

Package ‘EpiModel’

September 16, 2026

Version 2.6.2

Date 2026-09-15

Title Mathematical Modeling of Infectious Disease Dynamics

Description Tools for simulating mathematical models of infectious disease dynamics. Epidemic model classes include deterministic compartmental models, stochastic individual-contact models, and stochastic network models. Network models use the robust statistical methods of exponential-family random graph models (ERGMs) from the Statnet suite of software packages in R. Standard templates for epidemic modeling include SI, SIR, and SIS disease types. EpiModel features an API for extending these templates to address novel scientific research aims. Full methods for EpiModel are detailed in Jenness et al. (2018, <[doi:10.18637/jss.v084.i08](https://doi.org/10.18637/jss.v084.i08)>).

Maintainer Samuel Jenness <samuel.m.jenness@emory.edu>

License GPL-3

URL <https://www.epimodel.org/>, <https://epimodel.github.io/EpiModel/>

BugReports <https://github.com/EpiModel/EpiModel/issues/>

Depends R (>= 4.1), deSolve (>= 1.21), networkDynamic (>= 0.11.3),
tergm (>= 4.2.1), statnet.common (>= 4.10.0)

Imports future, future.apply, graphics, grDevices, stats, utils,
collections, ergm (>= 4.7.1), network (>= 1.18.1),
RColorBrewer, ape, lazyeval, ggplot2, tibble, methods, rlang,
dplyr, coda, networkLite (>= 1.0.5)

Suggests bslib (>= 0.6.0), DT, ergm.ego (>= 1.1.0), egor, knitr, ndtv,
plotly (>= 4.10.0), rmarkdown, shiny, testthat (>= 3.1.8),
progressr, tidyr

VignetteBuilder knitr

LinkingTo Rcpp, ergm

Encoding UTF-8

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Samuel Jenness [cre, aut],
 Steven M. Goodreau [aut],
 Martina Morris [aut],
 Adrien Le Guillou [aut],
 Chad Klumb [aut],
 Skye Bender-deMoll [ctb]

Repository CRAN

Date/Publication 2026-09-16 05:40:09 UTC

Contents

EpiModel-package	4
add_vertices	8
apportion_lr	9
arrive_nodes	10
as.data.frame.dcm	11
as.data.frame.icm	12
as.data.frame.netdx	14
as.epi.data.frame	15
as.network.transmat	16
as.phylo.transmat	16
as_cumulative_edgelist	18
as_tibble_edgelist	19
check_degdist_bal	19
color_tea	20
comp_plot	21
control.dcm	23
control.icm	25
control.net	26
create_dat_object	32
create_scenario_list	33
dcm	33
dedup_cumulative_edgelist	35
delete_edges	36
delete_vertices	37
depart_nodes	38
dissolution_coefs	39
edgelist_censor	41
epiweb	42
generate_random_params	44
geom_bands	46
get_adj_list	47
get_attr_history	48
get_connected_nodes	48
get_cumulative_degree	49
get_cumulative_edgelist	50
get_cumulative_edgelist_df	51

get_current_timestep	53
get_degree	53
get_degree_dist	54
get_edgelist	56
get_edgelist_df	57
get_formula_term_attr	57
get_network	58
get_network_attributes	60
get_network_term_attr	60
get_nwstats	61
get_param_set	62
get_partners	63
get_sims	65
get_subnet_adj_list	66
get_vertex_attribute	67
icm	67
increment_timestep	69
init.dcm	70
init.icm	71
init.net	72
InitErgmTerm.absdiffby	73
InitErgmTerm.absdiffnodemix	74
InitErgmTerm.fuzzynodematch	74
is.transmat	75
is_active_posit_ids	76
is_active_unique_ids	77
make_restart_point	77
merge.icm	79
merge.netsim	80
modules.icm	82
modules.net	83
multilayer	85
mutate_epi	85
net-accessor	86
netdx	90
netest	94
netsim	99
network_initialize	103
nwupdate.net	104
overwrite_attrs	104
padded_vector	105
param.dcm	105
param.icm	108
param.net	111
param.net_from_table	115
param.net_to_table	116
param_random	117
plot.dcm	117

plot.epi.data.frame	120
plot.icm	122
plot.netdx	125
plot.netsim	130
plot.transmat	135
print.dcm	136
print.icm	137
print.netdx	137
print.netsim	138
reachable-nodes	140
record_attr_history	142
record_raw_object	143
set_current_timestep	144
set_transmat	145
set_vertex_attribute	145
summary.dcm	146
summary.icm	147
summary.netsim	148
trim_netest	149
truncate_sim	151
unique_id-tools	152
update_cumulative_edgelist	153
update_dissolution	154
update_params	155
use_scenario	157
Index	158

EpiModel-package

Mathematical Modeling of Infectious Disease Dynamics

Description

Tools for simulating mathematical models of infectious disease dynamics. Epidemic model classes include deterministic compartmental models, stochastic individual-contact models, and stochastic network models. Network models use the robust statistical methods of exponential-family random graph models (ERGMs) from the Statnet suite of software packages in R. Standard templates for epidemic modeling include SI, SIR, and SIS disease types. EpiModel features an API for extending these templates to address novel scientific research aims. Full methods for EpiModel are detailed in Jenness et al. (2018, [doi:10.18637/jss.v084.i08](https://doi.org/10.18637/jss.v084.i08)).

Details

EpiModel provides tools for building, simulating, and analyzing mathematical models of infectious disease dynamics. Models may be built for pedagogical purposes using the built-in disease types, or for research purposes using the extension API to define arbitrary disease states and processes.

The distinguishing capability of EpiModel is stochastic network modeling, in which partnerships form, persist, and dissolve according to a statistical model estimated from network data. Simpler

compartmental and individual-based classes are also provided, both for teaching and for comparison against the network results.

Model Classes and Infectious Disease Types

EpiModel provides functionality for three classes of epidemic models:

- **Deterministic Compartmental Models (DCMs):** continuous-time models solved with ordinary differential equations, under an assumption of random mixing within the population. Sensitivity analyses across parameter values or initial conditions are specified by passing vectors to `param.dcm()` or `init.dcm()`.
- **Stochastic Individual Contact Models (ICMs):** discrete-time, individual-based analogs of the DCM class, which add random variation to each component of the transmission system, from infection to recovery to vital dynamics (arrivals and departures). Contacts are randomly mixed, with no persistent partnership structure.
- **Stochastic Network Models:** discrete-time, individual-based models in which the contact structure is represented explicitly as a dynamic network. Partnership formation and dissolution are simulated from a temporal exponential-family random graph model (TERGM), using the statistical framework implemented in the **Statnet** suite of R packages. Disease then spreads across the edges of that evolving network. This class represents features that random mixing cannot, including degree distributions, partnership duration, concurrency, assortative mixing, and feedback between network structure and epidemic dynamics.

EpiModel supports three built-in infectious disease types, available across all three model classes, in one-group and two-group configurations and with or without demography:

- **Susceptible-Infectious (SI):** a two-state disease with life-long infection and no recovery. HIV is one example, although for that case it is common to model infection stages as separate compartments.
- **Susceptible-Infectious-Recovered (SIR):** a three-state disease with life-long immunity following infection. Measles is one example, although modern models for measles also require consideration of vaccination patterns in the population.
- **Susceptible-Infectious-Susceptible (SIS):** a two-state disease in which one may transition back and forth between the susceptible and infected states throughout life. Examples include bacterial sexually transmitted infections such as gonorrhea.

These built-in types are starting points. The extension API described below allows them to be replaced with arbitrarily complex disease systems.

Model Parameterization and Simulation

Each model class uses three setup functions to collect the epidemic parameters, initial conditions, and control settings:

- `param.dcm()`, `param.icm()`, and `param.net()` set the epidemic parameters, such as the rate of contacts or acts between agents, the probability of transmission per act, and the recovery and demographic rates for models that include those transitions.
- `init.dcm()`, `init.icm()`, and `init.net()` set the initial conditions, which for built-in types are the numbers or, where applicable, the specific agents infected or recovered at the outset.

- `control.dcm()`, `control.icm()`, and `control.net()` set the remaining simulation controls, including the disease type, the number of time steps, and the number of simulations or runs.

With the model parameterized, the simulation functions are:

- `dcm()` for deterministic compartmental models.
- `icm()` for stochastic individual contact models.
- Network models are simulated in a three-step process:
 1. `netest()` estimates the statistical model for the network structure itself, that is, how partnerships form and dissolve over time. This function wraps the estimation routines in the `ergm` and `tergm` packages. The typical input is a set of target statistics summarizing the formation process together with a mean partnership duration, an approach known as egocentric inference because those targets may be calculated from an egocentric sample of the population. Dissolution coefficients are calculated with `dissolution_coefs()`.
 2. `netdx()` runs diagnostics on the fitted model by simulating the network forward in time without disease, to check that the simulated network recovers the formation and dissolution targets.
 3. `netsim()` simulates the stochastic network epidemic model, given the fitted network model from `netest()` along with the parameters, initial conditions, and controls defined above.

Network models may also use multiple network layers over a shared node set, for example separate sexual and needle-sharing contact networks. Each layer is estimated separately and the fits are passed together to `netsim()`, with `multilayer()` specifying any controls that vary by layer.

Extending the Built-In Models

Research applications typically require disease states or processes beyond the built-in types. The extension API supports this for the network class: setting `type = NULL` in `control.net()` and supplying module functions (for example, `infection.FUN = my_infection_module`) replaces or supplements the built-in modules. A module is a function with the signature `function(dat, at)` that reads from and returns the `netsim_dat` object holding all simulation state.

Modules should use the accessor functions rather than manipulating that object directly. These include `get_attr()` and `set_attr()` for nodal attributes, `get_param()` for parameters, `get_epi()` and `set_epi()` for epidemic summary statistics, `append_core_attr()` and `append_attr()` for new arrivals, `discord_edgelist()` for serodiscordant pairs, and `set_transmat()` for recording transmission events.

Extension support is specific to the network class. ICMs support the built-in SI, SIR, and SIS types only; users needing custom individual-based dynamics should use `netsim()`. DCMs accept an original system of differential equations through the `new.mod` argument to `control.dcm()`.

Working with Simulation Output

All three model classes return objects with `print`, `summary`, `plot`, and `as.data.frame` methods, so the same analysis workflow applies throughout. Simulation output may be converted to a long data frame with `as.data.frame.dcm()`, `as.data.frame.icm()`, or `as.data.frame.netsim()`, extended with derived statistics using `mutate_epi()`, subset with `get_sims()`, and combined across runs with `merge.icm()` or `merge.netsim()`.

Network models additionally record the simulated networks, retrieved with `get_network()`; the network statistics, retrieved with `get_nwstats()`; and the transmission tree, retrieved with `get_transmat()` and convertible to a phylogeny with `as.phylo.transmat()`. Partnership histories may be tracked over time with the cumulative edgelist and queried with `get_partners()`, `get_forward_reachable()`, and `get_backward_reachable()`.

Learning Resources

The package vignettes cover getting started (Intro) and then working with model parameters, network objects, and nodal attributes and summary statistics in extension models.

Beyond the package documentation:

- The **Network Modeling for Epidemics** course materials provide a full treatment of network model theory, estimation, diagnostics, and epidemic simulation, followed by the extension API and advanced topics such as multi-layer networks and epidemics over fully observed networks, at <https://epimodel.github.io/sismid/>.
- The **EpiModel Gallery** provides worked extension models, from adding a single compartment through multi-stage disease models with interventions and cost-effectiveness analysis, at <https://epimodel.github.io/EpiModel-Gallery/>.
- The **EpiModel website** collects these resources and a suggested learning pathway, at <https://www.epimodel.org/>.

Author(s)

Maintainer: Samuel Jenness <samuel.m.jenness@emory.edu>

Authors:

- Samuel Jenness <samuel.m.jenness@emory.edu>
- Steven M. Goodreau <goodreau@uw.edu>
- Martina Morris <morris@uw.edu>
- Adrien Le Guillou <contact@aleguillou.org>
- Chad Klumb <cklumb@gmail.com>

Other contributors:

- Skye Bender-deMoll <skyebend@uw.edu> [contributor]

References

The EpiModel website is at <https://www.epimodel.org/>, and the source code is at <https://github.com/EpiModel/EpiModel>. Bug reports and feature requests are welcome.

Our primary methods paper on EpiModel is published in the **Journal of Statistical Software**. If you use EpiModel for any research or teaching purposes, please cite this reference:

Jenness SM, Goodreau SM, and Morris M. EpiModel: An R Package for Mathematical Modeling of Infectious Disease over Networks. *Journal of Statistical Software*. 2018; 84(8): 1-47. doi:[10.18637/jss.v084.i08](https://doi.org/10.18637/jss.v084.i08).

EpiModel is the foundation for a set of extension packages maintained by our group. EpiModelHIV (<https://github.com/EpiModel/EpiModelHIV>) models HIV and bacterial sexually transmitted infections, and EpiModelCOVID (<https://github.com/EpiModel/EpiModelCOVID>) models SARS-CoV-2. Simulation on high-performance computing clusters is supported by EpiModelHPC (<https://github.com/EpiModel/EpiModelHPC>).

See Also

Useful links:

- <https://www.epimodel.org/>
- <https://epimodel.github.io/EpiModel/>
- Report bugs at <https://github.com/EpiModel/EpiModel/issues/>

add_vertices

Fast Version of network::add.vertices for Edgelist-formatted Network

Description

This function performs a simple operation of updating the edgelist attribute `n` that tracks the total network size implicit in an edgelist representation of the network.

Usage

```
add_vertices(el, nv)
```

Arguments

- | | |
|-----------------|--|
| <code>el</code> | A two-column matrix of current edges (edgelist) with an attribute variable <code>n</code> containing the total current network size. |
| <code>nv</code> | A integer equal to the number of nodes to add to the network size at the given time step. |

Details

This function is used in EpiModel modules to add vertices (nodes) to the edgelist object to account for entries into the population (e.g., births and in-migration).

Value

Returns the matrix of current edges, `el`, with the population size attribute updated based on the number of new vertices specified in `nv`.

Examples

```
## Not run:
library("EpiModel")
nw <- network_initialize(100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
x <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

param <- param.net(inf.prob = 0.3)
init <- init.net(i.num = 10)
control <- control.net(type = "SI", nsteps = 100, nsims = 5,
                      tergmLite = TRUE)

# networkLite representation after initialization
dat <- crosscheck.net(x, param, init, control)
dat <- initialize.net(x, param, init, control)

# Check current network size
attributes(dat$el[[1]])$n

# Add 10 vertices
dat$el[[1]] <- add_vertices(dat$el[[1]], 10)

# Check new network size
attributes(dat$el[[1]])$n

## End(Not run)
```

apportion_lr

Apportion Using the Largest Remainder Method

Description

Apportions a vector of values given a specified frequency distribution of those values such that the length of the output is robust to rounding and other instabilities.

Usage

```
apportion_lr(vector.length, values, proportions, shuffled = FALSE)
```

Arguments

vector.length	Length for the output vector.
values	Values for the output vector.
proportions	Proportion distribution with one number for each value. This must sum to 1.
shuffled	If TRUE, randomly shuffle the order of the vector.

Value

A vector of length `vector.length` containing the apportioned values from `values`.

Examples

```
## Not run:
## Example 1: Without rounding
apportioned_vec_1 <- apportion_lr(4, c(1, 2, 3, 4, 5),
                                   c(0.25, 0, 0.25, 0.25, 0.25))

## Example 2: With rounding
apportioned_vec_2 <- apportion_lr(5, c(1, 2, 3, 4, 5),
                                   c(0.21, 0, 0.29, 0.25, 0.25))

## End(Not run)
```

arrive_nodes

Arrive New Nodes to the netsim_dat Object

Description

Arrive New Nodes to the netsim_dat Object

Usage

```
arrive_nodes(dat, nArrivals)
```

Arguments

<code>dat</code>	the netsim_dat object
<code>nArrivals</code>	number of new nodes to arrive

Details

`nArrivals` new nodes are added to the network data stored on the netsim_dat object. If `tergmLite` is `FALSE`, these nodes are activated from the current timestep onward. Attributes for the new nodes must be set separately.

Note that this function only supports arriving new nodes; returning to an active state nodes that were previously active in the network is not supported.

Value

the updated netsim_dat object with `nArrivals` new nodes added

as.data.frame.dcm *Extract Model Data for Deterministic Compartmental Models*

Description

This function extracts a model run from an object of class dcm into a data frame using the generic as.data.frame function.

Usage

```
## S3 method for class 'dcm'
as.data.frame(x, row.names = NULL, optional = FALSE, run = NULL, ...)
```

Arguments

x	An EpiModel object of class <code>dcm()</code> .
row.names	See <code>as.data.frame.default()</code> .
optional	See <code>as.data.frame.default()</code> .
run	Run number for model; used for multiple-run sensitivity models. If NULL (default), will output data from all runs in a stacked data frame.
...	See <code>as.data.frame.default()</code> .

Details

Model output from `dcm()` simulations are available as a data frame with this helper function. The output data frame will include columns for time, the size of each compartment, the overall population size (the sum of compartment sizes), and the size of each flow.

For models with multiple runs (i.e., varying parameters - see example below), the default with the run parameter not specified will output all runs in a single stacked data frame.

Value

A data frame containing the data from x.

Examples

```
## Example 1: One-group SIS model with varying act.rate
param <- param.dcm(inf.prob = 0.2, act.rate = seq(0.05, 0.5, 0.05),
                  rec.rate = 1/50)
init <- init.dcm(s.num = 500, i.num = 1)
control <- control.dcm(type = "SIS", nsteps = 10)
mod1 <- dcm(param, init, control)
as.data.frame(mod1)
as.data.frame(mod1, run = 1)
as.data.frame(mod1, run = 10)

## Example 2: Two-group SIR model with vital dynamics
```

```

param <- param.dcm(inf.prob = 0.2, inf.prob.g2 = 0.1,
                  act.rate = 3, balance = "g1",
                  rec.rate = 1/50, rec.rate.g2 = 1/50,
                  a.rate = 1/100, a.rate.g2 = NA,
                  ds.rate = 1/100, ds.rate.g2 = 1/100,
                  di.rate = 1/90, di.rate.g2 = 1/90,
                  dr.rate = 1/100, dr.rate.g2 = 1/100)
init <- init.dcm(s.num = 500, i.num = 1, r.num = 0,
                s.num.g2 = 500, i.num.g2 = 1, r.num.g2 = 0)
control <- control.dcm(type = "SIR", nsteps = 10)
mod2 <- dcm(param, init, control)
as.data.frame(mod2)

```

as.data.frame.icm	<i>Extract Model Data for Stochastic Models</i>
-------------------	---

Description

This function extracts model simulations for objects of classes `icm` and `netsim` into a data frame using the generic `as.data.frame` function.

Usage

```

## S3 method for class 'icm'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  out = "vals",
  sim = NULL,
  qval = NULL,
  repair = "drop",
  ...
)

## S3 method for class 'netsim'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  out = "vals",
  sim = NULL,
  repair = "drop",
  ...
)

```

Arguments

x	An EpiModel object of class icm or netsim.
row.names	See as.data.frame.default() .
optional	See as.data.frame.default() .
out	Data output to data frame: "mean" for row means across simulations, "sd" for row standard deviations across simulations, "qnt" for row quantiles at the level specified in qval, or "vals" for values from individual simulations.
sim	If out="vals", the simulation number to output. If not specified, then data from all simulations will be output.
qval	Quantile value required when out="qnt".
repair	What to do with epi trackers that are too short. "drop" will remove them from the output, "pad" will add NA rows to get to the right size(Default = "drop").
...	See as.data.frame.default() .

Details

These methods work for both icm and netsim class models. The available output includes time-specific means, standard deviations, quantiles, and simulation values (compartment and flow sizes) from these stochastic model classes. Means, standard deviations, and quantiles are calculated by taking the row summary (i.e., each row of data corresponds to a time step) across all simulations in the model output.

Value

A data frame containing the data from x.

Examples

```
## Stochastic ICM SIS model
param <- param.icm(inf.prob = 0.8, act.rate = 2, rec.rate = 0.1)
init <- init.icm(s.num = 500, i.num = 1)
control <- control.icm(type = "SIS", nsteps = 10,
                      nsims = 3, verbose = FALSE)
mod <- icm(param, init, control)

# Default output all simulation runs, default to all in stacked data.frame
as.data.frame(mod)
as.data.frame(mod, sim = 2)

# Time-specific means across simulations
as.data.frame(mod, out = "mean")

# Time-specific standard deviations across simulations
as.data.frame(mod, out = "sd")

# Time-specific quantile values across simulations
as.data.frame(mod, out = "qnt", qval = 0.25)
as.data.frame(mod, out = "qnt", qval = 0.75)
```

```
## Not run:
## Stochastic SI Network Model
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

param <- param.net(inf.prob = 0.5)
init <- init.net(i.num = 10)
control <- control.net(type = "SI", nsteps = 10, nsims = 3, verbose = FALSE)
mod <- netsim(est, param, init, control)

# Same data extraction methods as with ICMs
as.data.frame(mod)
as.data.frame(mod, sim = 2)
as.data.frame(mod, out = "mean")
as.data.frame(mod, out = "sd")
as.data.frame(mod, out = "qnt", qval = 0.25)
as.data.frame(mod, out = "qnt", qval = 0.75)

## End(Not run)
```

as.data.frame.netdx *Extract Timed Edgelists for netdx Objects*

Description

This function extracts timed edgelists for objects of class `netdx` into a data frame using the generic `as.data.frame` function.

Usage

```
## S3 method for class 'netdx'
as.data.frame(x, row.names = NULL, optional = FALSE, sim = NULL, ...)
```

Arguments

<code>x</code>	An <code>EpiModel</code> object of class <code>netdx</code> .
<code>row.names</code>	See as.data.frame.default() .
<code>optional</code>	See as.data.frame.default() .
<code>sim</code>	The simulation number to output. If <code>NULL</code> , then data from all simulations will be output.
<code>...</code>	See as.data.frame.default() .

Value

A data frame containing the data from x.

Examples

```
# Initialize and parameterize the network model
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)

# Model estimation
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Simulate the network with netdx
dx <- netdx(est, nsims = 3, nsteps = 10, keep.tedgelist = TRUE,
            verbose = FALSE)

# Extract data from the first simulation
as.data.frame(dx, sim = 1)

# Extract data from all simulations
as.data.frame(dx)
```

as.epi.data.frame	<i>Validate and Convert to epi.data.frame</i>
-------------------	---

Description

This methods ensures that the data.frame is correctly formatted as an epi.data.frame

Usage

```
as.epi.data.frame(df)
```

Arguments

df	A data.frame to convert into an epi.data.frame
----	--

as.network.transmat	<i>Convert transmat Infection Tree into a network Object</i>
---------------------	--

Description

Converts a transmission matrix from the `get_transmat` function into a `network::network` class object.

Usage

```
## S3 method for class 'transmat'
as.network(x, ...)
```

Arguments

x	An object of class <code>transmat</code> to be converted into a network class object.
...	Unused.

Details

When converting from a `transmat` to a network object, this functions copies the edge attributes within the transmission matrix ('at', 'infDur', 'transProb', 'actRate', and 'finalProb') into edge attributes on the network.

Value

A `network::network` object.

as.phylo.transmat	<i>Convert transmat Infection Tree into a phylo Object</i>
-------------------	--

Description

Converts a transmission matrix from the `get_transmat` function into a `phylo` class object.

Usage

```
## S3 method for class 'transmat'
as.phylo(x, vertex.exit.times = NULL, ...)
```

Arguments

`x` An object of class `transmat`, the output from `get_transmat()`.

`vertex.exit.times` Optional numeric vector providing the time of departure of vertices, to be used to scale the lengths of branches reaching to the tips. Index position on vector corresponds to network id. NA indicates no departure, so branch will extend to the end of the tree.

`...` Further arguments (unused).

Details

Converts a `transmat()` object containing information about the history of a simulated infection into a `ape::phylo` object representation suitable for plotting as a tree with `ape::plot.phylo()`. Each infection event becomes a 'node' (horizontal branch) in the resulting phylo tree, and each network vertex becomes a 'tip' of the tree. The infection events are labeled with the vertex ID of the infector to make it possible to trace the path of infection.

The infection timing information is included to position the phylo-nodes, with the lines to the tips drawn to the max time value +1 (unless `vertex.exit.times` are passed in it effectively assumes all vertices are active until the end of the simulation).

If the `transmat` contains multiple infection seeds (there are multiple trees with separate root nodes), this function will return a list of class `multiPhylo`, each element of which is a phylo object. See `ape::read.tree()`.

Value

A phylo class object.

Examples

```
set.seed(13)

# Fit a random mixing TERGM with mean degree of 1 and mean edge
# duration of 20 time steps
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Parameterize the epidemic model as SI with one infected seed
param <- param.net(Inf.prob = 0.5)
init <- init.net(i.num = 1)
control <- control.net(type = "SI", nsteps = 40, nsims = 1, verbose = FALSE)

# Simulate the model
mod1 <- netsim(est, param, init, control)

# Extract the transmission matrix
tm <- get_transmat(mod1)
```

```
head(tm, 15)

# Convert to phylo object and plot
tmPhylo <- as.phylo.transmat(tm)
par(mar = c(1,1,1,1))
plot(tmPhylo, show.node.label = TRUE,
      root.edge = TRUE,
      cex = 0.75)
```

as_cumulative_edgelist

Convert an Object to a Cumulative Edgelist

Description

Converts an object that carries timed-edge information into a `cumulative_edgelist`: a `data.frame` with `head`, `tail`, `start`, `stop` columns matching the format used elsewhere in `EpiModel`. The typical use is feeding a `networkDynamic` object (from a non-`tergmLite` simulation, an external source, or a post-hoc construction) into the reachability tools.

Usage

```
as_cumulative_edgelist(x)
```

Arguments

`x` An object to be converted to a cumulative edgelist.

Details

The `networkDynamic` method subtracts 1 from each spell's terminus so that the resulting start/stop range is inclusive on both ends, matching the convention used by `get_cumulative_edgelist()` and the reachability functions.

Value

A `cumulative_edgelist` object: a `data.frame` with at least the columns `head`, `tail`, `start`, `stop`. Edges are active from time `start` to time `stop` inclusive; NA in `stop` means the edge was never observed to dissolve.

See Also

`get_forward_reachable()` / `get_backward_reachable()` for typical downstream uses; `dedup_cumulative_edgelist()` when the source contains overlapping spells for the same pair of nodes.

Other `cumulative_edgelist`: `dedup_cumulative_edgelist()`, `get_cumulative_degree()`, `get_cumulative_edgelist()`, `get_cumulative_edgelist_df()`, `get_partners()`, `reachable-nodes`, `update_cumulative_edgelist()`

Examples

```
## Not run:
# Convert a saved networkDynamic from a non-tergmLite netsim run
nd <- get_network(sim, sim = 1, network = 1)
el_cuml <- as_cumulative_edgelist(nd)
fwd <- get_forward_reachable(el_cuml, from_step = 1, to_step = 100)

## End(Not run)
```

as_tibble_edgelist	<i>Convert an Edgelist into a Tibble</i>
--------------------	--

Description

Convert an Edgelist into a Tibble

Usage

```
as_tibble_edgelist(el)
```

Arguments

el An edgelist in matrix or data frame form.

Value

The edgelist in tibble form with two columns named head and tail.

check_degdist_bal	<i>Check Degree Distribution for Balance in Target Statistics</i>
-------------------	---

Description

Checks for consistency in the implied network statistics of a two-group network in which the group size and group-specific degree distributions are specified.

Usage

```
check_degdist_bal(num.g1, num.g2, deg.dist.g1, deg.dist.g2)
```

Arguments

num.g1	Number of nodes in group 1.
num.g2	Number of nodes in group 2.
deg.dist.g1	Vector with fractional degree distribution for group 1.
deg.dist.g2	Vector with fractional degree distribution for group 2.

Details

This function outputs the number of nodes of degree 0 to g, where g is the length of a fractional degree distribution vector, given that vector and the size of the group. This utility is used to check for balance in implied degree given that fractional distribution within two-group network simulations, in which the degree-constrained counts must be equal across groups.

Examples

```
# An unbalanced distribution
check_degdist_bal(num.g1 = 500, num.g2 = 500,
  deg.dist.g2 = c(0.40, 0.55, 0.03, 0.02),
  deg.dist.g1 = c(0.48, 0.41, 0.08, 0.03))

# A balanced distribution
check_degdist_bal(num.g1 = 500, num.g2 = 500,
  deg.dist.g1 = c(0.40, 0.55, 0.04, 0.01),
  deg.dist.g2 = c(0.48, 0.41, 0.08, 0.03))
```

color_tea

Create a TEA Variable for Infection Status for ndtv Animations

Description

Creates a new color-named temporally-extended attribute (TEA) variable in a networkDynamic object containing a disease status TEA in numeric format.

Usage

```
color_tea(
  nd,
  old.var = "testatus",
  old.sus = "s",
  old.inf = "i",
  old.rec = "r",
  new.var = "ndtvcol",
  new.sus = NULL,
  new.inf = NULL,
  new.rec = NULL,
  verbose = TRUE
)
```

Arguments

nd	An object of class networkDynamic.
old.var	Old TEA variable name.
old.sus	Status value for susceptible in old TEA variable.

old.inf	Status value for infected in old TEA variable.
old.rec	Status value for recovered in old TEA variable.
new.var	New TEA variable name to be stored in networkDynamic object.
new.sus	Status value for susceptible in new TEA variable.
new.inf	Status value for infected in new TEA variable.
new.rec	Status value for recovered in new TEA variable.
verbose	If TRUE, print progress to console.

Details

The ndtv package (<https://cran.r-project.org/package=ndtv>) produces animated visuals for dynamic networks with evolving edge structures and nodal attributes. Nodal attribute dynamics in ndtv movies require a temporally extended attribute (TEA) containing a standard R color for each node at each time step. By default, the EpiModel package uses TEAs to store disease status history in network model simulations run in `netsim()`. But that status TEA is in numeric format (0, 1, 2). The `color_tea` function transforms those numeric values of that disease status TEA into a TEA with color values in order to visualize status changes in ndtv.

The convention in `plot.netsim()` is to color the susceptible nodes as blue, infected nodes as red, and recovered nodes as green. Alternate colors may be specified using the `new.sus`, `new.inf`, and `new.rec` parameters, respectively.

Using the `color_tea` function with a `netsim` object requires that TEAs for disease status be used and that the `networkDynamic` object be saved in the output: `tergmListe` must be set to FALSE in `control.net()`.

Value

The updated object of class `networkDynamic`.

See Also

`netsim()` and the ndtv package documentation.

comp_plot	<i>Plot Compartment Diagram for Epidemic Models</i>
-----------	---

Description

Plots a compartment flow diagram for deterministic compartmental models, stochastic individual contact models, and stochastic network models.

Usage

```
comp_plot(x, at, digits, ...)

## S3 method for class 'netsim'
comp_plot(x, at = 1, digits = 3, ...)

## S3 method for class 'icm'
comp_plot(x, at = 1, digits = 3, ...)

## S3 method for class 'dcm'
comp_plot(x, at = 1, digits = 3, run = 1, ...)
```

Arguments

x	An EpiModel object of class dcm, icm, or netsim.
at	Time step for model statistics. For dcm models, must be one of the times the model was solved over, held in control\$timesteps.
digits	Number of significant digits to print.
...	Additional arguments passed to plot (not currently used).
run	Model run number, for dcm class models with multiple runs (sensitivity analyses).

Details

The comp_plot function provides a visual summary of an epidemic model at a specific time step. The information contained in comp_plot is the same as in the summary functions for a model, but presented graphically as a compartment flow diagram.

For dcm class plots, specify the model run number if the model contains multiple runs, as in a sensitivity analysis. For icm and netsim class plots, the run argument is not used; the plots show the means and standard deviations across simulations at the specified time step.

These plots are currently limited to one-group models for each of the three model classes. That functionality may be expanded in future software releases.

Examples

```
## Example 1: DCM SIR model with varying act.rate
param <- param.dcm(inf.prob = 0.2, act.rate = 5:7,
  rec.rate = 1/3, a.rate = 1/90, ds.rate = 1/100,
  di.rate = 1/35, dr.rate = 1/100)
init <- init.dcm(s.num = 1000, i.num = 1, r.num = 0)
control <- control.dcm(type = "SIR", nsteps = 25, verbose = FALSE)
mod1 <- dcm(param, init, control)
comp_plot(mod1, at = 25, run = 3)

## Example 2: ICM SIR model with 3 simulations
param <- param.icm(inf.prob = 0.2, act.rate = 3, rec.rate = 1/50,
  a.rate = 1/100, ds.rate = 1/100,
  di.rate = 1/90, dr.rate = 1/100)
```

```

init <- init.icm(s.num = 500, i.num = 1, r.num = 0)
control <- control.icm(type = "SIR", nsteps = 25,
                      nsims = 3, verbose = FALSE)
mod2 <- icm(param, init, control)
comp_plot(mod2, at = 25, digits = 1)

```

control.dcm

*Control Settings for Deterministic Compartmental Models***Description**

Sets the controls for deterministic compartmental models simulated with `dcm()`.

Usage

```

control.dcm(
  type,
  nsteps,
  dt = 1,
  odemethod = "lsoda",
  dede = FALSE,
  new.mod = NULL,
  sens.param = TRUE,
  print.mod = FALSE,
  verbose = FALSE,
  ...
)

```

Arguments

type	Disease type to be modeled, with the choice of "SI" for Susceptible-Infected diseases, "SIR" for Susceptible-Infected-Recovered diseases, and "SIS" for Susceptible-Infected-Susceptible diseases.
nsteps	Number of time steps to solve the model over or vector of times to solve the model over. If the number of time steps, then this must be a positive integer of length 1.
dt	Time unit for model solutions, with the default of 1. Model solutions for fractional time steps may be obtained by setting this to a number between 0 and 1.
odemethod	Ordinary differential equation (ODE) integration method, with the default of "lsoda" (see <code>deSolve::ode</code> for other options).
dede	If TRUE, use the delayed differential equation solver, which allows for time-lagged variables.
new.mod	If not running a base model type, a function with a new model to be simulated (see details).

<code>sens.param</code>	If TRUE, evaluate arguments in parameters or initial conditions with length greater than 1 as sensitivity analyses, with one model run per value. If FALSE, one model will be run with parameters and initial conditions of arbitrary length (the model may error unless the model function is designed to accommodate vectors).
<code>print.mod</code>	If TRUE, print the model form to the console.
<code>verbose</code>	If TRUE, print model progress to the console.
<code>...</code>	additional control settings passed to model.

Details

`control.dcm` sets the required control settings for any deterministic compartmental models solved with the `dcm()` function. Controls are required for both base model types and original models. For all base models, the `type` argument is a necessary parameter and it has no default.

Value

An `EpiModel` object of class `control.dcm`.

ODE Solver

As of `EpiModel` 2.5.5, the default ODE solver was changed from "rk4" (fixed-step Runge-Kutta 4) to "lsoda" (adaptive step-size). The "lsoda" method automatically adjusts its internal step size, providing numerical stability across a wide range of parameter values. The fixed-step "rk4" solver could produce numerical instability (negative compartment sizes, NaN values) for models with high transmission rates or stiff dynamics. Users who require the previous behavior can set `odemethod = "rk4"` explicitly.

New Model Functions

The form of the model function for base models may be displayed with the `print.mod` argument set to TRUE. In this case, the model will not be run. These model forms may be used as templates to write original model functions.

These new models may be input and solved with `dcm()` using the `new.mod` argument, which requires as input a model function.

See Also

Use `param.dcm()` to specify model parameters and `init.dcm()` to specify the initial conditions. Run the parameterized model with `dcm()`.

Description

Sets the controls for stochastic individual contact models simulated with `icm()`.

Usage

```
control.icm(type, nsteps, nsims = 1, verbose = FALSE, verbose.int = 0)
```

Arguments

type	Disease type to be modeled, with the choice of "SI" for Susceptible-Infected diseases, "SIR" for Susceptible-Infected-Recovered diseases, and "SIS" for Susceptible-Infected-Susceptible diseases.
nsteps	Number of time steps to solve the model over. This must be a positive integer.
nsims	Number of simulations to run.
verbose	If TRUE, print model progress to the console.
verbose.int	Time step interval for printing progress to console, where 0 (the default) prints completion status of entire simulation and positive integer x prints progress after every x time steps.

Details

`control.icm` sets the required control settings for any stochastic individual contact model solved with the `icm()` function. ICM simulations use the built-in SI, SIR, and SIS disease types only. The `type` argument is required and has no default. For custom or extension epidemic models, use the network model class via `control.net()` instead.

Value

An `EpiModel` object of class `control.icm`.

See Also

Use `param.icm()` to specify model parameters and `init.icm()` to specify the initial conditions. Run the parameterized model with `icm()`.

Description

Sets the controls for stochastic network models simulated with [netsim](#).

Usage

```
control.net(
  type,
  nsteps,
  start = 1,
  nsims = 1,
  ncores = 1,
  resimulate.network = FALSE,
  tergmLite = FALSE,
  edges.correct.attr = NULL,
  cumulative.edgelist = FALSE,
  truncate.el.cuml = 0,
  attr.rules,
  epi.by,
  initialize.FUN = initialize.net,
  resim_nets.FUN = resim_nets,
  summary_nets.FUN = summary_nets,
  infection.FUN = NULL,
  recovery.FUN = NULL,
  departures.FUN = NULL,
  arrivals.FUN = NULL,
  nwupdate.FUN = nwupdate.net,
  prevalence.FUN = prevalence.net,
  verbose.FUN = verbose.net,
  module.order = NULL,
  save.nwstats = TRUE,
  nwstats.formula = "formation",
  save.transmat = TRUE,
  save.network,
  save.run = FALSE,
  save.cumulative.edgelist = FALSE,
  save.other,
  verbose = TRUE,
  verbose.int = 1,
  skip.check = FALSE,
  raw.output = FALSE,
  future.use.plan = FALSE,
  tergmLite.track.duration = FALSE,
  set.control.ergm = control.simulate.formula(MCMC.burnin = 2e+05),
```

```

    set.control.tergm = control.simulate.formula.tergm(MCMC.maxchanges =
      .Machine$integer.max),
    save.diss.stats = TRUE,
    dat.updates = NULL,
    ...
  )

```

Arguments

type	Disease type to be modeled, with the choice of "SI" for Susceptible-Infected diseases, "SIR" for Susceptible-Infected-Recovered diseases, and "SIS" for Susceptible-Infected-Susceptible diseases.
nsteps	Number of time steps to simulate the model over. This must be a positive integer that is equal to the final step of a simulation. If a simulation is restarted with start argument, this number must be at least one greater than that argument's value.
start	For models with network resimulation, time point to start up the simulation. For restarted simulations, this must be one greater than the final time step in the prior simulation and must be less than the value in nsteps.
nsims	The total number of disease simulations.
ncores	Number of processor cores to run multiple simulations on, using the future framework with multisession. See <code>future.use.plan</code> for other future back-ends.
resimulate.network	If TRUE, resimulate the network at each time step. This is required when the epidemic or demographic processes impact the network structure (e.g., vital dynamics). This parameter controls whether <code>resim_nets.FUN</code> performs actual network resimulation; it does not affect the order in which modules are executed (see <code>module.order</code>). Setting <code>tergmLite = TRUE</code> forces <code>resimulate.network = TRUE</code> with a warning.
tergmLite	Logical indicating usage of either <code>tergm</code> (<code>tergmLite = FALSE</code>), or <code>tergmLite</code> (<code>tergmLite = TRUE</code>). Default of FALSE. When TRUE, <code>resimulate.network</code> is automatically set to TRUE (with a warning if the user explicitly set it to FALSE).
edges.correct.attr	Name of a binary nodal attribute marking the nodes eligible to form ties, used by edges_correct when it rescales the edges coefficient to preserve mean degree as the population changes. NULL by default, which counts every active node and is correct whenever every active node can form a tie. Set it when the model carries a subpopulation that stays active but is structurally excluded from the network, such as an age band past a sexual-cessation age whose target statistics are all zero. Without it the correction counts nodes that can never hold an edge, and the whole of the adjustment lands on the nodes that can, thinning mean degree among them by the excluded share. Nodes with a value of 1 are eligible; NA is treated as ineligible.
cumulative.edgelist	If TRUE, <code>EpiModel</code> maintains a running record of every edge across the simulation (the <i>cumulative edgelist</i>) by calling update_cumulative_edgelist once

per network from the built-in network-resimulation module ([resim_nets](#)) at every time step. Off by default. Enabling it is the canonical way to query partnership histories under `tergmLite = TRUE`, where the full `networkDynamic` history is not retained. Inside a custom module the live data is read via [get_cumulative_edgelist](#) or [get_cumulative_edgelist_df](#), and derived helpers [get_partners](#), [get_cumulative_degree](#), [get_forward_reachable](#), and [get_backward_reachable](#). See the vignette("network-objects", package = "EpiModel") for a worked example.

<code>truncate.el.cuml</code>	Number of time steps of the cumulative edgelist to retain, passed as the <code>truncate</code> argument to each automatic update_cumulative_edgelist call. Default is 0, which keeps only currently active edges; use <code>Inf</code> to retain the full history (memory permitting) or a positive integer to keep dissolved edges for that many steps after they ended. Only relevant when <code>cumulative.edgelist = TRUE</code> .
<code>attr.rules</code>	A list containing the rules for setting the attributes of incoming nodes, with one list element per attribute to be set (see details below).
<code>epi.by</code>	A character vector of length 1 containing a nodal attribute for which subgroup stratified prevalence summary statistics are calculated. This nodal attribute must be contained in the network model formation formula, otherwise it is ignored.
<code>initialize.FUN</code>	Module to initialize the model at time 1, with the default function of initialize.net .
<code>resim_nets.FUN</code>	Module to resimulate the network at each time step, with the default function of resim_nets .
<code>summary_nets.FUN</code>	Module to extract summary statistics of the network at each time step, with the default function of summary_nets .
<code>infection.FUN</code>	Module to simulate disease infection, with the default function of infection.net .
<code>recovery.FUN</code>	Module to simulate disease recovery, with the default function of recovery.net .
<code>departures.FUN</code>	Module to simulate departure or exit, with the default function of departures.net .
<code>arrivals.FUN</code>	Module to simulate arrivals or entries, with the default function of arrivals.net .
<code>nwupdate.FUN</code>	Module to handle updating of network structure and nodal attributes due to exogenous epidemic model processes, with the default function of nwupdate.net .
<code>prevalence.FUN</code>	Module to calculate disease prevalence at each time step, with the default function of prevalence.net .
<code>verbose.FUN</code>	Module to print simulation progress to screen, with the default function of verbose.net .
<code>module.order</code>	A character vector of module names that lists modules in the order in which they should be evaluated within each time step. If <code>NULL</code> , the modules will be evaluated as follows: first any new modules supplied through <code>...</code> in the order in which they are listed, then the built-in modules in the order in which they are listed as arguments above. <code>initialize.FUN</code> will always be run first and <code>verbose.FUN</code> will always be run last. Module ordering is independent of <code>resimulate.network</code> : the specified order is always respected regardless of whether network resimulation is enabled. In the default ordering, <code>resim_nets.FUN</code> runs before <code>infection.FUN</code> , so the network is resimulated before transmission is evaluated at each time step. Important: when set, <code>module.order</code> replaces the entire dispatch order: built-in modules not listed will not run. <code>control.net()</code> validates the entries at construction time: each name must correspond to a <code>.FUN</code>

argument that has been supplied (either as a formal argument or through ...), and initialize.FUN / verbose.FUN may not appear because they run outside the per-step module loop. Omitting resim_nets.FUN, summary_nets.FUN, or nwupdate.FUN produces a warning, since these built-ins are typically required for correct semantics: resim_nets.FUN advances the TERGM each step, summary_nets.FUN records network statistics, and nwupdate.FUN applies vertex (de)activation from active/exitTime/entrTime and copies nodal attributes to the network.

save.nwstats	If TRUE, save network statistics in a data frame. The statistics to be saved are specified in the nwstats.formula argument.
nwstats.formula	A right-hand sided ERGM formula that includes network statistics of interest, with the default to the formation formula terms. Supports multilayer specification.
save.transmat	If TRUE, complete transmission matrix is saved at simulation end.
save.network	If TRUE, networkDynamic or networkLite object is saved at simulation end.
save.run	If TRUE, the run sublist of dat is saved, allowing a simulation to restart from this output.
save.cumulative.edgelist	If TRUE, the cumulative edgelist is attached to the returned netsim object as sim\$cumulative.edgelist, a list with one element per simulation (the same data.frame shape produced by get_cumulative_edgelist_df). Requires cumulative.edgelist = TRUE; without it, no history is collected to save. Off by default to keep output objects small.
save.other	A character vector of elements on the netsim_dat main data list to save out after each simulation. One example for base models is the attribute list, "attr", at the final time step.
verbose	If TRUE, print model progress to the console.
verbose.int	Time step interval for printing progress to console, where 0 prints completion status of entire simulation and positive integer x prints progress after every x time steps. The default is to print progress after each time step.
skip.check	If TRUE, skips the default error checking for the structure and consistency of the parameter values, initial conditions, and control settings before running base epidemic models. Setting this to FALSE is recommended when running models with new modules specified.
raw.output	If TRUE, netsim will output a list of raw data (one per simulation) instead of a cleaned and formatted netsim object.
future.use.plan	If FALSE, netsim uses multisession with workers = ncores for its parallelization. If TRUE, netsim uses the user-defined plan from globalEnv. It may also be given the output of a future::tweak() call, which sets up a user-defined temporary plan within netsim; this can be useful for distributed computation (HPC).
tergmLite.track.duration	If TRUE, track duration information for models in tergmLite simulations. Supports multilayer specification.

<code>set.control.ergm</code>	Control arguments passed to <code>ergm::simulate_formula.network</code> . In <code>netsim</code> , this is only used when initializing the network with <code>edapprox = TRUE</code> . All other simulations in <code>netsim</code> use <code>tergm</code> . Supports multilayer specification.
<code>set.control.tergm</code>	Control arguments passed to <code>tergm::simulate_formula.network</code> . See the help file for netdx for details and examples on specifying this parameter. Supports multilayer specification.
<code>save.diss.stats</code>	If <code>TRUE</code> , <code>netsim</code> will compute and save duration and dissolution statistics for plotting and printing, provided <code>save.network</code> is <code>TRUE</code> , <code>tergmLite</code> is <code>FALSE</code> , and the dissolution model is homogeneous.
<code>dat.updates</code>	Either <code>NULL</code> , a single function taking arguments <code>dat</code> , <code>at</code> , and <code>network</code> , or a list of functions of length one greater than the number of networks being simulated, with each function in the list taking arguments <code>dat</code> and <code>at</code> . Here <code>dat</code> is the main <code>netsim_dat</code> class object, <code>at</code> is the current timestep, and <code>network</code> is an index indicating the current position within the sequence of network (re)simulations on each time step. If a single function is passed, it will be called before the first network is simulated and after each network is simulated, with <code>network = 0L</code> before the first network is simulated and with <code>network = i</code> after the <i>i</i> th network is simulated. If a list of functions is passed, the first function will be called before the first network is simulated, and the <i>i</i> + 1th function will be called after the <i>i</i> th network is simulated. (Note that <code>at = 0L</code> is used for initial cross-sectional simulations in sim_nets_t1 .) The function(s) should return the <code>netsim_dat</code> object with any updates needed to correctly represent the network states for calls to <code>simulate</code> and/or <code>summary</code> . This can be useful if nodal attributes appearing in one network model depend on nodal degrees in a different network.
<code>...</code>	Additional control settings passed to model.

Details

`control.net` sets the required control settings for any network model solved with the `netsim` function. Controls are required for both base model types and when passing original process modules. For an overview of control settings for base models, consult the [Network Modeling for Epidemics](#) course materials. For all base models, the `type` argument is a necessary parameter and it has no default.

Value

An `EpiModel` object of class `control.net`.

The `attr.rules` Argument

The `attr.rules` parameter is used to specify the rules for how nodal attribute values for incoming nodes should be set. These rules are only necessary for models in which there are incoming nodes (i.e., arrivals). There are three rules available for each attribute value:

- `current`: new nodes will be assigned this attribute in proportion to the distribution of that attribute among existing nodes at that current time step.

- `t1`: new nodes will be assigned this attribute in proportion to the distribution of that attribute among nodes at time 1 (that is, the proportions set in the original network for `netest`).
- `Value`: all new nodes will be assigned this specific value, with no variation. For example, the rules list `attr.rules = list(race = "t1", sex = "current", status = "s")` specifies how the race, sex, and status attributes should be set for incoming nodes. By default, the rule is "current" for all attributes except status, in which case it is "s" (that is, all incoming nodes are susceptible).

Checkpointing Simulations

`netsim` has a built-in checkpoint system to prevent losing computation work if the function is interrupted (SIGINT, power loss, time limit exceeded on a computation cluster). When enabled, each simulation will be saved every `.checkpoint.steps` time steps. Then, if a checkpoint enabled simulation is launched again with `netsim`, it will restart at the last checkpoint available in the saved data.

To enable the checkpoint capabilities of `netsim`, two control arguments have to be set: `.checkpoint.steps`, which is a positive number of time steps to be run between each file save; and `.checkpoint.dir`, which is the path to a directory to save the checkpointed data. If `.checkpoint.dir` directory does not exist, `netsim` will attempt to create it on the first checkpoint save. With these two controls defined, one can simply re-run `netsim` with the same arguments to restart a set of simulations that were interrupted.

Simulations are checkpointed individually: for example, if 3 simulations are run on a single core, the first 2 are finished, then the interruption occurs during the third, `netsim` will only restart the third one from the last checkpoint.

A `.checkpoint.compress` argument can be set to overwrite the `compress` argument in `saveRDS` used to save the checkpointed data. The current default for `saveRDS` is `gzip` (`gz`), which provides fast compression that usually works well on `netsim` objects.

By default, if `netsim` reaches the end of all simulations, the checkpoint data directory and its content are removed before returning the `netsim` object. The `.checkpoint.keep` argument can be set to `TRUE` to prevent this removal to inspect the raw simulation objects.

New Modules

Base network models use a set of module functions that specify how the individual nodes in the network are subjected to infection, recovery, demographics, and other processes. Core modules are those listed in the `.FUN` arguments. For each module, there is a default function used in the simulation. The default infection module, for example, is contained in the `infection.net` function.

For original models, one may substitute replacement module functions for any of the default functions. New modules may be added to the workflow at each time step by passing a module function via the `...` argument. Consult the [Extending EpiModel](#) section of the Network Modeling for Epidemics course materials. One may remove existing modules, such as `arrivals.FUN`, from the workflow by setting the parameter value for that argument to `NULL`.

End Horizon

`netsim` implements an "End Horizon" mechanism, where a set of modules are removed from the simulation at a specific time step. This is enabled through the `end.horizon` parameter to `control.net`.

This parameter must receive a list with fields at, the time step at which the end horizon occurs, and modules, a character vector with the names of the modules to remove. For example, `list(at = 208, modules = c("arrivals.FUN", "infections.FUN"))`.

See Also

Use [param.net](#) to specify model parameters and [init.net](#) to specify the initial conditions. Run the parameterized model with [netsim](#).

create_dat_object	<i>Create a Minimal netsim_dat Main List Object for a Network Model</i>
-------------------	---

Description

This helper function populates a `netsim_dat` main list object with the minimal required elements. All parameters are optional. When none are given the resulting object is only a shell list of class `netsim_dat` with the different named elements defined as empty lists.

Usage

```
create_dat_object(
  param = list(),
  init = list(),
  control = list(),
  run = list()
)
```

Arguments

param	An EpiModel object of class param.net() .
init	An EpiModel object of class init.net() .
control	An EpiModel object of class control.net() .
run	A list that will contains the objects created by netsim() that are required for between step communication. This list must be preserved for restarting models.

Value

A `netsim_dat` main list object.

create_scenario_list *Make a list of EpiModel scenarios from a data.frame of scenarios*

Description

An EpiModel scenario allows one or multiple set of parameters to be applied to a model a predefined timesteps. They are usually used by a researcher who wants to model counterfactuals using a pre calibrated model.

Usage

```
create_scenario_list(scenarios.df)
```

Arguments

scenarios.df a data.frame

Value

a list of EpiModel scenarios

scenarios.df

The scenarios.df is a data.frame of values to be used as parameters.

It must contain a ".at" column, specifying when the changes should occur. It requires the "updater" module of EpiModel. *See, vignette*. If the ".at" value of a row is less than two, the changes will be applied to the parameter list itself. The second mandatory column is ".scenario.id". It is used to distinguish the different scenarios. If multiple rows share the same ".scenario.id", the resulting scenario will contain one updater per row. This permits modifying parameters at multiple points in time. (e.g. an intervention limited in time).

The other column names must correspond either to: the name of one parameter if this parameter is of size 1 or the name of the parameter with "_1", "_2", "N" with the second part being the position of the value for a parameter of size > 1. This means that the parameter names cannot contain any underscore ". (e.g "a.rate", "d.rate_1", "d.rate_2")

dcm

Deterministic Compartmental Models

Description

Solves deterministic compartmental epidemic models for infectious disease.

Usage

```
dcm(param, init, control)
```

Arguments

<code>param</code>	Model parameters, as an object of class <code>param.dcm()</code> .
<code>init</code>	Initial conditions, as an object of class <code>init.dcm()</code> .
<code>control</code>	Control settings, as an object of class <code>control.dcm()</code> .

Details

The `dcm` function uses the ordinary differential equation solver in the `deSolve` package to model disease as a deterministic compartmental system. The parameterization for these models follows the standard approach in `EpiModel`, with epidemic parameters, initial conditions, and control settings.

The `dcm` function performs modeling of both base model types and original models with new structures. Base model types include one-group and two-group models with disease types for Susceptible-Infected (SI), Susceptible-Infected-Recovered (SIR), and Susceptible-Infected-Susceptible (SIS). Both base and original models require the `param`, `init`, and `control` inputs.

Value

A list of class `dcm` with the following elements:

- **param:** the epidemic parameters passed into the model through `param`, with additional parameters added as necessary.
- **control:** the control settings passed into the model through `control`, with additional controls added as necessary.
- **epi:** a list of data frames, one for each epidemiological output from the model. Outputs for base models always include the size of each compartment, as well as flows in, out of, and between compartments.

References

Soetaert K, Petzoldt T, Setzer W. Solving Differential Equations in R: Package `deSolve`. *Journal of Statistical Software*. 2010; 33(9): 1-25. doi:10.18637/jss.v033.i09.

See Also

Extract the model results with `as.data.frame.dcm()`. Summarize the time-specific model results with `summary.dcm()`. Plot the model results with `plot.dcm()`. Plot a compartment flow diagram with `comp_plot()`.

Examples

```
## Example 1: SI Model (One-Group)
# Set parameters
param <- param.dcm(
  inf.prob = 0.2, act.rate = 0.25
)
init <- init.dcm(
  s.num = 500, i.num = 1
)
control <- control.dcm(
  type = "SI", nsteps = 500
)
mod1 <- dcm(param, init, control)
mod1
plot(mod1)
```

```
## Example 2: SIR Model with Vital Dynamics (One-Group)
param <- param.dcm(inf.prob = 0.2, act.rate = 5,
                   rec.rate = 1/3, a.rate = 1/90, ds.rate = 1/100,
                   di.rate = 1/35, dr.rate = 1/100)
init <- init.dcm(s.num = 500, i.num = 1, r.num = 0)
control <- control.dcm(type = "SIR", nsteps = 500)
mod2 <- dcm(param, init, control)
mod2
plot(mod2)

## Example 3: SIS Model with act.rate Sensitivity Parameter
param <- param.dcm(inf.prob = 0.2, act.rate = seq(0.1, 0.5, 0.1),
                   rec.rate = 1/50)
init <- init.dcm(s.num = 500, i.num = 1)
control <- control.dcm(type = "SIS", nsteps = 500)
mod3 <- dcm(param, init, control)
mod3
plot(mod3)

## Example 4: SI Model with Vital Dynamics (Two-Group)
param <- param.dcm(inf.prob = 0.4, inf.prob.g2 = 0.1,
                   act.rate = 0.25, balance = "g1",
                   a.rate = 1/100, a.rate.g2 = NA,
                   ds.rate = 1/100, ds.rate.g2 = 1/100,
                   di.rate = 1/50, di.rate.g2 = 1/50)
init <- init.dcm(s.num = 500, i.num = 1,
                 s.num.g2 = 500, i.num.g2 = 0)
control <- control.dcm(type = "SI", nsteps = 500)
mod4 <- dcm(param, init, control)
mod4
plot(mod4)

## Example 5: SI Model with Varying Initial Conditions
param <- param.dcm(inf.prob = 0.2, act.rate = 0.25)
init <- init.dcm(s.num = 500, i.num = c(1, 5, 25))
control <- control.dcm(type = "SI", nsteps = 500)
mod5 <- dcm(param, init, control)
mod5
plot(mod5)
```

dedup_cumulative_edgelist

Deduplicate a Cumulative Edgelist

Description

Merges overlapping spells for the same (head, tail) pair into a single row. Useful when concatenating cumulative edgelist across multiple simulations or sources, where the same partnership can appear in more than one segment. The reachability functions assume non-overlapping input.

Usage

```
dedup_cumulative_edgelist(el)
```

Arguments

`el` A cumulative edgelist with potentially overlapping edges.

Value

A cumulative edgelist with no overlapping edges for the same (head, tail) pair.

See Also

`as_cumulative_edgelist()` for one source of duplicates; `get_forward_reachable()` / `get_backward_reachable()`, which expect non-overlapping input.

Other cumulative_edgelist: `as_cumulative_edgelist()`, `get_cumulative_degree()`, `get_cumulative_edgelist()`, `get_cumulative_edgelists_df()`, `get_partners()`, `reachable-nodes`, `update_cumulative_edgelist()`

Examples

```
## Not run:
el_cum1 <- dedup_cumulative_edgelist(el_raw)

## End(Not run)
```

delete_edges

Remove Edges That Include Specified Vertices

Description

Given a current two-column matrix of edges and a vector of vertex IDs, this function removes any rows of the edgelist in which the IDs are present.

Usage

```
delete_edges(el, vid)
```

Arguments

`el` A two-column matrix of current edges (edgelist).

`vid` A vector of vertex IDs whose edges are to be deleted from the edgelist.

Value

Returns an updated edgelist object, with any edges including the specified vertices removed.

delete_vertices	<i>Fast Version of network::delete.vertices for Edgelist-formatted Network</i>
-----------------	--

Description

Given a current two-column matrix of edges and a vector of IDs to delete from the matrix, this function first removes any rows of the edgelist in which the IDs are present and then permutes downward the index of IDs on the edgelist that were numerically larger than the IDs deleted.

Usage

```
delete_vertices(el, vid)
```

Arguments

el	A two-column matrix of current edges (edgelist) with an attribute variable n containing the total current network size.
vid	A vector of IDs to delete from the edgelist.

Details

This function is used in EpiModel modules to remove vertices (nodes) from the edgelist object to account for exits from the population (e.g., deaths and out-migration).

Value

Returns an updated edgelist object, el, with the edges of deleted vertices removed from the edgelist and the ID numbers of the remaining edges permuted downward.

Examples

```
## Not run:
library("EpiModel")
set.seed(12345)
nw <- network_initialize(100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
x <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

param <- param.net(inf.prob = 0.3)
init <- init.net(i.num = 10)
control <- control.net(type = "SI", nsteps = 100, nsims = 5,
                      tergmLite = TRUE)

# Set seed for reproducibility
set.seed(123456)
```

```

# networkLite representation structure after initialization
dat <- crosscheck.net(x, param, init, control)
dat <- initialize.net(x, param, init, control)

# Current edges
head(dat$el[[1]], 20)

# Remove nodes 1 and 2
nodes.to.delete <- 1:2
dat$el[[1]] <- delete_vertices(dat$el[[1]], nodes.to.delete)

# Newly permuted edges
head(dat$el[[1]], 20)

## End(Not run)

```

depart_nodes

Depart Nodes from the netsim_dat Object

Description

Depart Nodes from the netsim_dat Object

Usage

```
depart_nodes(dat, departures)
```

Arguments

dat	the netsim_dat object
departures	the vertex ids of nodes to depart

Details

If `tergmLite` is `FALSE`, the vertex ids departures are deactivated (from the current timestep onward) in each `networkDynamic` stored in `dat$nw`. If `tergmLite` is `TRUE`, the vertex ids departures are deleted from `dat$el`, `dat$attr`, and `dat$net_attr`.

Value

the updated netsim_dat object with the nodes in departures departed

dissolution_coefs	<i>Dissolution Coefficients for Stochastic Network Models</i>
-------------------	---

Description

Calculates dissolution coefficients, given a dissolution model and average edge duration, to pass as offsets to an ERGM/TERGM model fit in `netest`.

Usage

```
dissolution_coefs(dissolution, duration, d.rate = 0)
```

Arguments

dissolution	Right-hand sided STERGM dissolution formula (see <code>netest()</code>). See below for list of supported dissolution models.
duration	A vector of mean edge durations in arbitrary time units.
d.rate	Departure or exit rate from the population, as a single homogeneous rate that applies to the entire population.

Details

This function performs two calculations for dissolution coefficients used in a network model estimated with `netest()`:

1. **Transformation:** the mean durations of edges in a network are mathematically transformed to logit coefficients.
2. **Adjustment:** in a dynamic network simulation in an open population (in which there are departures), it is further necessary to adjust these coefficients; this upward adjustment accounts for departure as a competing risk to edge dissolution.

The current dissolution models supported by this function and in network model estimation in `netest()` are as follows:

- `~offset(edges)`: a homogeneous dissolution model in which the edge duration is the same for all partnerships. This requires specifying one duration value.
- `~offset(edges) + offset(nodematch("<attr>"))`: a heterogeneous model in which the edge duration varies by whether the nodes in the dyad have similar values of a specified attribute. The duration vector should now contain two values: the first is the mean edge duration of non-matched dyads, and the second is the duration of the matched dyads.
- `~offset(edges) + offset(nodemix("<attr>"))`: a heterogeneous model that extends the `nodematch` model to include non-binary attributes for homophily. The duration vector should first contain the base value, then the values for every other possible combination in the term.

Value

A list of class `dissscoef` with the following elements:

- **dissolution:** right-hand sided STERGM dissolution formula passed in the function call.
- **duration:** mean edge durations passed into the function.
- **coef.crude:** mean durations transformed into logit coefficients.
- **coef.adj:** crude coefficients adjusted for the risk of departure on edge persistence, if the `d.rate` argument is supplied.
- **coef.form.corr:** corrections to be subtracted from formation coefficients.
- **d.rate:** the departure rate.
- **diss.model.type:** the form of the dissolution model; options include `edgesonly`, `nodematch`, and `nodemix`.

Examples

```
## Homogeneous dissolution model with no departures
dissolution_coefs(dissolution = ~offset(edges), duration = 25)

## Homogeneous dissolution model with departures
dissolution_coefs(dissolution = ~offset(edges), duration = 25,
                  d.rate = 0.001)

## Heterogeneous dissolution model in which same-race edges have
## shorter duration compared to mixed-race edges, with no departures
dissolution_coefs(dissolution = ~offset(edges) + offset(nodematch("race")),
                  duration = c(20, 10))

## Heterogeneous dissolution model in which same-race edges have
## shorter duration compared to mixed-race edges, with departures
dissolution_coefs(dissolution = ~offset(edges) + offset(nodematch("race")),
                  duration = c(20, 10), d.rate = 0.001)

## Not run:
## Extended example for differential homophily by age group
# Set up the network with nodes categorized into 5 age groups
nw <- network_initialize(n = 1000)
age.grp <- sample(1:5, 1000, TRUE)
nw <- set_vertex_attribute(nw, "age.grp", age.grp)

# durations = non-matched, age.grp1 & age.grp1, age.grp2 & age.grp2, ...
# TERGM will include differential homophily by age group with nodematch term
# Target stats for the formation model are overall edges, and then the number
# matched within age.grp 1, age.grp 2, ..., age.grp 5
form <- ~edges + nodematch("age.grp", diff = TRUE)
target.stats <- c(450, 100, 125, 40, 80, 100)

# Target stats for the dissolution model are duration of non-matched edges,
# then duration of edges matched within age.grp 1, age.grp 2, ..., age.grp 5
durs <- c(60, 30, 80, 100, 125, 160)
diss <- dissolution_coefs(~offset(edges) +
```

```

        offset(nodematch("age.grp", diff = TRUE)),
        duration = durs)

# Fit the TERGM
fit <- netest(nw, form, target.stats, diss)

# Full diagnostics to evaluate model fit
dx <- netdx(fit, nsims = 10, ncores = 4, nsteps = 300)
print(dx)

# Simulate one long time series to examine timed edgelist
dx <- netdx(fit, nsims = 1, nsteps = 5000, keep.tedgelist = TRUE)

# Extract timed-edgelist
te <- as.data.frame(dx)
head(te)

# Limit to non-censored edges
te <- te[which(te$onset.censored == FALSE & te$terminus.censored == FALSE),
        c("head", "tail", "duration")]
head(te)

# Look up the age group of head and tail nodes
te$ag.head <- age.grp[te$head]
te$ag.tail <- age.grp[te$tail]
head(te)

# Recover average edge durations for age-group pairing
mean(te$duration[te$ag.head != te$ag.tail])
mean(te$duration[te$ag.head == 1 & te$ag.tail == 1])
mean(te$duration[te$ag.head == 2 & te$ag.tail == 2])
mean(te$duration[te$ag.head == 3 & te$ag.tail == 3])
mean(te$duration[te$ag.head == 4 & te$ag.tail == 4])
mean(te$duration[te$ag.head == 5 & te$ag.tail == 5])
durs

## End(Not run)

```

edgelist_censor

Table of Edge Censoring

Description

Outputs a table of the number and percent of edges that are left-censored, right-censored, both-censored, or uncensored for a `networkDynamic` object.

Usage

```
edgelist_censor(el)
```

Arguments

e1 A timed edgelist with start and end times extracted from a `networkDynamic` object using the `as.data.frame.networkDynamic` function.

Details

Given a STERGM simulation over a specified number of time steps, the edges within that simulation may be left-censored (started before the first step), right-censored (continued after the last step), right and left-censored, or uncensored. The amount of censoring will increase when the average edge duration approaches the length of the simulation.

Value

A 4 x 2 table containing the number and percent of edges in `e1` that are left-censored, right-censored, both-censored, or uncensored.

Examples

```
# Initialize and parameterize network model
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)

# Model estimation
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Simulate the network and extract a timed edgelist
dx <- netdx(est, nsims = 1, nsteps = 100, keep.tedgelist = TRUE,
  verbose = FALSE)
e1 <- as.data.frame(dx)

# Calculate censoring
edgelist_censor(e1)
```

Description

Launches interactive Shiny applications for exploring deterministic compartmental models (DCMs) and stochastic individual contact models (ICMs) of infectious disease transmission.

Usage

```
epiweb(class, ...)
```

Arguments

<code>class</code>	Model class, with options of "dcm" and "icm".
<code>...</code>	Additional arguments passed to shiny::runApp , such as port, host, or <code>launch.browser</code> .

Details

epiweb launches a web-based graphical interface for configuring, running, and analyzing epidemic models built on EpiModel. Each application provides interactive controls for model parameters, real-time (or on-demand) visualization with plotly, narrative model summaries, and downloadable data tables. Both applications include a **Guide** tab with a comprehensive user manual covering the underlying theory, parameter definitions, and instructions for interpreting output.

DCM Application (`class = "dcm"`)

The DCM app solves systems of ordinary differential equations for one-group SI, SIR, and SIS models. Models run instantly and the interface is fully reactive (no run button). Features include:

- **Scenario presets:** Flu-like (SIR), STI-like (SIS), and Measles-like (SIR) configurations that set all parameters to epidemiologically plausible values.
- **Interventions:** An optional mid-epidemic intervention that reduces the transmission probability by a user-specified efficacy, shown as a vertical dashed line on the plot.
- **Sensitivity analysis:** Vary any epidemic parameter across a range of values to compare trajectories side by side with a blue-to-red color ramp.
- **Vital dynamics:** Optional births and deaths to model endemic equilibria over longer time horizons.
- **Summary tab:** Displays the basic reproduction number (R_0), peak timing, cumulative infections, attack rate, and contextual interpretation.
- **Data tab:** Searchable table of model output with CSV download.

ICM Application (`class = "icm"`)

The ICM app runs agent-based stochastic microsimulations for one-group SI, SIR, and SIS models. Because ICMs are computationally more expensive, a **Run Model** button controls when simulations execute. Features include:

- **Stochastic visualization:** Multiple simulations are summarized with mean trajectory lines and interquartile range (IQR) ribbons showing the middle 50\ individual trajectories.
- **Scenario presets:** The same Flu-like, STI-like, and Measles-like configurations as the DCM app, adapted for smaller populations suitable for agent-based simulation.
- **Interventions:** Same intervention system as DCM, natively supported by [param.icm](#).
- **Vital dynamics:** Optional births and deaths, as in DCM.
- **Summary tab:** Displays R_0 , epidemic timeline from the mean trajectory, and a stochastic variation section showing the range and standard deviation of peak infections and final prevalence across simulations.
- **Data tab:** View simulation output as means, standard deviations, or individual simulation values, with CSV download.

Required Packages

Both applications require the shiny, bslib, DT, and plotly packages. Install any missing packages with `install.packages(c("shiny", "bslib", "DT", "plotly"))`.

See Also

[dcm](#) and [param.dcm](#) for the DCM modeling API; [icm](#) and [param.icm](#) for the ICM modeling API.

Examples

```
## Not run:
## Deterministic compartmental models
epiweb(class = "dcm")

## Stochastic individual contact models
epiweb(class = "icm")

## End(Not run)
```

generate_random_params

Generate Values for Random Parameters

Description

This function uses the generative functions in the `random.params` list to create values for the parameters.

Usage

```
generate_random_params(param, verbose = FALSE)
```

Arguments

<code>param</code>	The param argument received by the <code>netsim</code> functions.
<code>verbose</code>	Should the function output the generated values (default = FALSE)?

Value

A fully instantiated param list.

random.params

The `random.params` argument to the [param.net\(\)](#) function must be a named list of functions that each return a value that can be used as the argument with the same name. In the example below, `param_random` is a function factory provided by `EpiModel` for `act.rate` and for `tx.halt.part.prob` we provide bespoke functions. A function factory is a function that returns a new function (see <https://adv-r.hadley.nz/function-factories.html>).

Generator Functions

The functions used inside `random_params` must be 0 argument functions returning a valid value for the parameter with the same name.

param_random_set

The `random_params` list can optionally contain a `param_random_set` element. It must be a `data.frame` of possible values to be used as parameters.

The column names must correspond either to: the name of one parameter, if this parameter is of size 1; or the name of one parameter with `"_1"`, `"_2"`, etc. *appended, with the number representing the position of the value, if this parameter is of size > 1. This means that the parameter names cannot contain any underscores ""* if you intend to use `param_random_set`.

The point of the `param.random.set` `data.frame` is to allow the random parameters to be correlated. To achieve this, a whole row of the `data.frame` is selected for each simulation.

Examples

```
## Not run:

## Example with only the generator function

# Define random parameter list
my_randoms <- list(
  act.rate = param_random(c(0.25, 0.5, 0.75)),
  tx.prob = function() rbeta(1, 1, 2),
  stratified.test.rate = function() c(
    rnorm(1, 0.05, 0.01),
    rnorm(1, 0.15, 0.03),
    rnorm(1, 0.25, 0.05)
  )
)

# Parameter model with fixed and random parameters
param <- param.net(inf.prob = 0.3, random.params = my_randoms)

# Below, `tx.prob` is set first to 0.3 then assigned a random value using
# the function from `my_randoms`. A warning notifying of this overwrite is
# therefore produced.
param <- param.net(tx.prob = 0.3, random.params = my_randoms)

# Parameters are drawn automatically in netsim by calling the function
# within netsim_loop. Demonstrating draws here but this is not used by
# end user.
paramDraw <- generate_random_params(param, verbose = TRUE)
paramDraw

## Addition of the `param.random.set` `data.frame`

# This function will generate sets of correlated parameters
```

```

generate_correlated_params <- function() {
  param.unique <- runif(1)
  param.set.1 <- param.unique + runif(2)
  param.set.2 <- param.unique * rnorm(3)

  return(list(param.unique, param.set.1, param.set.2))
}

# Data.frame set of random parameters :
correlated_params <- t(replicate(10, unlist(generate_correlated_params()))))
correlated_params <- as.data.frame(correlated_params)
colnames(correlated_params) <- c(
  "param.unique",
  "param.set.1_1", "param.set.1_2",
  "param.set.2_1", "param.set.2_2", "param.set.2_3"
)

# Define random parameter list with the `param.random.set` element
my_randoms <- list(
  act.rate = param_random(c(0.25, 0.5, 0.75)),
  param.random.set = correlated_params
)

# Parameter model with fixed and random parameters
param <- param.net(inf.prob = 0.3, random.params = my_randoms)

# Parameters are drawn automatically in netsim by calling the function
# within netsim_loop. Demonstrating draws here but this is not used by
# end user.
paramDraw <- generate_random_params(param, verbose = TRUE)
paramDraw

## End(Not run)

```

geom_bands

ggplot2 *Geom for Quantile Bands*

Description

Plots quantile bands given a data.frame with stochastic model results from `icm()` or `netsim()`.

Usage

```
geom_bands(mapping, lower = 0.25, upper = 0.75, alpha = 0.25, ...)
```

Arguments

mapping	Standard aesthetic mapping <code>aes()</code> input for ggplot2.
lower	Lower quantile for the time series.

upper	Upper quantile for the time series.
alpha	Transparency of the ribbon fill.
...	Additional arguments passed to stat_summary.

Details

This is a wrapper around `ggplot2::stat_summary` with a ribbon geom as aesthetic output.

Examples

```
param <- param.icm(inf.prob = 0.2, act.rate = 0.25)
init <- init.icm(s.num = 500, i.num = 1)
control <- control.icm(type = "SI", nsteps = 250, nsims = 5)
mod1 <- icm(param, init, control)
df <- as.data.frame(mod1)
df.mean <- as.data.frame(mod1, out = "mean")

library(ggplot2)
ggplot() +
  geom_line(data = df, mapping = aes(time, i.num, group = sim),
    alpha = 0.25, lwd = 0.25, color = "firebrick") +
  geom_bands(data = df, mapping = aes(time, i.num),
    lower = 0.1, upper = 0.9, fill = "firebrick") +
  geom_line(data = df.mean, mapping = aes(time, i.num)) +
  theme_minimal()
```

get_adj_list

Returns an adjacency list from an edge list

Description

Returns an adjacency list from an edge list

Usage

```
get_adj_list(el, n_nodes)
```

Arguments

el	An edge list as a data.frame with columns head and tail
n_nodes	The size number of node in the network

Details

The adjacency list is a list of length `n_nodes`. The entry for each node is a integer vector containing the index of all the nodes connected to it. This layout makes it directly subetable in $O(1)$ at the expanse of memory usage. To get all connections to the nodes 10 and 15: `unlist(adj_list[c(10, 15)])`

Value

An adjacency list for the network

get_attr_history	<i>Extract the Attributes History from Network Simulations</i>
------------------	--

Description

Extract the Attributes History from Network Simulations

Usage

```
get_attr_history(sims)
```

Arguments

sims An EpiModel object of class netsim.

Value

A list of data.frames, one for each "measure" recorded in the simulation by the record_attr_history function.

Examples

```
## Not run:

# With `sims` the result of a `netsim` call
get_attr_history(sims)

## End(Not run)
```

get_connected_nodes	<i>Returns all the node connected directly or indirectly to a set of nodes</i>
---------------------	--

Description

Returns all the node connected directly or indirectly to a set of nodes

Usage

```
get_connected_nodes(adj_list, nodes)
```

Arguments

adj_list	The network represented as an adjacency list
nodes	A set of nodes

Value

A vector of nodes indexes that are connected together with the ones provided in the nodes argument. The nodes themselves are not listed in this output

get_cumulative_degree *Return the Cumulative Degree of a Set of Index Nodes*

Description

Counts the number of distinct partners each of a set of index nodes has accumulated over the tracked history. A thin wrapper around `get_partners()` that collapses the partner list to a count per index.

Usage

```
get_cumulative_degree(
  dat,
  index_posit_ids,
  networks = NULL,
  truncate = Inf,
  only.active.nodes = FALSE
)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
index_posit_ids	The positional IDs of the indexes of interest.
networks	Numerical indexes of the networks to extract the partnerships from. (May be > 1 for models with multi-layer networks.) If NULL, extract from all networks.
truncate	After how many time steps a partnership that is no longer active should be removed from the output. See the Truncation section.
only.active.nodes	If TRUE, then inactive (e.g., deceased) partners will be removed from the output.

Value

A data.frame with 2 columns:

- index_pid: the positional ID (see `get_posit_ids()`) of the indexes.
- degree: the cumulative degree of the index.

Cumulative Degree

The cumulative degree of a node is the number of distinct partners connected to it during the tracked time window. The window is whatever history the cumulative edgelist currently contains (controlled at simulation time by `control$truncate.el.cuml` and at call time by the `truncate` argument).

See Also

[get_partners\(\)](#) for the full partner list backing this count.

Other cumulative_edgelist: [as_cumulative_edgelist\(\)](#), [dedup_cumulative_edgelist\(\)](#), [get_cumulative_edgelist\(\)](#), [get_cumulative_edgelist_df\(\)](#), [get_partners\(\)](#), [reachable-nodes](#), [update_cumulative_edgelist\(\)](#)

Examples

```
## Not run:
get_cumulative_degree(dat, index_posit_ids = 1:50, truncate = 52)

## End(Not run)
```

get_cumulative_edgelist

Get a Cumulative Edgelist From a Specified Network

Description

Returns the cumulative edgelist for a given network: a record of every edge EpiModel has tracked, with the time steps at which each edge formed and (if no longer active) dissolved. This is the canonical mechanism for querying partnership histories during or after a simulation, and is particularly important under `tergmLite = TRUE`, where the full `networkDynamic` history is not retained.

Usage

```
get_cumulative_edgelist(dat, network)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from netsim() .
<code>network</code>	Numerical index of the network from which the cumulative edgelist should be extracted. (May be > 1 for models with multiple overlapping networks.)

Details

Cumulative-edgelist tracking is opt-in. It must be enabled via `control.net(cumulative.edgelist = TRUE)` (see [control.net\(\)](#)); calling this function on a simulation that did not enable tracking raises an error.

Inside a custom module (where `dat` is the live `netsim_dat` object), call this function directly. After a `netsim()` run, the saved edgelists are attached to the returned object as `sim$cumulative.edgelist[[s]]` (one element per simulation, populated when `save.cumulative.edgelist = TRUE`); read those directly rather than calling this function on the processed output.

For the *current* (active-only, no history) edgelist, see [get_edgelist\(\)](#) and [get_edgelists_df\(\)](#).

Value

A tibble with four columns:

- `head`: the unique ID (see [get_unique_ids\(\)](#)) of the head node.
- `tail`: the unique ID of the tail node.
- `start`: the time step at which the edge formed.
- `stop`: the time step at which the edge dissolved, or NA if the edge is still active. Edges are active over `[start, stop]` inclusive.

See Also

[control.net\(\)](#) for the controlling flags (`cumulative.edgelist`, `truncate.el.cuml`, `save.cumulative.edgelist`). [vignette\("network-objects", package = "EpiModel"\)](#) walks through the full lifecycle.

Other `cumulative_edgelist`: [as_cumulative_edgelist\(\)](#), [dedup_cumulative_edgelist\(\)](#), [get_cumulative_degree\(\)](#), [get_cumulative_edgelists_df\(\)](#), [get_partners\(\)](#), [reachable-nodes](#), [update_cumulative_edgelist\(\)](#)

Examples

```
## Not run:
# Inside a custom module (dat is the live netsim_dat object):
el <- get_cumulative_edgelist(dat, network = 1)

# Post-simulation on a processed netsim object, read the saved slot:
sim$cumulative.edgelist[[1]]

## End(Not run)
```

Description

Combines the cumulative edgelists from one or more network layers into a single `data.frame`, adding a `network` column to identify the layer. Like `get_cumulative_edgelist()`, this requires `control.net(cumulative.edgelist = TRUE)`.

Usage

```
get_cumulative_edgelists_df(dat, networks = NULL)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
<code>networks</code>	Numerical indexes of the networks to extract the partnerships from. (May be > 1 for models with multiple overlapping networks.) If <code>NULL</code> , extract from all networks.

Value

A `data.frame` with 5 columns:

- `head`: the unique ID (see `get_unique_ids()`) of the head node.
- `tail`: the unique ID of the tail node.
- `start`: the time step at which the edge formed.
- `stop`: the time step at which the edge dissolved, or `NA` if still active.
- `network`: the numerical index of the network on which the partnership lives.

Note: column names `head/tail` here match the single-network `get_cumulative_edgelist()` output. The `index/partner` naming used by `get_partners()` is reserved for queries about specific index nodes.

See Also

`get_cumulative_edgelist()` (single network), `control.net()` (`cumulative.edgelist`, `save.cumulative.edgelist`). `vignette("network-objects", package = "EpiModel")` walks through the full lifecycle.

Other `cumulative_edgelist`: `as_cumulative_edgelist()`, `dedup_cumulative_edgelist()`, `get_cumulative_degree()`, `get_cumulative_edgelist()`, `get_partners()`, `reachable-nodes`, `update_cumulative_edgelist()`

get_current_timestep *Return the Current Timestep*

Description

Return the Current Timestep

Usage

```
get_current_timestep(dat)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
-----	--

Value

The current timestep.

get_degree *Get Individual Degree from Network or Edgelist*

Description

A fast method for querying the current degree of all individuals within a network.

Usage

```
get_degree(x)
```

Arguments

x	Either an object of class network or edgelist generated from a network. If x is an edgelist, then it must contain an attribute for the total network size, n.
---	---

Details

Individual-level data on the current degree of nodes within a network is often useful for summary statistics. Given a network class object, net, one way to look up the current degree is to get a summary of the ERGM term, sociality, as in: `summary(net ~ sociality(nodes = NULL))`. But that is computationally inefficient for a number of reasons. This function provides a fast method for generating the vector of degrees using a query of the edgelist. It is even faster if the parameter x is already transformed into an edgelist.

Value

A vector of length equal to the total network size, containing the current degree of each node in the network.

Examples

```
nw <- network_initialize(n = 500)

set.seed(1)
fit <- ergm(nw ~ edges, target.stats = 250)
sim <- simulate(fit)

# Slow ERGM-based method
ergm.method <- unname(summary(sim ~ sociality(nodes = NULL)))
ergm.method

# Fast tabulate method with network object
deg.net <- get_degree(sim)
deg.net

# Even faster if network already transformed into an edgelist
el <- as.edgelist(sim)
deg.el <- get_degree(el)
deg.el

identical(as.integer(ergm.method), deg.net, deg.el)
```

get_degree_dist

Extract Degree Distributions from Network Diagnostics

Description

Calculates the cumulative or momentary degree distribution from the timed edgelist of a dynamic [netdx\(\)](#) simulation.

Usage

```
get_degree_dist(
  x,
  type = c("cumulative", "momentary"),
  sims = NULL,
  window = NULL,
  unique.partners = TRUE
)
```

Arguments

<code>x</code>	An <code>EpiModel</code> object of class <code>netdx</code> , run with <code>dynamic = TRUE</code> and <code>keep.tedgelist = TRUE</code> .
<code>type</code>	Distribution to calculate, either "cumulative" for the number of partners each node accumulates over the observation window, or "momentary" for the number of partners each node has at a single point in time, averaged over the time steps in the window.
<code>sims</code>	A vector of simulation numbers to include. The default is all simulations in <code>x</code> .
<code>window</code>	A vector of length 2 giving the first and last time step of the observation window, on the time scale of the timed edgelist (which starts at 0). The default is the full simulation, <code>c(0, x\$ntsteps)</code> .
<code>unique.partners</code>	If <code>TRUE</code> , a dyad that forms, dissolves, and then re-forms within the window contributes one partner to the cumulative degree of each node; if <code>FALSE</code> , it contributes one per partnership. Only used when <code>type = "cumulative"</code> .

Details

The momentary degree distribution is the distribution plotted by `plot.netdx()` when `degree(k)` terms are included in the `nwstats.formula` of `netdx()`: it counts the partners a node has at one moment in time. The cumulative degree distribution counts the partners a node accumulates across the whole observation window, so a node with one partner at a time and a node with several simultaneous partners may have the same cumulative degree.

Partnerships are counted if they are active at any point in the window: a partnership contributes to the cumulative degree of both of its nodes when its spell overlaps window, and to the momentary degree of both nodes at each time step of the window in which it is active. Partnerships that are ongoing when the simulation ends are counted for the steps they are observed.

Because the cumulative degree grows with the length of the observation window, window should be set to a substantively meaningful period (for example, one year of time steps) when comparing across models or against data.

Value

A data frame with one row per simulation and degree value, and the columns:

- `sim`: the simulation number.
- `degree`: the degree value, from 0 to the maximum observed across the selected simulations.
- `count`: the number of nodes with that degree. For `type = "momentary"`, this is the mean number of nodes per time step, and so need not be a whole number.
- `prop`: count divided by the network size.

See Also

`plot.netdx()` with `type = "cumldeg"` to plot these distributions, and `netdx()` to generate the input object.

Examples

```
nw <- network_initialize(n = 100)
est <- netest(nw, formation = ~edges, target.stats = 50,
             coef.diss = dissolution_coefs(~offset(edges), duration = 10),
             verbose = FALSE)
dx <- netdx(est, nsims = 2, nsteps = 50, keep.tedgelist = TRUE,
           verbose = FALSE)

# Cumulative degree over the full simulation
cd <- get_degree_dist(dx, type = "cumulative")
head(cd)

# Mean cumulative degree per simulation
tapply(cd$degree * cd$prop, cd$sim, sum)

# Momentary degree over the last 25 steps
get_degree_dist(dx, type = "momentary", window = c(25, 50))
```

get_edgelist

Get an Edgelist From the Specified Network

Description

This function outputs an edgelist from the specified network, selecting the method depending on the stored network type.

Usage

```
get_edgelist(dat, network)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
network	Numerical index of the network from which the edgelist should be extracted. (May be > 1 for models with multiple overlapping networks.)

Value

An edgelist in matrix form with two columns. Each column contains the `posit_ids` (see `get_posit_ids`) of the nodes in each edge.

get_edgelists_df	<i>Get the Edgelist(s) from the Specified Network(s)</i>
------------------	--

Description

Get the Edgelist(s) from the Specified Network(s)

Usage

```
get_edgelists_df(dat, networks = NULL)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
networks	Numerical indexes of the networks to extract the partnerships from. (May be > 1 for models with multiple overlapping networks.) If NULL, extract from all networks.

Value

A data.frame with the following columns:

- head: Positional ID of the head node.
- tail: Positional ID of the tail node.
- network: The numerical index of the network on which the edge is located.

get_formula_term_attr	<i>Output ERGM Formula Attributes into a Character Vector</i>
-----------------------	---

Description

Given a formation formula for a network model, outputs a character vector of vertex attributes to be used in `netsim()` simulations.

Usage

```
get_formula_term_attr(form, nw)
```

Arguments

form	An ERGM model formula.
nw	A network object.

Value

A character vector of vertex attributes.

get_network

*Extract Network Objects from Network Simulations***Description**

Extracts the network object from either a network epidemic model object generated with `netsim`, a network diagnostic simulation generated with `netdx`, or a `netsim_dat` object used internally in `netsim`. For `netdx` or `netsim` with `tergmLite == FALSE`, the extracted network object is a `networkDynamic`, which can be collapsed down to a static network object with the `collapse` and `at` arguments. For `netsim` with `tergmLite == TRUE`, the extracted network object is the final `networkLite`, the `collapse` argument should be `FALSE`, and the `at` argument should be missing. For `netsim_dat`, the `collapse` and `at` arguments are not supported, and the network object is either the current `networkLite` (if `tergmLite == TRUE`) or the current `networkDynamic` (if `tergmLite == FALSE`).

Usage

```
get_network(x, ...)

## S3 method for class 'netdx'
get_network(x, sim = 1, collapse = FALSE, at = NULL, ...)

## S3 method for class 'netsim'
get_network(x, sim = 1, network = 1, collapse = FALSE, at = NULL, ...)

## S3 method for class 'netsim_dat'
get_network(x, network = 1L, ...)
```

Arguments

<code>x</code>	An EpiModel object of class <code>netsim()</code> , <code>netdx()</code> , or <code>netsim_dat</code> .
<code>...</code>	Additional arguments.
<code>sim</code>	Simulation number of extracted network, for <code>netdx</code> and <code>netsim</code> .
<code>collapse</code>	If <code>TRUE</code> , collapse the <code>networkDynamic</code> object to a static network object at a specified time step. Applicable to <code>netdx</code> objects and <code>netsim</code> objects with <code>tergmLite == FALSE</code> .
<code>at</code>	If <code>collapse</code> is <code>TRUE</code> , the time step at which the extracted network should be collapsed. Applicable to <code>netdx</code> objects and <code>netsim</code> objects with <code>tergmLite == FALSE</code> .
<code>network</code>	Network number, for <code>netsim</code> or <code>netsim_dat</code> objects with multiple overlapping networks (advanced use, and not applicable to <code>netdx</code> objects).

Details

This function requires that the network object is saved during the network simulation while running either `netsim()` or `netdx()`. For the former, that is specified by setting the `save.network` parameter in `control.net()` to `TRUE`. For the latter, that is specified with the `keep.tnetwork` parameter directly in `netdx()`.

Value

For `netdx` or `netsim` with `tergmLite == FALSE`, a `networkDynamic` object (if `collapse = FALSE`) or a static network object (if `collapse = TRUE`). For `netsim` with `tergmLite == TRUE` or `netsim_dat` with `tergmLite == TRUE`, a `networkLite` object. For `netsim_dat` with `tergmLite == FALSE`, a `networkDynamic` object.

Examples

```
# Set up network and TERGM formula
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)

# Estimate the model
est <- netest(nw, formation, target.stats, coef.diss)

# Run diagnostics, saving the networkDynamic objects
dx <- netdx(est, nsteps = 10, nsims = 3, keep.tnetwork = TRUE,
            verbose = FALSE)

# Extract the network for simulation 2 from dx object
get_network(dx, sim = 2)

# Extract and collapse the network from simulation 1 at time step 5
get_network(dx, collapse = TRUE, at = 5)

# Parameterize the epidemic model, and simulate it
param <- param.net(inf.prob = 0.3, inf.prob.g2 = 0.15)
init <- init.net(i.num = 10, i.num.g2 = 10)
control <- control.net(type = "SI", nsteps = 10, nsims = 3, verbose = FALSE)
mod <- netsim(est, param, init, control)

# Extract the network for simulation 2 from mod object
get_network(mod, sim = 2)

## Extract and collapse the network from simulation 1 at time step 5
get_network(mod, collapse = TRUE, at = 5)
```

```
get_network_attributes
```

Get Network Attributes from a Network Object

Description

Gets all network attributes except "mnext" from its network argument.

Usage

```
get_network_attributes(x)
```

Arguments

x An object of class network or networkLite.

Details

This function is used in EpiModel workflows to copy relevant network attributes from the network object to the netsim_dat object when initializing netsim runs.

Value

Returns the named list of network attributes.

Examples

```
nw <- network_initialize(100)
get_network_attributes(nw)
```

```
get_network_term_attr    Output Network Attributes into a Character Vector
```

Description

Given a simulated network, outputs a character vector of vertex attributes to be used in `netsim()` simulations.

Usage

```
get_network_term_attr(nw)
```

Arguments

nw A network object.

Value

A character vector of vertex attributes.

get_nwstats

Extract Network Statistics from netsim or netdx Object

Description

Extracts network statistics from a network epidemic model simulated with `netsim` or a network diagnostics object simulated with `netdx`. Statistics can be returned either as a single data frame or as a list of matrices (one matrix for each simulation).

Usage

```
get_nwstats(x, sim = NULL, network = 1, mode = c("data.frame", "list"))
```

Arguments

<code>x</code>	An EpiModel object of class <code>netsim()</code> or <code>netdx()</code> .
<code>sim</code>	A vector of simulation numbers from the extracted object. (Default = NULL, all simulations are included)
<code>network</code>	Network number, for <code>netsim</code> objects with multiple overlapping networks (advanced use, and not applicable to <code>netdx</code> objects).
<code>mode</code>	Either "data.frame" or "list", indicating the desired output.

Value

A data frame or list of matrices containing the network statistics.

Examples

```
# Two-group Bernoulli random graph TERGM
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

dx <- netdx(est, nsim = 3, nsteps = 10, verbose = FALSE,
            nwstats.formula = ~edges + isolates)
get_nwstats(dx)
get_nwstats(dx, sim = 1)

# SI epidemic model
param <- param.net(inf.prob = 0.3, inf.prob.g2 = 0.15)
init <- init.net(i.num = 10, i.num.g2 = 10)
```

```

control <- control.net(type = "SI", nsteps = 10, nsims = 3,
                      nwstats.formula = ~edges + meandeg + degree(0:5),
                      verbose = FALSE)
mod <- netsim(est, param, init, control)

# Extract the network statistics from all or sets of simulations
get_nwstats(mod)
get_nwstats(mod, sim = 2)
get_nwstats(mod, sim = c(1, 3))

# On the fly summary stats
summary(get_nwstats(mod))
colMeans(get_nwstats(mod))

```

get_param_set

Extract the Parameter Set from Network Simulations

Description

Extract the Parameter Set from Network Simulations

Usage

```
get_param_set(sims)
```

Arguments

sims An EpiModel object of class netsim.

Value

A data.frame with one row per simulation and one column per parameter or parameter element where the parameters are of size > 1.

Output Format

The outputted data.frame has one row per simulation and the columns correspond to the parameters used in this simulation.

The column name will match the parameter name if it is a size 1 parameter or if the parameter is of size > 1, there will be N columns (with N being the size of the parameter) named parameter.name_1, parameter.name_2, ..., parameter.name_N.

Examples

```

# Setup network
nw <- network_initialize(n = 50)

est <- netest(
  nw, formation = ~edges,
  target.stats = c(25),
  coef.diss = dissolution_coefs(~offset(edges), 10, 0),
  verbose = FALSE
)

init <- init.net(i.num = 10)

n <- 5

related.param <- data.frame(
  dummy.param = rbeta(n, 1, 2)
)

my.randoms <- list(
  act.rate = param_random(c(0.25, 0.5, 0.75)),
  dummy.param = function() rbeta(1, 1, 2),
  dummy.strat.param = function() c(
    rnorm(1, 0, 10),
    rnorm(1, 10, 1)
  )
)

param <- param.net(
  inf.prob = 0.3,
  dummy = c(0, 1, 2),
  random.params = my.randoms
)

control <- control.net(type = "SI", nsims = 3, nsteps = 5, verbose = FALSE)
mod <- netsim(est, param, init, control)

get_param_set(mod)

```

get_partners

*Return the Historical Contacts (Partners) of a Set of Index Nodes***Description**

Pulls every cumulative-edgelist row touching one of the supplied "index" nodes, returning each (index, partner) pair together with the partnership start/stop times and the network layer. This is the building block for contact tracing over the simulated network history.

Usage

```
get_partners(
  dat,
  index_posit_ids,
  networks = NULL,
  truncate = Inf,
  only.active.nodes = FALSE
)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
<code>index_posit_ids</code>	The positional IDs of the indexes of interest.
<code>networks</code>	Numerical indexes of the networks to extract the partnerships from. (May be > 1 for models with multi-layer networks.) If <code>NULL</code> , extract from all networks.
<code>truncate</code>	After how many time steps a partnership that is no longer active should be removed from the output. See the Truncation section.
<code>only.active.nodes</code>	If <code>TRUE</code> , then inactive (e.g., deceased) partners will be removed from the output.

Details

Indexes are passed as positional IDs but the output uses unique IDs, because partners may include nodes that have already departed (and thus no longer have a positional ID). Use `get_unique_ids()` and `get_posit_ids()` to convert between the two systems.

The `truncate` argument here filters by edge age: only edges whose stop step is within `truncate` time steps of the current step are kept (active edges are always included). It operates on whatever history is already stored in the cumulative edgelist; it cannot recover edges that were dropped earlier by `update_cumulative_edgelist()` or by `control$truncate.el.cuml`.

Value

A `data.frame` with 5 columns:

- `index`: the unique IDs of the indexes.
- `partner`: the unique IDs of the partners/contacts.
- `start`: the time step at which the edge started.
- `stop`: the time step in which the edge stopped; if ongoing, then `NA` is returned.
- `network`: the numerical index for the network on which the partnership/contact is located.

See Also

`get_cumulative_degree()` for a partner count per index; `get_cumulative_edgelist()` / `get_cumulative_edgelist_df()` for the underlying edgelist; `get_unique_ids()` / `get_posit_ids()` for ID conversion.

Other cumulative_edgelist: `as_cumulative_edgelist()`, `dedup_cumulative_edgelist()`, `get_cumulative_degree()`, `get_cumulative_edgelist()`, `get_cumulative_edgelist_df()`, `reachable-nodes`, `update_cumulative_edgelist()`.

Examples

```
## Not run:
# Contacts of nodes 1..10 across all networks within the last 30 steps,
# excluding partners who have since departed:
get_partners(dat,
             index_posit_ids = 1:10,
             truncate = 30,
             only.active.nodes = TRUE)

## End(Not run)
```

get_sims

*Extract Network Simulations***Description**

Subsets the entire netsim object to a subset of simulations, essentially functioning like a reverse of merge.

Usage

```
get_sims(x, sims = NULL, var = NULL)
```

Arguments

x	An object of class netsim.
sims	Either a numeric vector of simulation numbers to retain in the output object, or "mean", which selects the one simulation with the value of the variable specified in var closest to the mean of var across all simulations at the final step.
var	A character vector of variables to retain from x if sims is a numeric vector, or a single variable name for selecting the average simulation from the set if sims = "mean".

Value

An updated object of class netsim containing only the simulations specified in sims and the variables specified in var.

Examples

```
# Network model estimation
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est1 <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)
```

```

# Epidemic model
param <- param.net(inf.prob = 0.3)
init <- init.net(i.num = 10)
control <- control.net(type = "SI", nsteps = 10, nsims = 3, verbose.int = 0)
mod1 <- netsim(est1, param, init, control)

# Get sim 2
s.g2 <- get_sims(mod1, sims = 2)

# Get sims 2 and 3 and keep only a subset of variables
s.g2.small <- get_sims(mod1, sims = 2:3, var = c("i.num", "si.flow"))

# Extract the mean simulation for the variable i.num
sim.mean <- get_sims(mod1, sims = "mean", var = "i.num")

```

get_subnet_adj_list *Return an adjacency list of subnets*

Description

Return an adjacency list of subnets

Usage

```
get_subnet_adj_list(adj_list)
```

Arguments

adj_list A normal adjacency list

Details

A graph with 4 components: 1, 2, 3, 4, and 5 and 6, 7, 8 would yield a list like so: 1: 2, 3, 4 2: 1 3: 1 4: 1 5: numeric(0) 6: 7, 8 7: 6, 8: 6

This format speeds up the construction of reachable sets on dense networks

Value

An adjacency list where only the first node of a subnet contains the subnet and all other contain only the first node

get_vertex_attribute *Get Vertex Attribute on Network Object*

Description

Gets a vertex attribute from an object of class network. This function simplifies the related function in the network package.

Usage

```
get_vertex_attribute(x, attrname)
```

Arguments

x	An object of class network.
attrname	The name of the attribute to get.

Details

This function is used in EpiModel workflows to query vertex attributes on an initialized empty network object (see [network_initialize\(\)](#)).

Value

Returns a vector of vertex attribute values for the attribute specified by attrname.

Examples

```
nw <- network_initialize(100)
nw <- set_vertex_attribute(nw, "age", runif(100, 15, 65))
get_vertex_attribute(nw, "age")
```

icm

Stochastic Individual Contact Models

Description

Simulates stochastic individual contact epidemic models for infectious disease.

Usage

```
icm(param, init, control)
```

Arguments

<code>param</code>	Model parameters, as an object of class <code>param.icm()</code> .
<code>init</code>	Initial conditions, as an object of class <code>init.icm()</code> .
<code>control</code>	Control settings, as an object of class <code>control.icm()</code> .

Details

Individual contact models are intended to be the stochastic microsimulation analogs to deterministic compartmental models. ICMs simulate disease spread on individual agents in discrete time as a function of processes with stochastic variation. The stochasticity is inherent in all transition processes: infection, recovery, and demographics.

The `icm` function supports the built-in SI, SIR, and SIS disease types only, in one-group or two-group configurations, with or without demography. For custom or extension epidemic models, use the network model class via `netsim()` with `control.net(type = NULL, ...)` and user-supplied module functions.

Value

A list of class `icm` with the following elements:

- **param:** the epidemic parameters passed into the model through `param`, with additional parameters added as necessary.
- **control:** the control settings passed into the model through `control`, with additional controls added as necessary.
- **epi:** a list of data frames, one for each epidemiological output from the model. Outputs always include the size of each compartment, as well as flows in, out of, and between compartments.

See Also

Extract the model results with `as.data.frame.icm()`. Summarize the time-specific model results with `summary.icm()`. Plot the model results with `plot.icm()`. Plot a compartment flow diagram with `comp_plot()`.

Examples

```
## Not run:
## Example 1: SI Model
param <- param.icm(inf.prob = 0.2, act.rate = 0.25)
init <- init.icm(s.num = 500, i.num = 1)
control <- control.icm(type = "SI", nsteps = 500, nsims = 10)
mod1 <- icm(param, init, control)
mod1
plot(mod1)

## Example 2: SIR Model
param <- param.icm(inf.prob = 0.2, act.rate = 0.25, rec.rate = 1/50)
init <- init.icm(s.num = 500, i.num = 1, r.num = 0)
control <- control.icm(type = "SIR", nsteps = 500, nsims = 10)
mod2 <- icm(param, init, control)
```

```

mod2
plot(mod2)

## Example 3: SIS Model
param <- param.icm(inf.prob = 0.2, act.rate = 0.25, rec.rate = 1/50)
init <- init.icm(s.num = 500, i.num = 1)
control <- control.icm(type = "SIS", nsteps = 500, nsims = 10)
mod3 <- icm(param, init, control)
mod3
plot(mod3)

## Example 4: SI Model with Vital Dynamics (Two-Group)
param <- param.icm(inf.prob = 0.4, inf.prob.g2 = 0.1,
                  act.rate = 0.25, balance = "g1",
                  a.rate = 1/100, a.rate.g2 = NA,
                  ds.rate = 1/100, ds.rate.g2 = 1/100,
                  di.rate = 1/50, di.rate.g2 = 1/50)
init <- init.icm(s.num = 500, i.num = 1,
                s.num.g2 = 500, i.num.g2 = 0)
control <- control.icm(type = "SI", nsteps = 500, nsims = 10)
mod4 <- icm(param, init, control)
mod4
plot(mod4)

## End(Not run)

```

increment_timestep	<i>Increment the Current Timestep</i>
--------------------	---------------------------------------

Description

This function adds 1 to the timestep counter stored in the `netsim_dat` main list object.

Usage

```
increment_timestep(dat)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
------------------	--

Value

The updated `netsim_dat` main list object.

Mutability

This DOES NOT modify the `netsim_dat` object in place. The result must be assigned back to `dat` in order to be registered: `dat <- increment_timestep(dat)`.

init.dcm

*Initial Conditions for Deterministic Compartmental Models***Description**

Sets the initial conditions for deterministic compartmental models simulated with dcm.

Usage

```
init.dcm(s.num, i.num, r.num, s.num.g2, i.num.g2, r.num.g2, ...)
```

Arguments

s.num	Number of initial susceptible persons. For two-group models, this is the number of initial group 1 susceptible persons.
i.num	Number of initial infected persons. For two-group models, this is the number of initial group 1 infected persons.
r.num	Number of initial recovered persons. For two-group models, this is the number of initial group 1 recovered persons. This parameter is only used for the SIR model type.
s.num.g2	Number of initial susceptible persons in group 2. This parameter is only used for two-group models.
i.num.g2	Number of initial infected persons in group 2. This parameter is only used for two-group models.
r.num.g2	Number of initial recovered persons in group 2. This parameter is only used for two-group SIR models.
...	Additional initial conditions passed to model.

Details

The initial conditions for a model solved with `dcm()` should be input into the `init.dcm` function. This function handles initial conditions for both base model types and original models.

Original models may use the parameter names listed as arguments here, a new set of names, or a combination of both. With new models, initial conditions must be input in the same order that the solved derivatives from the model are output.

Value

An EpiModel object of class `init.dcm`.

Sensitivity Analyses

Like `param.dcm()`, initial conditions may be specified as vectors of length greater than one to run sensitivity analyses over initial conditions. When `sens.param = TRUE` in `control.dcm()` (the default), each element of the vector produces a separate model run. If both parameters and initial conditions have vector values, all vectors must have the same length.

See Also

Use `param.dcm()` to specify model parameters and `control.dcm()` to specify the control settings. Run the parameterized model with `dcm()`.

Examples

```
# SI model initial conditions
init <- init.dcm(s.num = 500, i.num = 1)

# Sensitivity analysis over initial infected count
init <- init.dcm(s.num = 500, i.num = c(1, 5, 25))
```

init.icm

*Initial Conditions for Stochastic Individual Contact Models***Description**

Sets the initial conditions for stochastic individual contact models simulated with `icm`.

Usage

```
init.icm(s.num, i.num, r.num, s.num.g2, i.num.g2, r.num.g2, ...)
```

Arguments

<code>s.num</code>	Number of initial susceptible persons. For two-group models, this is the number of initial group 1 susceptible persons.
<code>i.num</code>	Number of initial infected persons. For two-group models, this is the number of initial group 1 infected persons.
<code>r.num</code>	Number of initial recovered persons. For two-group models, this is the number of initial group 1 recovered persons. This parameter is only used for the SIR model type.
<code>s.num.g2</code>	Number of initial susceptible persons in group 2. This parameter is only used for two-group models.
<code>i.num.g2</code>	Number of initial infected persons in group 2. This parameter is only used for two-group models.
<code>r.num.g2</code>	Number of initial recovered persons in group 2. This parameter is only used for two-group SIR models.
<code>...</code>	Additional initial conditions passed to model.

Details

The initial conditions for a model solved with `icm()` should be input into the `init.icm` function. This function handles initial conditions for the built-in SI, SIR, and SIS disease types; for custom or extension epidemic models, use `init.net()` with the network model class.

Value

An EpiModel object of class `init.icm`.

See Also

Use `param.icm()` to specify model parameters and `control.icm()` to specify the control settings. Run the parameterized model with `icm()`.

init.net

Initial Conditions for Stochastic Network Models

Description

Sets the initial conditions for stochastic network models simulated with `netsim`.

Usage

```
init.net(i.num, r.num, i.num.g2, r.num.g2, status.vector, infTime.vector, ...)
```

Arguments

<code>i.num</code>	Number of initial infected persons. For two-group models, this is the number of initial group 1 infected persons.
<code>r.num</code>	Number of initial recovered persons. For two-group models, this is the number of initial group 1 recovered persons. This parameter is only used for the SIR model type.
<code>i.num.g2</code>	Number of initial infected persons in group 2. This parameter is only used for two-group models.
<code>r.num.g2</code>	Number of initial recovered persons in group 2. This parameter is only used for two-group SIR models.
<code>status.vector</code>	A vector of length equal to the size of the input network, containing the status of each node. Setting status here overrides any inputs passed in the <code>.num</code> arguments.
<code>infTime.vector</code>	A vector of length equal to the size of the input network, containing the (historical) time of infection for each of those nodes with a current status of "i". Can only be used if <code>status.vector</code> is used, and must contain NA values for any nodes whose status is not "i".
<code>...</code>	Additional initial conditions passed to model.

Details

The initial conditions for a model solved with `netsim()` should be input into the `init.net` function. This function handles initial conditions for both base models and new modules. For an overview of specifying initial conditions across a variety of base network models, consult the [Network Modeling for Epidemics](#) tutorials.

Value

An EpiModel object of class `init.net`.

See Also

Use `param.net()` to specify model parameters and `control.net()` to specify the control settings. Run the parameterized model with `netsim()`.

Examples

```
# Example of using status.vector and infTime.vector together
n <- 100
status <- sample(c("s", "i"), size = n, replace = TRUE, prob = c(0.8, 0.2))
infTime <- rep(NA, n)
infTime[which(status == "i")] <- -rgeom(sum(status == "i"), prob = 0.01) + 2

init.net(status.vector = status, infTime.vector = infTime)
```

InitErgmTerm.absdiffby

Definition for absdiffby ERGM Term

Description

This function defines and initializes the `absdiffby` ERGM term that allows for representing homophily with respect to a non-binary attribute (e.g., age) differentially by a binary attribute (e.g., sex).

Usage

```
InitErgmTerm.absdiffby(nw, arglist, ...)
```

Arguments

<code>nw</code>	An object of class <code>network</code> .
<code>arglist</code>	A list of arguments as specified in the <code>ergm.userterms</code> package framework.
<code>...</code>	Additional data passed into the function as specified in the <code>ergm.userterms</code> package framework.

Details

This ERGM user term was written to allow for age-based homophily in partnership formation that is asymmetric by sex. The `absdiff` component targets age-based homophily while the `by` component allows that to be structured by a binary attribute such as "male", in order to enforce an offset in the average difference. This allows, for example, a average age difference in partnerships, but with males (on average) older than females.

InitErgmTerm.absdiffnodemix

Definition for absdiffnodemix ERGM Term

Description

This function defines and initializes the absdiffnodemix ERGM term that allows for targeting homophily based on a non-binary attribute (e.g., age) by combinations of a binary attribute (e.g., race).

Usage

```
InitErgmTerm.absdiffnodemix(nw, arglist, ...)
```

Arguments

nw	An object of class network.
arglist	A list of arguments as specified in the ergm.userterms package framework.
...	Additional data passed into the function as specified in the ergm.userterms package framework.

Details

This ERGM user term was written to allow for age-based homophily in partnership formation that is heterogeneous by race. The absdiff component targets the distribution of age mixing on that continuous variable, and the nodemix component differentiates this for black-black, black-white, and white-white couples.

InitErgmTerm.fuzzynodematch

Definition for fuzzynodematch ERGM Term

Description

This function defines and initializes the fuzzynodematch ERGM term that allows for generalized homophily.

Usage

```
InitErgmTerm.fuzzynodematch(nw, arglist, ...)
```

Arguments

nw	An object of class network.
arglist	A list of arguments as specified in the ergm.userterms package framework.
...	Additional data passed into the function as specified in the ergm.userterms package framework.

Details

This ERGM user term was written to allow for generalized homophily. The `attr` term argument should specify a character vertex attribute encoding the "venues" associated to each node. The `split` argument should specify a string that separates different "venues" in the attribute value for each node, as handled by `strsplit` with `fixed = TRUE`. For example, if `split` is "|" (the default), and the attribute value for a given node is "a12|b476", then the associated venues for this node are "a12" and "b476". The empty string "" is interpreted as "no venues".

If the `binary` term argument is `FALSE` (the default), the change statistic for an on-toggle is the number of unique venues associated to both nodes (informally speaking, this could be described as the number of venues on which the two nodes "match"); if `binary` is `TRUE`, the change statistic for an on-toggle is 1 if any venue is associated to both nodes, and 0 otherwise.

is.transmat

Extract Transmissions Matrix from Network Epidemic Model

Description

Extracts the matrix of transmission data for each transmission event that occurred within a network epidemic model.

Usage

```
is.transmat(x)
```

```
get_transmat(x, sim = 1, deduplicate = TRUE)
```

Arguments

<code>x</code>	An <code>EpiModel</code> object of class <code>netsim()</code> .
<code>sim</code>	Simulation number of extracted network.
<code>deduplicate</code>	If <code>TRUE</code> , randomly select one transmission event in the case that multiple events occur per newly infected agent within a time step.

Value

A data frame with the following standard columns:

- **at**: the time step at which the transmission occurred.
- **sus**: the ID number of the susceptible (newly infected) node.
- **inf**: the ID number of the infecting node.
- **infDur**: the duration of the infecting node's disease at the time of the transmission.
- **transProb**: the probability of transmission per act.
- **actRate**: the rate of acts per unit time.
- **finalProb**: the final transmission probability for the transmission event.

Examples

```
## Simulate SI epidemic on two-group Bernoulli random graph
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)
param <- param.net(inf.prob = 0.3, inf.prob.g2 = 0.15)
init <- init.net(i.num = 10, i.num.g2 = 10)
control <- control.net(type = "SI", nsteps = 10, nsims = 3, verbose = FALSE)
mod <- netsim(est, param, init, control)

## Extract the transmission matrix from simulation 2
get_transmat(mod, sim = 2)
```

is_active_posit_ids	<i>Are These Nodes Active (Positional IDs)</i>
---------------------	--

Description

Are These Nodes Active (Positional IDs)

Usage

```
is_active_posit_ids(dat, posit_ids)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
posit_ids	A vector of node positional identifiers.

Value

A logical vector with TRUE if the node is still active and FALSE otherwise.

is_active_unique_ids *Are These Nodes Active (Unique IDs)*

Description

Are These Nodes Active (Unique IDs)

Usage

```
is_active_unique_ids(dat, unique_ids)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
unique_ids	A vector of node unique identifiers.

Value

A logical vector with TRUE if the node is still active and FALSE otherwise.

make_restart_point *Make a Lightweight Restart Point From a netsim Object with termLite*

Description

Extract the elements required for re-initializing a netsim simulation from a completed simulation. This function also resets the Unique IDs and Time values to reduce the size of the simulation. This function only works for simulations where `control$termLite = TRUE`

Usage

```
make_restart_point(sim_obj, time_attrs, sim_num = 1, keep_steps = 1)
```

Arguments

sim_obj	a netsim object from an ended netsim call.
time_attrs	a character vector containing the names of the attributes that are expressed in time-steps. These will be offsetted so the last step in the original simulation become the step 1 (default) in the new ones. If no such attributes exist, pass <code>c()</code> .
sim_num	the number of the simulation to extract from the netsim object (default = 1).
keep_steps	The number of simulation steps to keep from the previous run. By default only keep one but more is possible if some back-history is wanted.

Details

The restart point created always contains a single simulation and drops the `attr.history`, the `raw.records` and `stats` from the initial simulation.

The `epi trackers`, `cumulative edgelists`, `transmission matrix` and `nwstats` are truncated to only contain the last `keep_steps` entries.

Warning: the `time_attrs` argument is mandatory. Almost all simulation worth restarting have such attributes (e.g. `time.of.hiv.infection`). If no such argument exists, passing `c()` will allow the function to run while ensuring that this was done on purpose.

When restarting from the output of this function, it is suggested to express the time steps in a relative manner in `control.net`:

```
control.net(
  start = restart_point$control$nsteps + 1,
  nsteps = restart_point$control$nsteps + 1 + 104)
)
```

Value

a trimmed `netsim` object with only one simulation that is ready to be used as a restart point.

Examples

```
## Not run:
# With pre-existing `sim`, `param` and `init` object (see `netsim`)

# List all attributes that store a time step
time_attrs <- c(
  "inf.time",
  "stage.time",
  "aids.time",
  "prep.start.last"
)
# Make a restart point a re-run for 10 more timesteps
x <- make_restart_point(sim, time_attrs, sim_num = 1, keep_steps = 1)
control <- control_msm(
  start = x$control$nsteps + 1,
  nsteps = x$control$nsteps + 1 + 10
)
sim <- netsim(x, param, init, control)

## End(Not run)
```

merge.icm

*Merge Data across Stochastic Individual Contact Model Simulations***Description**

Merges epidemiological data from two independent simulations of stochastic individual contact models from `icm()`.

Usage

```
## S3 method for class 'icm'
merge(x, y, ...)
```

Arguments

<code>x</code>	An EpiModel object of class <code>icm()</code> .
<code>y</code>	Another EpiModel object of class <code>icm()</code> , with the identical model parameterization as <code>x</code> .
<code>...</code>	Additional merge arguments (not used).

Details

This merge function combines the results of two independent simulations of `icm()` class models, simulated under separate function calls. The model parameterization between the two calls must be exactly the same, except for the number of simulations in each call. This allows for manual parallelization of model simulations.

This merge function does not work the same as the default merge, which allows for a combined object where the structure differs between the input elements. Instead, the function checks that objects are identical in model parameterization in every respect (except number of simulations) and binds the results.

Value

An EpiModel object of class `icm()` containing the data from both `x` and `y`.

Examples

```
param <- param.icm(inf.prob = 0.2, act.rate = 0.8)
init <- init.icm(s.num = 1000, i.num = 100)
control <- control.icm(type = "SI", nsteps = 10,
                      nsims = 3, verbose = FALSE)
x <- icm(param, init, control)

control <- control.icm(type = "SI", nsteps = 10,
                      nsims = 1, verbose = FALSE)
y <- icm(param, init, control)
```

```

z <- merge(x, y)

# Examine separate and merged data
as.data.frame(x)
as.data.frame(y)
as.data.frame(z)

```

merge.netsim

*Merge Model Simulations across netsim Objects***Description**

Merges epidemiological data from two independent simulations of stochastic network models from netsim.

Usage

```

## S3 method for class 'netsim'
merge(
  x,
  y,
  keep.transmat = TRUE,
  keep.network = TRUE,
  keep.nwstats = TRUE,
  keep.other = TRUE,
  param.error = TRUE,
  keep.diss.stats = TRUE,
  ...
)

```

Arguments

x	An EpiModel object of class <code>netsim()</code> .
y	Another EpiModel object of class <code>netsim()</code> , with the identical model parameterization as x.
keep.transmat	If TRUE, keep the transmission matrices from the original x and y elements. Note: transmission matrices only saved when (save.transmat == TRUE).
keep.network	If TRUE, keep the networkDynamic objects from the original x and y elements. Note: network only saved when (tergmLite == FALSE).
keep.nwstats	If TRUE, keep the network statistics (as set by the nwstats.formula parameter in control.net) from the original x and y elements.
keep.other	If TRUE, keep the other simulation elements (as set by the save.other parameter in control.net) from the original x and y elements.
param.error	If TRUE, if x and y have different params (in <code>param.net()</code>) or controls (passed in <code>control.net()</code>) an error will prevent the merge. Use FALSE to override that check.

```

keep.diss.stats      If TRUE, keep diss.stats from the original x and y objects.
...                  Additional merge arguments (not currently used).

```

Details

This merge function combines the results of two independent simulations of `netsim()` class models, simulated under separate function calls. The model parameterization between the two calls must be exactly the same, except for the number of simulations in each call. This allows for manual parallelization of model simulations.

This merge function does not work the same as the default merge, which allows for a combined object where the structure differs between the input elements. Instead, the function checks that objects are identical in model parameterization in every respect (except number of simulations) and binds the results.

Value

An EpiModel object of class `netsim()` containing the data from both x and y.

Examples

```

# Network model
nw <- network_initialize(n = 100)
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 10)
est <- netest(nw, formation = ~edges, target.stats = 25,
             coef.diss = coef.diss, verbose = FALSE)

# Epidemic models
param <- param.net(inf.prob = 1)
init <- init.net(i.num = 1)
control <- control.net(type = "SI", nsteps = 20, nsims = 2,
                      save.nwstats = TRUE,
                      nwstats.formula = ~edges + degree(0),
                      verbose = FALSE)
x <- netsim(est, param, init, control)
y <- netsim(est, param, init, control)

# Merging
z <- merge(x, y)

# Examine separate and merged data
as.data.frame(x)
as.data.frame(y)
as.data.frame(z)

```

Description

Stochastic individual contact models of infectious disease simulate epidemics in which contacts between individuals are instantaneous events in discrete time. They are intended to be the stochastic microsimulation analogs to deterministic compartmental models.

The `icm()` function handles both the simulation tasks. Within this function are a series of modules that initialize the simulation and then simulate new infections, recoveries, and vital dynamics at each time step. A module also handles the basic bookkeeping calculations for disease prevalence.

This help page presents a brief overview of the built-in module functions in the order in which they are used within `icm()`. ICMs support only the built-in SI, SIR, and SIS disease types. For custom or extension epidemic models, use the network model class via `netsim()` and `control.net()` instead.

Initialization Module

This function sets up agent attributes, like disease status, on the network at the starting time step of disease simulation, t_1 . For multiple-simulation function calls, these are reset at the beginning of each simulation.

- `initialize.icm()`: sets which agents are initially infected, through the initial conditions passed in `init.icm()`.

Disease Status Modification Modules

The main disease simulation occurs at each time step given the current state of the population at that step. Infection of agents is simulated as a function of disease parameters and population composition. Recovery of agents is likewise simulated with respect to infected nodes. These functions also analyze the flows for summary measures such as disease incidence.

- `infection.icm()`: randomly draws an edgelist given the parameters, subsets the list for discordant pairs, and simulates transmission on those discordant pairs through a series of draws from a binomial distribution.
- `recovery.icm()`: simulates recovery from infection either to a lifelong immune state (for SIR models) or back to the susceptible state (for SIS models), as a function of the recovery rate specified in the `rec.rate` parameter. The recovery rate may vary for two-group models.

Demographic Modules

Vital dynamics such as arrival and departure processes are simulated at each time step to update entries into and exits from the population. These are used in open-population ICMs.

- `departures.icm()`: randomly simulates departures or exits for agents given the departure rate specified in the disease-state and group-specific departure parameters in `param.icm()`. This involves deactivating agents from the population, but their historical data is preserved in the simulation.

- `arrivals.icm()`: randomly simulates new arrivals into the population given the current population size and the arrival rate parameters. This involves adding new agents into the population.

Bookkeeping Module

Simulations require bookkeeping at each time step to calculate the summary epidemiological statistics used in the model output analysis.

- `prevalence.icm()`: calculates the number in each disease state (susceptible, infected, recovered) at each time step for those active agents in the population.

modules.net

Modules for Stochastic Network Models

Description

Stochastic network models of infectious disease in EpiModel require statistical modeling of networks, simulation of those networks forward through time, and simulation of epidemic dynamics on top of those evolving networks. The `netsim()` function handles both the network and epidemic simulation tasks. Within this function are a series of modules that initialize the simulation and then simulate new infections, recoveries, and demographics on the network. Modules also handle the resimulation of the network and some bookkeeping calculations for disease prevalence.

Writing original network models that expand upon our "base" model set will require modifying the existing modules or adding new modules to the workflow in `netsim()`. The existing modules may be used as a template for replacement or new modules.

This help page provides an orientation to these module functions, in the order in which they are used within `netsim()`, to help guide users in writing their own functions. These module functions are not shown on the help index since they are not called directly by the end-user. To understand these functions in more detail, review the separate help pages listed below.

Initialization Module

This function sets up nodal attributes, like disease status, on the network at the starting time step of disease simulation, t_1 . For multiple-simulation function calls, these are reset at the beginning of each individual simulation.

- `initialize.net()`: sets up the main `netsim_dat` data structure used in the simulation, initializes which nodes are infected (via the initial conditions passed in `init.net()`), and simulates a first time step of the networks given the network model fit from `netest()`.

Disease Status Modification Modules

The main disease simulation occurs at each time step given the current state of the network at that step. Infection of nodes is simulated as a function of attributes of the nodes and the edges. Recovery of nodes is likewise simulated as a function of nodal attributes of those infected nodes. These functions also calculate summary flow measures such as disease incidence.

- `infection.net()`: simulates disease transmission given an edgelist of discordant partnerships by calculating the relevant transmission and act rates for each edge, and then updating the nodal attributes and summary statistics.
- `recovery.net()`: simulates recovery from infection either to a lifelong immune state (for SIR models) or back to the susceptible state (for SIS models), as a function of the recovery rate parameters specified in `param.net()`.

Demographic Modules

Demographics such as arrival and departure processes are simulated at each time step to update entries into and exits from the network. These are used in epidemic models with network feedback, in which the network is resimulated at each time step to account for the nodal changes affecting the edges.

- `departures.net()`: randomly simulates departure for nodes given their disease status (susceptible, infected, recovered), and their group-specific departure rates specified in `param.net()`. Departures involve deactivating nodes.
- `arrivals.net()`: randomly simulates new arrivals into the network given the current population size and the arrival rate specified in the `a.rate` parameters. This involves adding new nodes into the network.

Network Resimulation Module

In dependent network models, the network object is resimulated at each time step to account for changes in the size of the network (changed through entries and exits), and the disease status of the nodes.

- `resim_nets()`: resimulates the network object one time step forward given the set of formation and dissolution coefficients estimated in `netest()`.

Bookkeeping Module

Network simulations require bookkeeping at each time step to calculate the summary epidemiological statistics used in the model output analysis.

- `prevalence.net()`: calculates the number in each disease state (susceptible, infected, recovered) at each time step for those active nodes in the network. If the `epi.by` control is used, it calculates these statistics by a set of specified nodal attributes.
- `verbose.net()`: summarizes the current state of the simulation and prints this to the console.

One- & Two-Group Modules

If epidemic type is supplied within `control.net()`, EpiModel defaults each of the base epidemic and demographic modules described above (`arrivals.FUN`, `departures.FUN`, `infection.FUN`, `recovery.FUN`) to the correct `.net` function based on variables passed to `param.net()` (e.g. `num.g2`, denoting population size of group two, would select the two-group variants of the aforementioned modules). Two-group modules are denoted by a `.2g` affix (e.g., `recovery.2g.net`)

multilayer	<i>Specify Controls by Network</i>
------------	------------------------------------

Description

This utility function allows specification of certain `netsim()` controls to vary by network. The `netsim()` control arguments currently supporting multilayer specifications are `nwstats.formula`, `set.control.ergm`, `set.control.tergm`, and `tergmLite.track.duration`.

Usage

```
multilayer(...)
```

Arguments

... control arguments to apply to each network, with the index of the network corresponding to the index of the control argument

Value

an object of class `multilayer` containing the specified control arguments

See Also

`netsim()` for passing a list of `netest()` fits, one per layer. The [Multi-Layer Networks](#) chapter of the Network Modeling for Epidemics course materials works through a two-layer model in full.

mutate_epi	<i>Add New Epidemiology Variables</i>
------------	---------------------------------------

Description

Inspired by `dplyr::mutate`, `mutate_epi` adds new variables to the epidemiological and related variables within simulated model objects of any class in `EpiModel`.

Usage

```
mutate_epi(x, ...)
```

Arguments

`x` An `EpiModel` object of class `dcm`, `icm`, or `netsim`.
 ... Name-value pairs of expressions (see examples below).

Value

The updated `EpiModel` object of class `dcm`, `icm`, or `netsim`.

Examples

```
# DCM example
param <- param.dcm(inf.prob = 0.2, act.rate = 0.25)
init <- init.dcm(s.num = 500, i.num = 1)
control <- control.dcm(type = "SI", nsteps = 500)
mod1 <- dcm(param, init, control)
mod1 <- mutate_epi(mod1, prev = i.num/num)
plot(mod1, y = "prev")

# Network model example
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est1 <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

param <- param.net(inf.prob = 0.3, inf.prob.g2 = 0.15)
init <- init.net(i.num = 1, i.num.g2 = 0)
control <- control.net(type = "SI", nsteps = 10, nsims = 3,
                      verbose = FALSE)
mod1 <- netsim(est1, param, init, control)
mod1

# Add the prevalences to the dataset
mod1 <- mutate_epi(mod1, i.prev = i.num / num,
                  i.prev.g2 = i.num.g2 / num.g2)
plot(mod1, y = c("i.prev", "i.prev.g2"), qnts = 0.5, legend = TRUE)

# Add incidence rate per 100 person years (assume time step = 1 week)
mod1 <- mutate_epi(mod1, ir100 = 5200*(si.flow + si.flow.g2) /
                  (s.num + s.num.g2))

as.data.frame(mod1)
as.data.frame(mod1, out = "mean")
```

net-accessor

Functions to Access and Edit the Main netsim_dat Object in Network Models

Description

These `get_`, `set_`, `append_`, and `add_` functions allow a safe and efficient way to retrieve and mutate the main `netsim_dat` class object of network models (typical variable name `dat`). They are intended for use *inside module functions* that run during a `netsim()` simulation, not for editing the user-facing `param.net`, `init.net`, or `control.net` inputs before a run or the output object returned by `netsim()`. See the **Intended Usage** section below for details and alternatives.

This function returns an exhaustive named list of the attributes managed by EpiModel itself. It can be used to check the validity of an attributes list and of its types.

Usage

```
get_attr_list(dat, item = NULL)

get_attr(dat, item, posit_ids = NULL, override.null.error = FALSE)

add_attr(dat, item)

set_attr(dat, item, value, posit_ids = NULL, override.length.check = FALSE)

append_attr(dat, item, value, n.new)

remove_node_attr(dat, posit_ids)

get_epi_list(dat, item = NULL)

get_epi(dat, item, at = NULL, override.null.error = FALSE)

add_epi(dat, item)

set_epi(dat, item, at, value)

get_param_list(dat, item = NULL)

get_param(dat, item, override.null.error = FALSE)

add_param(dat, item)

set_param(dat, item, value)

get_control_list(dat, item = NULL)

get_control(dat, item, override.null.error = FALSE)

get_network_control(dat, network, item, override.null.error = FALSE)

add_control(dat, item)

set_control(dat, item, value)

get_init_list(dat, item = NULL)

get_init(dat, item, override.null.error = FALSE)

add_init(dat, item)
```

```

set_init(dat, item, value)

get_core_attributes()

append_core_attr(dat, at, n.new)

```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
<code>item</code>	A character vector containing the name of the element to access (for <code>get_</code> functions), create (for <code>add_</code> functions), or edit (for <code>set_</code> and <code>append_</code> functions). Can be of length greater than 1 for <code>get_*_list</code> functions.
<code>posit_ids</code>	For <code>set_attr</code> and <code>get_attr</code> , a numeric vector of <code>posit_ids</code> to subset the desired item.
<code>override.null.error</code>	If TRUE, <code>get_</code> will return NULL if the <code>item</code> does not exist instead of throwing an error. (default = FALSE).
<code>value</code>	New value to be attributed in the <code>set_</code> and <code>append_</code> functions.
<code>override.length.check</code>	If TRUE, <code>set_attr</code> allows the modification of the <code>item</code> size. (default = FALSE).
<code>n.new</code>	For <code>append_core_attr</code> , the number of new nodes to initiate with core attributes; for <code>append_attr</code> , the number of new elements to append at the end of <code>item</code> .
<code>at</code>	For <code>get_epi</code> , the timestep at which to access the specified <code>item</code> ; for <code>set_epi</code> , the timestep at which to add the new value for the epi output <code>item</code> ; for <code>append_core_attr</code> , the current time step.
<code>network</code>	index of network for which to get control

Value

A vector or a list of vectors for `get_` functions; the main list object for `set_`, `append_`, and `add_` functions.

Intended Usage

These accessors operate on the live `netsim_dat` object that `EpiModel` constructs internally when a simulation starts and passes through each module at every time step. The expected calling context is **inside a user-supplied module function** (e.g., a custom `infection.FUN`, `recovery.FUN`, or `arrivals/departures` module) registered with `control.net()`.

They are *not* a general-purpose tool for editing the user-facing input or output objects of `netsim()`:

- Calling them on a `param.net()`, `init.net()`, or `control.net()` object before a run will appear to succeed (those objects are also list-like) but produces an object that is not a valid simulation input.
- Calling them on the object returned by `netsim()` modifies only the saved input slot, not anything that would change the result of a re-run. Note also that `netsim()` strips the `*.FUN` entries from its returned control slot, so feeding that slot back into a new simulation will fail.

For modifications outside a running simulation, use the following instead:

- **Editing parameters before a run:** `update_params()` for a `param.net` object, or direct list assignment (e.g., `p$inf.prob <- 0.5`).
- **Editing init or control before a run:** direct list assignment (e.g., `ctrl$steps <- 1000`), or rebuild with a fresh call to the `init.net()` / `control.net()` constructor.
- **Scheduled mid-simulation changes:** the `.param.updater.list` argument to `param.net()` and the `.control.updater.list` argument to `control.net()`. See the "model-parameters" vignette for the underlying updater module and the scenario API built on top of it.

Core Attribute

The `append_core_attr` function initializes the attributes necessary for `EpiModel` to work (the four core attributes are: "active", "unique_id", "entrTime", and "exitTime"). These attributes are used in the initialization phase of the simulation, to create the nodes (see `initialize.net()`); and also used when adding nodes during the simulation (see `arrivals.net()`).

Mutability

The `set_`, `append_`, and `add_` functions DO NOT modify the `netsim_dat` object in place. The result must be assigned back to `dat` in order to be registered: `dat <- set_*(dat, item, value)`.

`set_` and `append_` vs `add_`

The `set_` and `append_` functions edit a pre-existing element or create a new one if it does not exist already by calling the `add_` functions internally.

One Value Per Node

Every nodal attribute holds exactly one value per node, and `netsim()` checks this at the end of each time step. When `set_attr` and `append_attr` create an attribute, the nodes they do not write to are filled with NA to preserve that: `set_attr` with `posit_ids` fills the unselected nodes, and `append_attr` fills the nodes that predate the ones being appended. This means `append_core_attr` must be called before `append_attr` when adding new nodes, as it is what makes those nodes known to the rest of the object.

See Also

`update_params()` for editing a `param.net` object outside a simulation; `param.net()`, `init.net()`, `control.net()` for input constructors; `netsim()` for running a simulation.

Examples

```
dat <- create_dat_object(control = list(nsteps = 150))
dat <- append_core_attr(dat, 1, 100)

dat <- add_attr(dat, "age")
dat <- set_attr(dat, "age", runif(100))
dat <- set_attr(dat, "status", rbinom(100, 1, 0.9))
dat <- append_attr(dat, "status", 1, 10)
```

```

dat <- append_attr(dat, "age", NA, 10)
get_attr_list(dat)
get_attr_list(dat, c("age", "active"))
get_attr(dat, "status")
get_attr(dat, "status", c(1, 4))

dat <- add_epi(dat, "i.num")
dat <- set_epi(dat, "i.num", 150, 10)
dat <- set_epi(dat, "s.num", 150, 90)
get_epi_list(dat)
get_epi_list(dat, c("i.num", "s.num"))
get_epi(dat, "i.num")
get_epi(dat, "i.num", c(1, 4))

dat <- add_param(dat, "x")
dat <- set_param(dat, "x", 0.4)
dat <- set_param(dat, "y", 0.8)
get_param_list(dat)
get_param_list(dat, c("x", "y"))
get_param(dat, "x")

dat <- add_init(dat, "x")
dat <- set_init(dat, "x", 0.4)
dat <- set_init(dat, "y", 0.8)
get_init_list(dat)
get_init_list(dat, c("x", "y"))
get_init(dat, "x")

dat <- add_control(dat, "x")
dat <- set_control(dat, "x", 0.4)
dat <- set_control(dat, "y", 0.8)
get_control_list(dat)
get_control_list(dat, c("x", "y"))
get_control(dat, "x")

```

netdx

Dynamic Network Model Diagnostics

Description

Runs diagnostic simulations on an ERGM/STERGM estimated with [netest](#) to assess whether the fitted model reproduces the intended network features. Both static (cross-sectional) and dynamic (temporal) diagnostics are supported. This is the recommended second step in the network modeling pipeline, after estimation with [netest](#) and before epidemic simulation with [netsim](#).

Usage

```

netdx(
  x,

```

```

nsims = 1,
dynamic = TRUE,
nsteps = NULL,
nwstats.formula = "formation",
set.control.ergm = control.simulate.formula(),
set.control.tergm = control.simulate.formula.tergm(MCMC.maxchanges =
  .Machine$integer.max),
sequential = TRUE,
keep.tedgelist = FALSE,
keep.tnetwork = FALSE,
verbose = TRUE,
ncores = 1,
skip.dissolution = FALSE,
future.use.plan = FALSE
)

```

Arguments

<code>x</code>	An EpiModel object of class <code>netest</code> .
<code>nsims</code>	Number of simulations to run. For dynamic diagnostics, 5–10 simulations are usually sufficient to assess model fit. For static diagnostics, use 10,000+ draws to obtain stable estimates.
<code>dynamic</code>	If TRUE, runs dynamic diagnostics that simulate the temporal network forward in time, checking both formation targets and partnership duration/dissolution. If FALSE, draws from the static ERGM fit to check cross-sectional network structure only (faster, but does not verify dissolution dynamics). Static diagnostics are only available when the model was fit with the edges dissolution approximation (<code>edapprox = TRUE</code> in netest).
<code>nsteps</code>	Number of time steps per simulation (dynamic simulations only). Should be at least several multiples of the longest target partnership duration to allow the duration and dissolution statistics to stabilize. For example, if the target duration is 50, running for 500 time steps is a reasonable starting point.
<code>nwstats.formula</code>	A right-hand sided ERGM formula with the network statistics of interest. The default is the formation formula of the network model contained in <code>x</code> . You may track additional network statistics beyond the formation terms by specifying them here, such as <code>~ edges + meandeg + concurrent + degree(0:4)</code> . This is useful for verifying that the model produces reasonable values for network features that were not directly targeted in the formation model.
<code>set.control.ergm</code>	Control arguments passed to <code>ergm</code> 's <code>simulate_formula.network</code> (see details).
<code>set.control.tergm</code>	Control arguments passed to <code>tergm</code> 's <code>simulate_formula.network</code> (see details).
<code>sequential</code>	For static diagnostics (<code>dynamic=FALSE</code>): if FALSE, each of the <code>nsims</code> simulated Markov chains begins at the initial network; if TRUE, the end of one simulation is used as the start of the next.

<code>keep.tedgelist</code>	If TRUE, keep the timed edgelist generated from the dynamic simulations. Returned in the form of a list of matrices, with one entry per simulation. Accessible at <code>\$edgelist</code> .
<code>keep.tnetwork</code>	If TRUE, keep the full <code>networkDynamic</code> objects from the dynamic simulations. Returned in the form of a list of <code>nD</code> objects, with one entry per simulation. Accessible at <code>\$network</code> .
<code>verbose</code>	If TRUE, print progress to the console.
<code>ncores</code>	Number of processor cores to run multiple simulations on, using the future framework.
<code>skip.dissolution</code>	If TRUE, skip over the calculations of duration and dissolution stats in <code>netdx</code> .
<code>future.use.plan</code>	If FALSE, <code>netdx</code> uses <code>multisession</code> with <code>workers = ncores</code> for its parallelization. If TRUE, <code>netdx</code> uses the user-defined plan from <code>globalEnv</code> . It may also be given the output of a <code>future::tweak()</code> call, which sets up a user-defined temporary plan within <code>netdx</code> ; this can be useful for distributed computation (HPC).

Details

The `netdx` function handles dynamic network diagnostics for network models fit with the `netest` function. Given the fitted model, `netdx` simulates a specified number of dynamic networks for a specified number of time steps per simulation. The network statistics in `nwstats.formula` are saved for each time step. Summary statistics for the formation model terms, as well as dissolution model and relational duration statistics, are then calculated and can be accessed when printing or plotting the `netdx` object. See `print.netdx` and `plot.netdx` for details on printing and plotting.

Value

A list of class `netdx`. Use `print()` to view summary tables of formation statistics, duration, and dissolution diagnostics. Use `plot.netdx` to visualize these diagnostics over time. Use `as.data.frame.netdx()` to extract timed edgelists (if `keep.tedgelist = TRUE`).

Control Arguments

Models fit with the full STERGM method in `netest` (setting the `edapprox` argument to FALSE) require only a call to `tergm`'s `simulate_formula.network`. Control parameters for those simulations may be set using `set.control.tergm` in `netdx`. The parameters should be input through the `control.simulate_formula.tergm` function, with the available parameters listed in the `tergm::control.simulate_formula.tergm` help page in the `tergm` package.

Models fit with the ERGM method with the edges dissolution approximation (setting `edapprox` to TRUE) require a call first to `ergm`'s `simulate_formula.network` for simulating an initial network, and second to `tergm`'s `simulate_formula.network` for simulating that static network forward through time. Control parameters may be set for both processes in `netdx`. For the first, the parameters should be input through the `control.simulate_formula()` function, with the available parameters listed in the `ergm::control.simulate_formula` help page in the `ergm` package. For the second, parameters should be input through the `control.simulate_formula.tergm()` function, with the available parameters listed in the `tergm::control.simulate_formula.tergm` help page in the `tergm` package. An example is shown below.

Static vs. Dynamic Diagnostics

Static diagnostics (`dynamic = FALSE`) draw many independent networks from the fitted ERGM and compare the resulting statistics to the target values. This is fast and checks whether the cross-sectional structure is correct, but it does not verify partnership durations or dissolution rates. Dynamic diagnostics (`dynamic = TRUE`) simulate the full temporal network forward in time, checking both formation targets and dissolution/duration dynamics. Dynamic diagnostics are slower but more comprehensive, and are required to verify models that will be used with vital dynamics (arrivals/departures).

Interpreting Diagnostics

After running `netdx`, use `print()` and `plot.netdx` to inspect the results. Key indicators of a good model fit include:

- **Formation statistics:** The "Sim Mean" should be close to the "Target" value. A small "Pct Diff" ($< 5\%$) indicate good fit.
- **Duration statistics** (dynamic only): The simulated mean edge durations should match the values passed to `dissolution_coefs`.
- **Dissolution statistics** (dynamic only): The simulated dissolution rates should be approximately $1 / \text{duration}$.

Common problems: If formation statistics are off, the ERGM may need increased burn-in (via `set.control.ergm`), or the target statistics may be incompatible (e.g., specifying more edges than the network can support). If durations are off but formation is correct, verify that `d.rate` was correctly specified in `dissolution_coefs` for models with vital dynamics.

See Also

Estimate the network model with `netest` before running diagnostics. Plot diagnostics with `plot.netdx` and print summary tables with `print.netdx`. With `keep.tedgelist = TRUE`, extract the timed edgelist with `as.data.frame.netdx()` and the cumulative or momentary degree distribution with `get_degree_dist`. After diagnostics confirm a good fit, simulate the epidemic with `netsim`.

Examples

```
# Static diagnostics on a simple model
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 25)
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)
dx <- netdx(est, nsims = 1e4, dynamic = FALSE, verbose = FALSE)
dx
plot(dx)
```

```
## Not run:
# Static diagnostics with additional network statistics
dx1 <- netdx(est,
  nsims = 1e4, dynamic = FALSE,
```

```

nwstats.formula = ~ edges + meandeg + concurrent
)
dx1
plot(dx1, method = "b", stats = c("edges", "concurrent"))

# Dynamic diagnostics on the STERGM approximation
dx2 <- netdx(est,
  nsims = 5, nsteps = 500,
  nwstats.formula = ~ edges + meandeg + concurrent,
  set.control.ergm = control.simulate.formula(MCMC.burnin = 1e6)
)
dx2
plot(dx2, stats = c("edges", "meandeg"), plots.joined = FALSE)
plot(dx2, type = "duration")
plot(dx2, type = "dissolution", qnts.col = "orange2")
plot(dx2, type = "dissolution", method = "b", col = "bisque")

# Dynamic diagnostics on a more complex model
nw <- network_initialize(n = 1000)
nw <- set_vertex_attribute(nw, "neighborhood", rep(1:10, 100))
formation <- ~edges + nodematch("neighborhood", diff = TRUE)
target.stats <- c(800, 45, 81, 24, 16, 32, 19, 42, 21, 24, 31)
coef.diss <- dissolution_coefs(dissolution = ~offset(edges) +
  offset(nodematch("neighborhood", diff = TRUE)),
  duration = c(52, 58, 61, 55, 81, 62, 52, 64, 52, 68, 58))
est2 <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)
dx3 <- netdx(est2, nsims = 5, nsteps = 100)
print(dx3)
plot(dx3)
plot(dx3, type = "duration", plots.joined = TRUE, qnts = 0.2, legend = TRUE)
plot(dx3, type = "dissolution", mean.smooth = FALSE, mean.col = "red")

## End(Not run)

```

netest

Dynamic Network Model Estimation

Description

Estimates statistical network models using the exponential random graph modeling (ERGM) framework with extensions for dynamic/temporal models (STERGM). This is typically the first step in the network modeling pipeline, followed by [netdx](#) for model diagnostics and [netsim](#) for epidemic simulation.

Usage

```

netest(
  nw,
  formation,

```

```

    target.stats,
    coef.diss,
    constraints = NULL,
    coef.form = NULL,
    edapprox = TRUE,
    set.control.ergm = control.ergm(),
    set.control.tergm = control.tergm(MCMC.maxchanges = .Machine$integer.max),
    set.control.ergm.ego = NULL,
    verbose = FALSE,
    nested.edapprox = TRUE,
    ergm.ego.popsizes = 0,
    ...
)

```

Arguments

nw	An object of class network or egor, with the latter indicating an ergm.ego fit.
formation	Right-hand sided STERGM formation formula in the form <code>~edges + ...</code> , where <code>...</code> are additional network statistics (ERGM terms). This formula specifies which structural features of the network should be reproduced by the model. Common terms include edges (overall connectivity), nodematch (homophily by attribute), concurrent (overlapping partnerships), and degree (degree distribution constraints). See ergm::ergm-terms for the full list of available terms.
target.stats	Vector of target statistics for the formation model, with one number for each network statistic in the model. These are the observed (or desired) values for each term in the formation formula. For example, if <code>formation = ~edges + concurrent</code> , then <code>target.stats = c(175, 40)</code> means the model should produce approximately 175 edges and 40 nodes with 2 or more partners. For an edges-only model, a useful starting value is <code>mean_degree * network_size / 2</code> . Ignored if fitting via <code>ergm.ego</code> .
coef.diss	An object of class <code>disscoef</code> output from the dissolution_coefs function. This encodes the average partnership duration(s) and the corresponding dissolution model coefficients. For models with vital dynamics (arrivals and departures), the <code>d.rate</code> argument in dissolution_coefs should be set to adjust for the competing risk of node departure.
constraints	Right-hand sided formula specifying constraints for the modeled network, in the form <code>~...</code> , where <code>...</code> are constraint terms. By default, no constraints are set.
coef.form	Vector of coefficients for the offset terms in the formation formula.
edapprox	If TRUE, use the indirect edges dissolution approximation method for the dynamic model fit, otherwise use the more time-intensive full STERGM estimation (see details). The approximation is recommended for most use cases, especially when average partnership durations are moderate to long (> 25 time steps). Direct STERGM estimation (<code>edapprox = FALSE</code>) is slower but may be preferred for very short durations or when inferential quantities (standard errors, p-values) on formation coefficients are needed. For nw of class egor, only <code>edapprox = TRUE</code> is supported.

```

set.control.ergm
    Control arguments passed to ergm (see details).
set.control.tergm
    Control arguments passed to tergm (see details).
set.control.ergm.ego
    Control arguments passed to ergm.ego (see details).
verbose
    If TRUE, print model fitting progress to console.
nested.edapprox
    Logical, only used when edapprox = TRUE. When TRUE (the default), the dissolution model must be an initial segment of the formation model; when FALSE, the dissolution terms need not be nested (see details).
ergm.ego.popsiz
    Numeric. The popsize argument passed to ergm.ego::ergm.ego() when nw is an egor. Shifts the edges coefficient to reflect a target population size. The default of 0 scales the edges term to the size of the egor sample, matching the coefficient an equivalent ergm fit would produce. Set to 1 for per-capita scaling, which lets the fitted model be applied to a network of arbitrary size without knowing the population size assumed during estimation. See ?ergm.ego for other accepted values.
...
    Additional arguments passed to other functions.

```

Details

netest is a wrapper function for the `ergm`, `ergm.ego`, and `tergm` functions that estimate static and dynamic network models. Network model estimation is the first step in simulating a stochastic network epidemic model in `EpiModel`. The output from `netest` is a necessary input for running the epidemic simulations in `netsim`. With a fitted network model, one should always first proceed to model diagnostics, available through the `netdx` function, to check model fit. A detailed description of fitting these models, along with examples, may be found in the [Network Modeling for Epidemics](#) tutorials.

Value

A fitted network model object of class `netest`. This object is passed to `netdx` for diagnostics and to `netsim` for epidemic simulation. Use `print()` to view the model form, including the formation formula, target statistics, and dissolution model. Use `summary()` to view the estimated model coefficients and goodness-of-fit statistics from the underlying ERGM or STERGM fit.

Edges Dissolution Approximation

The edges dissolution approximation method is described in Carnegie et al. This approximation requires that the dissolution coefficients are known, that the formation model is being fit to cross-sectional data conditional on those dissolution coefficients, and that the terms in the dissolution model are a subset of those in the formation model. Under certain additional conditions, the formation coefficients of a STERGM model are approximately equal to the coefficients of that same model fit to the observed cross-sectional data as an ERGM, minus the corresponding coefficients in the dissolution model. The approximation thus estimates this ERGM (which is typically much faster than estimating a STERGM) and subtracts the dissolution coefficients.

The conditions under which this approximation best hold are when there are few relational changes from one time step to another; i.e. when either average relational durations are long, or density is low, or both. Conveniently, these are the same conditions under which STERGM estimation is slowest. Note that the same approximation is also used to obtain starting values for the STERGM estimate when the latter is being conducted. The estimation does not allow for calculation of standard errors, p-values, or likelihood for the formation model; thus, this approach is of most use when the main goal of estimation is to drive dynamic network simulations rather than to conduct inference on the formation model. The user is strongly encouraged to examine the behavior of the resulting simulations to confirm that the approximation is adequate for their purposes. For an example, see the vignette for the package `tergm`.

It has recently been found that subtracting a modified version of the dissolution coefficients from the formation coefficients provides a more principled approximation, and this is now the form of the approximation applied by `netest` for both settings of `nested.edapprox`. The modified values subtracted from the formation coefficients are equivalent to the (crude) dissolution coefficients with their target durations increased by 1. The `nested.edapprox` argument does not change this approximation; it only controls how it is implemented. When `TRUE` (the default), the dissolution terms are an initial segment of the formation model, so the correction values are subtracted in place from the corresponding initial formation model coefficients. When `FALSE`, the dissolution terms need not be nested in the formation model: they are appended to the formation model and the corresponding correction values are appended to the vector of formation model coefficients.

The nesting requirement is a property of the approximation, so it is only checked when `edapprox = TRUE`. A direct STERGM fit (`edapprox = FALSE`) estimates formation and persistence jointly and places no such requirement on the dissolution terms. Note, however, that `tergm` derives the starting values for its EGMME fit from the same approximation, so a direct fit whose persistence terms have no formation analog needs starting values supplied through `set.control.tergm`, with `control.tergm(init = ...)` or `control.tergm(init.method = "zeros")`.

Control Arguments

The `ergm`, `ergm.ego`, and `tergm` functions allow control settings for the model fitting process. When fitting a STERGM directly (setting `edapprox` to `FALSE`), control parameters may be passed to the `tergm` function with the `set.control.tergm` argument in `netest`. The controls should be input through the `control.tergm()` function, with the available parameters listed in the [tergm::control.tergm](#) help page in the `tergm` package.

When fitting a STERGM indirectly (setting `edapprox` to `TRUE`), control settings may be passed to the `ergm` function using `set.control.ergm`, or to the `ergm.ego` function using `set.control.ergm.ego`. The controls should be input through the `control.ergm()` and `control.ergm.ego()` functions, respectively, with the available parameters listed in the [ergm::control.ergm](#) help page in the `ergm` package and the [ergm.ego::control.ergm.ego](#) help page in the `ergm.ego` package. An example is below.

Typical Workflow

The network modeling pipeline in `EpiModel` typically follows these steps:

1. Initialize a network: [network_initialize](#)
2. Specify formation and dissolution: a formation formula (e.g., `~edges + concurrent`) with target statistics, and dissolution coefficients via [dissolution_coefs](#)

3. Estimate the network model: `netest()`
4. Diagnose model fit: [netdx](#)
5. Simulate the epidemic: [netsim](#) with [param.net](#), [init.net](#), and [control.net](#)

References

Krivitsky PN, Handcock MS. "A Separable Model for Dynamic Networks." JRSS(B). 2014; 76.1: 29-46.

Carnegie NB, Krivitsky PN, Hunter DR, Goodreau SM. An Approximation Method for Improving Dynamic Network Model Fitting. Journal of Computational and Graphical Statistics. 2014; 24(2): 502-519.

Jenness SM, Goodreau SM and Morris M. EpiModel: An R Package for Mathematical Modeling of Infectious Disease over Networks. Journal of Statistical Software. 2018; 84(8): 1-47.

See Also

Use [dissolution_coefs](#) to compute dissolution coefficients before estimation. Use [netdx](#) to diagnose the fitted network model, and [netsim](#) to simulate epidemic spread over a simulated dynamic network consistent with the model fit. Parameterize the epidemic simulation with [param.net](#), [init.net](#), and [control.net](#).

Examples

```
# Initialize a network of 100 nodes
nw <- network_initialize(n = 100)

# Set formation formula
formation <- ~edges + concurrent

# Set target statistics for formation
target.stats <- c(50, 25)

# Obtain the offset coefficients
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 10)

# Estimate the STERGM using the edges dissolution approximation
est <- netest(nw, formation, target.stats, coef.diss,
              set.control.ergm = control.ergm(MCMC.burnin = 1e5,
                                              MCMC.interval = 1000))

# View the model form (formation, targets, dissolution)
est

# View the estimated coefficients
summary(est)

## Not run:
# Model with homophily on a nodal attribute
nw2 <- network_initialize(n = 500)
nw2 <- set_vertex_attribute(nw2, "risk", rep(0:1, each = 250))
formation2 <- ~edges + nodematch("risk")
```

```

target.stats2 <- c(175, 110)
coef.diss2 <- dissolution_coefs(dissolution = ~offset(edges), duration = 50)
est2 <- netest(nw2, formation2, target.stats2, coef.diss2)
est2

# Direct STERGM estimation (slower, for short durations or inference)
est3 <- netest(nw, formation, target.stats, coef.diss, edapprox = FALSE)

## End(Not run)

```

netsim

Stochastic Network Models

Description

Simulates stochastic network epidemic models for infectious disease. This is typically the final step in the network modeling pipeline, after network estimation with [netest](#) and model diagnostics with [netdx](#).

Usage

```
netsim(x, param, init, control)
```

Arguments

x	If <code>control\$start == 1</code> , either a fitted network model object of class <code>netest</code> or a list of such objects. If <code>control\$start > 1</code> , an object of class <code>netsim</code> . When multiple networks are used (multi-layer models), pass a list of <code>netest</code> objects, one per network layer; the node sets (including network size and nodal attributes) are assumed to be the same for all networks.
param	Model parameters, as an object of class param.net . Includes transmission probability (<code>inf.prob</code>), act rate (<code>act.rate</code>), recovery rate (<code>rec.rate</code>), and demographic rates for models with vital dynamics. Custom parameters for extended models may also be passed through <code>param.net</code> .
init	Initial conditions, as an object of class init.net . Specifies the initial number of infected nodes (<code>i.num</code>) and, for SIR models, recovered nodes (<code>r.num</code>). For two-group models, the corresponding <code>.g2</code> parameters are also required.
control	Control settings, as an object of class control.net . Key settings include type (disease model: "SI", "SIR", or "SIS"), <code>nsteps</code> (number of time steps), <code>nsims</code> (number of simulations), <code>tergmLite</code> (lightweight mode for performance), and <code>resimulate.network</code> (required for models with vital dynamics). For extended models, custom module functions are also passed here.

Details

Stochastic network models explicitly represent phenomena within and across edges (pairs of nodes that remain connected) over time. This enables edges to have duration, allowing for repeated transmission-related acts within the same dyad, specification of edge formation and dissolution rates, control over the temporal sequencing of multiple edges, and specification of network-level features. A detailed description of these models, along with examples, is found in the [Network Modeling for Epidemics](#) course materials.

The `netsim` function performs modeling of both the base model types and original models. Base model types include one-group and two-group models with disease types for Susceptible-Infected (SI), Susceptible-Infected-Recovered (SIR), and Susceptible-Infected-Susceptible (SIS).

Original models may be parameterized by writing new process modules that either take the place of existing modules (for example, disease recovery), or supplement the set of existing processes with a new one contained in a new module. This functionality is documented in the [Extending EpiModel](#) section of the [Network Modeling for Epidemics](#) course materials. The list of modules within `netsim` available for modification is listed in [modules.net\(\)](#).

Value

A list of class `netsim` with the following elements:

- **param:** the epidemic parameters passed into the model through `param`, with additional parameters added as necessary.
- **control:** the control settings passed into the model through `control`, with additional controls added as necessary.
- **epi:** a list of data frames, one for each epidemiological output from the model. Outputs for base models always include the size of each compartment, as well as flows in, out of, and between compartments.
- **stats:** a list containing two sublists, `nwstats` for any network statistics saved in the simulation, and `transmat` for the transmission matrix saved in the simulation. See [control.net\(\)](#) for further details.
- **network:** a list of lists of `networkDynamic` or `networkLite` objects, with one list of objects for each model simulation.

If `control$raw.output == TRUE`: A list of the raw (pre-processed) `netsim_dat` objects, for use in simulation continuation.

The epi data can be extracted as a data frame with [as.data.frame.netsim\(\)](#), with options for per-simulation values (`out = "vals"`), means (`out = "mean"`), standard deviations (`out = "sd"`), or quantiles (`out = "qnt"`).

Module Pipeline

At each time step, `netsim` executes a series of modules in sequence. For base models, the default pipeline is:

1. **Network resimulation** ([resim_nets](#)): updates the network structure (if `resimulate.network == TRUE`).

2. **Infection** ([infection.net](#)): simulates disease transmission across discordant edges (where one partner is susceptible and the other is infected).
3. **Recovery** ([recovery.net](#)): simulates recovery from infection (SIR and SIS models only).
4. **Departures** ([departures.net](#)): simulates node exits from the network (if vital dynamics are enabled).
5. **Arrivals** ([arrivals.net](#)): simulates new node entries into the network (if vital dynamics are enabled).
6. **Prevalence** ([prevalence.net](#)): calculates summary statistics.

See [modules.net\(\)](#) for full details on each module.

Performance and `tergmLite`

Setting `tergmLite = TRUE` in [control.net](#) uses a lightweight network representation (`networkLite`) that is substantially faster than the full `networkDynamic` representation, often by a factor of 20–50x. This is recommended for large networks or when running many simulations. The trade-off is that full `networkDynamic` objects are not available for post-hoc analysis; use the cumulative edgelist (`cumulative.edgelist = TRUE`) instead to track partnership histories.

The `resimulate.network` control must be set to `TRUE` when demographic processes (arrivals and departures) change the network composition over time. Without it, the network structure evolves independently of the epidemic and demographic dynamics.

Partnership Histories (Cumulative Edgelist)

For contact tracing, partnership-duration analysis, or reachability analysis, enable cumulative-edgelist tracking via `control.net(cumulative.edgelist = TRUE, save.cumulative.edgelist = TRUE)`. This records every edge formed during the simulation along with its start and stop time steps, and attaches the result to the returned object as `sim$cumulative.edgelist`. Under `tergmLite = TRUE` this is the recommended substitute for the full `networkDynamic` history.

The helper family ([get_cumulative_edgelist](#), [get_cumulative_edgelist_df](#), [get_partners](#), [get_cumulative_degree](#), [get_forward_reachable](#), [get_backward_reachable](#)) operates on the tracked data. See `vignette("network-objects", package = "EpiModel")` for the full lifecycle.

Multi-Network Models

For models with multiple overlapping network layers (e.g., sexual and needle-sharing networks), pass a list of [netest](#) objects to the `x` argument, one per network layer. Each layer has its own formation/dissolution dynamics but shares the same node set. See the [multilayer](#) documentation for specifying controls that vary by layer, and the [Multi-Layer Networks](#) chapter of the Network Modeling for Epidemics course materials for a worked example building from independent to cross-dependent layers.

Restarting and Checkpointing

Simulations can be checkpointed and restarted if interrupted. Set `.checkpoint.steps` and `.checkpoint.dir` in [control.net](#) to enable automatic checkpointing. To restart a simulation from a prior `netsim` output, pass the `netsim` object as `x` and set `control$start` to one greater than the final time step of the prior simulation. See the Checkpointing Simulations section of [control.net](#) for full details.

References

Jenness SM, Goodreau SM and Morris M. EpiModel: An R Package for Mathematical Modeling of Infectious Disease over Networks. *Journal of Statistical Software*. 2018; 84(8): 1-47.

See Also

Estimate the network model with `netest` and diagnose model fit with `netdx` before running simulations. Extract model results with `as.data.frame.netsim()`. Summarize the time-specific model results with `summary.netsim()`. Plot the model results with `plot.netsim()`. Extract the network with `get_network`, the transmission matrix with `get_transmat`, and derive new epi statistics with `mutate_epi`.

Examples

```
## Not run:
## Example 1: SI Model without Network Feedback
# Network model estimation
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est1 <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Epidemic model
param <- param.net(inf.prob = 0.3)
init <- init.net(i.num = 10)
control <- control.net(type = "SI", nsteps = 100, nsims = 5, verbose.int = 0)
mod1 <- netsim(est1, param, init, control)

# Print, plot, and summarize the results
mod1
plot(mod1)
summary(mod1, at = 50)

## Example 2: SIR Model with Network Feedback (Two-Group)
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50

# Recalculate dissolution coefficient with departure rate
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20,
                             d.rate = 0.0021)

# Reestimate the model with new coefficient
est2 <- netest(nw, formation, target.stats, coef.diss)

# Set parameters to include demographic rates
param <- param.net(inf.prob = 0.3, inf.prob.g2 = 0.15,
                  rec.rate = 0.02, rec.rate.g2 = 0.02,
                  a.rate = 0.002, a.rate.g2 = NA,
```

```

        ds.rate = 0.001, ds.rate.g2 = 0.001,
        di.rate = 0.001, di.rate.g2 = 0.001,
        dr.rate = 0.001, dr.rate.g2 = 0.001)
init <- init.net(i.num = 10, i.num.g2 = 10,
               r.num = 0, r.num.g2 = 0)
control <- control.net(type = "SIR", nsteps = 100, nsims = 5,
                      resimulate.network = TRUE, tergmLite = TRUE)

# Simulate the model with new network fit
mod2 <- netsim(est2, param, init, control)

# Print, plot, and summarize the results
mod2
plot(mod2)
summary(mod2, at = 40)

## Example 3: Post-Simulation Analysis
# Extract epi data as a data frame
as.data.frame(mod1)
as.data.frame(mod1, out = "mean")

# Extract the transmission matrix from simulation 1
get_transmat(mod1, sim = 1)

# Derive new epi statistics and plot them
mod1 <- mutate_epi(mod1, prev = i.num / num)
plot(mod1, y = "prev")

## End(Not run)

```

network_initialize	<i>Initialize Network Object</i>
--------------------	----------------------------------

Description

Initialize an undirected network object for use in EpiModel workflows.

Usage

```
network_initialize(n)
```

Arguments

n	Network size.
---	---------------

Details

This function is used in EpiModel workflows to initialize an empty network object. The network attributes directed, bipartite, hyper, loops, and multiple are set to FALSE.

Value

Returns an object of class network.

Examples

```
nw <- network_initialize(100)
nw
```

nwupdate.net	<i>Dynamic Network Updates</i>
--------------	--------------------------------

Description

This function handles all calls to the network object contained on the main netsim_dat object handled in netsim.

Usage

```
nwupdate.net(dat, at)
```

Arguments

- | | |
|-----|--|
| dat | Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> . |
| at | Current time step. |

Value

The updated netsim_dat main list object.

overwrite_attrs	<i>Helper to use a data.frame to initialize some attributes</i>
-----------------	---

Description

Uses dat\$init\$init_attr to overwrite some attributes of the nodes at initialization

Usage

```
overwrite_attrs(dat)
```

Arguments

- | | |
|-----|--|
| dat | Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> . |
|-----|--|

Details

If an `init_attr` data.frame is present in `dat$init`, use it to overwrite the attributes it contains. `init_attr` must have a number of rows equal to the number of nodes in the model as the attributes will be overwritten one to one, ensuring the correct ordering. `init_attr` columns **MUST** have a corresponding attribute already initialized. See "R/default_attributes.R" for adding new attributes to the model. `init_attr` is removed from `dat$init` at the end of the function to free up its memory.

Value

The updated `netsim_dat` main list object.

padded_vector	<i>Grow a Vector to a Given Size, Padding it With Empty Elements</i>
---------------	--

Description

Grow a vector to a given size, padding it with NULL if `orig` is a list and with NA otherwise

Usage

```
padded_vector(orig, size)
```

Arguments

<code>orig</code>	A vector to grow.
<code>size</code>	The final size of the vector.

Value

A vector of size `size` padded with NULLs or NAs at the end.

param.dcm	<i>Epidemic Parameters for Deterministic Compartmental Models</i>
-----------	---

Description

Sets the epidemic parameters for deterministic compartmental models simulated with `dcm`.

Usage

```

param.dcm(
  inf.prob,
  inter.eff,
  inter.start,
  act.rate,
  rec.rate,
  a.rate,
  ds.rate,
  di.rate,
  dr.rate,
  inf.prob.g2,
  act.rate.g2,
  rec.rate.g2,
  a.rate.g2,
  ds.rate.g2,
  di.rate.g2,
  dr.rate.g2,
  balance,
  ...
)

```

Arguments

inf.prob	Probability of infection per transmissible act between a susceptible and an infected person. In two-group models, this is the probability of infection for the group 1 members.
inter.eff	Efficacy of an intervention which affects the per-act probability of infection. Efficacy is defined as 1 - the relative hazard of infection given exposure to the intervention, compared to no exposure.
inter.start	Time step at which the intervention starts, between 1 and the number of time steps specified in the model. This will default to 1 if inter.eff is defined but this parameter is not.
act.rate	Average number of transmissible acts per person per unit time. For two-group models, this is the number of acts per group 1 person per unit time; a balance between the acts in groups 1 and 2 is necessary, and set using the balance parameter (see details).
rec.rate	Average rate of recovery with immunity (in SIR models) or re-susceptibility (in SIS models). The recovery rate is the reciprocal of the disease duration. For two-group models, this is the recovery rate for group 1 persons only. This parameter is only used for SIR and SIS models.
a.rate	Arrival or entry rate. For one-group models, the arrival rate is the rate of new arrivals per person per unit time. For two-group models, the arrival rate is parameterized as a rate per group 1 person per unit time, with the a.rate.g2 rate set as described below.
ds.rate	Departure or exit rate for susceptible persons. For two-group models, it is the rate for the group 1 susceptible persons only.

di.rate	Departure or exit rate for infected persons. For two-group models, it is the rate for the group 1 infected persons only.
dr.rate	Departure or exit rate for recovered persons. For two-group models, it is the rate for the group 1 recovered persons only. This parameter is only used for SIR models.
inf.prob.g2	Probability of infection per transmissible act between a susceptible group 2 person and an infected group 1 person. It is the probability of infection to group 2 members.
act.rate.g2	Average number of transmissible acts per group 2 person per unit time; a balance between the acts in groups 1 and 2 is necessary, and set using the balance parameter (see details).
rec.rate.g2	Average rate of recovery with immunity (in SIR models) or re-susceptibility (in SIS models) for group 2 persons. This parameter is only used for two-group SIR and SIS models.
a.rate.g2	Arrival or entry rate for group 2. This may either be specified numerically as the rate of new arrivals per group 2 persons per unit time, or as NA in which case the group 1 rate, a.rate, governs the group 2 rate. The latter is used when, for example, the first group is conceptualized as female, and the female population size determines the arrival rate. Such arrivals are evenly allocated between the two groups.
ds.rate.g2	Departure or exit rate for group 2 susceptible persons.
di.rate.g2	Departure or exit rate for group 2 infected persons.
dr.rate.g2	Departure or exit rate for group 2 recovered persons. This parameter is only used for SIR model types.
balance	For two-group models, balance the act.rate to the rate set for group 1 (with balance="g1") or group 2 (with balance="g2"). See details.
...	Additional arguments passed to model.

Details

param.dcm sets the epidemic parameters for deterministic compartmental models solved with the [dcm\(\)](#) function. The models may use the base types, for which these parameters are used, or original model specifications for which these parameters may be used (but not necessarily).

For base models, the model specification will be selected as a function of the model parameters entered here and the control settings in [control.dcm\(\)](#). One-group and two-group models are available, where the former assumes a homogeneous mixing in the population and the latter assumes some form of heterogeneous mixing between two distinct partitions in the population (e.g., men and women). Specifying any group two parameters (those with a .g2) implies the simulation of a two-group model. All the parameters for a desired model type must be specified, even if they are zero.

Value

An EpiModel object of class param.dcm.

Act Balancing

In two-group models, a balance between the number of acts for group 1 members and those for group 2 members must be maintained. With purely heterogeneous mixing, the product of one group size and act rate must equal the product of the other group size and act rate: $N_1\alpha_1 = N_2\alpha_2$, where N_i is the group size and α_i the group-specific act rate at time t . The balance parameter here specifies which group's act rate should control the others with respect to balancing.

Sensitivity Analyses

dcm has been designed to easily run DCM sensitivity analyses, where a series of models varying one or more of the model parameters is run. This is possible by setting any parameter as a vector of length greater than one.

New Model Types

An original model may use either the existing model parameters named here, an original set of parameters, or a combination of both. The `...` argument allows the user to pass an arbitrary set of new model parameters into `param.dcm`. Whereas there are strict checks for base models that the model parameters are valid, parameter validity is the user's responsibility with these original models.

See Also

Use `init.dcm()` to specify the initial conditions and `control.dcm()` to specify the control settings. Run the parameterized model with `dcm()`.

Examples

```
# SI model parameters
param <- param.dcm(inf.prob = 0.2, act.rate = 0.25)

# SIS model with sensitivity analysis over act.rate
param <- param.dcm(inf.prob = 0.2, act.rate = seq(0.1, 0.5, 0.1),
                  rec.rate = 1/50)
```

 param.icm

Epidemic Parameters for Stochastic Individual Contact Models

Description

Sets the epidemic parameters for stochastic individual contact models simulated with `icm`.

Usage

```

param.icm(
  inf.prob,
  inter.eff,
  inter.start,
  act.rate,
  rec.rate,
  a.rate,
  ds.rate,
  di.rate,
  dr.rate,
  inf.prob.g2,
  act.rate.g2,
  rec.rate.g2,
  a.rate.g2,
  ds.rate.g2,
  di.rate.g2,
  dr.rate.g2,
  balance,
  ...
)

```

Arguments

inf.prob	Probability of infection per transmissible act between a susceptible and an infected person. In two-group models, this is the probability of infection for the group 1 members.
inter.eff	Efficacy of an intervention which affects the per-act probability of infection. Efficacy is defined as 1 - the relative hazard of infection given exposure to the intervention, compared to no exposure.
inter.start	Time step at which the intervention starts, between 1 and the number of time steps specified in the model. This will default to 1 if inter.eff is defined but this parameter is not.
act.rate	Average number of transmissible acts per person per unit time. For two-group models, this is the number of acts per group 1 person per unit time; a balance between the acts in groups 1 and 2 is necessary, and set using the balance parameter (see details).
rec.rate	Average rate of recovery with immunity (in SIR models) or re-susceptibility (in SIS models). The recovery rate is the reciprocal of the disease duration. For two-group models, this is the recovery rate for group 1 persons only. This parameter is only used for SIR and SIS models.
a.rate	Arrival or entry rate. For one-group models, the arrival rate is the rate of new arrivals per person per unit time. For two-group models, the arrival rate is parameterized as a rate per group 1 person per unit time, with the a.rate.g2 rate set as described below.
ds.rate	Departure or exit rate for susceptible persons. For two-group models, it is the rate for the group 1 susceptible persons only.

di.rate	Departure or exit rate for infected persons. For two-group models, it is the rate for the group 1 infected persons only.
dr.rate	Departure or exit rate for recovered persons. For two-group models, it is the rate for the group 1 recovered persons only. This parameter is only used for SIR models.
inf.prob.g2	Probability of infection per transmissible act between a susceptible group 2 person and an infected group 1 person. It is the probability of infection to group 2 members.
act.rate.g2	Average number of transmissible acts per group 2 person per unit time; a balance between the acts in groups 1 and 2 is necessary, and set using the balance parameter (see details).
rec.rate.g2	Average rate of recovery with immunity (in SIR models) or re-susceptibility (in SIS models) for group 2 persons. This parameter is only used for two-group SIR and SIS models.
a.rate.g2	Arrival or entry rate for group 2. This may either be specified numerically as the rate of new arrivals per group 2 persons per unit time, or as NA in which case the group 1 rate, a.rate, governs the group 2 rate. The latter is used when, for example, the first group is conceptualized as female, and the female population size determines the arrival rate. Such arrivals are evenly allocated between the two groups.
ds.rate.g2	Departure or exit rate for group 2 susceptible persons.
di.rate.g2	Departure or exit rate for group 2 infected persons.
dr.rate.g2	Departure or exit rate for group 2 recovered persons. This parameter is only used for SIR model types.
balance	For two-group models, balance the act.rate to the rate set for group 1 (with balance="g1") or group 2 (with balance="g2"). See details.
...	Additional arguments passed to model.

Details

param.icm sets the epidemic parameters for the stochastic individual contact models simulated with the `icm()` function. ICMs support the built-in SI, SIR, and SIS disease types only; for custom or extension epidemic models, use `param.net()` with the network model class.

The model specification will be chosen as a result of the model parameters entered here and the control settings in `control.icm()`. One-group and two-group models are available, where the former assumes a homogeneous mixing in the population and the latter assumes some form of heterogeneous mixing between two distinct partitions in the population (e.g., men and women). Specifying any group two parameters (those with a .g2) implies the simulation of a two-group model. All the parameters for a desired model type must be specified, even if they are zero.

Value

An EpiModel object of class param.icm.

Act Balancing

In two-group models, a balance between the number of acts for group 1 members and those for group 2 members must be maintained. With purely heterogeneous mixing, the product of one group size and act rate must equal the product of the other group size and act rate: $N_1\alpha_1 = N_2\alpha_2$, where N_i is the group size and α_i the group-specific act rate at time t . The balance parameter here specifies which group's act rate should control the others with respect to balancing.

See Also

Use `init.icm()` to specify the initial conditions and `control.icm()` to specify the control settings. Run the parameterized model with `icm()`.

 param.net

Epidemic Parameters for Stochastic Network Models

Description

Sets the epidemic parameters for stochastic network models simulated with `netsim()`.

Usage

```
param.net(
  inf.prob,
  inter.eff,
  inter.start,
  act.rate,
  rec.rate,
  a.rate,
  ds.rate,
  di.rate,
  dr.rate,
  inf.prob.g2,
  rec.rate.g2,
  a.rate.g2,
  ds.rate.g2,
  di.rate.g2,
  dr.rate.g2,
  ...
)
```

Arguments

inf.prob	Probability of infection per transmissible act between a susceptible and an infected person. In two-group models, this is the probability of infection to the group 1 nodes. This may also be a vector of probabilities, with each element corresponding to the probability in that time step of infection (see Time-Varying Parameters below).
----------	---

<code>inter.eff</code>	Efficacy of an intervention which affects the per-act probability of infection. Efficacy is defined as 1 - the relative hazard of infection given exposure to the intervention, compared to no exposure.
<code>inter.start</code>	Time step at which the intervention starts, between 1 and the number of time steps specified in the model. This will default to 1 if <code>inter.eff</code> is defined but this parameter is not.
<code>act.rate</code>	Average number of transmissible acts <i>per partnership</i> per unit time (see <code>act.rate</code> Parameter below). This may also be a vector of rates, with each element corresponding to the rate in that time step of infection (see Time-Varying Parameters below).
<code>rec.rate</code>	Average rate of recovery with immunity (in SIR models) or re-susceptibility (in SIS models). The recovery rate is the reciprocal of the disease duration. For two-group models, this is the recovery rate for group 1 persons only. This parameter is only used for SIR and SIS models. This may also be a vector of rates, with each element corresponding to the rate in that time step of infection (see Time-Varying Parameters below).
<code>a.rate</code>	Arrival or entry rate. For one-group models, the arrival rate is the rate of new arrivals per person per unit time. For two-group models, the arrival rate is parameterized as a rate per group 1 person per unit time, with the <code>a.rate.g2</code> rate set as described below.
<code>ds.rate</code>	Departure or exit rate for susceptible persons. For two-group models, it is the rate for group 1 susceptible persons only.
<code>di.rate</code>	Departure or exit rate for infected persons. For two-group models, it is the rate for group 1 infected persons only.
<code>dr.rate</code>	Departure or exit rate for recovered persons. For two-group models, it is the rate for group 1 recovered persons only. This parameter is only used for SIR models.
<code>inf.prob.g2</code>	Probability of transmission given a transmissible act between a susceptible group 2 person and an infected group 1 person. It is the probability of transmission to group 2 members.
<code>rec.rate.g2</code>	Average rate of recovery with immunity (in SIR models) or re-susceptibility (in SIS models) for group 2 persons. This parameter is only used for two-group SIR and SIS models.
<code>a.rate.g2</code>	Arrival or entry rate for group 2. This may either be specified numerically as the rate of new arrivals per group 2 person per unit time, or as NA, in which case the group 1 rate, <code>a.rate</code> , governs the group 2 rate. The latter is used when, for example, the first group is conceptualized as female, and the female population size determines the arrival rate. Such arrivals are evenly allocated between the two groups.
<code>ds.rate.g2</code>	Departure or exit rate for group 2 susceptible persons.
<code>di.rate.g2</code>	Departure or exit rate for group 2 infected persons.
<code>dr.rate.g2</code>	Departure or exit rate for group 2 recovered persons. This parameter is only used for SIR model types.
<code>...</code>	Additional arguments passed to model.

Details

param.net sets the epidemic parameters for the stochastic network models simulated with the `netsim()` function. Models may use the base types, for which these parameters are used, or new process modules which may use these parameters (but not necessarily). A detailed description of network model parameterization for base models is found in the [Network Modeling for Epidemics](#) tutorials.

For base models, the model specification will be chosen as a result of the model parameters entered here and the control settings in `control.net()`. One-group and two-group models are available, where the latter assumes a heterogeneous mixing between two distinct partitions in the population (e.g., men and women). Specifying any two-group parameters (those with a `.g2`) implies the simulation of a two-group model. All the parameters for a desired model type must be specified, even if they are zero.

Value

An EpiModel object of class param.net.

The act.rate Parameter

A key difference between these network models and DCM/ICM classes is the treatment of transmission events. With DCM and ICM, contacts or partnerships are mathematically instantaneous events: they have no duration in time, and thus no changes may occur within them over time. In contrast, network models allow for partnership durations defined by the dynamic network model, summarized in the model dissolution coefficients calculated in `dissolution_coefs()`. Therefore, the `act.rate` parameter has a different interpretation here, where it is the number of transmissible acts *per partnership* per unit time.

Time-Varying Parameters

The `inf.prob`, `act.rate`, `rec.rate` arguments (and their `.g2` companions) may be specified as time-varying parameters by passing in a vector of probabilities or rates, respectively. The value in each position on the vector then corresponds to the probability or rate at that discrete time step for the infected partner. For example, an `inf.prob` of `c(0.5, 0.5, 0.1)` would simulate a 0.5 transmission probability for the first two time steps of a person's infection, followed by a 0.1 for the third time step. If the infected person has not recovered or exited the population by the fourth time step, the third element in the vector will carry forward until one of those events occurs or the simulation ends. For further examples, see the [Network Modeling for Epidemics](#) tutorials.

Random Parameters

In addition to deterministic parameters in either fixed or time-varying varieties above, one may also include a generator for random parameters. These might include a vector of potential parameter values or a statistical distribution definition; in either case, one draw from the generator would be completed per individual simulation. This is possible by passing a list named `random.params` into `param.net`, with each element of `random.params` a named generator function. See the help page and examples in `generate_random_params()`. A simple factory function for sampling is provided with `param_random()` but any function will do.

Using a Parameter data.frame

It is possible to set input parameters using a specifically formatted `data.frame` object. The first 3 columns of this `data.frame` must be:

- `param`: The name of the parameter. If this is a non-scalar parameter (a vector of length > 1), end the parameter name with the position on the vector (e.g., "p_1", "p_2", ...).
- `value`: the value for the parameter (or the value of the parameter in the Nth position if non-scalar).
- `type`: a character string containing either "numeric", "logical", or "character" to define the parameter object class.

In addition to these 3 columns, the `data.frame` can contain any number of other columns, such as details or source columns to document parameter meta-data. However, these extra columns will not be used by EpiModel.

This `data.frame` is then passed in to `param.net` under a `data.frame.params` argument. Further details and examples are provided in the "Working with Model Parameters in EpiModel" vignette. Note that releases through v2.6.1 documented this argument as `data.frame.parameters`, which was never an accepted name; passing it now produces an error pointing to `data.frame.params`.

Parameters with New Modules

To build original models outside of the base models, new process modules may be constructed to replace the existing modules or to supplement the existing set. These are passed into the control settings in `control.net()`. New modules may use either the existing model parameters named here, an original set of parameters, or a combination of both. The `...` allows the user to pass an arbitrary set of original model parameters into `param.net`. Whereas there are strict checks with default modules for parameter validity, this becomes a user responsibility when using new modules.

See Also

Use `init.net()` to specify the initial conditions and `control.net()` to specify the control settings. Run the parameterized model with `netsim()`.

Examples

```
## Example SIR model parameterization with fixed and random parameters
# Network model estimation
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Random epidemic parameter list (here act.rate values are sampled uniformly
# with helper function param_random, and inf.prob follows a general Beta
# distribution with the parameters shown below)
my_randoms <- list(
  act.rate = param_random(1:3),
  inf.prob = function() rbeta(1, 1, 2)
)
```

```

# Parameters, initial conditions, and control settings
param <- param.net(rec.rate = 0.02, random.params = my_randoms)

# Printing parameters shows both fixed and random parameter functions
param

# Set initial conditions and controls
init <- init.net(i.num = 10, r.num = 0)
control <- control.net(type = "SIR", nsteps = 10, nsims = 3, verbose = FALSE)

# Simulate the model
sim <- netsim(est, param, init, control)

# Printing the sim object shows the randomly drawn values for each simulation
sim

# Parameter sets can be extracted with:
get_param_set(sim)

```

param.net_from_table	<i>Parameters List for Stochastic Network Models from a Formatted Data Frame</i>
----------------------	--

Description

Sets the epidemic parameters for stochastic network models with `netsim()` using a specially formatted data frame of parameters.

Usage

```
param.net_from_table(long.param.df)
```

Arguments

`long.param.df` A data.frame of parameters. See details for the expected format.

Value

A list object of class `param.net`, which can be passed to `netsim()`.

long.param.df

It is possible to set input parameters using a specifically formatted data.frame object. The first 3 columns of this data.frame must be:

- `param`: The name of the parameter. If this is a non-scalar parameter (a vector of length > 1), end the parameter name with the position on the vector (e.g., "p_1", "p_2", ...).

- `value`: the value for the parameter (or the value of the parameter in the Nth position if non-scalar).
- `type`: a character string containing either "numeric", "logical", or "character" to define the parameter object class.

In addition to these 3 columns, the `data.frame` can contain any number of other columns, such as details or source columns to document parameter meta-data. However, these extra columns will not be used by `EpiModel`.

param.net_to_table	<i>Coerce a list of parameters to a long.param.df</i>
--------------------	---

Description

Coerce a list of parameters to a `long.param.df`

Usage

```
param.net_to_table(params)
```

Arguments

params	A list of parameters to be formatted into a <code>long.param.df</code>
--------	--

Value

A `data.frame` of parameters.

long.param.df

It is possible to set input parameters using a specifically formatted `data.frame` object. The first 3 columns of this `data.frame` must be:

- `param`: The name of the parameter. If this is a non-scalar parameter (a vector of length > 1), end the parameter name with the position on the vector (e.g., "p_1", "p_2", ...).
- `value`: the value for the parameter (or the value of the parameter in the Nth position if non-scalar).
- `type`: a character string containing either "numeric", "logical", or "character" to define the parameter object class.

In addition to these 3 columns, the `data.frame` can contain any number of other columns, such as details or source columns to document parameter meta-data. However, these extra columns will not be used by `EpiModel`.

param_random

Create a Value Sampler for Random Parameters

Description

This function returns a 0 argument function that can be used as a generator function in the `random.params` argument of the [param.net\(\)](#) function.

Usage

```
param_random(values, prob = NULL)
```

Arguments

values	A vector of values to sample from.
prob	A vector of weights to use during sampling. If NULL, all values have the same probability of being picked (default = NULL).

Value

A 0 argument generator function to sample one of the values from the values vector.

See Also

[param.net\(\)](#) and [generate_random_params\(\)](#)

Examples

```
# Define function with equal sampling probability
a <- param_random(1:5)
a()

# Define function with unequal sampling probability
b <- param_random(1:5, prob = c(0.1, 0.1, 0.1, 0.1, 0.6))
b()
```

plot.dcm

Plot Data from a Deterministic Compartmental Epidemic Model

Description

Plots epidemiological data from a deterministic compartment epidemic model solved with [dcm\(\)](#).

Usage

```
## S3 method for class 'dcm'
plot(
  x,
  y = NULL,
  popfrac = FALSE,
  run = NULL,
  col = NULL,
  lwd = NULL,
  lty = NULL,
  alpha = 0.9,
  legend = NULL,
  leg.name = NULL,
  leg.cex = 0.8,
  grid = FALSE,
  add = FALSE,
  main = "",
  xlim = NULL,
  ylim = NULL,
  xlab = "Time",
  ylab = NULL,
  ...
)
```

Arguments

<code>x</code>	An EpiModel object of class dcm.
<code>y</code>	Output compartments or flows from dcm object to plot.
<code>popfrac</code>	If TRUE, plot prevalence of values rather than numbers (see details).
<code>run</code>	Run number to plot, for models with multiple runs (default is run 1).
<code>col</code>	Color for lines, either specified as a single color in a standard R color format, or alternatively as a color palette from RColorBrewer::RColorBrewer (see details).
<code>lwd</code>	Line width for output lines.
<code>lty</code>	Line type for output lines.
<code>alpha</code>	Transparency level for lines, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
<code>legend</code>	Type of legend to plot. Values are "n" for no legend, "full" for full legend, and "lim" for limited legend (see details).
<code>leg.name</code>	Character string to use for legend, with the default determined automatically based on the y input.
<code>leg.cex</code>	Legend scale size.
<code>grid</code>	If TRUE, a grid is added to the background of plot (see grid() for details), with default of nx by ny.
<code>add</code>	If TRUE, new plot window is not called and lines are added to existing plot window.

main	a main title for the plot, see also title .
xlim	the x limits (x1, x2) of the plot. Note that $x1 > x2$ is allowed and leads to a ‘reversed axis’. The default value, NULL, indicates that the range of the finite values to be plotted should be used.
ylim	the y limits of the plot; see xlim for default and details.
xlab	a label for the x axis, defaults to a description of x.
ylab	a label for the y axis, defaults to a description of y.
...	Additional arguments to pass to main plot window (see plot.default()).

Details

This function plots epidemiological outcomes from a deterministic compartmental model solved with [dcm\(\)](#). Depending on the number of model runs (sensitivity analyses) and number of groups, the default plot is the fractional proportion of each compartment in the model over time. The specific compartments or flows to plot may be set using the y parameter, and in multiple run models the specific run may also be specified.

The popfrac Argument

Compartment prevalence is the size of a compartment over some denominator. To plot the raw numbers from any compartment, use popfrac=FALSE; this is the default. The popfrac parameter calculates and plots the denominators of all specified compartments using these rules:

1. for one-group models, the prevalence of any compartment is the compartment size divided by the total population size; 2) for two-group models, the prevalence of any compartment is the compartment size divided by the group size.

Color Palettes

Since [dcm\(\)](#) supports multiple run sensitivity models, plotting the results of such models uses a complex color scheme for distinguishing runs. This is accomplished using the [RColorBrewer::RColorBrewer](#) color palettes, which include a range of linked colors using named palettes. For [plot.dcm](#), one may either specify a brewer color palette listed in [RColorBrewer::brewer.pal.info](#), or, alternatively, a vector of standard R colors (named, hexadecimal, or positive integers; see [col2rgb\(\)](#)).

Plot Legends

There are three automatic legend types available, and the legend is added by default for plots. To turn off the legend, use legend="n". To plot a legend with values for every line in a sensitivity analysis, use legend="full". With models with many runs, this may be visually overwhelming. In those cases, use legend="lim" to plot a legend limited to the highest and lowest values of the varying parameter in the model. In cases where the default legend names are not helpful, one may override those names with the leg.name argument.

See Also

[dcm\(\)](#), [RColorBrewer::brewer.pal.info](#)

Examples

```
# Deterministic SIR model with varying act rate
param <- param.dcm(inf.prob = 0.2, act.rate = 1:10,
                   rec.rate = 1/3, a.rate = 0.011, ds.rate = 0.01,
                   di.rate = 0.03, dr.rate = 0.01)
init <- init.dcm(s.num = 1000, i.num = 1, r.num = 0)
control <- control.dcm(type = "SIR", nsteps = 100, dt = 0.25)
mod <- dcm(param, init, control)

# Plot disease prevalence by default
plot(mod)

# Plot prevalence of susceptibles
plot(mod, y = "s.num", popfrac = TRUE, col = "Greys")

# Plot number of susceptibles
plot(mod, y = "s.num", popfrac = FALSE, col = "Greys", grid = TRUE)

# Plot multiple runs of multiple compartments together
plot(mod, y = c("s.num", "i.num"),
      run = 5, xlim = c(0, 50), grid = TRUE)
plot(mod, y = c("s.num", "i.num"),
      run = 10, lty = 2, legend = "n", add = TRUE)
```

plot.epi.data.frame	<i>Plot Epidemic Model Results From a Netsim Data.Frame</i>
---------------------	---

Description

This function is a wrapper around `plot.netsim` accepting a `data.frame` obtain with `as.data.frame(netsim_object)`.

Usage

```
## S3 method for class 'epi.data.frame'
plot(
  x,
  y = NULL,
  sims = NULL,
  legend = NULL,
  mean.col = NULL,
  qnts.col = NULL,
  sim.lwd = NULL,
  sim.col = NULL,
  sim.alpha = NULL,
  popfrac = FALSE,
  qnts = 0.5,
  qnts.alpha = 0.5,
```

```

    qnts.smooth = TRUE,
    mean.line = TRUE,
    mean.smooth = TRUE,
    add = FALSE,
    mean.lwd = 2,
    mean.lty = 1,
    xlim = NULL,
    ylim = NULL,
    main = NULL,
    xlab = NULL,
    ylab = NULL,
    sim.lines = FALSE,
    grid = FALSE,
    leg.cex = 0.8,
    ...
)

```

Arguments

<code>x</code>	A data.frame obtain with <code>as.data.frame(netsim_object)</code> .
<code>y</code>	Output compartments or flows from netsim object to plot.
<code>sims</code>	If type="epi" or "formation", a vector of simulation numbers to plot. The plot then describes those simulations alone: the mean line and quantile bands are calculated over them, not over the full set. If type="network", a single simulation number for which to plot the network, or else "min" to plot the simulation number with the lowest disease prevalence, "max" for the simulation with the highest disease prevalence, or "mean" for the simulation with the prevalence closest to the mean across simulations at the specified time step.
<code>legend</code>	If TRUE, plot default legend.
<code>mean.col</code>	Vector of any standard R color format for mean lines.
<code>qnts.col</code>	Vector of any standard R color format for polygons.
<code>sim.lwd</code>	Line width for simulation lines.
<code>sim.col</code>	Vector of any standard R color format for simulation lines.
<code>sim.alpha</code>	Transparency level for simulation lines, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
<code>popfrac</code>	If TRUE, plot prevalence of values rather than numbers (see details).
<code>qnts</code>	If numeric, plot polygon of simulation quantiles based on the range implied by the argument (see details). If FALSE, suppress polygon from plot.
<code>qnts.alpha</code>	Transparency level for quantile polygons, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
<code>qnts.smooth</code>	If TRUE, use a loess smoother on quantile polygons.
<code>mean.line</code>	If TRUE, plot mean of simulations across time.
<code>mean.smooth</code>	If TRUE, use a loess smoother on the mean line.
<code>add</code>	If TRUE, new plot window is not called and lines are added to existing plot window.

mean.lwd	Line width for mean lines.
mean.lty	Line type for mean lines.
xlim	the x limits (x1, x2) of the plot. Note that $x1 > x2$ is allowed and leads to a 'reversed axis'. The default value, NULL, indicates that the range of the finite values to be plotted should be used.
ylim	the y limits of the plot; see xlim for default and details.
main	a main title for the plot, see also title .
xlab	a label for the x axis, defaults to a description of x.
ylab	a label for the y axis, defaults to a description of y.
sim.lines	If TRUE, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
grid	If TRUE, a grid is added to the background of plot (see grid() for details), with default of nx by ny.
leg.cex	Legend scale size.
...	Additional arguments to pass.

plot.icm

Plot Data from a Stochastic Individual Contact Epidemic Model

Description

Plots epidemiological data from a stochastic individual contact model simulated with [icm\(\)](#).

Usage

```
## S3 method for class 'icm'
plot(
  x,
  y = NULL,
  popfrac = FALSE,
  sim.lines = FALSE,
  sims = NULL,
  sim.col = NULL,
  sim.lwd = NULL,
  sim.alpha = NULL,
  mean.line = TRUE,
  mean.smooth = TRUE,
  mean.col = NULL,
  mean.lwd = 2,
  mean.lty = 1,
  qnts = 0.5,
  qnts.col = NULL,
  qnts.alpha = 0.5,
```

```

    qnts.smooth = TRUE,
    legend = NULL,
    leg.cex = 0.8,
    grid = FALSE,
    add = FALSE,
    xlim = NULL,
    ylim = NULL,
    main = "",
    xlab = "Time",
    ylab = NULL,
    ...
)

```

Arguments

<code>x</code>	An EpiModel model object of class <code>icm</code> .
<code>y</code>	Output compartments or flows from <code>icm</code> object to plot. ———
<code>popfrac</code>	If TRUE, plot prevalence of values rather than numbers (see details).
<code>sim.lines</code>	If TRUE, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
<code>sims</code>	A vector of simulation numbers to plot. The plot then describes those simulations alone: the mean line and quantile bands are calculated over them, not over the full set.
<code>sim.col</code>	Vector of any standard R color format for simulation lines.
<code>sim.lwd</code>	Line width for simulation lines.
<code>sim.alpha</code>	Transparency level for simulation lines, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
<code>mean.line</code>	If TRUE, plot mean of simulations across time.
<code>mean.smooth</code>	If TRUE, use a loess smoother on the mean line.
<code>mean.col</code>	Vector of any standard R color format for mean lines.
<code>mean.lwd</code>	Line width for mean lines.
<code>mean.lty</code>	Line type for mean lines.
<code>qnts</code>	If numeric, plot polygon of simulation quantiles based on the range implied by the argument (see details). If FALSE, suppress polygon from plot.
<code>qnts.col</code>	Vector of any standard R color format for polygons.
<code>qnts.alpha</code>	Transparency level for quantile polygons, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
<code>qnts.smooth</code>	If TRUE, use a loess smoother on quantile polygons.
<code>legend</code>	If TRUE, plot default legend.
<code>leg.cex</code>	Legend scale size.
<code>grid</code>	If TRUE, a grid is added to the background of plot (see <code>grid()</code> for details), with default of <code>nx</code> by <code>ny</code> .

add	If TRUE, new plot window is not called and lines are added to existing plot window.
xlim	the x limits (x1, x2) of the plot. Note that $x1 > x2$ is allowed and leads to a 'reversed axis'. The default value, NULL, indicates that the range of the finite values to be plotted should be used.
ylim	the y limits of the plot; see xlim for default and details.
main	a main title for the plot, see also title .
xlab	a label for the x axis, defaults to a description of x.
ylab	a label for the y axis, defaults to a description of y.
...	Additional arguments to pass.

Details

This plotting function will extract the epidemiological output from a model object of class `icm` and plot the time series data of disease prevalence and other results. The summary statistics that the function calculates and plots are individual simulation lines, means of the individual simulation lines, and quantiles of those individual simulation lines. The mean line, toggled on with `mean.line=TRUE`, is calculated as the row mean across simulations at each time step.

Compartment prevalences are the size of a compartment over some denominator. To plot the raw numbers from any compartment, use `popfrac=FALSE`; this is the default for any plots of flows. The `popfrac` parameter calculates and plots the denominators of all specified compartments using these rules: 1) for one-group models, the prevalence of any compartment is the compartment size divided by the total population size; 2) for two-group models, the prevalence of any compartment is the compartment size divided by the group population size. For any prevalences that are not automatically calculated, the `mutate_epi()` function may be used to add new variables to the `icm` object to plot or analyze.

The quantiles show the range of outcome values within a certain specified quantile range. By default, the interquartile range is shown: that is the middle 50\ middle 95\ where they are plotted by default, specify `qnts=FALSE`.

See Also

[icm\(\)](#)

Examples

```
## Example 1: Plotting multiple compartment values from SIR model
param <- param.icm(inf.prob = 0.5, act.rate = 0.5, rec.rate = 0.02)
init <- init.icm(s.num = 500, i.num = 1, r.num = 0)
control <- control.icm(type = "SIR", nsteps = 100,
                        nsims = 3, verbose = FALSE)
mod <- icm(param, init, control)
plot(mod, grid = TRUE)

## Example 2: Plot only infected with specific output from SI model
param <- param.icm(inf.prob = 0.25, act.rate = 0.25)
```

```

init <- init.icm(s.num = 500, i.num = 10)
control <- control.icm(type = "SI", nsteps = 100,
                      nsims = 3, verbose = FALSE)
mod2 <- icm(param, init, control)

# Plot prevalence
plot(mod2, y = "i.num", mean.line = FALSE, sim.lines = TRUE)

# Plot incidence
par(mfrow = c(1, 2))
plot(mod2, y = "si.flow", mean.smooth = TRUE, grid = TRUE)
plot(mod2, y = "si.flow", qnts.smooth = FALSE, qnts = 1)

```

plot.netdx

Plot Dynamic Network Model Diagnostics

Description

Plots dynamic network model diagnostics calculated in `netdx()`.

Usage

```

## S3 method for class 'netdx'
plot(
  x,
  type = "formation",
  method = "l",
  sims = NULL,
  stats = NULL,
  duration.imputed = TRUE,
  sim.lines = FALSE,
  sim.col = NULL,
  sim.lwd = NULL,
  mean.line = TRUE,
  mean.smooth = TRUE,
  mean.col = NULL,
  mean.lwd = 2,
  mean.lty = 1,
  qnts = 0.5,
  qnts.col = NULL,
  qnts.alpha = 0.5,
  qnts.smooth = TRUE,
  targ.line = TRUE,
  targ.col = NULL,
  targ.lwd = 2,
  targ.lty = 2,

```

```

plots.joined = NULL,
legend = NULL,
grid = FALSE,
xlim = NULL,
xlab = NULL,
ylim = NULL,
ylab = NULL,
momentary = FALSE,
window = NULL,
maxdeg = NULL,
unique.partners = TRUE,
...
)

```

Arguments

<code>x</code>	An EpiModel object of class <code>netdx</code> .
<code>type</code>	Plot type, with options of "formation" for network model formation statistics, "duration" for dissolution model statistics for average edge duration, "dissolution" for dissolution model statistics for proportion of ties dissolved per time step, or "cumldeg" for the cumulative degree distribution.
<code>method</code>	Plot method, with options of "l" for line plots and "b" for box plots. For type = "cumldeg", "b" gives a bar plot of the degree distribution.
<code>sims</code>	A vector of simulation numbers to plot.
<code>stats</code>	Statistics to plot. For type = "formation", stats are among those specified in the call to <code>netdx()</code> ; for type = "duration", "dissolution", stats are among those of the dissolution model (without <code>offset()</code>). The default is to plot all statistics.
<code>duration.imputed</code>	If type = "duration", a logical indicating whether or not to impute starting times for relationships extant at the start of the simulation. Defaults to TRUE when type = "duration".
<code>sim.lines</code>	If TRUE, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
<code>sim.col</code>	Vector of any standard R color format for simulation lines.
<code>sim.lwd</code>	Line width for simulation lines.
<code>mean.line</code>	If TRUE, plot mean of simulations across time.
<code>mean.smooth</code>	If TRUE, use a loess smoother on the mean line.
<code>mean.col</code>	Vector of any standard R color format for mean lines.
<code>mean.lwd</code>	Line width for mean lines.
<code>mean.lty</code>	Line type for mean lines.
<code>qnts</code>	If numeric, plot polygon of simulation quantiles based on the range implied by the argument (see details). If FALSE, suppress polygon from plot.
<code>qnts.col</code>	Vector of any standard R color format for polygons.

qnts.alpha	Transparency level for quantile polygons, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
qnts.smooth	If TRUE, use a loess smoother on quantile polygons.
targ.line	If TRUE, plot target or expected value line for the statistic of interest.
targ.col	Vector of standard R colors for target statistic lines, with default colors based on RColorBrewer color palettes.
targ.lwd	Line width for the line showing the target statistic values.
targ.lty	Line type for the line showing the target statistic values.
plots.joined	If TRUE, combine all statistics in one plot, versus one plot per statistic if FALSE.
legend	If TRUE, plot default legend.
grid	If TRUE, a grid is added to the background of plot (see <code>grid()</code> for details), with default of nx by ny.
xlim	the x limits (x1, x2) of the plot. Note that x1 > x2 is allowed and leads to a 'reversed axis'. The default value, NULL, indicates that the range of the <code>finite</code> values to be plotted should be used.
xlab	a label for the x axis, defaults to a description of x.
ylim	the y limits of the plot; see xlim for default and details.
ylab	a label for the y axis, defaults to a description of y.
momentary	If TRUE, add the momentary degree distribution to the cumulative degree plot for comparison. Used only when type = "cumldeg".
window	A vector of length 2 giving the first and last time step over which degrees are counted, with the default of the full simulation. Used only when type = "cumldeg".
maxdeg	Highest degree to show on the x-axis, with degrees above it collapsed into a final "maxdeg+" category. Used only when type = "cumldeg".
unique.partners	If TRUE, a dyad that forms, dissolves, and then re-forms within the window counts as one cumulative partner rather than two. Used only when type = "cumldeg".
...	Additional arguments to pass.

Details

The `plot` function for `netdx` objects will generate plots of two types of model diagnostic statistics that run as part of the diagnostic tools within that function. The `formation` plot shows the summary statistics requested in `nwstats.formula`, where the default includes those statistics in the network model formation formula specified in the original call to `netest()`.

The `duration` plot shows the average age of existing edges at each time step, up until the maximum time step requested. The age is used as an estimator of the average duration of edges in the equilibrium state. When `duration.imputed = FALSE`, edges that exist at the beginning of the simulation are assumed to start with an age of 1, yielding a burn-in period before the observed mean approaches

its target. When `duration.imputed = TRUE`, expected ages prior to the start of the simulation are calculated from the dissolution model, typically eliminating the need for a burn-in period.

The dissolution plot shows the proportion of the extant ties that are dissolved at each time step, up until the maximum time step requested. Typically, the proportion of ties that are dissolved is the reciprocal of the mean relational duration. This plot thus contains similar information to that in the duration plot, but should reach its expected value more quickly, since it is not subject to censoring.

The `plots.joined` argument will control whether the statistics are joined in one plot or plotted separately, assuming there are multiple statistics in the model. The default is based on the number of network statistics requested. The layout of the separate plots within the larger plot window is also based on the number of statistics.

The `cumldeg` plot shows the cumulative degree distribution: the proportion of nodes with each count of partners accumulated over the simulation, rather than the count held at one point in time. It requires a `netdx` object run with `keep.tedgelist = TRUE`, since it is calculated from the timed edgelist with `get_degree_dist()`. Setting `momentary = TRUE` adds the momentary degree distribution over the same window, calculated from the same edgelist, which is the distribution summarized by the `degree(k)` terms of the formation plot. Two models with the same mean degree and mean partnership duration have the same mean cumulative degree even when their momentary degree distributions differ, as the example below shows for models with high and low concurrency. Legend entries report the mean of each distribution.

Because the cumulative degree grows with the length of the observation window, use `window` to restrict it to a substantively meaningful period when comparing across models. The lines or bars show the mean across simulations, with `qnts` giving the inter-simulation quantile band; the `targ.line`, `mean.smooth`, `qnts.smooth`, and `plots.joined` arguments do not apply to this plot type.

See Also

`netdx()`, and `get_degree_dist()` for the degree distributions behind `type = "cumldeg"`.

Examples

```
## Not run:
# Network initialization and model parameterization
nw <- network_initialize(n = 500)
nw <- set_vertex_attribute(nw, "sex", rbinom(500, 1, 0.5))
formation <- ~edges + nodematch("sex")
target.stats <- c(500, 300)
coef.diss <- dissolution_coefs(dissolution = ~offset(edges) +
                             offset(nodematch("sex")), duration = c(50, 40))

# Estimate the model
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Static diagnostics
dx1 <- netdx(est, nsims = 1e4, dynamic = FALSE,
            nwstats.formula = ~edges + meandeg + concurrent +
                           nodefactor("sex", levels = NULL) +
                           nodematch("sex"))
dx1
```

```

# Plot diagnostics
plot(dx1)
plot(dx1, stats = c("edges", "concurrent"), mean.col = "black",
      sim.lines = TRUE, plots.joined = FALSE)
plot(dx1, stats = "edges", method = "b",
      col = "seagreen3", grid = TRUE)

# Dynamic diagnostics
dx2 <- netdx(est, nsims = 10, nsteps = 500,
             nwstats.formula = ~edges + meandeg + concurrent +
                               nodefactor("sex", levels = NULL) +
                               nodematch("sex"))

dx2

# Formation statistics plots, joined and separate
plot(dx2, grid = TRUE)
plot(dx2, type = "formation", plots.joined = TRUE)
plot(dx2, type = "formation", sims = 1, plots.joined = TRUE,
      qnts = FALSE, sim.lines = TRUE, mean.line = FALSE)
plot(dx2, type = "formation", plots.joined = FALSE,
      stats = c("edges", "concurrent"), grid = TRUE)

plot(dx2, method = "b", col = "bisque", grid = TRUE)
plot(dx2, method = "b", stats = "meandeg", col = "dodgerblue")

# Duration statistics plot
par(mfrow = c(1, 2))
# With duration imputed
plot(dx2, type = "duration", sim.line = TRUE, sim.lwd = 0.3,
      targ.lty = 1, targ.lwd = 0.5)
# Without duration imputed
plot(dx2, type = "duration", sim.line = TRUE, sim.lwd = 0.3,
      targ.lty = 1, targ.lwd = 0.5, duration.imputed = FALSE)

# Dissolution statistics plot
plot(dx2, type = "dissolution", qnts = 0.25, grid = TRUE)
plot(dx2, type = "dissolution", method = "b", col = "pink1")

# Cumulative degree distribution: two models with the same mean degree and
# the same mean partnership duration, differing only in how much
# concurrency (overlapping partnerships) they allow
nw <- network_initialize(n = 500)
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 25)

est.hi <- netest(nw, formation = ~edges + concurrent,
                 target.stats = c(200, 150), coef.diss = coef.diss,
                 verbose = FALSE)
est.lo <- netest(nw, formation = ~edges + concurrent,
                 target.stats = c(200, 40), coef.diss = coef.diss,
                 verbose = FALSE)

dx.hi <- netdx(est.hi, nsims = 5, nsteps = 250, keep.tedgelist = TRUE,
               nwstats.formula = ~edges + concurrent + degree(0:3))

```

```

dx.lo <- netdx(est.lo, nsims = 5, nsteps = 250, keep.tedgelist = TRUE,
              nwstats.formula = ~edges + concurrent + degree(0:3))

# The momentary degree distributions are far apart: about half the nodes in
# the high-concurrency model have no partner at any given time and a quarter
# have two, while most nodes in the low-concurrency model have exactly one.
# Over 250 time steps both accumulate about 8.5 partners per node.
par(mfrow = c(1, 2))
plot(dx.hi, type = "cumldeg", momentary = TRUE, xlim = c(0, 25),
     main = "High concurrency")
plot(dx.lo, type = "cumldeg", momentary = TRUE, xlim = c(0, 25),
     main = "Low concurrency")

# As bars, with the tail of the distribution collapsed into one category
# and the inter-simulation range shown
par(mfrow = c(1, 1))
plot(dx.hi, type = "cumldeg", method = "b", maxdeg = 15, qnts = 0.95)

# Partners accumulated over the last 100 steps only
plot(dx.hi, type = "cumldeg", window = c(150, 250), momentary = TRUE)

# The plotted distributions are returned invisibly
dd <- plot(dx.hi, type = "cumldeg")
tapply(dd$degree * dd$prop, dd$sim, sum)

## End(Not run)

```

plot.netsim

Plot Data from a Stochastic Network Epidemic Model

Description

Plots epidemiological and network data from a stochastic network model simulated with `netsim()`.

Usage

```

## S3 method for class 'netsim'
plot(
  x,
  type = "epi",
  y = NULL,
  popfrac = FALSE,
  sim.lines = FALSE,
  sims = NULL,
  sim.col = NULL,
  sim.lwd = NULL,
  sim.alpha = NULL,
  mean.line = TRUE,

```

```

mean.smooth = TRUE,
mean.col = NULL,
mean.lwd = 2,
mean.lty = 1,
qnts = 0.5,
qnts.col = NULL,
qnts.alpha = 0.5,
qnts.smooth = TRUE,
legend = NULL,
leg.cex = 0.8,
grid = FALSE,
add = FALSE,
network = 1,
at = 1,
col.status = FALSE,
shp.g2 = NULL,
vertex.cex = NULL,
stats = NULL,
targ.line = TRUE,
targ.col = NULL,
targ.lwd = 2,
targ.lty = 2,
plots.joined = NULL,
duration.imputed = TRUE,
method = "l",
main = NULL,
xlim = NULL,
xlab = NULL,
ylim = NULL,
ylab = NULL,
...
)

```

Arguments

<code>x</code>	An EpiModel model object of class <code>netsim</code> .
<code>type</code>	Type of plot: "epi" for epidemic model results, "network" for a static network plot (<code>plot.network</code>), or "formation", "duration", or "dissolution" for network formation, duration, or dissolution statistics.
<code>y</code>	Output compartments or flows from <code>netsim</code> object to plot.
<code>popfrac</code>	If TRUE, plot prevalence of values rather than numbers (see details).
<code>sim.lines</code>	If TRUE, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
<code>sims</code>	If <code>type="epi"</code> or "formation", a vector of simulation numbers to plot. The plot then describes those simulations alone: the mean line and quantile bands are calculated over them, not over the full set. If <code>type="network"</code> , a single simulation number for which to plot the network, or else "min" to plot the simulation number with the lowest disease prevalence, "max" for the simulation with

	the highest disease prevalence, or "mean" for the simulation with the prevalence closest to the mean across simulations at the specified time step.
sim.col	Vector of any standard R color format for simulation lines.
sim.lwd	Line width for simulation lines.
sim.alpha	Transparency level for simulation lines, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
mean.line	If TRUE, plot mean of simulations across time.
mean.smooth	If TRUE, use a loess smoother on the mean line.
mean.col	Vector of any standard R color format for mean lines.
mean.lwd	Line width for mean lines.
mean.lty	Line type for mean lines.
qnts	If numeric, plot polygon of simulation quantiles based on the range implied by the argument (see details). If FALSE, suppress polygon from plot.
qnts.col	Vector of any standard R color format for polygons.
qnts.alpha	Transparency level for quantile polygons, where 0 = transparent and 1 = opaque (see <code>adjustcolor</code> function).
qnts.smooth	If TRUE, use a loess smoother on quantile polygons.
legend	If TRUE, plot default legend.
leg.cex	Legend scale size.
grid	If TRUE, a grid is added to the background of plot (see <code>grid()</code> for details), with default of nx by ny.
add	If TRUE, new plot window is not called and lines are added to existing plot window.
network	Network number, for simulations with multiple networks representing the population.
at	If type = "network", time step for network graph.
col.status	If TRUE and type="network", automatic disease status colors (blue = susceptible, red = infected, green = recovered).
shp.g2	If type = "network" and x is for a two-group model, shapes for the Group 2 vertices, with acceptable inputs of "triangle" and "square". Group 1 vertices will remain circles.
vertex.cex	Relative size of plotted vertices if type="network", with implicit default of 1.
stats	If type="formation", "duration", "dissolution", statistics to plot. For type = "formation", stats are among those specified in <code>nwstats.formula</code> of <code>control.net()</code> ; for type = "duration", "dissolution", stats are among those of the dissolution model (without <code>offset()</code>). The default is to plot all statistics.
targ.line	If TRUE, plot target or expected value line for the statistic of interest.
targ.col	Vector of standard R colors for target statistic lines, with default colors based on RColorBrewer color palettes.
targ.lwd	Line width for the line showing the target statistic values.

targ.lty	Line type for the line showing the target statistic values.
plots.joined	If TRUE and type="formation", "duration", "dissolution", combine all statistics in one plot, versus one plot per statistic if FALSE.
duration.imputed	If type = "duration", a logical indicating whether or not to impute starting times for relationships extant at the start of the simulation. Defaults to TRUE when type = "duration".
method	Plot method for type="formation", "duration", "dissolution", with options of "l" for line plots and "b" for box plots.
main	a main title for the plot, see also title .
xlim	the x limits (x1, x2) of the plot. Note that x1 > x2 is allowed and leads to a 'reversed axis'. The default value, NULL, indicates that the range of the finite values to be plotted should be used.
xlab	a label for the x axis, defaults to a description of x.
ylim	the y limits of the plot; see xlim for default and details.
ylab	a label for the y axis, defaults to a description of y.
...	Additional arguments to pass.

Details

This plot function can produce three types of plots with a stochastic network model simulated through [netsim\(\)](#):

1. type="epi": epidemic model results (e.g., disease prevalence and incidence) may be plotted.
2. type="network": a static network plot will be generated. A static network plot of a dynamic network is a cross-sectional extraction of that dynamic network at a specific time point. This plotting function wraps the [network::plot.network](#) function in the network package. Consult the help page for plot.network for all of the plotting parameters. In addition, four plotting parameters specific to netsim plots are available: sim, at, col.status, and shp.g2.
3. type="formation": summary network statistics related to the network model formation are plotted. These plots are similar to the formation plots for netdx objects. When running a netsim simulation, one must specify there that save.nwstats=TRUE; the plot here will then show the network statistics requested explicitly in nwstats.formula, or will use the formation formula set in netest otherwise.
4. type="duration", "dissolution": as in [plot.netdx\(\)](#); supported in plot.netsim only when the dissolution model is ~offset(edges), tergmLite is FALSE, and save.network is TRUE.

When type="epi", this plotting function will extract the epidemiological output from a model object of class netsim and plot the time series data of disease prevalence and other results. The summary statistics that the function calculates and plots are individual simulation lines, means of the individual simulation lines, and quantiles of those individual simulation lines. The mean line, toggled on with mean.line=TRUE, is calculated as the row mean across simulations at each time step.

Compartment prevalences are the size of a compartment over some denominator. To plot the raw numbers from any compartment, use `popfrac=FALSE`; this is the default for any plots of flows. The `popfrac` parameter calculates and plots the denominators of all specified compartments using these rules: 1) for one-group models, the prevalence of any compartment is the compartment size divided by the total population size; 2) for two-group models, the prevalence of any compartment is the compartment size divided by the group population size. For any prevalences that are not automatically calculated, the `mutate_epi()` function may be used to add new variables to the `netsim` object to plot or analyze.

The quantiles show the range of outcome values within a certain specified quantile range. By default, the interquartile range is shown: that is the middle 50% of the data. This is specified by `qnts=0.5`. To show the middle 95% of the data, specify `qnts=0.95`. To toggle off the polygons where they are plotted by default, specify `qnts=FALSE`.

When `type="network"`, this function will plot cross sections of the simulated networks at specified time steps. Because it is only possible to plot one time step from one simulation at a time, it is necessary to enter these in the `at` and `sims` parameters. To aid in visualizing representative and extreme simulations at specific time steps, the `sims` parameter may be set to "mean" to plot the simulation in which the disease prevalence is closest to the average across all simulations, "min" to plot the simulation in which the prevalence is lowest, and "max" to plot the simulation in which the prevalence is highest.

See Also

`network:plot.network`, `mutate_epi()`

Examples

```
## SI Model without Network Feedback
# Initialize network and set network model parameters
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)

# Estimate the network model
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Simulate the epidemic model
param <- param.net(Inf.prob = 0.3, Inf.prob.g2 = 0.15)
init <- init.net(i.num = 10, i.num.g2 = 10)
control <- control.net(type = "SI", nsteps = 20, nsims = 3,
                      verbose = FALSE, save.nwstats = TRUE,
                      nwstats.formula = ~edges + meandeg + concurrent)
mod <- netsim(est, param, init, control)

# Plot epidemic trajectory
plot(mod)
plot(mod, type = "epi", grid = TRUE)
plot(mod, type = "epi", popfrac = TRUE)
plot(mod, type = "epi", y = "si.flow", qnts = 1, ylim = c(0, 4))
```

```

# Plot static networks
par(mar = c(0, 0, 0, 0))
plot(mod, type = "network", vertex.cex = 1.5)

# Automatic coloring of infected nodes as red
par(mfrow = c(1, 2), mar = c(0, 0, 2, 0))
plot(mod, type = "network", main = "Min Prev | Time 50",
      col.status = TRUE, at = 20, sims = "min", vertex.cex = 1.25)
plot(mod, type = "network", main = "Max Prev | Time 50",
      col.status = TRUE, at = 20, sims = "max", vertex.cex = 1.25)

# Automatic shape by group number (circle = group 1)
par(mar = c(0, 0, 0, 0))
plot(mod, type = "network", at = 20, col.status = TRUE,
      shp.g2 = "square")
plot(mod, type = "network", at = 20, col.status = TRUE,
      shp.g2 = "triangle", vertex.cex = 2)

# Plot formation statistics
par(mfrow = c(1,1), mar = c(3,3,1,1), mgp = c(2,1,0))
plot(mod, type = "formation", grid = TRUE)
plot(mod, type = "formation", plots.joined = FALSE)
plot(mod, type = "formation", sims = 2:3)
plot(mod, type = "formation", plots.joined = FALSE,
      stats = c("edges", "concurrent"))
plot(mod, type = "formation", stats = "meandeg",
      mean.lwd = 1, qnts.col = "seagreen", mean.col = "black")

```

plot.transmat

Plot transmat Infection Tree in Three Styles

Description

Plots the transmission matrix tree from from `get_transmat` in one of three styles: a phylogram, a directed network, or a transmission timeline.

Usage

```

## S3 method for class 'transmat'
plot(x, style = c("phylo", "network", "transmissionTimeline"), ...)

```

Arguments

<code>x</code>	A <code>transmat()</code> object to be plotted.
<code>style</code>	Character name of plot style. One of "phylo", "network", or "transmissionTimeline".
<code>...</code>	Additional plot arguments to be passed to lower-level plot functions (<code>plot.network</code> , <code>plot.phylo</code> , or <code>transmissionTimeline</code>).

Details

The phylo plot requires the ape package. The transmissionTimeline plot requires that the ndtv package.

See Also

`network::plot.network`, `ape::plot.phylo()`, `ndtv::transmissionTimeline()`.

<code>print.dcm</code>	<i>Print Method for DCM Objects</i>
------------------------	-------------------------------------

Description

Prints a summary of a deterministic compartmental model object, including the model type, number of runs, time steps, model parameters, and output variable names.

Usage

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

Arguments

<code>x</code>	An object of class <code>dcm</code> , from <code>dcm()</code> .
<code>...</code>	Additional arguments (currently ignored).

Details

Given a `dcm` object, `print.dcm` displays:

- **Simulation summary:** model class, model type (e.g., SI, SIR, SIS; omitted for new/custom models), number of runs, number of time steps, and number of groups.
- **Model parameters:** all parameters passed via `param.dcm()`, excluding internal bookkeeping parameters (`groups`, `vital`).
- **Model output:** the names of all epidemic output variables stored in the model.

print.icm	<i>Print Method for ICM Objects</i>
-----------	-------------------------------------

Description

Prints a summary of a stochastic individual contact model object, including the model type, number of simulations, time steps, model parameters, and output variable names.

Usage

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

Arguments

x	An object of class icm, from <code>icm()</code> .
...	Additional arguments (currently ignored).

Details

Given an icm object, `print.icm` displays:

- **Simulation summary:** model class, model type (e.g., SI, SIR, SIS), number of simulations, number of time steps, and number of groups.
- **Model parameters:** all parameters passed via `param.icm()`, excluding internal bookkeeping parameters (groups, vital).
- **Model output:** the names of all epidemic output variables stored in the model.

print.netdx	<i>Utility Function for Printing netdx Object</i>
-------------	---

Description

Prints basic information and statistics from a netdx object.

Usage

```
## S3 method for class 'netdx'
print(x, digits = 3, ...)
```

Arguments

x	an object of class netdx
digits	number of digits to print in statistics tables
...	additional arguments (currently ignored)

Details

Given a `netdx` object, `print.netdx` prints the diagnostic method (static/dynamic), number of simulations, and (if dynamic) the number of time steps per simulation used in generating the `netdx` object, as well as printing the formation statistics table and (if present) the duration and dissolution statistics tables. The statistics tables are interpreted as follows.

Each row has the name of a particular network statistic. In the formation table, these correspond to actual network statistics in the obvious way. In the duration and dissolution tables, these correspond to dissolution model dyad types: in a homogeneous dissolution model, all dyads are of the edges type; in a heterogeneous dissolution model, a dyad with a nonzero `nodematch` or `nodemix` change statistic in the dissolution model has type equal to that statistic, and has type equal to edges otherwise. The statistics of interest for the duration and dissolution tables are, respectively, the mean age of extant edges and the edge dissolution rate, broken down by dissolution model dyad type. (The current convention is to treat the mean age and dissolution rate for a particular dissolution dyad type as 0 on time steps with no edges of that type; this behavior may be changed in the future.)

The columns are named `Target`, `Sim Mean`, `Pct Diff`, `Sim SE`, `Z Score`, `SD(Sim Means)`, and `SD(Statistic)`. The `Sim Mean` column refers to the mean statistic value, across all time steps in all simulations in the dynamic case, and across all sampled networks in all simulations in the static case. The `Sim SE` column refers to the standard error in the mean, estimated using `coda::effectiveSize`. The `Target` column indicates the target value (if present) for the statistic, and the `Pct Diff` column gives $(\text{Sim Mean} - \text{Target})/\text{Target}$ when `Target` is present. The `Z Score` column gives $(\text{Sim Mean} - \text{Target})/(\text{Sim SE})$. The `SD(Sim Means)` column gives the empirical standard deviation across simulations of the mean statistic value within simulation, and `SD(Statistic)` gives the empirical standard deviation of the statistic value across all the simulated data.

```
print.netsim
```

Print Method for Network Model Simulations

Description

Prints a detailed summary of a stochastic network model simulation object, including simulation metadata, model parameters, output variables, and (optionally) network formation, duration, and dissolution statistics with target comparisons.

Usage

```
## S3 method for class 'netsim'
print(x, nwstats = TRUE, digits = 3, network = 1, ...)
```

Arguments

<code>x</code>	An object of class <code>netsim</code> , from <code>netsim()</code> .
<code>nwstats</code>	If <code>TRUE</code> (the default), print network statistics tables (formation, duration, and dissolution) when available.
<code>digits</code>	Number of digits to print in the network statistics tables.

network	Integer index of the network for which to display statistics, for multi-network models. Default is 1.
...	Additional arguments (currently ignored).

Details

Given a netsim object, `print.netsim` displays the following sections:

Simulation summary: model class, model type (e.g., SI, SIR, SIS), number of simulations, number of time steps, and number of network groups.

Model parameters: printed via `print.param.net()`, showing fixed and (if applicable) random parameters.

Model functions: for extension models (type = NULL), lists the names of all custom module functions.

Model output: the names of all epidemic output variables, saved network objects, transmission matrices, and any other saved elements.

Network statistics (when `nwstats = TRUE` and network statistics were saved during the simulation):

The formation, duration, and dissolution statistics tables are computed and displayed in the same way as for `print.netdx()`. Each table contains the columns Target, Sim Mean, Pct Diff, Sim SE, Z Score, SD(Sim Means), and SD(Statistic). The Sim Mean column is the mean statistic value across all time steps and simulations. Sim SE is the standard error estimated using `codar::effectiveSize`. Pct Diff gives $(\text{Sim Mean} - \text{Target}) / \text{Target}$ and Z Score gives $(\text{Sim Mean} - \text{Target}) / \text{Sim SE}$ when a target is available. SD(Sim Means) is the standard deviation of per-simulation means, and SD(Statistic) is the overall standard deviation of the statistic.

Formation statistics: each row corresponds to a network statistic from the formation formula (e.g., edges, nodematch), compared against the target statistics from network estimation. These statistics assess whether the network structure is maintained at target levels during the epidemic simulation (which is particularly important in open-population models where demographic turnover can shift network structure).

Duration statistics: each row corresponds to a dissolution model dyad type. In a homogeneous dissolution model (`~offset(edges)`), all dyads are of the edges type. The statistic of interest is the mean age of extant edges, compared against the target duration from `dissolution_coefs()`.

Dissolution statistics: same row structure as the duration table. The statistic of interest is the edge dissolution rate (proportion of edges dissolving per time step), compared against $1 / \text{target duration}$.

Duration and dissolution tables are only available when `control$save.diss.stats = TRUE`, `control$save.network = TRUE`, `control$tergmLite = FALSE`, and the dissolution formula is `~offset(edges)`. When these conditions are not met, a note listing the requirements is printed instead.

See Also

`netsim()`, `print.netdx()`, `summary.netsim()`.

Description

These functions return the forward or backward reachable nodes of a set of nodes in a network over a time period. Forward reachability answers "who could a node have infected?"; backward reachability answers "who could have infected this node?". Both operate on a cumulative edgelist, the same structure produced by `get_cumulative_edgelist()` / `get_cumulative_edgelists_df()` from a netsim run, or by `as_cumulative_edgelist()` applied to an external networkDynamic object.

Warning: these functions ignore nodes without edges in the period of interest. See the Number of Nodes section for details. The distance to each node can be back-calculated using the length of the reachable set at each time step and the fact that the reachable sets are ordered by the time to arrival. These functions are much faster than iterating `tsna::tPath()`.

Usage

```
get_forward_reachable(
  el_cuml,
  from_step,
  to_step,
  nodes = NULL,
  dense_optim = "auto"
)

get_backward_reachable(
  el_cuml,
  from_step,
  to_step,
  nodes = NULL,
  dense_optim = "auto"
)
```

Arguments

<code>el_cuml</code>	a cumulative edgelist object. That is a data.frame with at least columns: head, tail, start and stop. Start and stop are inclusive.
<code>from_step</code>	the beginning of the time period.
<code>to_step</code>	the end of the time period.
<code>nodes</code>	the subset of nodes to calculate the FRP for. (default = NULL, all nodes)
<code>dense_optim</code>	pre-process the adjacency list to speed up the computations on dense networks. "auto" (default), enable the optimisation when <code>n_edges > n_nodes</code> . "yes" always enables and "no" always disables. The overhead of the optimization is not worth it on sparse networks.

Value

A named list containing: `reached`: the set of reachable nodes for each of the nodes. `lengths`: A matrix of `length(nodes)` rows and one column per timestep + 1 with the length of the reachable set at each step from `from_step - 1` to `to_step`. The first column is always one as the set of reachables at the beginning is just the node itself.

Number of Nodes

To speed up the calculations and lower the memory usage, these functions only take into account nodes with edges in the cumulative edgelist over the period of interest. The nodes are identified in the `reached` and `lengths` sublists by names (e.g. `node_1093`). Nodes without any edges are therefore not calculated as the only node they reach is themselves (length of 1). Take this fact into account when exploring the distribution of Forward Reachable Paths for example. As the nodes with `FRP == 1` are not in the output.

Time and Memory Use

These functions may be used to efficiently calculate multiple sets of reachable nodes. As cumulative edgelists are way smaller than full `networkDynamic` objects, these functions are suited for large and dense networks. Also, as long as the size of the nodes set is greater than 5, these functions are faster than iterating over `tsna::tPath`.

Displaying Progress

These functions are using the [progressr package](#) to display its progression. Use `progressr::with_progress({ fwd_reach <- get_forward_reachable(el, from = 1, to = 260) })` to display the progress bar. Or see the [progressr package](#) for more information and customization.

See Also

`get_cumulative_edgelist()` / `get_cumulative_edgelists_df()` for the typical input source from a `netsim` run; `as_cumulative_edgelist()` to derive a cumulative edgelist from a `networkDynamic` object; `dedup_cumulative_edgelist()` when the input has been concatenated from multiple sources. `vignette("network-objects", package = "EpiModel")` walks through the full lifecycle.

Other cumulative_edgelist: `as_cumulative_edgelist()`, `dedup_cumulative_edgelist()`, `get_cumulative_degree()`, `get_cumulative_edgelist()`, `get_cumulative_edgelists_df()`, `get_partners()`, `update_cumulative_edgelist()`

Examples

```
## Not run:

# load a network dynamic object
nd <- readRDS("nd_obj.Rds")
# convert it to a cumulative edgelist
el_cuml <- as_cumulative_edgelist(nd)

# sample 100 node indexes
nnodes <- max(el_cuml$head, el_cuml$tail)
nodes <- sample(nnodes, 100)
```

```

# `get_forward_reachable` uses steps [from_step, to_step] inclusive
el_fwd <- get_forward_reachable(el_cum1, 1, 52, nodes)[["reached"]]

# check if the results are consistent with `tsna::tPath`
nodes <- strsplit(names(el_fwd), "_")
for (i in seq_along(el_fwd)) {
  node <- as.integer(nodes[[i]][2])
  t_fwd <- tsna::tPath(
    nd, v = node,
    start = 1, end = 52 + 1, # tPath works from [start, end) right exclusive
    direction = "fwd"
  )

  t_fwd_set <- which(t_fwd$tdist < Inf)
  if(!setequal(el_fwd[[i]], t_fwd_set))
    stop("Mismatch on node: ", node)
}

# Backward:
el_bkwd <- get_backward_reachable(el_cum1, 1, 52, nodes = 1)[["reached"]]
nodes <- strsplit(names(el_bkwd), "_")
t_bkwd <- tsna::tPath(
  nd, v = nodes[i][2],
  start = 1, end = 52 + 1,
  direction = "bkwd", type = "latest.depart"
)
t_bkwd_set <- which(t_bkwd$tdist < Inf)
setequal(el_bkwd[[1]], t_bkwd_set)

## End(Not run)

```

record_attr_history *Record Attribute History*

Description

This function records values specific to a time-step and a group of nodes. In the records, the `posit_ids` are converted to `unique_ids` which allows the recording of data for nodes that are no longer in the network by the end of the run. The records are stored in `dat[["attr.history"]]` where `dat` is the main `netsim_dat` class object, and can be accessed from the `netsim` object with `get_attr_history`.

Usage

```
record_attr_history(dat, at, attribute, posit_ids, values)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
<code>at</code>	The time where the recording happens.
<code>attribute</code>	The name of the value to record.
<code>posit_ids</code>	A numeric vector of <code>posit_ids</code> to which the measure applies. (see <code>get_posit_ids</code>).
<code>values</code>	The values to be recorded.

Details

See the "Time-Varying Parameters" section of the "Working With Model Parameters" vignette.

Value

The updated `netsim_dat` main list object.

Examples

```
## Not run:
# This function must be used inside a custom module
dat <- record_attr_history(dat, at, "attr_1", get_posit_ids(dat), 5)
some_nodes <- get_posit_ids(dat)
some_nodes <- some_nodes[runif(length(some_nodes)) < 0.2]
dat <- record_attr_history(
  dat, at,
  "attr_2",
  some_nodes,
  rnorm(length(some_nodes))
)

## End(Not run)
```

<code>record_raw_object</code>	<i>Record an Arbitrary Object During a Simulation</i>
--------------------------------	---

Description

This function records any object during a simulation to allow its inspection afterward. The records are stored in `dat[["raw.records"]]` during the simulation, where `dat` is the main `netsim_dat` class object, and in the `netsim` object under the `raw.records` collections::queue object.

Usage

```
record_raw_object(dat, at, label, object)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
at	The time where the recording happens.
label	The name to give to the recorded object.
object	The object to be recorded.

Details

See the "Time-Varying Parameters" section of the "Working With Model Parameters" vignette.

Value

The updated netsim_dat main list object.

Examples

```
## Not run:

dat <- record_raw_object(dat, at, "a.df", data.frame(x = 2:200))
dat <- record_raw_object(dat, at, "a.message", "I recorded something")

## End(Not run)
```

set_current_timestep *Set the Current Timestep*

Description

Changes the current timestep in the netsim_dat object. Use with caution. This function exists to work around unforeseen corner cases. In most situation, increment_timestep is preferred.

Usage

```
set_current_timestep(dat, timestep)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
timestep	The new value for the timestep.

Value

The updated netsim_dat main list object.

Mutability

This DOES NOT modify the netsim_dat object in place. The result must be assigned back to dat in order to be registered: `dat <- increment_timestep(dat)`.

set_transmat	<i>Save Transmission Matrix</i>
--------------	---------------------------------

Description

This function appends the transmission matrix created during `infection.net` and `infection.2g.net`.

Usage

```
set_transmat(dat, del, at)
```

Arguments

dat	Main netsim_dat object containing a networkDynamic object and other initialization information passed from <code>netsim()</code> .
del	Discordant edgelist created within <code>infection.net()</code> and <code>infection.2g.net()</code> .
at	Current time step.

Details

This internal function works within the parent `infection.net()` functions to save the transmission matrix created at time step at to the main netsim_dat class object dat.

Value

The updated netsim_dat main list object.

set_vertex_attribute	<i>Set Vertex Attribute on Network Object</i>
----------------------	---

Description

Sets a vertex attribute on an object of class network. This function simplifies the related function in the network package.

Usage

```
set_vertex_attribute(x, attrname, value, v = NULL)
```

Arguments

x	An object of class network.
attrname	The name of the attribute to set.
value	A vector of values of the attribute to be set.
v	IDs for the vertices whose attributes are to be altered.

Details

This function is used in EpiModel workflows to set vertex attributes on an initialized empty network object (see [network_initialize\(\)](#)).

Value

Returns an object of class network.

Examples

```
nw <- network_initialize(100)
nw <- set_vertex_attribute(nw, "age", runif(100, 15, 65))
nw
```

summary.dcm

*Summary Model Statistics***Description**

Extracts and prints model statistics solved with dcm.

Usage

```
## S3 method for class 'dcm'
summary(object, at, run = 1, digits = 3, ...)
```

Arguments

object	An EpiModel object of class dcm.
at	Time step for model statistics. Must be one of the times the model was solved over, held in control\$timesteps, which for a model with a non-integer dt or an explicit vector of times in nsteps need be neither integer nor evenly spaced.
run	Model run number, for dcm class models with multiple runs (sensitivity analyses).
digits	Number of significant digits to print.
...	Additional summary function arguments (not used).

Details

This function provides summary statistics for the main epidemiological outcomes (state and transition size and prevalence) from a dcm model. Time-specific summary measures are provided, so it is necessary to input a time of interest. For multiple-run models (sensitivity analyses), input a model run number. See examples below.

See Also

[dcm\(\)](#)

Examples

```
## Deterministic SIR model with varying act.rate
param <- param.dcm(inf.prob = 0.2, act.rate = 2:4, rec.rate = 1/3,
                  a.rate = 0.011, ds.rate = 0.01,
                  di.rate = 0.03, dr.rate = 0.01)
init <- init.dcm(s.num = 1000, i.num = 1, r.num = 0)
control <- control.dcm(type = "SIR", nsteps = 50)
mod <- dcm(param, init, control)
summary(mod, at = 25, run = 1)
summary(mod, at = 25, run = 3)
summary(mod, at = 26, run = 3)
```

summary.icm	<i>Summary Model Statistics</i>
-------------	---------------------------------

Description

Extracts and prints model statistics simulated with icm.

Usage

```
## S3 method for class 'icm'
summary(object, at, digits = 3, ...)
```

Arguments

- object An EpiModel object of class icm.
- at Time step for model statistics.
- digits Number of significant digits to print.
- ... Additional summary function arguments.

Details

This function provides summary statistics for the main epidemiological outcomes (state and transition size and prevalence) from an icm model. Time-specific summary measures are provided, so it is necessary to input a time of interest.

See Also

[icm\(\)](#)

Examples

```
## Stochastic ICM SI model with 3 simulations
param <- param.icm(inf.prob = 0.2, act.rate = 1)
init <- init.icm(s.num = 500, i.num = 1)
control <- control.icm(type = "SI", nsteps = 50,
                       nsims = 5, verbose = FALSE)
mod <- icm(param, init, control)
summary(mod, at = 25)
summary(mod, at = 50)
```

summary.netsim	<i>Summary Model Statistics</i>
----------------	---------------------------------

Description

Extracts and prints model statistics simulated with netsim.

Usage

```
## S3 method for class 'netsim'
summary(object, at, digits = 3, ...)
```

Arguments

- object An EpiModel object of class netsim.
- at Time step for model statistics.
- digits Number of significant digits to print.
- ... Additional summary function arguments.

Details

This function provides summary statistics for the main epidemiological outcomes (state and transition size and prevalence) from a netsim model. Time-specific summary measures are provided, so it is necessary to input a time of interest.

See Also

[netsim\(\)](#)

Examples

```
## Not run:
## SI Model without Network Feedback
# Initialize network and set network model parameters
nw <- network_initialize(n = 100)
nw <- set_vertex_attribute(nw, "group", rep(1:2, each = 50))
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 20)

# Estimate the ERGM models (see help for netest)
est1 <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)

# Parameters, initial conditions, and controls for model
param <- param.net(inf.prob = 0.3, inf.prob.g2 = 0.15)
init <- init.net(i.num = 10, i.num.g2 = 10)
control <- control.net(type = "SI", nsteps = 100, nsims = 5, verbose.int = 0)

# Run the model simulation
mod <- netsim(est1, param, init, control)

summary(mod, at = 1)
summary(mod, at = 50)
summary(mod, at = 100)

## End(Not run)
```

trim_netest

Function to Reduce the Size of a netest Object

Description

Trims formula environments from the netest object. Optionally converts the newnetwork element of the netest object to a networkLite class, and removes the fit element (if present) from the netest object.

Usage

```
trim_netest(
  object,
  as.networkLite = TRUE,
  keep.fit = FALSE,
  keep = character(0)
)
```

Arguments

<code>object</code>	A netest class object.
<code>as.networkLite</code>	If TRUE, converts <code>object\$newnetwork</code> to a <code>networkLite</code> .
<code>keep.fit</code>	If FALSE, removes the <code>object\$fit</code> (if present) on the netest object.
<code>keep</code>	Character vector of object names to keep in formula environments. By default, all objects are removed.

Details

With larger, more complex network structures with epidemic models, it is generally useful to reduce the memory footprint of the fitted TERGM model object (estimated with `netest()`). This utility function removes all but the bare essentials needed for simulating a network model with `netsim()`.

The function always trims the environments of `object$constraints` and `object$coef.diss$dissolution`.

When both `edapprox = TRUE` and `nested.edapprox = TRUE` in the `netest` call, also trims the environments of `object$formula` and `object$formation`.

When both `edapprox = TRUE` and `nested.edapprox = FALSE` in the `netest` call, also trims the environments of `object$formula`, `environment(object$formation)$formation`, and `environment(object$formation)$dissolution`.

When `edapprox = FALSE` in the `netest` call, also trims the environments of `object$formation`, `environment(object$formula)$formation` and `environment(object$formula)$dissolution`.

By default all objects are removed from these trimmed environments. Specific objects may be retained by passing their names as the `keep` argument. For the output of `trim_netest` to be usable in `netsim()` simulation, any objects referenced in the formulas should be included in the `keep` argument.

If `as.networkLite = TRUE`, converts `object$newnetwork` to a `networkLite` object. If `keep.fit = FALSE`, removes `fit` (if present) from object.

Value

A netest object with formula environments trimmed, optionally with the `newnetwork` element converted to a `networkLite` and the `fit` element removed.

Examples

```
nw <- network_initialize(n = 100)
formation <- ~edges + concurrent
target.stats <- c(50, 25)
coef.diss <- dissolution_coefs(dissolution = ~offset(edges), duration = 10)
est <- netest(nw, formation, target.stats, coef.diss,
              set.control.ergm = control.ergm(MCMC.burnin = 1e5,
                                              MCMC.interval = 1000))

print(object.size(est), units = "KB")

est.small <- trim_netest(est)
print(object.size(est.small), units = "KB")
```

truncate_sim	<i>Truncate Simulation Time Series</i>
--------------	--

Description

Left-truncates simulation epidemiological summary statistics and network statistics at a specified time step.

Usage

```
truncate_sim(x, at, reset.time)

## S3 method for class 'dcm'
truncate_sim(x, at, reset.time = TRUE)

## S3 method for class 'icm'
truncate_sim(x, at, reset.time = TRUE)

## S3 method for class 'netsim'
truncate_sim(x, at, reset.time = TRUE)
```

Arguments

x	Object of class dcm, netsim, or icm.
at	Time step at which to left-truncate the time series.
reset.time	If TRUE, the time step sequence in the truncated model will be reset to start at 1. If FALSE, the original time step values will be preserved. Default is TRUE.

Details

This function would be used when running a follow-up simulation from time steps b to c after a burn-in period from time a to b, where the final time window of interest for data analysis is b to c only.

Value

The updated object of class dcm, netsim, or icm.

Examples

```
# DCM examples
param <- param.dcm(inf.prob = 0.2, act.rate = 0.25)
init <- init.dcm(s.num = 500, i.num = 1)
control <- control.dcm(type = "SI", nsteps = 200)
mod1 <- dcm(param, init, control)
plot(mod1)

# Reset time
```

```

mod2a <- truncate_sim(mod1, at = 150)
plot(mod2a)
head(as.data.frame(mod2a))

# Do not reset time
mod2b <- truncate_sim(mod1, at = 150, reset.time = FALSE)
plot(mod2b)
head(as.data.frame(mod2b))

# ICM example
param <- param.icm(inf.prob = 0.2, act.rate = 0.25)
init <- init.icm(s.num = 500, i.num = 1)
control <- control.icm(type = "SI", nsteps = 200, nsims = 1)
mod1 <- icm(param, init, control)

# Reset time
mod2a <- truncate_sim(mod1, at = 150)
plot(mod2a)
head(as.data.frame(mod2a))

# Do not reset time
mod2b <- truncate_sim(mod1, at = 150, reset.time = FALSE)
plot(mod2b)
head(as.data.frame(mod2b))

```

unique_id-tools

Convert Unique Identifiers to/from Positional Identifiers

Description

EpiModel refers to its nodes either by positional identifiers (`posit_ids`), which describe the position of a node in the `attr` vector, or by unique identifiers (`unique_ids`), which allow references to nodes even after they are deactivated.

Usage

```
get_unique_ids(dat, posit_ids = NULL)
```

```
get_posit_ids(dat, unique_ids = NULL)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
<code>posit_ids</code>	A vector of node positional identifiers (default = <code>NULL</code>).
<code>unique_ids</code>	A vector of node unique identifiers (default = <code>NULL</code>).

Value

A vector of unique or positional identifiers.

All elements

When `unique_ids` or `posit_ids` is `NULL` (default) the full list of positional IDs or unique IDs is returned.

Deactivated nodes

When providing `unique_ids` of deactivated nodes to `get_posit_ids`, NAs are returned instead and a warning is produced.

update_cumulative_edgelist

Update a Cumulative Edgelist of the Specified Network

Description

Records the current network state into the cumulative edgelist, stamping newly observed edges with their start step and edges that have just dissolved with their stop step. Returns `dat` unchanged when cumulative-edgelist tracking is disabled.

Usage

```
update_cumulative_edgelist(dat, network, truncate = 0)
```

Arguments

<code>dat</code>	Main <code>netsim_dat</code> object containing a <code>networkDynamic</code> object and other initialization information passed from <code>netsim()</code> .
<code>network</code>	Numerical index of the network for which the cumulative edgelist will be updated. (May be > 1 for models with multiple overlapping networks.)
<code>truncate</code>	After how many time steps a partnership that is no longer active should be removed from the output. See the Truncation section.

Details

Calling this function is a no-op unless `control.net(cumulative.edgelist = TRUE)`. With tracking enabled, the built-in network-resimulation module `resim_nets()` calls this once per network at every time step, using `truncate = control$truncate.el.cuml`. Custom modules that mutate the network outside the TERGM machinery (e.g., forming or dissolving edges directly) should call this manually after the change so the cumulative record stays consistent.

Value

The updated `netsim_dat` main list object.

Truncation

To avoid storing a cumulative edgelist too long, the `truncate` parameter defines a number of steps after which an edge that is no longer active is dropped from the stored history. When `truncate = Inf`, no edges are ever removed. When `truncate = 0` (the default), only currently active edges are kept; this is useful for tracking the start step of each active edge while keeping memory low.

See Also

`control.net()` (`cumulative.edgelist`, `truncate.el.cum1`), `get_cumulative_edgelist()` to read the result.

Other cumulative_edgelist: `as_cumulative_edgelist()`, `dedup_cumulative_edgelist()`, `get_cumulative_degree()`, `get_cumulative_edgelist()`, `get_cumulative_edgelists_df()`, `get_partners()`, `reachable-nodes`

Examples

```
## Not run:
# Inside a custom module, after editing the network outside the TERGM:
for (n in seq_len(dat$num.nw)) {
  dat <- update_cumulative_edgelist(dat, n,
    truncate = get_control(dat, "truncate.el.cum1"))
}

## End(Not run)
```

update_dissolution	<i>Adjust Dissolution Component of Network Model Fit</i>
--------------------	--

Description

Adjusts the dissolution component of a dynamic ERGM fit using the `netest()` function with the edges dissolution approximation method.

Usage

```
update_dissolution(old.netest, new.coef.diss, nested.edapprox = TRUE)
```

Arguments

<code>old.netest</code>	An object of class <code>netest</code> , from the <code>netest()</code> function.
<code>new.coef.diss</code>	An object of class <code>disscoef</code> , from the <code>dissolution_coefs()</code> function.
<code>nested.edapprox</code>	Logical, only used when <code>edapprox = TRUE</code> . When <code>TRUE</code> (the default), the dissolution model must be an initial segment of the formation model; when <code>FALSE</code> , the dissolution terms need not be nested (see details in <code>netest()</code>).

Details

Fitting an ERGM is a computationally intensive process when the model includes dyad dependent terms. With the edges dissolution approximation method of Carnegie et al, the coefficients for a temporal ERGM are approximated by fitting a static ERGM and adjusting the formation coefficients to account for edge dissolution. This function provides a very efficient method to adjust the coefficients of that model when one wants to use a different dissolution model; a typical use case may be to fit several different models with different average edge durations as targets. The example below exhibits that case.

Value

An updated network model object of class `netest`.

Examples

```
## Not run:
nw <- network_initialize(n = 1000)

# Two dissolutions: an average duration of 300 versus 200
diss.300 <- dissolution_coefs(~offset(edges), 300, 0.001)
diss.200 <- dissolution_coefs(~offset(edges), 200, 0.001)

# Fit the two reference models
est300 <- netest(nw = nw,
                 formation = ~edges,
                 target.stats = c(500),
                 coef.diss = diss.300)

est200 <- netest(nw = nw,
                 formation = ~edges,
                 target.stats = c(500),
                 coef.diss = diss.200)

# Alternatively, update the 300 model with the 200 coefficients
est200.compare <- update_dissolution(est300, diss.200)

identical(est200$coef.form, est200.compare$coef.form)

## End(Not run)
```

Description

Updates epidemic model parameters originally set with `param.net()` and adds new parameters.

Usage

```
update_params(param, new.param.list)
```

Arguments

`param` Object of class `param.net`, output from function of same name.
`new.param.list` Named list of new parameters to add to original parameters.

Details

This function can update any original parameters specified with `param.net()` and add new parameters. This function would be used if the inputs to `param.net()` were a long list of fixed model parameters that needed supplemental replacements or additions for particular model runs (e.g., changing an intervention efficacy parameter but leaving all other parameters fixed).

The `new.param.list` object should be a named list object containing named parameters matching those already in `x` (in which case those original parameter values will be replaced) or not matching (in which case new parameters will be added to `param`).

`update_params` modifies a `param.net` object **before** it is passed to `netsim()`; it is the recommended helper for tweaking parameters outside of a running simulation. To modify parameters from *inside* a module function during a simulation, use `set_param()` on the live `dat` object instead. There is no analogous `update_controls()` or `update_inits()` helper: for a `control.net` or `init.net` object, edit the list directly (e.g., `ctrl$steps <- 1000`) or rebuild it with a fresh call to the constructor. For *scheduled* mid-simulation changes to parameters or control settings, see the `.param.updater.list` argument to `param.net()` and the `.control.updater.list` argument to `control.net()`.

Value

An updated list object of class `param.net`, which can be passed to the `EpiModel` function `netsim()`.

See Also

`set_param()` for editing parameters inside a module function during a simulation; `param.net()` for the parameter constructor; `netsim()` for running a simulation.

Examples

```
x <- param.net(inf.prob = 0.5, act.rate = 2)
y <- list(inf.prob = 0.75, dx.rate = 0.2)
z <- update_params(x, y)
print(z)
```

use_scenario	<i>Apply a scenario object to a param.net object</i>
--------------	--

Description

Apply a scenario object to a param.net object

Usage

```
use_scenario(param, scenario)
```

Arguments

param	Object of class param.net, output from function of same name.
scenario	a scenario object usually created from a data.frame of scenarios using the create_scenario_list function. See the vignette "network-model-scenarios".

Value

An updated list object of class param.net, which can be passed to the EpiModel function [netsim\(\)](#).

scenario

A scenario is a list containing an "id" field, the name of the scenario and a ".param.updater.list" containing a list of updaters that modifies the parameters of the model at given time steps. If a scenario contains a parameter not defined in the param object, an error will be produced. See the vignette "model-parameters" for the technical detail of their implementation.

Index

- * **colorUtils**
 - color_tea, 20
- * **cumulative_edgelist**
 - as_cumulative_edgelist, 18
 - dedup_cumulative_edgelist, 35
 - get_cumulative_degree, 49
 - get_cumulative_edgelist, 50
 - get_cumulative_edgelist_df, 51
 - get_partners, 63
 - reachable-nodes, 140
 - update_cumulative_edgelist, 153
- * **extract**
 - as.data.frame.dcm, 11
 - as.data.frame.icm, 12
 - as.data.frame.netdx, 14
 - get_degree_dist, 54
 - get_network, 58
 - get_nwstats, 61
 - get_sims, 65
 - is.transmat, 75
 - merge.icm, 79
 - merge.netsim, 80
 - summary.dcm, 146
 - summary.icm, 147
 - summary.netsim, 148
- * **model**
 - dcm, 33
 - icm, 67
 - netest, 94
 - netstim, 99
- * **netUtils**
 - check_degdist_bal, 19
 - dissolution_coefs, 39
 - edgelist_censor, 41
- * **package**
 - EpiModel-package, 4
- * **parameterization**
 - control.dcm, 23
 - control.icm, 25
 - control.net, 26
 - init.dcm, 70
 - init.icm, 71
 - init.net, 72
 - param.dcm, 105
 - param.icm, 108
 - param.net, 111
- * **plot**
 - comp_plot, 21
 - geom_bands, 46
 - plot.dcm, 117
 - plot.epi.data.frame, 120
 - plot.icm, 122
 - plot.netdx, 125
 - plot.netstim, 130
- absdiffby (InitErgmTerm.absdiffby), 73
- absdiffnodemix
 - (InitErgmTerm.absdiffnodemix), 74
- add_attr (net-accessor), 86
- add_control (net-accessor), 86
- add_epi (net-accessor), 86
- add_init (net-accessor), 86
- add_param (net-accessor), 86
- add_vertices, 8
- ape::phylo, 17
- ape::plot.phylo(), 17, 136
- ape::read.tree(), 17
- append_attr (net-accessor), 86
- append_attr(), 6
- append_core_attr (net-accessor), 86
- append_core_attr(), 6
- apportion_lr, 9
- arrivals.icm(), 83
- arrivals.net, 28, 101
- arrivals.net(), 84, 89
- arrive_nodes, 10
- as.data.frame.dcm, 11
- as.data.frame.dcm(), 6, 34

- `as.data.frame.default()`, [11](#), [13](#), [14](#)
- `as.data.frame.icm`, [12](#)
- `as.data.frame.icm()`, [6](#), [68](#)
- `as.data.frame.netdx`, [14](#)
- `as.data.frame.netdx()`, [92](#), [93](#)
- `as.data.frame.netsim`
 (`as.data.frame.icm`), [12](#)
- `as.data.frame.netsim()`, [6](#), [100](#), [102](#)
- `as.epi.data.frame`, [15](#)
- `as.network.transmat`, [16](#)
- `as.phylo.transmat`, [16](#)
- `as.phylo.transmat()`, [7](#)
- `as_cumulative_edgelist`, [18](#)
- `as_cumulative_edgelist()`, [36](#), [50–52](#), [64](#),
 [140](#), [141](#), [154](#)
- `as_tibble_edgelist`, [19](#)
-
- `check_degdist_bal`, [19](#)
- `coda::effectiveSize`, [138](#), [139](#)
- `col2rgb()`, [119](#)
- `color_tea`, [20](#)
- `comp_plot`, [21](#)
- `comp_plot()`, [34](#), [68](#)
- `control.dcm`, [23](#)
- `control.dcm()`, [6](#), [34](#), [70](#), [71](#), [107](#), [108](#)
- `control.icm`, [25](#)
- `control.icm()`, [6](#), [68](#), [72](#), [110](#), [111](#)
- `control.net`, [26](#), [98](#), [99](#), [101](#)
- `control.net()`, [6](#), [21](#), [25](#), [32](#), [51](#), [52](#), [59](#), [73](#),
 [80](#), [82](#), [84](#), [88](#), [89](#), [100](#), [113](#), [114](#),
 [132](#), [154](#), [156](#)
- `create_dat_object`, [32](#)
- `create_scenario_list`, [33](#)
-
- `dcm`, [33](#), [44](#)
- `dcm()`, [6](#), [11](#), [23](#), [24](#), [70](#), [71](#), [107](#), [108](#), [117](#),
 [119](#), [136](#), [147](#)
- `dedup_cumulative_edgelist`, [35](#)
- `dedup_cumulative_edgelist()`, [18](#), [50–52](#),
 [64](#), [141](#), [154](#)
- `delete_edges`, [36](#)
- `delete_vertices`, [37](#)
- `depart_nodes`, [38](#)
- `departures.icm`, [82](#)
- `departures.net`, [28](#), [101](#)
- `departures.net()`, [84](#)
- `deSolve::ode`, [23](#)
- `discord_edgelist()`, [6](#)
- `dissolution_coefs`, [39](#), [93](#), [95](#), [97](#), [98](#)
-
- `dissolution_coefs()`, [6](#), [113](#), [139](#), [154](#)
-
- `edgelist_censor`, [41](#)
- `edges_correct`, [27](#)
- `EpiModel` (`EpiModel-package`), [4](#)
- `EpiModel-package`, [4](#)
- `epiweb`, [42](#)
- `ergm.ego::control.ergm.ego`, [97](#)
- `ergm::control.ergm`, [97](#)
- `ergm::control.simulate.formula`, [92](#)
-
- `finite`, [119](#), [122](#), [124](#), [127](#), [133](#)
- `fuzzynodematch`
 (`InitErgmTerm.fuzzynodematch`),
 [74](#)
-
- `generate_random_params`, [44](#)
- `generate_random_params()`, [113](#), [117](#)
- `geom_bands`, [46](#)
- `get_adj_list`, [47](#)
- `get_attr` (`net-accessor`), [86](#)
- `get_attr()`, [6](#)
- `get_attr_history`, [48](#)
- `get_attr_list` (`net-accessor`), [86](#)
- `get_backward_reachable`, [28](#), [101](#)
- `get_backward_reachable`
 (`reachable-nodes`), [140](#)
- `get_backward_reachable()`, [7](#), [18](#), [36](#)
- `get_connected_nodes`, [48](#)
- `get_control` (`net-accessor`), [86](#)
- `get_control_list` (`net-accessor`), [86](#)
- `get_core_attributes` (`net-accessor`), [86](#)
- `get_cumulative_degree`, [28](#), [49](#), [101](#)
- `get_cumulative_degree()`, [18](#), [36](#), [51](#), [52](#),
 [64](#), [141](#), [154](#)
- `get_cumulative_edgelist`, [28](#), [50](#), [101](#)
- `get_cumulative_edgelist()`, [18](#), [36](#), [50](#), [52](#),
 [64](#), [140](#), [141](#), [154](#)
- `get_cumulative_edgelist_df`, [28](#), [29](#), [51](#),
 [101](#)
- `get_cumulative_edgelist_df()`, [18](#), [36](#),
 [50](#), [51](#), [64](#), [140](#), [141](#), [154](#)
- `get_current_timestep`, [53](#)
- `get_degree`, [53](#)
- `get_degree_dist`, [54](#), [93](#)
- `get_degree_dist()`, [128](#)
- `get_edgelist`, [56](#)
- `get_edgelist()`, [51](#)
- `get_edgelist_df`, [57](#)

- `get_edgelist_df()`, 51
- `get_epi (net-accessor)`, 86
- `get_epi()`, 6
- `get_epi_list (net-accessor)`, 86
- `get_formula_term_attr`, 57
- `get_forward_reachable`, 28, 101
- `get_forward_reachable`
 (reachable-nodes), 140
- `get_forward_reachable()`, 7, 18, 36
- `get_init (net-accessor)`, 86
- `get_init_list (net-accessor)`, 86
- `get_network`, 58, 102
- `get_network()`, 7
- `get_network_attributes`, 60
- `get_network_control (net-accessor)`, 86
- `get_network_term_attr`, 60
- `get_nwstats`, 61
- `get_nwstats()`, 7
- `get_param (net-accessor)`, 86
- `get_param()`, 6
- `get_param_list (net-accessor)`, 86
- `get_param_set`, 62
- `get_partners`, 28, 63, 101
- `get_partners()`, 7, 18, 36, 49–52, 141, 154
- `get_posit_ids (unique_id-tools)`, 152
- `get_posit_ids()`, 49, 64
- `get_sims`, 65
- `get_sims()`, 6
- `get_subnet_adj_list`, 66
- `get_transmat`, 102
- `get_transmat (is.transmat)`, 75
- `get_transmat()`, 7, 17
- `get_unique_ids (unique_id-tools)`, 152
- `get_unique_ids()`, 51, 52, 64
- `get_vertex_attribute`, 67
- `grid()`, 118, 122, 123, 127, 132
- `icm`, 44, 67
- `icm()`, 6, 25, 46, 71, 72, 79, 82, 110, 111, 122, 124, 137, 148
- `increment_timestep`, 69
- `infection.2g.net()`, 145
- `infection.icm()`, 82
- `infection.net`, 28, 31, 101
- `infection.net()`, 84, 145
- `init.dcm`, 70
- `init.dcm()`, 5, 24, 34, 108
- `init.icm`, 71
- `init.icm()`, 5, 25, 68, 82, 111
- `init.net`, 32, 72, 98, 99
- `init.net()`, 5, 32, 71, 83, 88, 89, 114
- `InitErgmTerm.absdiffby`, 73
- `InitErgmTerm.absdiffnodemix`, 74
- `InitErgmTerm.fuzzynodematch`, 74
- `initialize.icm()`, 82
- `initialize.net`, 28
- `initialize.net()`, 83, 89
- `is.transmat`, 75
- `is_active_posit_ids`, 76
- `is_active_unique_ids`, 77
- `make_restart_point`, 77
- `merge.icm`, 79
- `merge.icm()`, 6
- `merge.netsim`, 80
- `merge.netsim()`, 6
- `modules.icm`, 82
- `modules.net`, 83
- `modules.net()`, 100, 101
- `multilayer`, 29, 30, 85, 101
- `multilayer()`, 6
- `mutate_epi`, 85, 102
- `mutate_epi()`, 6, 124, 134
- `ndtv::transmissionTimeline()`, 136
- `net-accessor`, 86
- `netdx`, 30, 90, 94, 96, 98, 99, 102
- `netdx()`, 6, 54, 55, 58, 59, 61, 125, 126, 128
- `netest`, 31, 90–93, 94, 99, 101, 102
- `netest()`, 6, 39, 83–85, 127, 150, 154
- `netsim`, 26, 30, 32, 90, 93, 94, 96, 98, 99
- `netsim()`, 6, 21, 32, 46, 49, 50, 52, 53, 56–61, 64, 68, 69, 72, 73, 75–77, 80–83, 85, 86, 88, 89, 104, 111, 113–115, 130, 133, 138, 139, 143–145, 148, 150, 152, 153, 156, 157
- `network::network`, 16
- `network::plot.network`, 133, 134, 136
- `network_initialize`, 97, 103
- `network_initialize()`, 67, 146
- `nwupdate.net`, 28, 104
- `overwrite_attrs`, 104
- `padded_vector`, 105
- `param.dcm`, 44, 105
- `param.dcm()`, 5, 24, 34, 70, 71, 136
- `param.icm`, 43, 44, 108

param.icm(), 5, 25, 68, 72, 82, 137
 param.net, 32, 98, 99, 111
 param.net(), 5, 32, 44, 73, 80, 84, 88, 89, 110, 117, 155, 156
 param.net_from_table, 115
 param.net_to_table, 116
 param_random, 117
 param_random(), 113
 plot.dcm, 117
 plot.dcm(), 34
 plot.default(), 119
 plot.epi.data.frame, 120
 plot.icm, 122
 plot.icm(), 68
 plot.netdx, 92, 93, 125
 plot.netdx(), 55, 133
 plot.netsim, 130
 plot.netsim(), 21, 102
 plot.transmat, 135
 prevalence.icm(), 83
 prevalence.net, 28, 101
 prevalence.net(), 84
 print.dcm, 136
 print.icm, 137
 print.netdx, 92, 93, 137
 print.netdx(), 139
 print.netsim, 138

 RColorBrewer::brewer.pal.info, 119
 RColorBrewer::RColorBrewer, 118, 119
 reachable-nodes, 140
 record_attr_history, 142
 record_raw_object, 143
 recovery.icm(), 82
 recovery.net, 28, 101
 recovery.net(), 84
 remove_node_attr (net-accessor), 86
 resim_nets, 28, 100
 resim_nets(), 84, 153

 set_attr (net-accessor), 86
 set_attr(), 6
 set_control (net-accessor), 86
 set_current_timestep, 144
 set_epi (net-accessor), 86
 set_epi(), 6
 set_init (net-accessor), 86
 set_param (net-accessor), 86
 set_param(), 156

 set_transmat, 145
 set_transmat(), 6
 set_vertex_attribute, 145
 shiny::runApp, 43
 sim_nets_t1, 30
 summary.dcm, 146
 summary.dcm(), 34
 summary.icm, 147
 summary.icm(), 68
 summary.netsim, 148
 summary.netsim(), 102, 139
 summary_nets, 28

 tergm::control.simulate.formula.tergm, 92
 tergm::control.tergm, 97
 title, 119, 122, 124, 133
 transmat (is.transmat), 75
 transmat(), 17, 135
 trim_netest, 149
 truncate_sim, 151

 unique_id-tools, 152
 update_cumulative_edgelist, 27, 28, 153
 update_cumulative_edgelist(), 18, 36, 50–52, 64, 141
 update_dissolution, 154
 update_params, 155
 update_params(), 89
 use_scenario, 157

 verbose.net, 28
 verbose.net(), 84