*. If the temporal extent of your model is known in advance, this variant of *time:go* is the recommended way to dispatch your model. This primitive can also be used to execute all the events scheduled before the next whole tick, which is useful if other model actions take place on whole ticks.

```
time:go-until 100.0
;; Execute events up to tick 100

time:go-until time:plus t-datetime 1.0 "hour"
;; Execute events within the next hour; t-datetime is the current time.
```

---

time:show-schedule

time:show-schedule

Reports all of the events in the schedule as a single string in tab-separated format with three columns: tick,semi-colon-separated-list-of-agents,anonymous-command.

```
print time:show-schedule
```

---

time:size-of-schedule

time:size-of-schedule

Reports the number of events in the discrete event schedule.

```
if time:size-of-schedule > 0[
  time:go
]
```

---

back to top

Building with SBT

Using SBT, create a package that will generate the time.jar. For example:

```
sbt package
```

If compilation succeeds, `time.jar` will be created. See Installation for instructions on where to put your compiled extension.

Authors

Colin Sheppard, Steve Railsback and Jacob Kelter

Feedback? Bugs? Feature Requests?

Please visit the github issue tracker to submit comments, bug reports, or feature requests. I'm also more than willing to accept pull requests.

Credits

This extension is inspired by the Ecoswarm Time Manager Library. Allison Campbell helped benchmark discrete event scheduling versus static scheduling. The extension was funded in part by the Swarm Development Group.

Terms of Use

back to top