Package ‘EpiModel’

July 19, 2022

Version 2.3.0

Date 2022-07-18

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>).

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

License GPL-3

URL `http://www.epimodel.org/`

BugReports `https://github.com/EpiModel/EpiModel/issues`

Depends R (>= 3.5), deSolve (>= 1.21), networkDynamic (>= 0.11.2), tergm (>= 4.1.0), statnet.common (>= 4.6.0)

Imports graphics, grDevices, stats, utils, doParallel, ergm (>= 4.2.2), foreach, network (>= 1.17.2), RColorBrewer, ape, lazyeval, ggplot2, tibble, methods, dplyr

Suggests knitr, ndtv, rmarkdown, shiny, testthat, tidyr

VignetteBuilder knitr

LinkingTo Rcpp, ergm

RoxygenNote 7.2.0

Encoding UTF-8

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 2022-07-19 03:30:06 UTC

R topics documented:

`EpiModel-package` *Mathematical Modeling of Infectious Disease Dynamics*

Description

Package:	EpiModel
Type:	Package
Version:	2.3.0
Date:	2022-07-18
License:	GPL-3
LazyLoad:	yes

Details

The EpiModel software package provides tools for building, solving, and visualizing mathematical models of infectious disease dynamics. These tools allow users to simulate epidemic models in multiple frameworks for both pedagogical purposes ("base models") and novel research purposes ("extension models").

Model Classes and Infectious Disease Types

EpiModel provides functionality for three classes of epidemic models:

- **Deterministic Compartmental Models:** these continuous-time models are solved using ordinary differential equations. EpiModel allows for easy specification of sensitivity analyses to compare multiple scenarios of the same model across different parameter values.
- **Stochastic Individual Contact Models:** a novel class of individual-based, microsimulation models that were developed to add random variation in all components of the transmission system, from infection to recovery to vital dynamics (arrivals and departures).
- **Stochastic Network Models:** with the underlying statistical framework of temporal exponential random graph models (ERGMs) recently developed in the **Statnet** suite of software in R, network models over epidemics simulate edge (e.g., partnership) formation and dissolution stochastically according to a specified statistical model, with disease spread across that network.

EpiModel supports three infectious disease types to be run across all of the three classes.

- **Susceptible-Infectious (SI):** a two-state disease in which there is life-long infection without recovery. HIV/AIDS is one example, although for this case it is common to model infection stages as separate compartments.
- **Susceptible-Infectious-Recovered (SIR):** a three-stage disease in which one has life-long recovery with immunity after infection. Measles is one example, but modern models for the disease also require consideration of vaccination patterns in the population.

- **Susceptible-Infectious-Susceptible (SIS):** a two-stage disease in which one may transition back and forth from the susceptible to infected states throughout life. Examples include bacterial sexually transmitted diseases like gonorrhea.

These basic disease types may be extended in any arbitrarily complex way to simulate specific diseases for research questions.

Model Parameterization and Simulation

EpiModel uses three model setup functions for each model class to input the necessary parameters, initial conditions, and control settings:

- `param.dcm`, `param.icm`, and `param.net` are used to input epidemic parameters for each of the three model classes. Parameters include the rate of contacts or acts between actors, the probability of transmission per contact, and recovery and demographic rates for models that include those transitions.
- `init.dcm`, `init.icm`, and `init.net` are used to input the initial conditions for each class. The main conditions are limited to the numbers or, if applicable, the specific agents in the population who are infected or recovered at the simulation outset.
- `control.dcm`, `control.icm`, and `control.net` are used to specify the remaining control settings for each simulation. The core controls for base model types include the disease type, number of time steps, and number of simulations. Controls are also used to input new model functions (for DCMs) and new model modules (for ICMs and network models) to allow the user to simulate fully original epidemic models in EpiModel. See the documentation for the specific control functions help pages.

With the models parameterized, the functions for simulating epidemic models are:

- `dcm` for deterministic compartmental models.
- `icm` for individual contact models.
- Network models are simulated in a three-step process:
  1. `netest` estimates the statistical model for the network structure itself (i.e., how partnerships form and dissolve over time given the parameterization of those processes). This function is a wrapper around the `ergm` and `stergm` functions in the `ergm` and `tergm` packages. The current statistical framework for model simulation is called "egocentric inference": target statistics summarizing these formation and dissolution processes collected from an egocentric sample of the population.
  2. `netdx` runs diagnostics on the dynamic model fit by simulating the base network over time to ensure the model fits the targets for formation and dissolution.
  3. `netsim` simulates the stochastic network epidemic models, with a given network model fit in `netest`. Here the function requires this model fit object along with the parameters, initial conditions, and control settings as defined above.

References

The EpiModel website is at `http://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.

We have also developed two extension packages for modeling specific disease dynamics. For HIV and bacterial sexually transmitted infections, we have developed `EpiModelHIV`, which is available on Github at `https://github.com/EpiModel/EpiModelHIV`. For COVID-19, we have developed `EpiModelCOVID`, which is available at `https://github.com/EpiModel/EpiModelCOVID`.

`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

`el`	A two-column matrix of current edges (edgelist) with an attribute variable `n` containing the total current network size.
`nv`	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)
```

`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, ...)
```

Arguments

`x`	An `EpiModel` object of class `dcm`.
`row.names`	See `as.data.frame.default`.
`optional`	See `as.data.frame.default`.
`run`	Run number for model; used for multiple-run sensitivity models. If not specified, will output data from all runs in a stacked data frame.
`...`	See `as.data.frame.default`.

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` *Extract Model Data for Stochastic Models*

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,
  qval,
  ...
)

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

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"`.
`...`	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 is 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, ...)
```

Arguments

`x`	An `EpiModel` object of class `netdx`.
`row.names`	See `as.data.frame.default`.
`optional`	See `as.data.frame.default`.
`sim`	The simulation number to output. If not specified, then data from all simulations will be output.
`...`	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.network.transmat` *Convert transmat Infection Tree into a network Object*

Description

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

Usage

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

Arguments

`x`	An object of class `transmat` 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` object.

`as.phylo.transmat` *Convert transmat Infection Tree into a phylo Object*

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, ...)
```

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 `phylo` object representation suitable for plotting as a tree with `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 `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)
```

`check_degdist_bal` *Check Degree Distribution for Balance in Target Statistics*

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,
  new.inf,
  new.rec,
  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` *Plot Compartment Diagram for Epidemic Models*

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 'dcm'
comp_plot(x, at = 1, digits = 3, run = 1, ...)

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

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

Arguments

`x`	An `EpiModel` object of class `dcm`, `icm`, or `netsim`.
`at`	Time step for model statistics.
`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 = "rk4",
  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 the "Runge-Kutta 4" method (see ode 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).
`sens.param`	If `TRUE`, evaluate arguments in parameters with length greater than 1 as sensitivity analyses, with one model run per value of the parameter. If `FALSE`, one model will be run with parameters of arbitrary length (the model may error unless the model function is designed to accomodate parameter vectors).
`print.mod`	If `TRUE`, print the model form to the console.
`verbose`	If `TRUE`, print model progress to the console.
`...`	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 an overview of control settings for base DCM class models, consult the Basic DCMs tutorial. For all base models, the `type` argument is a necessary parameter and it has no default.

Value

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

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. Details and examples are found in the New DCMs tutorial.

See Also

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

`control.icm` *Control Settings for Stochastic Individual Contact Models*

Description

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

Usage

```
control.icm(
  type,
  nsteps,
  nsims = 1,
  initialize.FUN = initialize.icm,
  infection.FUN = NULL,
  recovery.FUN = NULL,
  departures.FUN = NULL,
  arrivals.FUN = NULL,
  prevalence.FUN = NULL,
  verbose = FALSE,
  verbose.int = 0,
  skip.check = 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. This must be a positive integer.
`nsims`	Number of simulations to run.
`initialize.FUN`	Module to initialize the model at the outset, with the default function of `initialize.icm`.
`infection.FUN`	Module to simulate disease infection, with the default function of `infection.icm`.
`recovery.FUN`	Module to simulate disease recovery, with the default function of `recovery.icm`.
`departures.FUN`	Module to simulate departures or exits, with the default function of `departures.icm`.
`arrivals.FUN`	Module to simulate arrivals or entries, with the default function of `arrivals.icm`.
`prevalence.FUN`	Module to calculate disease prevalence at each time step, with the default function of `prevalence.icm`.
`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.
`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.
`...`	Additional control settings passed to model.

Details

`control.icm` sets the required control settings for any stochastic individual contact model solved with the `icm` function. Controls are required for both base model types and when passing original process modules. For an overview of control settings for base ICM class models, consult the Basic ICMs tutorial. For all base models, the `type` argument is a necessary parameter and it has no default.

Value

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

New Modules

Base ICM models use a set of module functions that specify how the individual agents in the population 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.icm` 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.

See Also

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

`control.net` *Control Settings for Stochastic Network Models*

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,
  cumulative.edgelist = FALSE,
  truncate.el.cuml = 0,
  attr.rules,
  epi.by,
  initialize.FUN = initialize.net,
  resim_nets.FUN = resim_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.other,
  verbose = TRUE,
  verbose.int = 1,
  skip.check = FALSE,
  raw.output = FALSE,
  tergmLite.track.duration = FALSE,
  set.control.ergm = control.simulate.formula(MCMC.burnin = 2e+05),
  set.control.stergm = NULL,
  set.control.tergm = control.simulate.formula.tergm(),
  save.diss.stats = TRUE,
  ...
)
```

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 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 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 `foreach` and `doParallel` implementations.
`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).
`tergmLite`	Logical indicating usage of either `tergm` (`tergmLite = FALSE`), or `tergmLite` (`tergmLite = TRUE`). Default of `FALSE`.
`cumulative.edgelist`	If `TRUE`, calculates a cumulative edgelist within the network simulation module. This is used when tergmLite is used and the entire networkDynamic object is not used.
`truncate.el.cuml`	Number of time steps of the cumulative edgelist to retain. See help file for `update_cumulative_edgelist` for options.
`attr.rules`	A list containing the rules for setting the attributes of incoming nodes, with one list element per attribute to be set (see details below).
`epi.by`	A character vector of length 1 containing a nodal attribute for which subgroup epidemic prevalences should be calculated. This nodal attribute must be contained in the network model formation formula, otherwise it is ignored.
`initialize.FUN`	Module to initialize the model at time 1, with the default function of `initialize.net`.
`resim_nets.FUN`	Module to resimulate the network at each time step, with the default function of `resim_nets`.
`infection.FUN`	Module to simulate disease infection, with the default function of `infection.net`.
`recovery.FUN`	Module to simulate disease recovery, with the default function of `recovery.net`.
`departures.FUN`	Module to simulate departure or exit, with the default function of `departures.net`.
`arrivals.FUN`	Module to simulate arrivals or entries, with the default function of `arrivals.net`.
`nwupdate.FUN`	Module to handle updating of network structure and nodal attributes due to exogenous epidemic model processes, with the default function of `nwupdate.net`.
`prevalence.FUN`	Module to calculate disease prevalence at each time step, with the default function of `prevalence.net`.
`verbose.FUN`	Module to print simulation progress to screen, with the default function of `verbose.net`.

`module.order`	A character vector of module names that lists modules in the order in which they should be evaluated within each time step. If `NULL`, the modules will be evaluated as follows: first any new modules supplied through `...` in the order in which they are listed, then the built-in modules in the order in which they are listed as arguments above. `initialize.FUN` will always be run first and `verbose.FUN` will always be run last.
`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.
`save.transmat`	If `TRUE`, complete transmission matrix is saved at simulation end. Default of `TRUE`.
`save.network`	If `TRUE`, networkDynamic/network object is saved at simulation end. Implicit reset to `FALSE` if `tergmLite = TRUE` (because network history is not saved with tergmLite).
`save.other`	A character vector of elements on the `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 netsim data (one per simulation) instead of a formatted `netsim` object.
`tergmLite.track.duration`	If `TRUE`, track duration information for models in `tergmLite` simulations.
`set.control.ergm`	Control arguments passed to `ergm`'s `simulate_formula.network`. In `netsim`, this is only used when initializing the network with `edapprox = TRUE`; all other simulations in `netsim` use `tergm`.
`set.control.stergm`	Deprecated control argument of class `control.simulate.network`; use `set.control.tergm` instead.
`set.control.tergm`	Control arguments passed to `tergm`'s `simulate_formula.network`. See the help file for `netdx` for details and examples on specifying this parameter.
`save.diss.stats`	If `TRUE`, `netsim` will compute and save duration/dissolution statistics for plotting and printing, provided `save.network` is `TRUE`, `tergmLite` is `FALSE`, and the dissolution model is homogeneous.
`...`	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 Basic Network Models tutorials. 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 `gunzip (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 New Network Models tutorials. One may remove existing modules, such as `arrivals.FUN`, from the workflow by setting the parameter value for that argument to `NULL`.

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` *Create a Minimal Main List Object for a Network Model*

Description

This helper function populates a `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 with the different named elements defined as empty lists.

Usage

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

Value

A `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 iteself. 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

`param`	Model parameters, as an object of class `param.dcm`.
`init`	Initial conditions, as an object of class `init.dcm`.
`control`	Control settings, as an object of class `control.dcm`.

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. A description of solving DCMs with the `dcm` function may be found in the Basic DCMs tutorial.

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). New model types may be written and input into `dcm` following the steps outlined in the New DCMs tutorial. 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)
```

`delete_vertices` *Fast Version of network::delete.vertices for Edgelist-formatted Network*

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)
```

`dissolution_coefs` *Dissolution Coefficients for Stochastic Network Models*

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 `netest`). 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.
- `~offset(edges) + offset(nodefactor("<attr>"))`: a heterogeneous model in which the edge duration varies by a specified attribute. The duration vector should first contain the base value, then the values for every other value of that attribute in the term. This option is deprecated.

Value

A list of class `disscoef` 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`, `nodemix`, and `nodefactor`.

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

`el`	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 `el` 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)
el <- as.data.frame(dx)

# Calculate censoring
edgelist_censor(el)
```

`epiweb` *EpiModel Web*

Description

Runs a web browser-based GUI of deterministic compartmental models, stochastic individual contact models, and basic network models.

Usage

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

Arguments

`class`	Model class, with options of `"dcm"`, `"icm"` and `"net"`.
`...`	Additional arguments passed to `shiny::runApp`.

Details

`epiweb` runs a web-based GUI of one-group deterministic compartmental models, stochastic individual contact models, and stochastic network models with user input on model type, state sizes, and parameters. Model output may be plotted, summarized, and saved as raw data using the core `EpiModel` functionality for these model classes. These applications are built using the `shiny` package framework.

References

RStudio. shiny: Web Application Framework for R. R package version 1.0.5. 2015. `https://shiny.rstudio.com/`

See Also

`dcm`, `icm`, `netsim`

Examples

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

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

## Stochastic network models
epiweb(class = "net")

## 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

`param`	The `param` argument received by the `netsim` functions.
`verbose`	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 `aes()` 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 `ggplot::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_attr_history` *Extract the Attributes History from Network Simulations*

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_cumulative_edgelist` *Get a Cumulative Edgelist From a Specified Network*

Description

Get a Cumulative Edgelist From a Specified Network

Usage

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

Arguments

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

Value

A cumulative edgelist in `data.frame` form with 4 columns:

- `head`: the unique ID (see `get_unique_ids`) of the head node on the edge.
- `tail`: the unique ID (see `get_unique_ids`) of the tail node on the edge.
- `start`: the time step in which the edge started.
- `stop`: the time step in which the edge stopped; if ongoing, then `NA` is returned.

`get_cumulative_edgelists_df` *Get the Cumulative Edgelists of a Model*

Description

Get the Cumulative Edgelists of a Model

Usage

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

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`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 5 columns:

- `index`: the unique ID (see `get_unique_ids`) of the indexes.
- `partner`: the unique ID (see `get_unique_ids`) of the partners/contacts.
- `start`: the time step in 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.

`get_current_timestep` *Return the Current Timestep*

Description

Return the Current Timestep

Usage

```
get_current_timestep(dat)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.

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_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 list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`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_formula_term_attr` *Output ERGM Formula Attributes into a Character Vector*

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 networkDynamic and network Objects from Network Simulations*

Description

Extracts the `networkDynamic` object from either a network epidemic model object generated with `netsim` or a network diagnostic simulation generated with `netdx`, with the option to collapse the extracted `networkDynamic` object down to a static `network` object.

Usage

```
get_network(
  x,
  sim = 1,
  network = 1,
  collapse = FALSE,
  at,
  ergm.create.nd = TRUE
)
```

Arguments

`x`	An `EpiModel` object of class `netsim` or `netdx`.
`sim`	Simulation number of extracted network.
`network`	Network number, for `netsim` objects with multiple overlapping networks (advanced use, and not applicable to `netdx` objects).
`collapse`	If `TRUE`, collapse the `networkDynamic` object to a static `network` object at a specified time step.
`at`	If `collapse` is `TRUE`, the time step at which the extracted network should be collapsed.
`ergm.create.nd`	If `TRUE` and `x` contains a static ERGM (i.e., a netest model with duration = 1), then create a `networkDynamic` object from the stored list of static `network` objects; if `FALSE`, output the network list directly.

Details

This function requires that the `networkDynamic` object is saved during the network simulation while running either `netsim` or `netdx`. For the former, that is specified by setting the `tergmLite` parameter in `control.net` to `FALSE`. For the latter, that is specified with the `keep.tedgelist` parameter directly in `netdx`.

Value

A `networkDynamic` object (if `collapse = FALSE`) or a static `network` object (if `collapse = TRUE`).

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_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 a data frame of network statistics from a network epidemic model simulated with `netsim` or a network diagnostics object simulated with `netdx`.

Usage

```
get_nwstats(x, sim, network = 1)
```

Arguments

`x`	An `EpiModel` object of class `netsim` or `netdx`.
`sim`	A vector of simulation numbers from the extracted object.
`network`	Network number, for `netsim` objects with multiple overlapping networks (advanced use, and not applicable to `netdx` objects).

Value

A data frame of 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 Partners (Contacts) of a Set of Index Patients*

Description

Return the Historical Partners (Contacts) of a Set of Index Patients

Usage

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

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`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 multiple overlapping 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.
`only.active.nodes`	If `TRUE`, then inactive (e.g., deceased) partners will be removed from the output.

Value

A `data.frame` with 5 columns:

- `index`: the unique ID (see `get_unique_ids`) of the indexes.
- `partner`: the unique ID (see `get_unique_ids`) of the partners/contacts.
- `start`: the time step in 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.

`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, var)
```

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.
`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_vertex_attribute` *Get Vertex Attribute on Network Object*

Description

Gets a vertex attribute from an object of class `network`. This functions 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 an object of class `network`.

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

`param`	Model parameters, as an object of class `param.icm`.
`init`	Initial conditions, as an object of class `init.icm`.
`control`	Control settings, as an object of class `control.icm`.

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. A detailed description of these models may be found in the Basic ICMs tutorial.

The `icm` 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 built 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 an original module.

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 for base models 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` *Increment the Current Timestep*

Description

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

Usage

```
increment_timestep(dat)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.

Value

The updated `dat` main list object.

Mutability

This DOES NOT modify the `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. For an overview of initial conditions for base DCM class models, consult the Basic DCMs tutorial.

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. More details on this requirement are outlined in the Solving New DCMs tutorial.

Value

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

See Also

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

`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

`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 `icm` should be input into the `init.icm` function. This function handles initial conditions for both base models and original models using new modules. For an overview of initial conditions for base ICM class models, consult the Basic ICMs tutorial.

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

`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.
`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.
`status.vector`	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 `.num` arguments.
`infTime.vector`	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 `status.vector` is used, and must contain `NA` values for any nodes whose status is not `"i"`.
`...`	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 Basic Network Models 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

`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 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.

`init_tergmLite` *Initialize EpiModel netsim Object for tergmLite Simulation*

Description

Initialize EpiModel netsim Object for tergmLite Simulation

Usage

```
init_tergmLite(dat)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.

Details

This function is typically used within the initialization modules of `EpiModel` to establish the necessary infrastructure needed for `tergmLite` network resimulation. The example below demonstrates the specific information returned.

Value

Returns the list object `dat` and adds the element `el` which is an edgelist representation of the network. Also converts the `nw` element to a `networkLite` representation.

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)
str(dat, max.level = 1)

# Element added is el (edgelist representation of network)...
dat$el

# ... and nw is now a networkLite
dat$nw[[1]]

## End(Not run)
```

`is.transmat` *Extract Transmissions Matrix from Network Epidemic Model*

Description

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

Usage

```
is.transmat(x)

get_transmat(x, sim = 1)
```

Arguments

`x`	An `EpiModel` object of class `netsim`.
`sim`	Simulation number of extracted network.

Value

A data frame with the following 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` *Are These Nodes Active (Positional IDs)*

Description

Are These Nodes Active (Positional IDs)

Usage

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

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`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 list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`unique_ids`	A vector of node unique identifiers.

Value

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

`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

`x`	An `EpiModel` object of class `icm`.
`y`	Another `EpiModel` object of class `icm`, with the identical model parameterization as `x`.
`...`	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 `netsim`.
`y`	Another `EpiModel` object of class `netsim`, 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.netsim`) 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.netsim`) from the original `x` and `y` elements.
`param.error`	If `TRUE`, if `x` and `y` have different params (in `param.net`) or controls (passed in `control.net`) 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)
```

`modules.icm` *Modules for Stochastic Individual Contact Models*

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.

Writing original ICMs will require modifying the existing modules or adding new modules to the workflow in `icm`. The existing modules may be used as a template for replacement or new modules.

This help page presents a brief overview of the module functions in the order in which they are used within `icm`, in order to help guide users in writing their own module 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 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 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)

`mutate_epi` *Add New Epidemiology Variables*

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 List Object in Network Models*

Description

These `get_`, `set_`, `append_`, and `add_` functions allow a safe and efficient way to retrieve and mutate the main list object of network models (`dat`).

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)

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)

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)

append_core_attr(dat, at, n.new)
```

Arguments

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

Value

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

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 `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.

Examples

```
dat <- list(
  attr = list(
    active = rbinom(100, 1, 0.9)
  ),
  epi = list(),
  param = list(),
  init = list(),
  control = list(
    nsteps = 150
  )
)

dat <- add_attr(dat, "age")
dat <- set_attr(dat, "age", runif(100))
dat <- set_attr(dat, "status", rbinom(100, 1, 0.9))
dat <- set_attr(dat, "status", rep(1, 150), override.length.check = TRUE)
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))
get_epi(dat, "i.num", rbinom(150, 1, 0.2) == 1)

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 dynamic diagnostics on an ERGM/STERGM estimated through `netest`.

Usage

```
netdx(
  x,
  nsims = 1,
  dynamic = TRUE,
  nsteps,
  nwstats.formula = "formation",
  set.control.ergm = control.simulate.formula(),
  set.control.stergm = control.simulate.network(),
  set.control.tergm = control.simulate.formula.tergm(),
  sequential = TRUE,
  keep.tedgelist = FALSE,
  keep.tnetwork = FALSE,
  verbose = TRUE,
  ncores = 1,
  skip.dissolution = FALSE
)
```

Arguments

`x`	An `EpiModel` object of class `netest`.
`nsims`	Number of simulations to run.
`dynamic`	If `TRUE`, runs dynamic diagnostics. If `FALSE` and the `netest` object was fit with the Edges Dissolution approximation method, simulates from the static ERGM fit.
`nsteps`	Number of time steps per simulation (dynamic simulations only).
`nwstats.formula`	A right-hand sided ERGM formula with the network statistics of interest. The default is the formation formula of the network model contained in `x`.
`set.control.ergm`	Control arguments passed to `ergm`'s `simulate_formula.network` (see details).
`set.control.stergm`	Deprecated control argument of class `control.simulate.network`; use `set.control.tergm` instead.

`set.control.tergm`	Control arguments passed to `tergm`'s `simulate_formula.network` (see details).
`sequential`	For static diagnostics (`dynamic=FALSE`): if `FALSE`, each of the `nsims` simulated Markov chains begins at the initial network; if `TRUE`, the end of one simulation is used as the start of the next.
`keep.tedgelist`	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 `$edgelist`.
`keep.tnetwork`	If `TRUE`, keep the full networkDynamic objects from the dynamic simulations. Returned in the form of a list of nD objects, with one entry per simulation. Accessible at `$network`.
`verbose`	If `TRUE`, print progress to the console.
`ncores`	Number of processor cores to run multiple simulations on, using the `foreach` and `doParallel` implementations.
`skip.dissolution`	If `TRUE`, skip over the calculations of duration and dissolution stats in `netdx`.

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`.

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 `control.simulate.formula.ter` 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 `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 `control.simulate.formula.tergm` help page in the `tergm` package. An example is shown below.

See Also

Plot these model diagnostics with `plot.netdx`.

Examples

```
## Not run:
# Network initialization and model parameterization
nw <- network_initialize(n = 100)
formation <- ~edges
target.stats <- 50
coef.diss <- dissolution_coefs(dissolution = ~ offset(edges), duration = 25)

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

# Static diagnostics on the ERGM fit
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))
est <- netest(nw, formation, target.stats, coef.diss, verbose = FALSE)
dx11 <- netdx(est, nsims = 5, nsteps = 100)
plot(dx11)
plot(dx11, type = "duration", plots.joined = TRUE, qnts = 0.2)
plot(dx11, 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).

Usage

```
netest(
  nw,
  formation,
  target.stats,
  coef.diss,
  constraints,
  coef.form = NULL,
  edapprox = TRUE,
  set.control.ergm = control.ergm(),
  set.control.stergm = control.stergm(),
  set.control.tergm = control.tergm(),
  verbose = FALSE,
  nested.edapprox = TRUE,
  ...
)
```

Arguments

`nw`	An object of class `network`.
`formation`	Right-hand sided STERGM formation formula in the form `~edges + ...`, where `...` are additional network statistics.
`target.stats`	Vector of target statistics for the formation model, with one number for each network statistic in the model.
`coef.diss`	An object of class `disscoef` output from the `dissolution_coefs` function.
`constraints`	Right-hand sided formula specifying constraints for the modeled network, in the form `~...`, where `...` 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).
`set.control.ergm`	Control arguments passed to `ergm` (see details).
`set.control.stergm`	Deprecated control argument of class `control.stergm`; use `set.control.tergm` instead.

`set.control.tergm`	Control arguments passed to `tergm` (see details).
`verbose`	If `TRUE`, print model fitting progress to console.
`nested.edapprox`	Logical. If `edapprox = TRUE` the dissolution model is an initial segment of the formation model (see details).
`...`	Additional arguments passed to other functions.

Details

`netest` is a wrapper function for the `ergm` and `stergm` functions that estimate static and dynamic network models, respectively. 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 Basic Network Models tutorials.

Value

A fitted network model object of class `netest`.

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`. 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 toggles whether to implement this modified version by appending the dissolution terms to the formation model and appending the relevant values to the vector of formation model coefficients (value = `FALSE`), whereas the standard version subtracts the relevant values from the initial formation model coefficients (value = `TRUE`).

Control Arguments

The `ergm` 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 `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` in `netest`. The controls should be input through the `control.ergm()` function, with the available parameters listed in the `control.ergm` help page in the `ergm` package. An example is below.

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 `netdx` to diagnose the fitted network model, and `netsim` to simulate epidemic spread over a simulated dynamic network consistent with the model fit.

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))
est

# To estimate the STERGM directly, use edapprox = FALSE
# est2 <- netest(nw, formation, target.stats, coef.diss, edapprox = FALSE)
```

`netsim` *Stochastic Network Models*

Description

Simulates stochastic network epidemic models for infectious disease.

Usage

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

Arguments

`x`	Fitted network model object, as an object of class `netest`. Alternatively, if restarting a previous simulation, may be an object of class `netsim`.
`param`	Model parameters, as an object of class `param.net`.
`init`	Initial conditions, as an object of class `init.net`.
`control`	Control settings, as an object of class `control.net`.

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 Basic Network Models tutorials.

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 Extension Network Models tutorials. 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` and the tutorials for further details.
- **network:** a list of `networkDynamic` objects, one for each model simulation.

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

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

Extract the model results with `as.data.frame.netsim`. Summarize the time-specific model results with `summary.netsim`. Plot the model results with `plot.netsim`.

Examples

```
## Not run:
## Example 1: SI Model without Network Feedback
# Network model estimation
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)

# 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 = 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
# 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)
```

```
# Reset 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)

## End(Not run)
```

`networkLite` *networkLite Constructor Utilities*

Description

Constructor methods for `networkLite` objects.

Usage

```
networkLite(x, ...)

## S3 method for class 'edgelist'
networkLite(
  x,
  attr = list(vertex.names = seq_len(attributes(x)[["n"]]), na =
    logical(attributes(x)[["n"]])),
  ...
)

## S3 method for class 'matrix'
networkLite(
  x,
  attr = list(vertex.names = seq_len(attributes(x)[["n"]]), na =
    logical(attributes(x)[["n"]])),
  ...
```

```
)

## S3 method for class 'numeric'
networkLite(x, directed = FALSE, bipartite = FALSE, ...)

networkLite_initialize(x, directed = FALSE, bipartite = FALSE, ...)
```

Arguments

`x`	Either an `edgelist` class network representation (including network attributes in its `attributes` list), or a number specifying the network size.
`...`	Additional arguments used by other methods.
`attr`	A named list of vertex attributes for the network represented by `x`.
`directed, bipartite`	Common network attributes that may be set via arguments to the `networkLite.numeric` method.

Details

Currently there are several distinct `networkLite` constructor methods available.

The `edgelist` method takes an `edgelist` class object `x` with network attributes attached in its `attributes` list, and a named list of vertex attributes `attr`, and returns a `networkLite` object, which is a named list with fields `el`, `attr`, and `gal`; the fields `el` and `attr` match the arguments `x` and `attr` (the latter coerced to `tibble`) respectively, and the field `gal` is the list of network attributes (copied from `attributes(x)`). Missing network attributes `directed` and `bipartite` are defaulted to `FALSE`; the network size attribute `n` must not be missing. Attributes `class`, `dim`, `dimnames`, `vnames`, and `mnext` (if present) are not copied from `x` to the `networkLite`. (For convenience, a `matrix` method, identical to the `edgelist` method, is also defined, to handle cases where the edgelist is, for whatever reason, not classed as an `edgelist`.)

The `numeric` method takes a number `x` as well as the network attributes `directed` and `bipartite` (defaulting to `FALSE`), and returns an empty `networkLite` with these network attributes and number of nodes `x`.

The constructor `networkLite_initialize` is also available for creating an empty `networkLite`, and its `x` argument should be a number indicating the size of the `networkLite` to create.

Within `tergmLite`, the `networkLite` data structure is used in the calls to `ergm` and `tergm` `simulate` functions.

Value

A networkLite object with edge list `el`, vertex attributes `attr`, and network attributes `gal`.

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)

# Conversion to networkLite class format
nwl <- networkLite(dat$el[[1]], dat$attr)
nwl

## End(Not run)
```

`networkLitemethods` *networkLite Methods*

Description

S3 methods for networkLite class, for generics defined in network package.

Usage

```
## S3 method for class 'networkLite'
get.vertex.attribute(x, attrname, ...)

## S3 method for class 'networkLite'
set.vertex.attribute(x, attrname, value, v = seq_len(network.size(x)), ...)

## S3 method for class 'networkLite'
list.vertex.attributes(x, ...)

## S3 method for class 'networkLite'
get.network.attribute(x, attrname, ...)

## S3 method for class 'networkLite'
set.network.attribute(x, attrname, value, ...)

## S3 method for class 'networkLite'
list.network.attributes(x, ...)

## S3 method for class 'networkLite'
get.edge.attribute(x, attrname, ...)
```

```
## S3 method for class 'networkLite'
get.edge.value(x, attrname, ...)

## S3 method for class 'networkLite'
set.edge.attribute(
  x,
  attrname,
  value,
  e = seq_len(network.edgecount(x, na.omit = FALSE)),
  ...
)

## S3 method for class 'networkLite'
set.edge.value(
  x,
  attrname,
  value,
  e = seq_len(network.edgecount(x, na.omit = FALSE)),
  ...
)

## S3 method for class 'networkLite'
list.edge.attributes(x, ...)

## S3 method for class 'networkLite'
network.edgecount(x, na.omit = TRUE, ...)

## S3 method for class 'networkLite'
as.edgelist(
  x,
  attrname = NULL,
  output = c("matrix", "tibble"),
  na.rm = TRUE,
  ...
)

## S3 method for class 'networkLite'
mixingmatrix(object, attr, ...)

## S3 replacement method for class 'networkLite'
x[i, j, names.eval = NULL, add.edges = FALSE] <- value

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

## S3 method for class 'networkLite'
network.naedgecount(x, ...)
```

```
## S3 method for class 'networkLite'
add.edges(x, tail, head, names.eval = NULL, vals.eval = NULL, ...)

as.networkLite(x, ...)

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

## S3 method for class 'networkLite'
as.networkLite(x, ...)

## S3 method for class 'networkLite'
as.networkDynamic(object, ...)

## S3 method for class 'networkLite'
as_tibble(x, attrnames = NULL, na.rm = TRUE, ...)

## S3 method for class 'networkLite'
as.matrix(
  x,
  matrix.type = c("adjacency", "incidence", "edgelist"),
  attrname = NULL,
  ...
)

## S3 method for class 'networkLite'
is.na(x)

## S3 method for class 'networkLite'
delete.vertex.attribute(x, attrname, ...)

## S3 method for class 'networkLite'
delete.edge.attribute(x, attrname, ...)

## S3 method for class 'networkLite'
delete.network.attribute(x, attrname, ...)

## S3 method for class 'networkLite'
add.vertices(x, nv, vattr = NULL, last.mode = TRUE, ...)

## S3 method for class 'networkLite'
e1 + e2

## S3 method for class 'networkLite'
e1 - e2
```

Arguments

`x`	A `networkLite` object.

`attrname`	The name of an attribute in `x`.
`...`	Any additional arguments.
`value`	The attribute value to set in vertex, edge, and network attribute setters; the value to set edges to (must be FALSE) for the `networkLite` replacement method.
`v`	Indices at which to set vertex attribute values.
`e`	edge indices to assign value
`na.omit`	logical; omit missing edges from edge count?
`output`	Type of edgelist to output.
`na.rm`	should missing edges be dropped from edgelist?
`object`	A `networkLite` object.
`attr`	Specification of a vertex attribute in `object` as described in `nodal_attributes`.
`i, j`	Nodal indices (must be missing for networkLite method).
`names.eval`	name(s) of edge attributes
`add.edges`	should edges being assigned to be added if not already present?
`tail`	Vector of tails of edges to add to the networkLite.
`head`	Vector of heads of edges to add to the networkLite.
`vals.eval`	value(s) of edge attributes
`attrnames`	vector specifying edge attributes to include in the tibble; may be logical, integer, or character vector, the former two being used to select attribute names from `list.edge.attributes(x)`, and the latter being used as the attribute names themselves
`matrix.type`	type of matrix to return from `as.matrix.networkLite`
`nv`	number of vertices to add to the `networkLite`
`vattr`	list (of length `nv`) of named lists of vertex attributes for added vertices, or `NULL` to indicate vertex attributes are not being passed
`last.mode`	logical; if `x` is bipartite, should the new vertices be added to the second mode?
`e1, e2`	networkLite objects

Details

Allows use of networkLite objects in `ergm_model`.

Value

An edgelist for `as.edgelist.networkLite`; an updated `networkLite` object for the replacement method. The other methods return no objects.

`network_initialize` *Initialize Network Object*

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` *Dynamic Network Updates*

Description

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

Usage

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

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`at`	Current time step.

Value

The updated `dat` main list object.

`param.dcm` *Epidemic Parameters for Deterministic Compartmental Models*

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). A detailed description of DCM parameterization for base models is found in the Basic DCMs tutorial.

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. See the Basic DCMs tutorial for further details.

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. See the Basic DCMs tutorial.

New Model Types

To build original model specifications outside of the base models, start by consulting the New DCMs with EpiModel tutorial. Briefly, 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`.

`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. 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 ICM parameterization for base models is found in the Basic ICMs tutorial.

For base models, 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 the Basic ICMs tutorial.

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.icm`. 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.icm`. Whereas there are strict checks with default modules for parameter validity, these checks are the user's responsibility with new modules.

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).
`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 partnership* per unit time (see `act.rate` 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).
`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. 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).
`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 group 1 susceptible persons only.
`di.rate`	Departure or exit rate for infected persons. For two-group models, it is the rate for group 1 infected persons only.
`dr.rate`	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.

`inf.prob.g2`	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.
`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 person 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.
`...`	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 Basic Network Models tutorial.

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 NME Course 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.parameters` argument. Further details and examples are provided in the "Working with Model Parameters in EpiModel" vignette.

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 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` *Parameters List for Stochastic Network Models from a Formatted Data Frame*

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.

Details

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.

Value

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

`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,
  popfrac = FALSE,
  run,
  col,
  lwd,
  lty,
  alpha = 0.9,
  legend,
  leg.name,
  leg.cex = 0.8,
  axs = "r",
  grid = FALSE,
  add = FALSE,
  ...
)
```

Arguments

`x`	An `EpiModel` object of class `dcm`.
`y`	Output compartments or flows from `dcm` object to plot.
`popfrac`	If `TRUE`, plot prevalence of values rather than numbers (see details).

`run`	Run number to plot, for models with multiple runs (default is run 1).
`col`	Color for lines, either specified as a single color in a standard R color format, or alternatively as a color palette from `RColorBrewer` (see details).
`lwd`	Line width for output lines.
`lty`	Line type for output lines.
`alpha`	Transparency level for lines, where 0 = transparent and 1 = opaque (see `adjustcolor` function).
`legend`	Type of legend to plot. Values are `"n"` for no legend, `"full"` for full legend, and `"lim"` for limited legend (see details).
`leg.name`	Character string to use for legend, with the default determined automatically based on the `y` input.
`leg.cex`	Legend scale size.
`axs`	Plot axis type (see `par` for details), with default of "r".
`grid`	If `TRUE`, a grid is added to the background of plot (see `grid` 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.
`...`	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` 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 `brewer.pal.info`, or, alternatively, a vector of standard R colors (named, hexidecimal, 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`, `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.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,
  popfrac = FALSE,
  sim.lines = FALSE,
  sims,
  sim.col,
  sim.lwd,
  sim.alpha,
  mean.line = TRUE,
  mean.smooth = TRUE,
  mean.col,
  mean.lwd = 2,
  mean.lty = 1,
  qnts = 0.5,
  qnts.col,
  qnts.alpha,
  qnts.smooth = TRUE,
  legend,
  leg.cex = 0.8,
  axs = "r",
  grid = FALSE,
  add = FALSE,
  ...
)
```

Arguments

`x`	An `EpiModel` model object of class `icm`.
`y`	Output compartments or flows from `icm` object to plot.
`popfrac`	If `TRUE`, plot prevalence of values rather than numbers (see details).
`sim.lines`	If `TRUE`, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
`sims`	A vector of simulation numbers to plot.
`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 `adjustcolor` 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 `adjustcolor` function).
`qnts.smooth`	If `TRUE`, use a loess smoother on quantile polygons.
`legend`	If `TRUE`, plot default legend.
`leg.cex`	Legend scale size.
`axs`	Plot axis type (see `par` for details), with default of `"r"`.
`grid`	If `TRUE`, a grid is added to the background of plot (see `grid` 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.
`...`	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% 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`.

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,
  stats,
  duration.imputed = TRUE,
  sim.lines = FALSE,
  sim.col,
  sim.lwd,
  mean.line = TRUE,
  mean.smooth = TRUE,
  mean.col,
  mean.lwd = 2,
  mean.lty = 1,
  qnts = 0.5,
  qnts.col,
  qnts.alpha = 0.5,
  qnts.smooth = TRUE,
```

```
  targ.line = TRUE,
  targ.col,
  targ.lwd = 2,
  targ.lty = 2,
  plots.joined,
  legend,
  grid = FALSE,
  ...
)
```

Arguments

`x`	An `EpiModel` object of class `netdx`.
`type`	Plot type, with options of `"formation"` for network model formation statistics, `"duration"` for dissolution model statistics for average edge duration, or `"dissolution"` for dissolution model statistics for proportion of ties dissolved per time step.
`method`	Plot method, with options of `"l"` for line plots and `"b"` for boxplots.
`sims`	A vector of simulation numbers to plot.
`stats`	Statistics to plot. For `type = "formation"`, `stats` are among those specified in the call to `netdx`; for `type = "duration","dissolution"`, `stats` are among those of the dissolution model (without `offset()`). The default is to plot all statistics.
`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"`.
`sim.lines`	If `TRUE`, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
`sim.col`	Vector of any standard R color format for simulation lines.
`sim.lwd`	Line width for simulation lines.
`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 `adjustcolor` 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 `grid` for details), with default of nx by ny.
`...`	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.

See Also

`netdx`

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")

## 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,
  popfrac = FALSE,
  sim.lines = FALSE,
  sims,
  sim.col,
  sim.lwd,
  sim.alpha,
  mean.line = TRUE,
  mean.smooth = TRUE,
  mean.col,
  mean.lwd = 2,
  mean.lty = 1,
  qnts = 0.5,
  qnts.col,
  qnts.alpha = 0.5,
  qnts.smooth = TRUE,
  legend,
  leg.cex = 0.8,
  axs = "r",
  grid = FALSE,
  add = FALSE,
  network = 1,
  at = 1,
  col.status = FALSE,
  shp.g2 = NULL,
  vertex.cex,
  stats,
  targ.line = TRUE,
  targ.col,
  targ.lwd = 2,
  targ.lty = 2,
  plots.joined,
  duration.imputed = TRUE,
  method = "l",
  ...
)
```

Arguments

`x`	An `EpiModel` model object of class `netsim`.
`type`	Type of plot: `"epi"` for epidemic model results, `"network"` for a static network plot (`plot.network`), or `"formation"`, `"duration"`, or `"dissolution"` for network formation, duration, or dissolution statistics.
`y`	Output compartments or flows from `netsim` object to plot.
`popfrac`	If `TRUE`, plot prevalence of values rather than numbers (see details).
`sim.lines`	If `TRUE`, plot individual simulation lines. Default is to plot lines for one-group models but not for two-group models.
`sims`	If `type="epi"` or `"formation"`, a vector of simulation numbers to plot. 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.
`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 `adjustcolor` 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 `adjustcolor` function).
`qnts.smooth`	If `TRUE`, use a loess smoother on quantile polygons.
`legend`	If `TRUE`, plot default legend.
`leg.cex`	Legend scale size.
`axs`	Plot axis type (see `par` for details), with default of `"r"`.
`grid`	If `TRUE`, a grid is added to the background of plot (see `grid` 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 `nwstats.formula` of `control.net`; for `type = "duration","dissolution"`, `stats` are among those of the dissolution model (without `offset()`). 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 boxplots.
`...`	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 `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.

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

`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

`x`	A `transmat` object to be plotted.
`style`	Character name of plot style. One of "`phylo`", "`network`", or "`transmissionTimeline`".

`...`	Additional plot arguments to be passed to lower-level plot functions (`plot.network`, `plot.phylo`, or `transmissionTimeline`).

Details

The `phylo` plot requires the `ape` package. The `transmissionTimeline` plot requires that the `ndtv` package.

See Also

`plot.network`, `plot.phylo`, `transmissionTimeline`.

`print.netdx` *Utility Function for Printing netdx Object*

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`, and `Sim SD`. For the formation table, `Sim Mean` refers to the mean statistic value and `Sim SD` refers to the standard deviation in the statistic value, across all time steps in all simulations in the dynamic case, and across all sampled networks in the static case. For the duration and dissolution tables, `Sim Mean` refers to the mean across simulations of the mean across time steps within simulation, and `Sim SD` refers to the standard deviation across simulations of the mean across time steps within simulation, for the age and dissolution statistics defined above. The `Target` column indicates the target value (if present) for the network statistic, mean edge age, or edge dissolution rate, and the `Pct Diff` column gives `(Sim Mean - Target)/Target` when `Target` is present.

`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"]]` and can be accessed from the `netsim` object with `get_attr_history`.

Usage

```
record_attr_history(dat, at, attribute, posit_ids, values)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`at`	The time where the recording happens.
`attribute`	The name of the value to record.
`posit_ids`	A numeric vector of posit_ids to which the measure applies. (see `get_posit_ids`).
`values`	The values to be recorded.

Details

See the "Time-Varying Parameters" section of the "Working With Model Parameters" vignette.

Value

The updated `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)
```

`record_raw_object` *Record an Arbitrary Object During a Simulation*

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 and in the `netsim` object under the `raw.records` sublists.

Usage

```
record_raw_object(dat, at, label, object)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`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 `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 `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 list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`timestep`	The new value for the timestep.

Value

The updated `dat` main list object.

Mutability

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

`set_transmat` *Save Transmission Matrix*

Description

This function appends the transmission matrix created during `infection.net` and `infection.2g.net`.

Usage

```
set_transmat(dat, del, at)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`del`	Discordant edgelist created within `infection.net` and `infection.2g.net`.
`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 list object `dat`.

Value

The updated `dat` main list object.

`set_vertex_attribute` *Set Vertex Attribute on Network Object*

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)
```

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.
`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` *Summary Model Statistics*

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` *Summary Model Statistics*

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)
```

Arguments

`object`	A `netest` class object.
`as.networkLite`	If `TRUE`, converts `object$newnetwork` to a `networkLite`.
`keep.fit`	If `FALSE`, removes the `object$fit` (if present) on the `netest` object.

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`.

Specifically, the function removes:

- `environment(object$constraints)`
- `environment(object$coef.diss$dissolution)`
- `environment(object$formation)`

When `edapprox` = `TRUE` in the `netest` call, also removes `environment(object$formula)`.

When `edapprox` = `FALSE`, also removes all but `formation` and `dissolution` from `environment(object$formula)`, as well as `environment(environment(object$formula)$formation)` and `environment(environment(object$formula`

If `as.networkLite` = `TRUE`, converts `object$newnetwork` to a `networkLite` object. If `keep.fit` = `FALSE`, removes `fit` (if present) from `object`.

For the output to be usable in `netsim` simulation, there should not be substitutions in the formulas, other than `formation` and `dissolution` in `object$formula` when `edapprox` = `FALSE`.

Value

A `netest` object with formula environments removed, 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` *Truncate Simulation Time Series*

Description

Left-truncates simulation epidemiological summary statistics and network statistics at a specified time step.

Usage

```
truncate_sim(x, at)
```

Arguments

`x`	Object of class `netsim` or `icm`.
`at`	Time step at which to left-truncate the time series.

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 `netsim` or `icm`.

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 = 200, nsims = 1)
mod1 <- icm(param, init, control)
df <- as.data.frame(mod1)
print(df)
plot(mod1)
mod1$control$nsteps

mod2 <- truncate_sim(mod1, at = 150)
df2 <- as.data.frame(mod2)
print(df2)
plot(mod2)
mod2$control$nsteps
```

`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

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`posit_ids`	A vector of node positional identifiers (default = NULL).
`unique_ids`	A vector of node unique identifiers (default = NULL).

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`, `NA`s are returned instead and a warning is produced.

`update_cumulative_edgelist`

Update a Cumulative Edgelist of the Specified Network

Description

Update a Cumulative Edgelist of the Specified Network

Usage

```
update_cumulative_edgelist(dat, network, truncate = 0)
```

Arguments

`dat`	Main list object containing a `networkDynamic` object and other initialization information passed from `netsim`.
`network`	Numerical index of the network for which the cumulative edgelist will be updated. (May be > 1 for models with multiple overlapping networks.)
`truncate`	After how many time steps a partnership that is no longer active should be removed from the output.

Value

The updated `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 truncated out of the cumulative edgelist. When `truncate = Inf`, no edges are ever removed. When `truncate = 0`, only the active edges are kept. You may want this behavior to keep track of the active edges' start step.

`update_dissolution` *Adjust Dissolution Component of Network Model Fit*

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

`old.netest`	An object of class `netest`, from the `netest` function.
`new.coef.diss`	An object of class `disscoef`, from the `dissolution_coefs` function.
`nested.edapprox`	Logical. If `edapprox = TRUE` the dissolution model is an initial segment of the formation model (see details in `netest`).
`...`	Additional arguments passed to other functions.

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)
```

`update_params` *Update Model Parameters for Stochastic Network Models*

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`).

Value

An updated list object of class `param.net`, which can be passed to the EpiModel function `netsim`.

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` *Apply a scenario object to a param.net object*

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