# Package 'nlrx'

November 14, 2020

**Type** Package

**Title** Setup, Run and Analyze 'NetLogo' Model Simulations from 'R' via 'XML'

**Version** 0.4.2

**Maintainer** Jan Salecker <jsaleck@gwdg.de>

**Description** Setup, run and analyze 'NetLogo' (<https://ccl.northwestern.edu/netlogo/>) model simulations in 'R'.
'nlrx' experiments use a similar structure as 'NetLogos' Behavior Space experiments.
However, 'nlrx' offers more flexibility and additional tools for running and analyzing complex simulation designs and sensitivity analyses.
The user defines all information that is needed in an intuitive framework, using class objects.
Experiments are submitted from 'R' to 'NetLogo' via 'XML' files that are dynamically written, based on specifications defined by the user.
By nesting model calls in future environments, large simulation design with many runs can be executed in parallel.
This also enables simulating 'NetLogo' experiments on remote high performance computing machines.
In order to use this package, 'Java' and 'NetLogo' (>= 5.3.1) need to be available on the executing system.

**License** GPL-3

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 7.1.0

**URL** https://docs.ropensci.org/nlrx/, https://github.com/ropensci/nlrx/

**BugReports** https://github.com/ropensci/nlrx/issues/

**Imports** XML, lhs, sensitivity, dplyr, readr, purrr, furrr, magrittr, stats, utils, GenSA, genalg, raster, rstudioapi, sf, tidyr, methods, tibble, crayon, igraph, stringr, progressr, EasyABC

**Depends** R (>= 3.3)

**Suggests** testthat, knitr, rmarkdown, covr

**VignetteBuilder** knitr

**SystemRequirements** NetLogo (>= 5.3.1), Java (JDK 1.8), pandoc, OpenSSL, udunits-2, PROJ.4, GEOS, GDAL, libxml2

**NeedsCompilation** no

**Author** Jan Salecker [aut, cre] (<https://orcid.org/0000-0002-9000-4229>), Marco Sciaini [aut] (<https://orcid.org/0000-0002-3042-5435>), Marina Papadopoulou [rev] (Marina reviewed the package for ropensci, see <https://github.com/ropensci/software-review/issues/262>)

**Repository** CRAN

**Date/Publication** 2020-11-14 00:00:02 UTC

# R topics documented:

---

`nlrx-package` *nlrx: A package for running NetLogo simulations from R.*

---

## Description

The nlrx package provides tools to setup NetLogo simulations in R. It uses a similar structure as NetLogos Behavior Space but offers more flexibility and additional tools for running sensitivity analyses.

## Details

Get started:

General information that is needed to run NetLogo simulations remotely, such as path to the NetLogo installation folder is stored within a `nl` class object. Nested within this `nl` class are the classes `experiment` and `simdesign`. The `experiment` class stores all experiment specifications. After attaching a valid experiment, a `simdesign` class object can be attached to the `nl` class object, by using one of the simdesign helper functions. These helper functions create different parameter input matrices from the experiment variable definitions that can then be executed by the `run_nl_one()` and `run_nl_all()` functions. The nested design allows to store everything related to the experiment within one R object. Additionally, different simdesign helper functions can be applied to the same `nl` object in order to repeat the same experiment with different parameter exploration methods (simdesigns).

Step by step application example

The "Wolf Sheep Predation" model from the NetLogo models library is used to present a basic example on how to setup and run NetLogo model simulations from R.

Step 1: Create a nl object:

The nl object holds all information on the NetLogo version, a path to the NetLogo directory with the defined version, a path to the model file, and the desired memory for the java virtual machine. Depending on the operation system, paths to NetLogo and the model need to be adjusted.

```
library(nlrx)
# Windows default NetLogo installation path (adjust to your needs!):
netlogopath <- file.path("C:/Program Files/NetLogo 6.0.3")
modelpath <- file.path(netlogopath, "app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo")
outpath <- file.path("C:/out")
# Unix default NetLogo installation path (adjust to your needs!):
netlogopath <- file.path("/home/NetLogo 6.0.3")
modelpath <- file.path(netlogopath, "app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo")
outpath <- file.path("/home/out")
nl <- nl(nlversion = "6.0.3",
         nlpath = netlogopath,
         modelpath = modelpath,
         jvmmem = 1024)
```

Step 2: Attach an experiment

The experiment object is organized in a similar fashion as NetLogo Behavior Space experiments. It contains all information that is needed to generate a simulation parameter matrix and to execute the NetLogo simulations. Details on the specific slots of the experiment class can be found in the package documentation (`?experiment`) and the "Advanced configuration" vignette.

```
nl@experiment <- experiment(expname="wolf-sheep",
                            outpath=outpath,
                            repetition=1,
                            tickmetrics="true",
                            idsetup="setup",
                            idgo="go",
                            runtime=50,
                            evalticks=seq(40,50),
                  metrics=c("count sheep", "count wolves", "count patches with [pcolor = green]"),
                  variables = list('initial-number-sheep' = list(min=50, max=150, qfun="qunif"),
                              'initial-number-wolves' = list(min=50, max=150, qfun="qunif")),
                   constants = list("model-version" = "\"sheep-wolves-grass\"",
                                    "grass-regrowth-time" = 30,
                                    "sheep-gain-from-food" = 4,
                                    "wolf-gain-from-food" = 20,
                                    "sheep-reproduce" = 4,
                                    "wolf-reproduce" = 5,
                                    "show-energy?" = "false"))
```

Step 3: Attach a simulation design

While the experiment defines the variables and specifications of the model, the simulation design creates a parameter input table based on these model specifications and the chosen simulation design method. nlrx provides a bunch of different simulation designs, such as full-factorial, latin-hypercube, sobol, morris and eFast (see "Simdesign Examples" vignette for more information on

simdesigns). All simdesign helper functions need a properly defined nl object with a valid experiment design. Each simdesign helper also allows to define a number of random seeds that are randomly generated and can be used to execute repeated simulations of the same parameter matrix with different random-seeds (see "Advanced configuration" vignette for more information on random-seed and repetition management). A simulation design is attached to a nl object by using one of the simdesign helper functions:

```
nl@simdesign <- simdesign_lhs(nl=nl,
                              samples=100,
                              nseeds=3,
                              precision=3)
```

Step 4: Run simulations

All information that is needed to run the simulations is now stored within the nl object. The `run_nl_one()` function allows to run one specific simulation from the siminput parameter table. The `run_nl_all()` function runs a loop over all simseeds and rows of the parameter input table siminput. The loops are constructed in a way that allows easy parallelisation, either locally or on remote HPC machines (see "Advanced configuration" vignette for more information on parallelisation).

```
results <- run_nl_all(nl = nl)
```

Step 5: Attach results to nl and run analysis

nlrx provides method specific analysis functions for each simulation design. Depending on the chosen design, the function reports a tibble with aggregated results or sensitivity indices. In order to run the `analyze_nl()` function, the simulation output has to be attached to the nl object first. After attaching the simulation results, these can also be written to the defined outpath of the experiment object.

```
# Attach results to nl object:
setsim(nl, "simoutput") <- results
# Write output to outpath of experiment within nl
write_simoutput(nl)
# Do further analysis:
analyze_nl(nl)
```

## Author(s)

Jan Salecker <jsaleck@gwdg.de>

## See Also

Useful links:

- https://docs.ropensci.org/nlrx/
- https://github.com/ropensci/nlrx/
- Report bugs at https://github.com/ropensci/nlrx/issues/

`analyze_nl` *Analyze NetLogo simulation output*

## Description

Analyze NetLogo simulation output

## Usage

```
analyze_nl(nl, metrics = getexp(nl, "metrics"), funs = list(mean = mean))
```

## Arguments

| | |
|---|---|
| `nl` | nl object |
| `metrics` | vector of strings defining metric columns for evaluation. Defaults to metrics of the experiment within the nl object |
| `funs` | list with the summary metrics for the sensitivity results |

## Details

The analyze_nl function runs basic analyses on NetLogo simulation output. In order to execute this function, simulation output needs to be attached to the simdesign first with `setsim(nl,"output") <-results`.

analyze_nl calls different post-processing analysis functions, depending on the specified method in the simdesign object of the nl object.

**The following simdesign are currently supported:**

simdesign_ff

Calls analyze_ff. The function calculates aggregated output metrics by dropping random seeds and aggregating values with the provided functions.

simdesign_lhs

Calls analyze_lhs. The function calculates aggregated output metrics by dropping random seeds and aggregating values with the provided functions.

simdesign_sobol

Calls analyze_sobol. The function calculates sobol sensitivity indices from the output results using the sensitivity package.

simdesign_sobol2007

Calls analyze_sobol2007. The function calculates sobol sensitivity indices from the output results using the sensitivity package.

simdesign_soboljansen

Calls analyze_soboljansen. The function calculates sobol sensitivity indices from the output results using the sensitivity package.

simdesign_morris

Calls analyze_morris. The function calculates morris sensitivity indices from the output results using the sensitivity package.

simdesign_eFast

Calls analyze_eFast. The function calculates eFast sensitivity indices from the output results using the sensitivity package.

**For the following simdesign no postprocessing analysis function has been implemented yet:**

simdesign_simple, simdesign_distinct, simdesign_GenSA, simdesign_GenAlg

## Value

analysis summary tibble

## Examples

```
# Load nl object including output data from testdata
nl <- nl_sobol

# Define aggregation measurements:
myfuns <- list(mean=mean, sd=sd, min=min, max=max)

# Calculate sensitivity indices:
analyze_nl(nl, funs = myfuns)
```

---

`download_netlogo` *Download NetLogo*

---

## Description

Auxiliary function to download NetLogo

## Usage

```
download_netlogo(to, version, extract = FALSE)
```

## Arguments

| | |
|---|---|
| `to` | Path to folder where the downloaded file is saved. |
| `version` | Character string naming which NetLogo Version to download (see Details) |
| `extract` | TRUE/FALSE, if TRUE downloaded archive is extracted to subfolder of `to` (only unix) |

## Details

Supported Versions for Download and Usage (parameter `version`):

- "6.1.1" = NetLogo Version 6.1.1
- "6.1.0" = NetLogo Version 6.1.0
- "6.0.4" = NetLogo Version 6.0.4
- "6.0.3" = NetLogo Version 6.0.3
- "6.0.2" = NetLogo Version 6.0.2
- "6.0.1" = NetLogo Version 6.0.1
- "6.0" = NetLogo Version 6.0
- "5.3.1" = NetLogo Version 5.3.1

## Examples

```
dlpath <- tempdir()  # adjust path to your needs
try(download_netlogo(dlpath, "6.0.3"))
```

---

# eval_simoutput *Evaluate input/output integrity*

---

## Description

Evaluate input/output integrity

## Usage

```
eval_simoutput(nl)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with attached simulation output |

## Details

This function checks if the attached simulation output in the simoutput slot of the simdesign, corresponds to the defined siminput matrix.

Warning messages are thrown if data is missing in the simoutput tibble. Additionally, missing combinations of siminputrow and random seed for which no data was found can be reported as tibble. Such a tibble can then be used directly to rerun missing combinations conveniently (see examples below)

## Examples

```
## Not run:
# Check eval_simoutput for testdata nl_lhs:
nl <- nl_lhs
eval_simoutput(nl)

# Now remove one row of simoutput and check output:
nl <- nl_lhs
nl@simdesign@simoutput <- nl@simdesign@simoutput[-1,]
check <- eval_simoutput(nl)
check

# Rerun missing combinations within check tibble:
rerun <- purrr::map_dfr(seq(nrow(check)), function(x) {
  res <- run_nl_one(nl, siminputrow=check$siminputrow[x], seed=check$seed[x])
    return(res)
    }) %>%
      dplyr::bind_rows(., nl@simdesign@simoutput)



## End(Not run)
```

---

`eval_variables_constants` *Evaluate variable validity*

---

## Description

Evaluate variables and constants defined in experiment

## Usage

```
eval_variables_constants(nl)
```

## Arguments

| | |
|---|---|
| `nl` | nl object |

## Details

This function checks if the variables and constants that are defined in the experiment are valid. It loads the model code of the NetLogo model and checks if these variables and constants really exist. In case of nonvalid entries, the function throws an error message, indicating which variables and constants are not valid. Please note, that this function might fail if the supported modelpath does not point to an existing nlogo file. This might for example happen, if the modelpath is set up for a remote cluster execution.

## Examples

```
## Not run:
nl <- nl_lhs
eval_variables_constants(nl)

## End(Not run)
```

---

`experiment` *Construct a new experiment object*

---

## Description

Construct a new experiment object

## Usage

```
experiment(
  expname = "defaultexp",
  outpath = NA_character_,
  repetition = 1,
  tickmetrics = "true",
  idsetup = "setup",
  idgo = "go",
  idfinal = NA_character_,
  idrunnum = NA_character_,
  runtime = 1,
  evalticks = NA_integer_,
  stopcond = NA_character_,
  metrics = c("count turtles"),
  metrics.turtles = list(),
  metrics.patches = NA_character_,
  metrics.links = list(),
  variables = list(),
  constants = list(),
  ...
)
```

## Arguments

| | |
|---|---|
| `expname` | A character string defining the name of the experiment, no whitespaces allowed |
| `outpath` | Path to a directory where experiment output will be stored |
| `repetition` | A number which gives the number of repetitions for each row of the simulation design input tibble |
| `tickmetrics` | Character string "true" runs defined metrics on each simulation tick. "false" runs metrics only after simulation is finished |

| | |
|---|---|
| idsetup | character string or vector of character strings, defining the name of the NetLogo setup procedure |
| idgo | character string or vector of character strings, defining the name of the NetLogo go procedure |
| idfinal | character string or vector of character strings, defining the name of NetLogo procedures that should be run after the last tick |
| idrunnum | character string, defining the name of a NetLogo global that should be used to parse the current siminputrow during model executions which can then be used for self-written output. |
| runtime | number of model ticks that should be run for each simulation |
| evalticks | vector of tick numbers defining when measurements are taken. NA_integer_ to measure each tick |
| stopcond | a NetLogo reporter that reports TRUE/FALSE. If it reports TRUE the current simulation run is stopped (e.g. "not any? turtles") |
| metrics | vector of strings defining valid NetLogo reporters that are taken as output measurements (e.g. c("count turtles", "count patches")) |
| metrics.turtles | a list with named vectors of strings defining valid turtles-own variables that are taken as output measurements (e.g. list("turtles" = c("who", "pxcor", "pycor", "color")) |
| metrics.patches | vector of strings defining valid patches-own variables that are taken as output measurements (e.g. c("pxcor", "pycor", "pcolor")) |
| metrics.links | a list with named vectors of strings defining valid links-own variables that are taken as output measurements (e.g. list("links" = c("end1", "end2"))) |
| variables | a nested list of variables that are changed within a simulation design. The name of each sublist item has to be a valid global of the defined NetLogo model. Depending on the desired simdesign each list item consist of a vector of values, a min value, a max value, a step value and a qfun (e.g. list("paramA" = list(values=c(0, 0.5, 1), min=0, max=1, step=0.1, qfun="qunif"))) |
| constants | a list of constants that are kept constant within a simulation design. The name of each list item has to be a valid global of the defined NetLogo model (e.g. list("pNUM" = 12, "pLOGIC"="TRUE", "pSTRING"="\"default\"")) |
| ... | ... |

## Details

The experiment class stores all information related to the NetLogo simulation experiment. The class holds all information that is typically entered into NetLogo Behavior Space experiments. When setting up an experiment, it is usually attached to an already defined nl object (see examples). After attaching an experiment, different simdesign helper functions can be used to attach a simdesign to the nl object simdesign. The simdesign helper functions use the variable definitions from the experiment within the nl object to generate a parameter tibble for simulations.

**The following class slots are obligatory to run an experiment:**

*repetition*

In cases, where the random seed is controlled by nlrx simdesigns, repitition should be set to one as random seeds would not differ between simulations. In cases, where the random seed is set within the NetLogo model, repitition can be increased to repeat the same parameterisation with different random seeds.

*tickmetrics*

If "true", the defined output reporters are collected on each simulation tick that is defined in evalticks. If "false" measurements are taken only on the last tick.

*idsetup, idgo*

These two class slots accept strings, or vectors of strings, defining NetLogo model procedures that should be executed for model setup (idsetup) and model execution (idgo).

*runtime*

Defines the maximum number of simulation ticks that are executed. Use 0 or NA_integer_ to execute simulations without predefined number of ticks. Warning: This is only recommended in combination with a stop condition (see slot stopcond), or NetLogo models with a built-in stop condition. Otherwise, simulations might get stuck in endless loops.

**Depending on the simdesign, the following slots may be obligatory:**

*metrics*

A vector of valid netlogo reporters that defines which measurements are taken. The optimization simdesigns need at least one defined metrics reporter for fitness calculation of the optimization algorithm.

*constants, variables*

These slots accept lists with NetLogo parameters that should be varied within a simdesign (variables) or should be kept constant (constants) for each simulation. Any NetLogo parameter that is not entered in at least one of these two lists will be set up as constant with the default value from the NetLogo interface. It is not possible to enter a NetLogo parameter in both lists (a warning message will appear when a simdesign is attached to such an experiment). All simdesigns except simdesign_simple need defined variables for setting up a parameter matrix. Variables can be defined as distinct values, value distributions or range with increment. The information that is needed, depends on the chosen simdesign (details on variable definition requirements can be found in the helpfiles of each simdesign helper function).

**All remaining slots are optional:**

*expname*

A character string defining the name of the experiment, useful for documentation purposes. The string must not contain any whitespaces.

*outpath*

A valid path to an existing directory. The directory is used by the write_simoutput function to store attached simulation results to disk in csv format.

*idfinal*

A character string or vector of strings defining NetLogo procedures that are executed at the end of each simulation (e.g. cleanup or self-written output procedures).

*idrunnum*

This slot can be used to transfer the current nlrx experiment name, random seed and runnumber (siminputrow) to NetLogo. To use this functionality, a string input field widget needs to be created on the GUI of your NetLogo model. The name of this widget can be entered into the "idrunnum" field of the experiment. During simulations, the value of this widget is automatically updated with a generated string that contains the current nlrx experiment name, random seed and siminputrow ("expname_seed_siminputrow"). For self-written output In NetLogo, we suggest to include this global variable which allows referencing the self-written output files to the collected output of the nlrx simulations in R.

*evalticks*

Only applied if tickmetrics = TRUE. Evalticks may contain a vector of integers, defining the ticks for which the defined metrics will be measured. Set evalticks to NA_integer_ to measure on every tick.

*stopcond*

The stopcond slot can be used to define a stop condition by providing a string with valid NetLogo code that reports either true or false. Each simulation will be stopped automatically, once the reporter reports true.

*metrics.patches*

The metrics.patches slot accepts a vector of valid patches-own variables of the NetLogo model. These patch variables are measured in addition to the defined metrics. Results of these metrics will be nested inside the output results tibble of the simulations. Please note that NetLogo models may contain a huge number of patches and output measurements of agent variables on each tick may need a lot of ressources.

*metrics.turtles*

The metrics.turtles slot accepts a list with named vectors of valid turtle breed metrics. Each name of a vector in this list defines a specified breed of the NetLogo model, whereas the vector defines the variables that are measured for this breed. For example metrics.turtles = list("sheep"=c("color"), "wolves"=c("who")) - would measure the color of each sheep and the who number of each wolf agent. To measure <turtles-own> variables for all turtles, use "turtles" = c(...). Be aware, that NetLogo will produce runtime errors if you measure <breed-own> variables for agents that do not belong to this breed. Please note that NetLogo models may contain a huge number of turtles and output measurements of agent variables on each tick may need a lot of ressources.

*metrics.links*

The metrics.links slot accepts a list with named vectors of valid link breed metrics. Each name of a vector in this list defines a specified link breed of the NetLogo model, whereas the vector defines the variables that are measured for this link breed. For example metrics.links = list("linktype-a"=c("end1"), "linktype-b"=c("end2")) - would measure the start agent of each linktype-a link and the end agent of each linktype-b link. To measure <links-own> variables for all links, use "links" = c(...). Be aware, that NetLogo will produce runtime errors if you measure <link-breed-own> variables for agents that do not belong to this breed. Please note that NetLogo models may contain a huge number of turtles and output measurements of agent variables on each tick may need a lot of ressources.

## Value

experiment S4 class object

## Examples

```
# To attach an experiment, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl_simple
nl@experiment <- experiment(expname="wolf-sheep",
                            outpath="C:/out/",
                            repetition=1,
                            tickmetrics="true",
                            idsetup="setup",
                            idgo="go",
                            idfinal=NA_character_,
                            idrunnum=NA_character_,
                            runtime=50,
                            evalticks=seq(40,50),
                            stopcond="not any? turtles",
                            metrics=c("count sheep",
                                      "count wolves",
                                     "count patches with [pcolor = green]"),
                            metrics.turtles=list("turtles" = c("who",
                                                 "pxcor",
                                                 "pycor",
                                                 "color")),
                            metrics.patches=c("pxcor", "pycor", "pcolor"),
                            variables = list('initial-number-sheep' =
                            list(min=50, max=150, step=10, qfun="qunif"),
                                             'initial-number-wolves' =
                            list(min=50, max=150, step=10, qfun="qunif")),
                            constants = list("model-version" =
                                             "\"sheep-wolves-grass\"",
                                             "grass-regrowth-time" = 30,
                                             "sheep-gain-from-food" = 4,
                                             "wolf-gain-from-food" = 20,
                                             "sheep-reproduce" = 4,
                                             "wolf-reproduce" = 5,
                                             "show-energy?" = "false"))
```

---

`export_nl` *Export NetLogo Experiment*

---

## Description

Export NetLogo Experiment as zip file

## Usage

```
export_nl(nl, path = dirname(getnl(nl, "modelpath")), tarfile)
```

### Arguments

| | |
|---|---|
| nl | nl object |
| path | Path to folder that contains files to run NetLogo experiment |
| tarfile | Path to folder where the experiments gets stored as zip file |

### Details

Exports your folder that contains data and scripts for NetLogo + nlrx analyses as a zip file. Furthermore, `export_nl` takes your nl object and saves it in the zipped folder. This enables other person to run the same experiment as you.

### Value

The status value returned by the external command, invisibly.

### Examples

```
## Not run:

# Load nl object from testdata:
nl <- nl_lhs
path <- getwd() # adjust path to your needs, path should point to a directory with model data
outfile <- tempfile(fileext = ".zip") # adjust file path to your needs
export_nl(nl, path = path, tarfile = outfile)

## End(Not run)
```

---

## getexp *Getter function to get a variable of an experiment object*

---

### Description

Getter function to get a variable of an experiment object

### Usage

```
getexp(nl, var)
```

### Arguments

| | |
|---|---|
| nl | nl object |
| var | valid experiment variable string |

## Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl(nlversion = "6.0.3",
nlpath = "/home/user/NetLogo 6.0.3/",
modelpath = "/home/user/NetLogo 6.0.3/app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo",
jvmmem = 1024)

# Set experiment name
setexp(nl, "expname") <- "experimentName"

# Get experiment name
getexp(nl, "experiment")
```

---

# getnl *Getter function to get a variable of a nl object*

---

## Description

Getter function to get a variable of a nl object

## Usage

```
getnl(nl, var)
```

## Arguments

| | |
|---|---|
| nl | nl object |
| var | valid nl variable string |

## Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl(nlversion = "6.0.3",
nlpath = "/home/user/NetLogo 6.0.3/",
modelpath = "/home/user/NetLogo 6.0.3/app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo",
jvmmem = 1024)

# get NetLogo version
getnl(nl, "nlversion")
```

## getsim *Getter function to get a variable of a simdesign object*

### Description

Getter function to get a variable of a simdesign object

### Usage

```
getsim(nl, var)
```

### Arguments

| | |
|---|---|
| nl | nl object |
| var | valid simdesign variable string |

### Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl(nlversion = "6.0.3",
nlpath = "/home/user/NetLogo 6.0.3/",
modelpath = "/home/user/NetLogo 6.0.3/app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo",
jvmmem = 1024)

# Set simulation seeds
setsim(nl, "simseeds") <- c(123, 456, 789)

# Set simulation seeds
getsim(nl, "simseeds")
```

## import_nl *Import NetLogo Experiment*

### Description

Import NetLogo Experiment from export_nl

### Usage

```
import_nl(tarfile, targetdir, new_session = FALSE)
```

### Arguments

| | |
|---|---|
| tarfile | Path to tarfile that contains files to run NetLogo experiment |
| targetdir | Path to folder where the experiments gets extracted |
| new_session | If TRUE, opens a new RStudio Session with an Rproj |

## Details

Imports NetLogo experiments that were saved with `export_nl`. If the folder comes with an .Rproj file (which is recommended because relative paths enhance the reproducability of your analysis), `import_nl` opens this project and loads the nl object in your R environment.

## Value

The status value returned by the external command, invisibly.

## Examples

```
## Not run:

infile <- "/home/user/test.zip"
targetdirectory <- "/home/user/test"
import_nl(infile, targetdirectory)

## End(Not run)
```

---

`nl` *Construct a new nl object*

---

## Description

Construct a new nl object

## Usage

```
nl(
  nlversion = "6.0.2",
  nlpath = character(),
  modelpath = character(),
  jvmmem = 1024,
  experiment = methods::new("experiment"),
  simdesign = methods::new("simdesign"),
  ...
)
```

## Arguments

| | |
|---|---|
| `nlversion` | A character string defining the NetLogo version that is used |
| `nlpath` | Path to the NetLogo main directory matching the defined version |
| `modelpath` | Path to the NetLogo model file (*.nlogo) that is used for simulations |
| `jvmmem` | Java virtual machine memory capacity in megabytes |
| `experiment` | Holds a experiment S4 class object |
| `simdesign` | Holds a simdesign S4 class object |
| `...` | ... |

## Details

nl objects are the main class objects used in the nlrx package. These objects store all information that is needed to run NetLogo simulations. nl objects are initialized with basic information on Netlogo and the model.

After setting up the nl object, an experiment needs to be attached. The experiment class stores all information related to the NetLogo simulation experiment, such as runtime, variables, constants, measurements, and more.

After attaching an experiment, different simdesign helper functions can be used to attach a simdesign to the nl object. The simdesign helper functions use the variable definitions from the experiment within the nl object to generate a parameter tibble for simulations.

## Value

nl S4 class object

## Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
# Windows default NetLogo installation path (adjust to your needs!):
netlogopath <- file.path("C:/Program Files/NetLogo 6.0.3")
modelpath <- file.path(netlogopath, "app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo")
outpath <- file.path("C:/out")
# Unix default NetLogo installation path (adjust to your needs!):
netlogopath <- file.path("/home/NetLogo 6.0.3")
modelpath <- file.path(netlogopath, "app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo")
outpath <- file.path("/home/out")

nl <- nl(nlversion = "6.0.3",
         nlpath = netlogopath,
         modelpath = modelpath,
         jvmmem = 1024)
```

---

# nldoc *Create NetLogo documentation*

---

## Description

Create NetLogo documentation

## Usage

```
nldoc(
  modelfiles,
  outpath,
  infotab = TRUE,
  gui = TRUE,
```

```
  bs = TRUE,
  output_format = "html",
  number_sections = TRUE,
  theme = "journal",
  date = today(),
  toc = TRUE
)
```

## Arguments

| | |
|---|---|
| modelfiles | vector of filepaths to model files |
| outpath | Path to folder where rendered documentation should be created |
| infotab | TRUE/FALSE, if TRUE infotab section will be included in the documentation |
| gui | TRUE/FALSE, if TRUE a table with GUI elements from the model will be included in the documentation |
| bs | TRUE/FALSE, if TRUE a table with behavior space experiments will be included in the documentation |
| output_format | either "html", "pdf" or "docx" |
| number_sections | TRUE/FALSE, if TRUE sections in the documentation will be numbered |
| theme | markdown theme, supported themes are "journal", "cerulean", "flatly", "readable", "spacelab", "united", "cosmo" |
| date | date that is printed in the documentation header |
| toc | TRUE/FALSE, if TRUE the documentation contains a table of contents - only for html and pdf output format |

## Details

nldoc reads model code from the provided model files. The code is then split into several groups (code, gui, behavior space). The procedures then finds noxygen commands within the NetLogo model code. For a complete list of noxygen commands type ?nldoc These commands are translated into a markdown documentation file. If needed, tables of gui elements and behavior space experiments are added to the markdown file. Finally, the document is rendered in the specified format.

## Examples

```
## Not run:

# List model files (.nls subfiles are also supported)
modelfiles <- c("https://raw.githubusercontent.com/nldoc/nldoc_pg/master/WSP.nlogo",
                "https://raw.githubusercontent.com/nldoc/nldoc_pg/master/WSP.nls")

# Define output directory:
outdir <- tempdir()  # adjust path to your needs

# Create documentation:
```

```
nldoc(modelfiles = modelfiles,
      infotab=TRUE,
      gui=TRUE,
      bs=TRUE,
      outpath = outdir,
      output_format = "html",
      number_sections = TRUE,
      theme = "cosmo",
      date = date(),
      toc = TRUE)

## End(Not run)
```

---

`nldoc_network` *Create NetLogo procedure network*

---

## Description

Create NetLogo procedure network

## Usage

```
nldoc_network(modelfiles)
```

## Arguments

| | |
|---|---|
| `modelfiles` | vector of filepaths to model files |

## Details

Reads model code from the provided model files. The procedure identifies NetLogo procedures and searches for procedure calls within the code. From this data, an igraph network is created and returned. This network can be used to plot the model procedure network and identify model components.

## Value

network of model procedures (igraph)

## Examples

```
## Not run:

# List model files (.nls subfiles are also supported)
modelfiles <- c("https://raw.githubusercontent.com/nldoc/nldoc_pg/master/WSP.nlogo",
                "https://raw.githubusercontent.com/nldoc/nldoc_pg/master/WSP.nls")

# Determine the function network:
```

```
nw <- nldoc_network(modelfiles)

# Determine communities within the network and plot using Igraph package:
library(igraph)
com <- walktrap.community(nw)
V(nw)$community <- com$membership
rain <- rainbow(14, alpha=.5)
V(nw)$color <- rain[V(nw)$community]

plot(nw,
     edge.arrow.size=.2,
     vertex.label.color="black",
     vertex.label.dist=1,
     vertex.size=5,
     edge.curved=0,
     vertex.label.cex=.5,
     layout=layout_with_fr(nw, niter = 2000))

# Interactive plot using igraph::tkplot
tkplot(nw, layout=layout_with_fr(nw, niter = 2000))

## End(Not run)
```

---

## nl_distinct *Wolf Sheep model sample data: simdesign distinct*

---

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a distinct simulation design. It also contains the model outputs within the simdesign object.

### Usage

```
nl_distinct
```

### Format

nl object with defined experiment, simdesign and model output

---

## nl_eFast *Wolf Sheep model sample data: simdesign eFast*

---

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run an eFast simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

## Usage

```
nl_eFast
```

## Format

nl object with defined experiment, simdesign and model output

---

`nl_ff` *Wolf Sheep model sample data: simdesign ff*

---

## Description

The dataset contains a complete nl object. The nl object has been used to setup and run a full-factorial simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

## Usage

```
nl_ff
```

## Format

nl object with defined experiment, simdesign and model output

---

`nl_gensa` *Wolf Sheep model sample data: gensa*

---

## Description

The dataset contains a complete nl object. The nl object has been used to setup and run a simulated annealing simulation design.

## Usage

```
nl_gensa
```

## Format

nl object with defined experiment, simdesign and model output

## nl_lhs *Wolf Sheep model sample data: simdesign lhs*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a latin hypercube simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

### Usage

```
nl_lhs
```

### Format

nl object with defined experiment, simdesign and model output

## nl_morris *Wolf Sheep model sample data: simdesign morris*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a morris simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

### Usage

```
nl_morris
```

### Format

nl object with defined experiment, simdesign and model output

## nl_simple *Wolf Sheep model sample data: simdesign simple*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a simple simulation design. It also contains the model outputs within the simdesign object.

### Usage

```
nl_simple
```

### Format

nl object with defined experiment, simdesign and model output

## nl_sobol *Wolf Sheep model sample data: simdesign sobol*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a sobol simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

### Usage

```
nl_sobol
```

### Format

nl object with defined experiment, simdesign and model output

## nl_sobol2007 *Wolf Sheep model sample data: simdesign sobol2007*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a sobol2007 simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

### Usage

```
nl_sobol2007
```

### Format

nl object with defined experiment, simdesign and model output

## nl_soboljansen *Wolf Sheep model sample data: simdesign soboljansen*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a soboljansen simulation design. It also contains the model outputs within the simdesign object. Further analysis output can be derived by submitting the dataset to analyze_nl().

### Usage

```
nl_soboljansen
```

### Format

nl object with defined experiment, simdesign and model output

## nl_spatial *Wolf Sheep model sample data: spatial*

### Description

The dataset contains a complete nl object. The nl object has been used to setup and run a simple simulation design. For these simulations, spatial agent output has been collected. metrics.turtles was set up to measure turtle coordinates. metrics.patches was set up to measure colors of cells. It also contains the model outputs within the simdesign object.

### Usage

```
nl_spatial
```

### Format

nl object with defined experiment, simdesign and model output

## nl_to_graph *Generate igraph objects from measured turtles and links metrics*

### Description

Generate igraph objects from measured turtles and links metrics

### Usage

```
nl_to_graph(nl)
```

### Arguments

| | |
|---|---|
| nl | nl object<br>Generate igraph objects from measured turtles and links metrics. Output has to be attached to the simdesign first with simoutput(nl) <- results I graph objects are created automatically for each combination of random-seed, siminputrow and step. An additional column with igraph objects is attached to the original output results tibble of the nl object. In order to generate igraph objects some metrics are mandatory: The metrics.turtles slot of the experiment must contain "who" numbers (see example experiment). Additional turtle metrics will be stored as properties of the igraph vertices. The metrics.links slot of the experiment must contain "who" numbers of link end1 and end2 (see example experiment). Additional link metrics will be stored as properties of the igraph edges. |

## Examples

```
## Not run:
## Example running the Giant Component model from the NetLogo models library:
library(nlrx)
library(igraph)
# Windows default NetLogo installation path (adjust to your needs!):
netlogopath <- file.path("C:/Program Files/NetLogo 6.0.4")
modelpath <- file.path(netlogopath, "app/models/Sample Models/Networks/Giant Component.nlogo")
outpath <- file.path("C:/out")
# Unix default NetLogo installation path (adjust to your needs!):
netlogopath <- file.path("/home/NetLogo 6.0.4")
modelpath <- file.path(netlogopath, "app/models/Sample Models/Networks/Giant Component.nlogo")
outpath <- file.path("/home/out")

nl <- nl(nlversion = "6.0.4",
         nlpath = netlogopath,
         modelpath = modelpath,
         jvmmem = 1024)

nl@experiment <- experiment(expname="networks",
                            outpath=outpath,
                            repetition=1,
                            tickmetrics="false",
                            idsetup="setup",
                            idgo="go",
                            runtime=50,
                            metrics.turtles = list("turtles" = c("who", "color")),
                      metrics.links = list("links" = c("[who] of end1", "[who] of end2")),
                            constants = list("num-nodes" = 80,
                                             "layout?" = "true"))

nl@simdesign <- simdesign_simple(nl, 1)
nl@simdesign@simoutput <- run_nl_all(nl)
nl.graph <- nl_to_graph(nl)

## Extract graph of tick 1:
nl.graph.i <- nl.graph$spatial.links[[1]]
## Set vertex colors by measured color variable:
vcols <- c("7" = "grey", "15" = "red")
V(nl.graph.i)$color <- vcols[as.character(V(nl.graph.i)$color)]
## Set edge colors by measured link breed:
ecols <- c("links" = "black")
E(nl.graph.i)$color <- ecols[E(nl.graph.i)$breed]

## Plot:
plot.igraph(nl.graph.i, vertex.size=8, vertex.label=NA, edge.arrow.size=0.2)


## End(Not run)
```

`nl_to_points` *Get spatial data from metrics.turtles output*

## Description

Turn turtle metrics from NetLogo in spatial data objects

## Usage

```
nl_to_points(nl, coords)
```

## Arguments

| | |
|---|---|
| `nl` | nl object |
| `coords` | nl object |

## Details

Converts measured metrics.turtles into spatial sf point objects. In order to so, a pair of turtle coordinates needs to be measured. Any additional metrics will be stored as properties of the spatial points. Because turtle coordinates in NetLogo can be measured in two formats, pxcor/pycor or xcor/ycor coordinates, the type of coordinate used for transformation to spatial objects need to be defined, using the parameter coords: "px" for pxcor/pycor coordinates, "x" for xcor/ycor coordinates.

In order to use this function, simulation results need to be attached to the nl object first.

## Value

tibble with spatial data objects

## Examples

```
# Load nl object (with spatial output data already attached) from test data:
nl <- nl_spatial

# Convert turtle metrics (pxcor/pycor) to spatial point objects:
results.sf <- nl_to_points(nl, coords="px")
```

## nl_to_raster *Get spatial data from metrics.patches output*

### Description

Turn patch metrics from NetLogo in spatial data objects

### Usage

```
nl_to_raster(nl)
```

### Arguments

| | |
|---|---|
| nl | nl object |

### Details

Converts measured metrics.patches into spatial raster objects. In order to so, a patch coordinates needs to be measured (pxcor/pycor). For each additional patch metric, a raster will be created using the measured coordinates as x and y and the additional metric as z dimension. In case of multiple measured metrics, a raster stack with one raster for each metric will be reported.

In order to use this function, simulation results need to be attached to the nl object first.

### Value

tibble with spatial data objects

### Examples

```
# Load nl object (with spatial output data already attached) from test data:
nl <- nl_spatial

# Convert patch metrics to spatial raster objects:
results.raster <- nl_to_raster(nl)
```

## print.nl *Print content of nl object*

### Description

Print content of nl object and embedded experiment and simdesign objects to console

### Usage

```
## S3 method for class 'nl'
print(x, ...)
```

### Arguments

| | |
|---|---|
| x | nl object to print |
| ... | further arguments passed to or from other methods |

### Details

Print content of the provided nl object in a readable format.

### Examples

```
print(nl_lhs)
```

---

## report_model_parameters

*Report globals from a NetLogo model that is defined within a nl object*

---

### Description

Report globals from a NetLogo model that is defined within a nl object

### Usage

```
report_model_parameters(nl)
```

### Arguments

| | |
|---|---|
| nl | nl object with a defined modelpath that points to a NetLogo model (*.nlogo) |

### Details

The function reads the NetLogo model file that is defined within the nl object and reports all global parameters that are defined as widget elements on the GUI of the NetLogo model. Only globals that are found by this function are valid globals that can be entered into the variables or constants vector of an experiment object.

## Examples

```
## Not run:
nl <- nl_lhs
report_model_parameters(nl)

## End(Not run)
```

---

`run_nl_all` *Execute all NetLogo simulations from a nl object*

---

## Description

Execute all NetLogo simulations from a nl object with a defined experiment and simdesign

## Usage

```
run_nl_all(
  nl,
  split = 1,
  cleanup.csv = TRUE,
  cleanup.xml = TRUE,
  cleanup.bat = TRUE
)
```

## Arguments

| | |
|---|---|
| `nl` | nl object |
| `split` | number of parts the job should be split into |
| `cleanup.csv` | TRUE/FALSE, if TRUE temporary created csv output files will be deleted after gathering results. |
| `cleanup.xml` | TRUE/FALSE, if TRUE temporary created xml output files will be deleted after gathering results. |
| `cleanup.bat` | TRUE/FALSE, if TRUE temporary created bat/sh output files will be deleted after gathering results. |

## Details

run_nl_all executes all simulations of the specified NetLogo model within the provided nl object. The function loops over all random seeds and all rows of the siminput table of the simdesign of nl. The loops are created by calling future_map_dfr, which allows running the function either locally or on remote HPC machines. The logical cleanup variables can be set to FALSE to preserve temporary generated output files (e.g. for debugging). cleanup.csv deletes/keeps the temporary generated model output files from each run. cleanup.xml deletes/keeps the temporary generated experiment xml files from each run. cleanup.bat deletes/keeps the temporary generated batch/sh commandline files from each run.

When using run_nl_all in a parallelized environment (e.g. by setting up a future plan using the future package), the outer loop of this function (random seeds) creates jobs that are distributed to available cores of the current machine. The inner loop (siminputrows) distributes simulation tasks to these cores. However, it might be advantageous to split up large jobs into smaller jobs for example to reduce the total runtime of each job. This can be done using the split parameter. If split is > 1 the siminput matrix is split into smaller parts. Jobs are created for each combination of part and random seed. If the split parameter is set such that the siminput matrix can not be splitted into equal parts, the procedure will stop and throw an error message.

## Value

tibble with simulation output results

## Examples

```
## Not run:

# Load nl object from test data:
nl <- nl_lhs

# Execute all simulations from an nl object with properly attached simdesign.
results <- run_nl_all(nl)

# Run in parallel on local machine:
library(future)
plan(multisession)
results <- run_nl_all(nl)


## End(Not run)
```

---

`run_nl_dyn` *Execute NetLogo simulation without pregenerated parametersets*

---

## Description

Execute NetLogo simulation from a nl object with a defined experiment and simdesign but no pregenerated input parametersets

## Usage

```
run_nl_dyn(
  nl,
  seed,
  cleanup.csv = TRUE,
  cleanup.xml = TRUE,
  cleanup.bat = TRUE
)
```

## Arguments

| | |
|---|---|
| nl | nl object |
| seed | a random seed for the NetLogo simulation |
| cleanup.csv | TRUE/FALSE, if TRUE temporary created csv output files will be deleted after gathering results. |
| cleanup.xml | TRUE/FALSE, if TRUE temporary created xml output files will be deleted after gathering results. |
| cleanup.bat | TRUE/FALSE, if TRUE temporary created bat/sh output files will be deleted after gathering results. |

## Details

run_nl_dyn can be used for simdesigns where no predefined parametersets exist. This is the case for dynamic designs, such as Simulated Annealing and Genetic Algorithms, where parametersets are dynamically generated, based on the output of previous simulations. The logical cleanup variables can be set to FALSE to preserve temporary generated output files (e.g. for debugging). cleanup.csv deletes/keeps the temporary generated model output files from each run. cleanup.xml deletes/keeps the temporary generated experiment xml files from each run. cleanup.bat deletes/keeps the temporary generated batch/sh commandline files from each run.

## Value

simulation output results can be tibble, list, ...

## Examples

```
## Not run:

# Load nl object form test data:
nl <- nl_lhs

# Add genalg simdesign:
nl@simdesign <- simdesign_GenAlg(nl=nl,
                                 popSize = 200,
                                 iters = 100,
                                 evalcrit = 1,
                                 nseeds = 1)

# Run simulations:
results <- run_nl_dyn(nl)


## End(Not run)
```

run_nl_one *Execute one NetLogo simulation from a nl object*

## Description

Execute one NetLogo simulation from a nl object with a defined experiment and simdesign

## Usage

```
run_nl_one(
  nl,
  seed,
  siminputrow,
  cleanup.csv = TRUE,
  cleanup.xml = TRUE,
  cleanup.bat = TRUE,
  writeRDS = FALSE
)
```

## Arguments

| | |
|---|---|
| nl | nl object |
| seed | a random seed for the NetLogo simulation |
| siminputrow | rownumber of the input tibble within the attached simdesign object that should be executed |
| cleanup.csv | TRUE/FALSE, if TRUE temporary created csv output files will be deleted after gathering results. |
| cleanup.xml | TRUE/FALSE, if TRUE temporary created xml output files will be deleted after gathering results. |
| cleanup.bat | TRUE/FALSE, if TRUE temporary created bat/sh output files will be deleted after gathering results. |
| writeRDS | TRUE/FALSE, if TRUE an rds file with the simulation results will be written to the defined outpath folder of the experiment within the nl object. |

## Details

run_nl_one executes one simulation of the specified NetLogo model within the provided nl object. The random seed is set within the NetLogo model to control stochasticity. The siminputrow number defines which row of the input data tibble within the simdesign object of the provided nl object is executed. The logical cleanup variables can be set to FALSE to preserve temporary generated output files (e.g. for debugging). cleanup.csv deletes/keeps the temporary generated model output files from each run. cleanup.xml deletes/keeps the temporary generated experiment xml files from each run. cleanup.bat deletes/keeps the temporary generated batch/sh commandline files from each run.

This function can be used to run single simulations of a NetLogo model.

## Value

tibble with simulation output results

## Examples

```
## Not run:

# Load nl object from test data:
nl <- nl_lhs

# Run one simulation:
results <- run_nl_one(nl = nl,
                      seed = getsim(nl, "simseeds")[1],
                      siminputrow = 1)


## End(Not run)
```

---

# setexp<- *Setter function to set a variable of an experiment object*

---

## Description

Setter function to set a variable of an experiment object

## Usage

```
setexp(nl, var) <- value
```

## Arguments

| | |
|---|---|
| nl | nl object |
| var | valid experiment variable string |
| value | valid value for the specified variable |

## Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl(nlversion = "6.0.3",
nlpath = "/home/user/NetLogo 6.0.3/",
modelpath = "/home/user/NetLogo 6.0.3/app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo",
jvmmem = 1024)

# Set experiment name
setexp(nl, "expname") <- "experimentName"
```

## setnl<- *Setter function to set a variable of a nl object*

### Description

Setter function to set a variable of a nl object

### Usage

```
setnl(nl, var) <- value
```

### Arguments

| | |
|---|---|
| nl | nl object |
| var | valid nl variable string |
| value | valid value for the specified variable |

### Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl(
nlpath = "/home/user/NetLogo 6.0.3/",
modelpath = "/home/user/NetLogo 6.0.3/app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo",
jvmmem = 1024)

# set NetLogo version
setnl(nl, "nlversion") <- "6.0.3"
```

## setsim<- *Setter function to set a variable of a simdesign object*

### Description

Setter function to set a variable of a simdesign object

### Usage

```
setsim(nl, var) <- value
```

### Arguments

| | |
|---|---|
| nl | nl object |
| var | valid simdesign variable string |
| value | valid value for the specified variable |

## Examples

```
# Example for Wolf Sheep Predation model from NetLogo models library:
nl <- nl(nlversion = "6.0.3",
nlpath = "/home/user/NetLogo 6.0.3/",
modelpath = "/home/user/NetLogo 6.0.3/app/models/Sample Models/Biology/Wolf Sheep Predation.nlogo",
jvmmem = 1024)

# Set simulation seeds
setsim(nl, "simseeds") <- c(123, 456, 789)
```

---

`simdesign` *Construct a new simdesign object*

---

## Description

Construct a new simdesign object

## Usage

```
simdesign(
  simmethod = character(),
  siminput = tibble::tibble(),
  simobject = list(),
  simseeds = NA_integer_,
  simoutput = tibble::tibble(),
  ...
)
```

## Arguments

| | |
|---|---|
| `simmethod` | character string defining the method of the simulation design |
| `siminput` | tibble providing input parameterisations for the NetLogo model (cols=parameter, rows=runs) |
| `simobject` | used for some methods to store additional information (sobol, morris, eFast) |
| `simseeds` | a vector or model random seeds |
| `simoutput` | tibble containing model results |
| `...` | ... |

## Details

The simulation design class holds information on the input parameter design of model simulations. It also stores information that is needed to run method specific analysis functions. The simseeds can be used to run all model simulations that are defined within the siminput tibble several times with changing random-seeds. While it is possible to add simdesign directly with this function, we suggest to use our simdesign_helper functions. A simulation design can be attached to a nl

object by using one of these simdesign_helper functions on an already defined nl object with a valid experiment. All simdesign helpers use the defined constants and variables of the experiment to create the siminput tibble. NetLogo parameters that are not defined in constants or variables will be set with their default value from the NetLogo interface.

Currently, following simdesign_helper functions are provided:

simdesign_simple

The simple simdesign only uses defined constants and reports a parameter matrix with only one parameterization. To setup a simple simdesign, no variables have to be defined.

simdesign_distinct

The distinct simdesign can be used to run distinct parameter combinations. To setup a distinct simdesign, vectors of values need to be defined for each variable. These vectors must have the same number of elements across all variables. The first simulation run consist of all 1st elements of these variable vectors; the second run uses all 2nd values, and so on.

simdesign_ff

The full factorial simdesign creates a full-factorial parameter matrix with all possible combinations of parameter values. To setup a full-factorial simdesign, vectors of values need to be defined for each variable. Alternatively, a sequence can be defined by setting min, max and step. However, if both (values and min, max, step) are defined, the values vector is prioritized.

simdesign_lhs

The latin hypercube simdesign creates a Latin Hypercube sampling parameter matrix. The method can be used to generate a near-random sample of parameter values from the defined parameter distributions. More Details on Latin Hypercube Sampling can be found in McKay 1979. nlrx uses the lhs package to generate the Latin Hypercube parameter matrix. To setup a latin hypercube sampling simdesign, variable distributions need to be defined (min, max, qfun).

Sensitivity Analyses: simdesign_sobol, simdesign_sobol2007, simdesign_soboljansen, simdesign_morris, simdesign_eFast

Sensitivity analyses are useful to estimate the importance of model parameters and to scan the parameter space in an efficient way. nlrx uses the sensitivity package to setup sensitivity analysis parameter matrices. All supported sensitivity analysis simdesigns can be used to calculate sensitivity indices for each parameter-output combination. These indices can be calculated by using the analyze_nl function after attaching the simulation results to the nl object. To setup sensitivity analysis simdesigns, variable distributions (min, max, qfun) need to be defined.

Optimization techniques: simdesign_GenSA, simdesign_GenAlg

Optimization techniques are a powerful tool to search the parameter space for specific solutions. Both approaches try to minimize a specified model output reporter by systematically (genetic algorithm, utilizing the genalg package) or randomly (simulated annealing, utilizing the genSA package) changing the model parameters within the allowed ranges. To setup optimization simdesigns, variable ranges (min, max) need to be defined. Optimization simdesigns can only be executed using the run_nl_dyn function instead of run_nl_all or run_nl_one.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load nl objects from test data.

# Simdesign examples for Wolf Sheep Predation model from NetLogo models library:

nl <- nl_simple
nl@simdesign <- simdesign_simple(nl = nl,
                                 nseeds = 3)

nl <- nl_distinct
nl@simdesign <- simdesign_distinct(nl = nl,
                                   nseeds = 3)

nl <- nl_ff
nl@simdesign <- simdesign_ff(nl = nl,
                             nseeds = 3)

nl <- nl_lhs
nl@simdesign <- simdesign_lhs(nl=nl,
                              samples=100,
                              nseeds=3,
                              precision=3)

nl <- nl_sobol
nl@simdesign <- simdesign_sobol(nl=nl,
                                samples=200,
                                sobolorder=2,
                                sobolnboot=20,
                                sobolconf=0.95,
                                nseeds=3,
                                precision=3)

nl <- nl_sobol2007
nl@simdesign <- simdesign_sobol2007(nl=nl,
                                    samples=200,
                                    sobolnboot=20,
                                    sobolconf=0.95,
                                    nseeds=3,
                                    precision=3)

nl <- nl_soboljansen
nl@simdesign <- simdesign_soboljansen(nl=nl,
                                      samples=200,
                                      sobolnboot=20,
                                      sobolconf=0.95,
                                      nseeds=3,
                                      precision=3)

nl <- nl_morris
nl@simdesign <- simdesign_morris(nl=nl,
```

```
                                  morristype="oat",
                                  morrislevels=4,
                                  morrisr=100,
                                  morrisgridjump=2,
                                  nseeds=3)

nl <- nl_eFast
nl@simdesign <- simdesign_eFast(nl=nl,
                                samples=100,
                                nseeds=3)

nl <- nl_lhs
nl@simdesign <- simdesign_GenAlg(nl=nl,
                                  popSize = 200,
                                  iters = 100,
                                  evalcrit = 1,
                                  elitism = NA,
                                  mutationChance = NA,
                                  nseeds = 1)

nl <- nl_lhs
nl@simdesign <- simdesign_GenSA(nl=nl,
                                par=NULL,
                                evalcrit=1,
                                control=list(max.time = 600),
                                nseeds=1)
```

---

`simdesign_ABCmcmc_Marjoram` *Add an Approximate Bayesian Computation (Monte-Carlo Markov-Chain) simdesign using the Majoram algorithm to a nl object*

---

## Description

Add an Approximate Bayesian Computation (Monte-Carlo Markov-Chain) simdesign using the Majoram algorithm to a nl object

## Usage

```
simdesign_ABCmcmc_Marjoram(
  nl,
  postpro_function = NULL,
  summary_stat_target,
  prior_test = NULL,
  n_rec,
  n_between_sampling = 10,
  n_cluster = 1,
```

```
  use_seed = FALSE,
  dist_weights = NULL,
  n_calibration = 10000,
  tolerance_quantile = 0.01,
  proposal_phi = 1,
  seed_count = 0,
  progress_bar = FALSE,
  nseeds
)
```

## Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| postpro_function | default is NULL. Allows to provide a function that is called to post-process the output Tibble of the NetLogo simulations. The function must accept the nl object with attached results as input argument. The function must return a one-dimensional vector of output metrics that corresponds in leght and order to the specified summary_stat_target. |
| summary_stat_target | a vector of target values in the same order as the defined metrics of the experiment |
| prior_test | a string expressing the constraints between model parameters. This expression will be evaluated as a logical expression, you can use all the logical operators including "<", ">", ... Each parameter should be designated with "X1", "X2", ... in the same order as in the prior definition. Set to NULL to disable. |
| n_rec | Number of samples along the MCMC |
| n_between_sampling | a positive integer equal to the desired spacing between sampled points along the MCMC. |
| n_cluster | number of cores to parallelize simulations. Due to the design of the EasyABC parallelization it is currently not possible to use this feature with cores > 1. |
| use_seed | if TRUE, seeds will be automatically created for each new model run |
| dist_weights | a vector containing the weights to apply to the distance between the computed and the targeted statistics. These weights can be used to give more importance to a summary statistisc for example. The weights will be normalized before applying them. Set to NULL to disable. |
| n_calibration | a positive integer. This is the number of simulations performed during the calibration step. Default value is 10000. |
| tolerance_quantile | a positive number between 0 and 1 (strictly). This is the percentage of simulations retained during the calibration step to determine the tolerance threshold to be used during the MCMC. Default value is 0.01. |
| proposal_phi | a positive number. This is a scaling factor defining the range of MCMC jumps. Default value is 1. |

| | |
|---|---|
| seed_count | a positive integer, the initial seed value provided to the function model (if use_seed=TRUE). This value is incremented by 1 at each call of the function model. |
| progress_bar | logical, FALSE by default. If TRUE, ABC_mcmc will output a bar of progression with the estimated remaining computing time. Option not available with multiple cores. |
| nseeds | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and a shape of the distribution (qunif, qnorm, qlnorm, qexp)(e.g. list(min=1, max=4, qfun="qunif")).

The function uses the EasyABC package to set up the ABC_mcmc function. For details on the ABC_mcmc function parameters see ?EasyABC::ABC_mcmc Finally, the function reports a simdesign object.

Approximate Bayesian Computation simdesigns can only be executed using the run_nl_dyn function instead of run_nl_all or run_nl_one.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_lhs

# Attach the simdesign to the nl object
nl@simdesign <- simdesign_ABCmcmc_Marjoram(nl = nl,
                                           summary_stat_target = c(100, 80),
                                           n_rec = 100,
                                           n_between_sampling = 10,
                                           n_cluster = 1,
                                           use_seed = FALSE,
                                           n_calibration = 10000,
                                           tolerance_quantile = 0.01,
                                           proposal_phi = 1,
                                           progress_bar = FALSE,
                                           nseeds = 1)
```

`simdesign_ABCmcmc_Marjoram_original` *Add an Approximate Bayesian Computation (Monte-Carlo Markov-Chain) simdesign using the Majoram Original algorithm to a nl object*

## Description

Add an Approximate Bayesian Computation (Monte-Carlo Markov-Chain) simdesign using the Majoram Original algorithm to a nl object

## Usage

```
simdesign_ABCmcmc_Marjoram_original(
  nl,
  postpro_function = NULL,
  summary_stat_target,
  prior_test = NULL,
  n_rec,
  n_between_sampling = 10,
  n_cluster = 1,
  use_seed = FALSE,
  dist_weights = NULL,
  dist_max = 0,
  tab_normalization = summary_stat_target,
 proposal_range = vector(mode = "numeric", length = length(getexp(nl, "variables"))),
  seed_count = 0,
  progress_bar = FALSE,
  nseeds
)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `postpro_function` | default is NULL. Allows to provide a function that is called to post-process the output Tibble of the NetLogo simulations. The function must accept the nl object with attached results as input argument. The function must return a one-dimensional vector of output metrics that corresponds in leght and order to the specified summary_stat_target. |
| `summary_stat_target` | a vector of target values in the same order as the defined metrics of the experiment |
| `prior_test` | a string expressing the constraints between model parameters. This expression will be evaluated as a logical expression, you can use all the logical operators including "<", ">", ... Each parameter should be designated with "X1", "X2", ... in the same order as in the prior definition. Set to NULL to disable. |

`n_rec` Number of samples along the MCMC

`n_between_sampling`
a positive integer equal to the desired spacing between sampled points along the MCMC.

`n_cluster` number of cores to parallelize simulations. Due to the design of the EasyABC parallelization it is currently not possible to use this feature with cores > 1.

`use_seed` if TRUE, seeds will be automatically created for each new model run

`dist_weights` a vector containing the weights to apply to the distance between the computed and the targeted statistics. These weights can be used to give more importance to a summary statistisc for example. The weights will be normalized before applying them. Set to NULL to disable.

`dist_max` a positive number. This is the tolerance threshold used during the MCMC. If not provided by the user, it is automatically computed as half the distance between the first simulation and the target summary statistics and a warning is printed.

`tab_normalization`
a vector of the same length as summary_stat_target. Each element contains a positive number by which each summary statistics must be divided before the computation of the Euclidean distance between simulations and data. If not provided by the user, the simulated summary statistics are divided by the target summary statistics and a warning is printed.

`proposal_range` a vector of the same length as the number of model parameters, used when method is "Marjoram_original". Each element contains a positive number defining the range of MCMC jumps for each model parameter. If not provided by the user, a default value is used for each parameter and a warning is printed. The default value is 1/50 of the prior range for uniform distributions, 1/20 of the standard deviation of the prior distribution for normal distributions, 1/20 * exp ( sigma * sigma

`seed_count` a positive integer, the initial seed value provided to the function model (if use_seed=TRUE). This value is incremented by 1 at each call of the function model.

`progress_bar` logical, FALSE by default. If TRUE, ABC_mcmc will output a bar of progression with the estimated remaining computing time. Option not available with multiple cores.

`nseeds` number of seeds for this simulation design

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and a shape of the distribution (qunif, qnorm, qlnorm, qexp)(e.g. list(min=1, max=4, qfun="qunif")).

The function uses the EasyABC package to set up the ABC_mcmc function. For details on the ABC_mcmc function parameters see ?EasyABC::ABC_mcmc Finally, the function reports a simdesign object.

Approximate Bayesian Computation simdesigns can only be executed using the run_nl_dyn function instead of run_nl_all or run_nl_one.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_lhs

# Attach the simdesign to the nl object
nl@simdesign <- simdesign_ABCmcmc_Marjoram_original(nl = nl,
                                                    summary_stat_target = c(100, 80),
                                                    n_rec = 100,
                                                    n_between_sampling = 10,
                                                    nseeds = 1)
```

---

# simdesign_ABCmcmc_Wegmann

*Add an Approximate Bayesian Computation (Monte-Carlo Markov-Chain) simdesign using the Wegmann algorithm to a nl object*

---

## Description

Add an Approximate Bayesian Computation (Monte-Carlo Markov-Chain) simdesign using the Wegmann algorithm to a nl object

## Usage

```
simdesign_ABCmcmc_Wegmann(
  nl,
  postpro_function = NULL,
  summary_stat_target,
  prior_test = NULL,
  n_rec,
  n_between_sampling = 10,
  n_cluster = 1,
  use_seed = FALSE,
  dist_weights = NULL,
  n_calibration = 10000,
  tolerance_quantile = 0.01,
  proposal_phi = 1,
  numcomp = 0,
  seed_count = 0,
  progress_bar = FALSE,
  nseeds
)
```

## Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| postpro_function | default is NULL. Allows to provide a function that is called to post-process the output Tibble of the NetLogo simulations. The function must accept the nl object with attached results as input argument. The function must return a one-dimensional vector of output metrics that corresponds in leght and order to the specified summary_stat_target. |
| summary_stat_target | a vector of target values in the same order as the defined metrics of the experiment |
| prior_test | a string expressing the constraints between model parameters. This expression will be evaluated as a logical expression, you can use all the logical operators including "<", ">", ... Each parameter should be designated with "X1", "X2", ... in the same order as in the prior definition. Set to NULL to disable. |
| n_rec | Number of samples along the MCMC |
| n_between_sampling | a positive integer equal to the desired spacing between sampled points along the MCMC. |
| n_cluster | number of cores to parallelize simulations. Due to the design of the EasyABC parallelization it is currently not possible to use this feature with cores > 1. |
| use_seed | if TRUE, seeds will be automatically created for each new model run |
| dist_weights | a vector containing the weights to apply to the distance between the computed and the targeted statistics. These weights can be used to give more importance to a summary statistisc for example. The weights will be normalized before applying them. Set to NULL to disable. |
| n_calibration | a positive integer. This is the number of simulations performed during the calibration step. Default value is 10000. |
| tolerance_quantile | a positive number between 0 and 1 (strictly). This is the percentage of simulations retained during the calibration step to determine the tolerance threshold to be used during the MCMC. Default value is 0.01. |
| proposal_phi | a positive number. This is a scaling factor defining the range of MCMC jumps. Default value is 1. |
| numcomp | a positive integer. This is the number of components to be used for PLS transformations. Default value is 0 which encodes that this number is equal to the number of summary statistics. |
| seed_count | a positive integer, the initial seed value provided to the function model (if use_seed=TRUE). This value is incremented by 1 at each call of the function model. |
| progress_bar | logical, FALSE by default. If TRUE, ABC_mcmc will output a bar of progression with the estimated remaining computing time. Option not available with multiple cores. |
| nseeds | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and a shape of the distribution (qunif, qnorm, qlnorm, qexp)(e.g. list(min=1, max=4, qfun="qunif")).

The function uses the EasyABC package to set up the ABC_mcmc function. For details on the ABC_mcmc function parameters see ?EasyABC::ABC_mcmc Finally, the function reports a simdesign object.

Approximate Bayesian Computation simdesigns can only be executed using the run_nl_dyn function instead of run_nl_all or run_nl_one.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_lhs

# Attach the simdesign to the nl object
nl@simdesign <- simdesign_ABCmcmc_Wegmann(nl = nl,
                                          summary_stat_target = c(100, 80),
                                          n_rec = 100,
                                          n_between_sampling = 10,
                                          n_cluster = 1,
                                          use_seed = FALSE,
                                          n_calibration = 10000,
                                          tolerance_quantile = 0.01,
                                          proposal_phi = 1,
                                          progress_bar = FALSE,
                                          nseeds = 1)
```

---

# simdesign_distinct *Add a distinct simdesign to a nl object*

---

## Description

Add a distinct simdesign to a nl object

## Usage

```
simdesign_distinct(nl, nseeds)
```

### Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| nseeds | number of seeds for this simulation design |

### Details

This function creates a simdesign S4 class which can be added to a nl object. The distinct simdesign allows to create a parameter matrix with distinct parameterisations.

Variables in the experiment variable list need to provide a vector of distinct values (e.g. list(values=c(1,2,3,4)). All vectors of values must have the same length across variables.

The distinct simdesign then creates one simulation run for all first elements of these values vectors, one run for all second items, and so on. With this function, multiple distinct simulations can be run at once. Finally, the function reports a simdesign object.

### Value

simdesign S4 class object

### Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_distinct
nl@simdesign <- simdesign_distinct(nl = nl, nseeds = 3)
```

---

## simdesign_eFast *Add an eFast simdesign to a nl object*

---

### Description

Add an eFast simdesign to a nl object

### Usage

```
simdesign_eFast(nl, samples, nseeds)
```

### Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| samples | number of samples for the eFast sensitivity analysis |
| nseeds | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and qfun (e.g. list(min=1, max=4, qfun="qunif")).

The eFast simdesign uses the sensitivity package to set up a fast99 elementary effects sensitivity analysis, including a simobject of class fast99 and a input tibble for simulations. For details on method specific sensitivity analysis function parameters see ?fast99 Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_eFast
nl@simdesign <- simdesign_eFast(nl=nl,
                                samples=100,
                                nseeds=1)
```

---

# simdesign_ff *Add a full-factorial simdesign to a nl object*

---

## Description

Add a full-factorial simdesign to a nl object

## Usage

```
simdesign_ff(nl, nseeds)
```

## Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| nseeds | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a vector of distinct values (e.g. list(values=c(1,2,3,4)). Or a sequence definition with min, max and step (e.g. list=(min=1, max=4, step=1)). If both (values and sequence) are defined, the full-factorial design gives priority to the values.

The full-factorial simdesign uses these defined parameter ranges within the nl object. A full-factorial matrix of all parameter combinations is created as input tibble for the simdesign. Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_ff
nl@simdesign <- simdesign_ff(nl = nl, nseeds = 3)
```

---

# simdesign_GenAlg *Add a Genetic Algorithm simdesign to a nl object*

---

## Description

Add a Genetic Algorithm simdesign to a nl object

## Usage

```
simdesign_GenAlg(
  nl,
  popSize = 200,
  iters = 100,
  evalcrit = 1,
  elitism = NA,
  mutationChance = NA,
  nseeds = 1
)
```

## Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| popSize | population Size parameter for genetic algorithm |
| iters | number of iterations for genetic algorithm function |
| evalcrit | position of evaluation criterion within defined NetLogo metrics of nl experiment or a function that reports a single numeric value |
| elitism | elitism rate of genetic algorithm function |
| mutationChance | mutation rate of genetic algorithm function |
| nseeds | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min and max (e.g. list(min=1, max=4)).

The GenAlg simdesign generates a Genetic Algorithm experiment within the defined min and max parameter boundaries that are defined in the variables field of the experiment object within the nl object.

The evalcrit reporter defines the evaluation criterion for the Genetic algorithm procedure. There are two options to evaluate the fitness value of each iteration of the algorithm:

1. Use a reporter that is defined within the experiment metrics vector. You can just enter the position of that metric within the experiment metrics vector (e.g. 1 would use the first defined metric of the experiment to evaluate each iteration). The algorithm automatically calculates the mean value of this reporter if evalticks is defined to measure multiple ticks during each simulation.
2. Use a self-defined evaluation function You can define a function that post-processes NetLogo output to calculate an evaluation value. This function must accept the nl object as input and return one single numeric value. The nl object that is then provided to the evaluation function will have results of the current iteration attached. The results can be accessed via the simoutput slot of the simdesign. You can pass this function to evalcrit. It is then applied to the output of each iteration.

The function uses the genalg package to set up a Genetic Algorithm function. For details on the genalg function parameters see ?genalg::rbga Finally, the function reports a simdesign object.

Genetic Algorithm simdesigns can only be executed using the run_nl_dyn function instead of run_nl_all or run_nl_one.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_lhs

# Example 1: Using a metric from the experiment metrics vector for evaluation:
nl@simdesign <- simdesign_GenAlg(nl=nl,
                                 evalcrit=1,
                                 nseeds=1)

# Example 2: Using a self-defined evaluation function
# For demonstration we define a simple function that calculates
# the maximum value of count sheep output.
critfun <- function(nl) {
results <- nl@simdesign@simoutput
crit <- as.integer(max(results$`count sheep`))
return(crit)
}

nl@simdesign <- simdesign_GenAlg(nl=nl,
                                 evalcrit=critfun,
                                 nseeds=1)
```

---

`simdesign_GenSA` *Add a Simulated Annealing simdesign to a nl object*

---

## Description

Add a Simulated Annealing simdesign to a nl object

## Usage

```
simdesign_GenSA(nl, par = NULL, evalcrit = 1, control = list(), nseeds = 1)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `par` | optional vector of start values for each parameter defined in variables of experiment |
| `evalcrit` | position of evaluation criterion within defined NetLogo metrics of nl experiment or a function that reports a single numeric value |
| `control` | list with further arguments passed to the GenSA function (see ?GenSA for details) |
| `nseeds` | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min and max (e.g. list(min=1, max=4)).

The GenSA simdesign generates a simulated Annealing experiment within the defined min and max parameter boundaries that are defined in the variables field of the experiment object within the nl object.

The evalcrit reporter defines the evaluation criterion for the simulated annealing procedure. There are two options to evaluate the fitness value of each iteration of the algorithm:

1. Use a reporter that is defined within the experiment metrics vector. You can just enter the position of that metric within the experiment metrics vector (e.g. 1 would use the first defined metric of the experiment to evaluate each iteration). The algorithm automatically calculates the mean value of this reporter if evalticks is defined to measure multiple ticks during each simulation. You can define a function that post-processes NetLogo output to calculate an evaluation value. This function must accept the nl object as input and return one single numeric value. The nl object that is then provided to the evaluation function will have results of the current iteration attached. The results can be accessed via the simoutput slot of the simdesign. You can pass this function to evalcrit. It is then applied to the output of each iteration.

The function uses the GenSA package to set up a Simulated Annealing function. For details on the GenSA function parameters see ?GenSA Finally, the function reports a simdesign object.

Simulated Annealing simdesigns can only be executed using the run_nl_dyn function instead of run_nl_all or run_nl_one.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_lhs

# Example 1: Using a metric from the experiment metrics vector for evaluation:
nl@simdesign <- simdesign_GenSA(nl=nl,
                                par=NULL,
                                evalcrit=1,
                                control=list(max.time = 600),
                                nseeds=1)


# Example 2: Using a self-defined evaluation function
# For demonstration we define a simple function that calculates
# the maximum value of count sheep output.
critfun <- function(nl) {
```

```
results <- nl@simdesign@simoutput
crit <- as.integer(max(results$`count sheep`))
return(crit)
}

nl@simdesign <- simdesign_GenSA(nl=nl,
                                par=NULL,
                                evalcrit=critfun,
                                control=list(max.time = 600),
                                nseeds=1)
```

---

`simdesign_lhs` *Add a latin-hypercube simdesign to a nl object*

---

## Description

Add a latin-hypercube simdesign to a nl object

## Usage

```
simdesign_lhs(nl, samples, nseeds, precision)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `samples` | number of samples for the latin hypercube |
| `nseeds` | number of seeds for this simulation design |
| `precision` | number of digits for the decimal fraction of parameter values |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and qfun (e.g. list(min=1, max=4, qfun="qunif")).

The latin hypercube simdesign creates a parameter matrix based on these defined distributions. Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_lhs
nl@simdesign <- simdesign_lhs(nl=nl,
                              samples=100,
                              nseeds=3,
                              precision=3)
```

---

`simdesign_morris` *Add a morris elementary effects simdesign to a nl object*

---

## Description

Add a morris elementary effects simdesign to a nl object

## Usage

```
simdesign_morris(nl, morristype, morrislevels, morrisr, morrisgridjump, nseeds)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `morristype` | morris design type |
| `morrislevels` | number of parameter levels |
| `morrisr` | morris r value |
| `morrisgridjump` | morris grid jump value |
| `nseeds` | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and qfun (e.g. list(min=1, max=4, qfun="qunif")).

The morris simdesign uses the sensitivity package to set up a morris elementary effects sensitivity analysis, including a simobject of class morris and a input tibble for simulations. For details on method specific sensitivity analysis function parameters see ?morris Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_morris
nl@simdesign <- simdesign_morris(nl=nl,
                                 morristype="oat",
                                 morrislevels=4,
                                 morrisr=20,
                                 morrisgridjump=2,
                                 nseeds=3)
```

---

`simdesign_simple` *Add a simple simdesign to a nl object*

---

## Description

Add a simple simdesign to a nl object

## Usage

```
simdesign_simple(nl, nseeds)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `nseeds` | number of seeds for this simulation design |

## Details

This function creates a simdesign S4 class which can be added to a nl object. The simple simdesign only uses model parameters that are defined in the constants field of the experiment object within the nl object. Thus, the resulting input tibble of the simdesign has only one run with constant parameterisations. This can be useful to run one simulation with a specific parameterset. Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_simple
nl@simdesign <- simdesign_simple(nl = nl, nseeds = 3)
```

---

`simdesign_sobol` *Add a sobol simdesign to a nl object*

---

## Description

Add a sobol simdesign to a nl object

## Usage

```
simdesign_sobol(
  nl,
  samples,
  sobolorder,
  sobolnboot,
  sobolconf,
  nseeds,
  precision
)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `samples` | number of samples for the sobol sensitivity analysis |
| `sobolorder` | order of interactions of the sobol sensitivity analysis |
| `sobolnboot` | number of bootstrap replicates of the sobol sensitivity analysis |
| `sobolconf` | the confidence level for bootstrap confidence intervals |
| `nseeds` | number of seeds for this simulation design |
| `precision` | number of digits for the decimal fraction of parameter values |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and qfun (e.g. list(min=1, max=4, qfun="qunif")).

The sobol simdesign uses the sensitivity package to set up a sobol sensitivity analysis, including a simobject of class sobol and a input tibble for simulations. For details on method specific sensitivity analysis function parameters see ?sobol Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_sobol
nl@simdesign <- simdesign_sobol(nl=nl,
samples=1000,
sobolorder=2,
sobolnboot=100,
sobolconf=0.95,
nseeds=3,
precision=3)
```

---

# simdesign_sobol2007 *Add a sobol2007 simdesign to a nl object*

---

## Description

Add a sobol2007 simdesign to a nl object

## Usage

```
simdesign_sobol2007(nl, samples, sobolnboot, sobolconf, nseeds, precision)
```

## Arguments

| | |
|---|---|
| nl | nl object with a defined experiment |
| samples | number of samples for the sobol sensitivity analysis |
| sobolnboot | number of bootstrap replicates of the sobol sensitivity analysis |
| sobolconf | the confidence level for bootstrap confidence intervals |
| nseeds | number of seeds for this simulation design |
| precision | number of digits for the decimal fraction of parameter values |

## Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and qfun (e.g. list(min=1, max=4, qfun="qunif")).

The sobol2007 simdesign uses the sensitivity package to set up a sobol2007 sensitivity analysis, including a simobject of class sobol and a input tibble for simulations. For details on method specific sensitivity analysis function parameters see ?sobol2007 Finally, the function reports a simdesign object.

## Value

simdesign S4 class object

## Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_sobol2007
nl@simdesign <- simdesign_sobol2007(nl=nl,
samples=1000,
sobolnboot=100,
sobolconf=0.95,
nseeds=3,
precision=3)
```

---

`simdesign_soboljansen` *Add a soboljansen simdesign to a nl object*

---

## Description

Add a soboljansen simdesign to a nl object

## Usage

```
simdesign_soboljansen(nl, samples, sobolnboot, sobolconf, nseeds, precision)
```

## Arguments

| | |
|---|---|
| `nl` | nl object with a defined experiment |
| `samples` | number of samples for the sobol sensitivity analysis |
| `sobolnboot` | number of bootstrap replicates of the sobol sensitivity analysis |
| `sobolconf` | the confidence level for bootstrap confidence intervals |
| `nseeds` | number of seeds for this simulation design |
| `precision` | number of digits for the decimal fraction of parameter values |

### Details

This function creates a simdesign S4 class which can be added to a nl object.

Variables in the experiment variable list need to provide a numeric distribution with min, max and qfun (e.g. list(min=1, max=4, qfun="qunif")).

The soboljansen simdesign uses the sensitivity package to set up a soboljansen sensitivity analysis, including a simobject of class sobol and a input tibble for simulations. For details on method specific sensitivity analysis function parameters see ?soboljansen Finally, the function reports a simdesign object.

### Value

simdesign S4 class object

### Examples

```
# To attach a simdesign, a nl object needs to be created first (see ?nl).
# For this example, we load a nl object from test data.

nl <- nl_soboljansen
nl@simdesign <- simdesign_soboljansen(nl=nl,
samples=1000,
sobolnboot=100,
sobolconf=0.95,
nseeds=3,
precision=3)
```

---

## unnest_simoutput *Get spatial data from metrics.turtles and metrics.patches output*

---

### Description

Turn results from NetLogo in spatial data objects

### Usage

```
unnest_simoutput(nl)
```

### Arguments

| | |
|---|---|
| nl | nl object |

### Details

Unnests output from run_nl into long format.

## Value

tibble with spatial data objects

## Examples

```
# To unnest data a nl object containing spatial output data is needed.
# For this example, we load a nl object from test data.

nl <- nl_spatial
unnest_simoutput(nl)
```

---

`write_simoutput` *Write attached NetLogo simulation output to file*

---

## Description

Write attached NetLogo simulation output to file

## Usage

```
write_simoutput(nl, outpath = NA)
```

## Arguments

| | |
|---|---|
| `nl` | nl object |
| `outpath` | optional path to directory where output is written<br>Write NetLogo simulation output to a csv file in the directory outpath of the nl object Output has to be attached to the simdesign first with simoutput(nl) <- results The outpath argument can be optionally used to write output to a different directory than the defined outpath of the nl object. |

## Examples

```
# Load nl object including output data from testdata
nl <- nl_lhs

# Write output to outpath directory
write_simoutput(nl, outpath=tempdir())
```

# Index