Package ‘hesim’

October 13, 2022

Type Package

Title Health Economic Simulation Modeling and Decision Analysis

Version 0.5.3

Description A modular and computationally efficient R package for parameterizing, simulating, and analyzing health economic simulation models. The package supports cohort discrete time state transition models (Briggs et al. 1998) <doi:10.2165/00019053-199813040-00003>, N-state partitioned survival models (Glasziou et al. 1990) <doi:10.1002/sim.4780091106>, and individual-level continuous time state transition models (Siebert et al. 2012) <doi:10.1016/j.jval.2012.06.014>, encompassing both Markov (time-homogeneous and time-inhomogeneous) and semi-Markov processes. Decision uncertainty from a cost-effectiveness analysis is quantified with standard graphical and tabular summaries of a probabilistic sensitivity analysis (Claxton et al. 2005, Barton et al. 2008) <doi:10.1002/hec.985>, <doi:10.1111/j.1524-4733.2008.00358.x>. Use of C++ and data.table make individual-patient simulation, probabilistic sensitivity analysis, and incorporation of patient heterogeneity fast.

URL `https://hesim-dev.github.io/hesim/`, `https://github.com/hesim-dev/hesim`

BugReports `https://github.com/hesim-dev/hesim/issues`

License GPL-3

LazyData true

LinkingTo Rcpp, RcppArmadillo

Depends R (>= 3.5.0)

Imports data.table, flexsurv, ggplot2, MASS, msm, Rcpp (>= 0.12.16), R6, stats, survival

Suggests covr, kableExtra, knitr, magrittr, mstate, nnet, numDeriv, pracma, rmarkdown, scales, testthat, truncnorm

VignetteBuilder knitr

RoxygenNote 7.2.1

SystemRequirements C++11

NeedsCompilation yes

Author Devin Incerti [aut, cre],
Jeroen P. Jansen [aut],
R Core Team [ctb] (hesim uses some slightly modified C functions from base R)

Maintainer Devin Incerti <devin.incerti@gmail.com>

Repository CRAN

Date/Publication 2022-09-03 21:40:10 UTC

R topics documented:

apply_rr 4
as.data.table.tparams_transprobs 5
as_array3 6
as_pfs_os 8
autoplot.stateprobs 9
autoplot.survival 10
ce 11
cea 12
CohortDtstm 15
CohortDtstmTrans 20
costs 23
create_CohortDtstm 24
create_CohortDtstmTrans 25
create_IndivCtstmTrans 26
create_params 28
create_PsmCurves 30
create_StateVals 32
create_trans_dt 34
define_model 34
define_rng 37
define_tparams 39
disprog 40
expand.hesim_data 41
expmat 42
fast_rgengamma 42
flexsurvreg_list 43
get_labels 44
hesim_data 45
icer 47
incr_effect 48
IndivCtstm 49
IndivCtstmTrans 54
input_mats 57
mom_beta 59
mom_gamma 60

mstate3_exdata 61
multinom3_exdata 62
multinom_list 64
onc3 64
onc3p 65
params 66
params_lm 67
params_mlogit 68
params_mlogit_list 69
params_surv 71
params_surv_list 73
plot_ceac 74
plot_ceaf 75
plot_ceplane 76
plot_evpi 77
Psm 77
psm4_exdata 81
PsmCurves 83
qalys 87
qmatrix.matrix 88
qmatrix.msm 89
rcat 90
rdirichlet_mat 91
rng_distributions 92
rpwexp 94
set_labels 95
sim_ev 96
sim_stateprobs.survival 100
stateprobs 102
StateVals 103
stateval_tbl 105
summary.ce 107
summary.eval_rng 109
summary.params 110
summary.tparams_mean 111
summary.tparams_transprobs 112
summary.tpmatrix 113
survival 114
surv_quantile 116
time_intervals 116
tparams 117
tparams_mean 118
tparams_transprobs 119
tpmatrix 122
tpmatrix_id 124
tpmatrix_names 125

Index **126**

`apply_rr` *Apply relative risks to transition probability matrices*

Description

Elements of transition probability matrices are multiplied by relative risks and the transition probability matrices are adjusted so that rows sum to 1. Operations are vectorized and each relative risk is multiplied by every transition matrix (stored in 3-dimensional arrays).

Usage

```
apply_rr(x, rr, index, complement = NULL)
```

Arguments

`x`	A 3-dimensional array where each slice is a square transition probability matrix.
`rr`	A 2-dimensional tabular object such as a matrix or data frame where each column is a vector of relative risks to apply to each transition matrix in `x`.
`index`	The indices of the transition probability matrices that `rr` is applied to. Should either be a matrix where the first column denotes a transition probability matrix row and the second column denotes a transition probability matrix column or a list where each element is a vector of length 2 with the first element denoting a transition probability matrix row and the second column denoting a transition probability matrix column.
`complement`	Denotes indices of transition probability matrices that are "complements" (i.e., computed as 1 less the sum of all other elements in that row). Should be in the same format as `index`. There can be at most one complementary column in each row of a transition probability matrix. If `NULL`, then the diagonals are assumed to be the complements.

Details

This function is useful for applying relative treatment effects measured using relative risks to an existing transition probability matrix. For example, a transition probability matrix for the reference treatment strategy may exist or have been estimated from the data. Relative risks estimated from a meta-analysis or network meta-analysis can then be applied to the reference transition probability matrix. If the number of rows in `rr` exceeds `x`, then the arrays in `x` are recycled to the number of rows in `rr`, which facilitates the application of relative risks from multiple treatment strategies to a reference treatment.

Value

A 3-dimensional array where each slice contains matrices of the same dimension as each matrix in `x` and the number of slices is equal to the number of rows in `rr`.

Examples

```
p_12 <- c(.7, .5)
p_23 <- c(.1, .2)
x <- as_array3(tpmatrix(
  C, p_12, .1,
  0, C,     p_23,
  0, 0,     1
))

# There are the same number of relative risk rows and transition probability matrices
rr_12 <- runif(2, .8, 1)
rr_13 <- runif(2, .9, 1)
rr <- cbind(rr_12, rr_13)
apply_rr(x, rr,
         index = list(c(1, 2), c(1, 3)),
         complement = list(c(1, 1), c(2, 2)))

# There are more relative risk rows than transition probability matrices
rr_12 <- runif(4, .8, 1)
rr_13 <- runif(4, .9, 1)
rr <- cbind(rr_12, rr_13)
apply_rr(x, rr,
         index = list(c(1, 2), c(1, 3)),
         complement = list(c(1, 1), c(2, 2)))
```

`as.data.table.tparams_transprobs` *Coerce to* `data.table`

Description

Creates a `data.table` that combines the transition probability matrices and ID variables from a `tparams_transprobs` object. This is often useful for debugging.

Usage

```
## S3 method for class 'tparams_transprobs'
as.data.table(x, ..., prefix = "prob_", sep = "_", long = FALSE)
```

Arguments

`x`	A `tparams_transprobs` object.
`...`	Currently unused.
`prefix, sep`	Arguments passed to `tpmatrix_names()` for naming the transition probability columns. The `states` argument is based on the column names (i.e., names of the second dimension) of the `$value` element of `x`; if `NULL`, then states are named `s1`, ..., `sh` where h is the number of states. Only used if `long = FALSE`.

`long`	If `TRUE`, then output is returned in a longer format with one row for each transition; if `FALSE`, then each row contains an entire flattened transition probability matrix.

Value

The output always contains columns for the ID variables and the transition probabilities, but the form depends on on the `long` argument. If `FALSE`, then a `data.table` with one row for each transition probability matrix is returned; otherwise, the `data.table` contains one row for each transition and columns `from` (the state being transitioned from) and `to` (the state being transitioned to) are added.

See Also

`tparams_transprobs()`

Examples

```
# Create tparams_transprobs object
hesim_dat <- hesim_data(strategies = data.frame(strategy_id = 1:2),
                        patients = data.frame(patient_id = 1:3))
input_data <- expand(hesim_dat, by = c("strategies", "patients"))
tpmat_id <- tpmatrix_id(input_data, n_samples = 2)
p_12 <- runif(nrow(tpmat_id), .6, .7) +
  .05 * (tpmat_id$strategy_id == 2)
tpmat <- tpmatrix(
  C, p_12,
  0, 1
)
tprobs <- tparams_transprobs(tpmat, tpmat_id)

# Convert to data.table in "wide" format
as.data.table(tprobs)
as.data.table(tprobs, prefix = "")
as.data.table(tprobs, prefix = "", sep = ".")

# Convert to data.table in "long: format
as.data.table(tprobs, long = TRUE)
```

`as_array3` *Convert between 2D tabular objects and 3D arrays*

Description

Convert a 2-dimensional tabular object where each row stores a flattened square matrix to a 3-dimensional array of square matrices and vice versa. This allows multiple transition matrices to be stored as either tabular objects (e.g., matrices, data frames, etc) or as arrays.

Usage

```
as_array3(x)

as_tbl2(
  x,
  output = c("data.table", "data.frame", "matrix", "tpmatrix"),
  prefix = "",
  sep = "_"
)
```

Arguments

`x`	For `as_array3()` a 2-dimensional tabular object where each row stores a flattened square matrix ordered rowwise. Reasonable classes are `matrix`, `data.frame`, `data.table`, and `tpmatrix`. For `as_tbl2()` a 3-dimensional array where each slice is a square matrix.
`output`	The class of the object returned by the function. Either a `data.table`, `data.frame`, `matrix`, or `tpmatrix`.
`prefix, sep`	Arguments passed to `tpmatrix_names()` for naming the transition probability columns. The `states` argument is based on the column names (i.e., names of the second dimension) of array; if `NULL`, then states are named `s1`, ..., `sh` where h is the number of states.

Value

For `as_array3()` a 3-dimensional array of square matrices; for `as_tbl2()` a 2-dimensional tabular object as specified by `output`.

See Also

`tpmatrix`

Examples

```
p_12 <- c(.7, .6)
pmat <- tpmatrix(
 C, p_12,
 0, 1
)
pmat

as_array3(pmat)
as_array3(as.matrix(pmat))
as_tbl2(as_array3(pmat))
as_tbl2(as_array3(pmat), prefix = "p_", sep = ".")
```

`as_pfs_os` *Convert multi-state data to PFS and OS data*

Description

Convert a multi-state dataset with irreversible transitions containing 3 health states to a dataset with one row per patient and progression-free survival (PFS) and overall survival (OS) time-to-event outcomes.

Usage

```
as_pfs_os(
  data,
  patient_vars,
  status = "status",
  time_stop = "time_stop",
  transition = "transition_id"
)
```

Arguments

`data`	A multi-state dataset.
`patient_vars`	Character vector of the names of patient specific variables.
`status`	Character string with the name of the status variable (1 = event, 0 = censored).
`time_stop`	Character string with the name of the stopping time variable (i.e., time patient transitions from state r to state s).
`transition`	Character string with the name of the variable identifying a transition. The transition variable should be integer valued with values 1, 2, and 3 for the Stable -> Progression, Stable -> Death, and Progression -> Death transitions, respectively.

Value

A `data.table` with one row per patient containing each variable in `patient_vars` as well as a time variable and status indicator for both PFS (`pfs_status`, `pfs_time`) and OS (`os_time`, `os_status`).

Examples

```
as_pfs_os(onc3, patient_vars = c("patient_id", "strategy_name", "female", "age"))
```

`autoplot.stateprobs` *Plot state probabilities*

Description

Quickly plot state probabilities stored in a `stateprobs` object.

Usage

```
## S3 method for class 'stateprobs'
autoplot(
  object,
  labels = NULL,
  ci = FALSE,
  prob = 0.95,
  ci_style = c("ribbon", "line"),
  geom_alpha = 0.3,
  ...
)
```

Arguments

`object`	A `stateprobs` object.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.
`ci`	A logical value indicating whether confidence intervals should be plotted. Default is `FALSE`.
`prob`	A numeric scalar in the interval `(0,1)` giving the confidence interval. Default is 0.95 for a 95 percent interval.
`ci_style`	Style to use for the confidence interval if `ci = TRUE`. If `"line"`, then dashed lines are used; if `"ribbon"`, then shaded confidence bands are plotted using `ggplot2::geom_ribbon()`.
`geom_alpha`	The opacity for the shaded confidence bands when `ci_style = "ribbon"`. This is the value of the value of the `alpha` aesthetic passed to `ggplot2::geom_ribbon()`.
`...`	Further arguments passed to and from methods. Currently unused.

Value

A `ggplot` object.

Note

If there are multiple patients/groups, then state probabilities are averaged across patients/groups (using the weights in `patient_wt` if available) prior to plotting.

See Also

`Psm` for an example.

`autoplot.survival` *Plot survival curves*

Description

Quickly plot survival curves stored in a `survival` object.

Usage

```
## S3 method for class 'survival'
autoplot(
  object,
  labels = NULL,
  ci = FALSE,
  prob = 0.95,
  ci_style = c("ribbon", "line"),
  geom_alpha = 0.3,
  ...
)
```

Arguments

`object`	A `survival` object.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.
`ci`	A logical value indicating whether confidence intervals should be plotted. Default is `FALSE`.
`prob`	A numeric scalar in the interval `(0,1)` giving the confidence interval. Default is 0.95 for a 95 percent interval.
`ci_style`	Style to use for the confidence interval if `ci = TRUE`. If `"line"`, then dashed lines are used; if `"ribbon"`, then shaded confidence bands are plotted using `ggplot2::geom_ribbon()`.
`geom_alpha`	The opacity for the shaded confidence bands when `ci_style = "ribbon"`. This is the value of the value of the `alpha` aesthetic passed to `ggplot2::geom_ribbon()`.
`...`	Further arguments passed to and from methods. Currently unused.

Value

A `ggplot` object.

Note

If there are multiple patients, then survival probabilities are averaged across patients (using the weights in `patient_wt` if available) prior to plotting.

See Also

`Psm` for an example.

`ce` *A cost-effectiveness object*

Description

An object that summarizes simulated measures of clinical effectiveness and costs from a simulation model for use in a cost-effectiveness analysis.

Format

A list containing two elements:

- `costs` Total (discounted) costs by category.
- `qalys` (Discounted) quality-adjusted life-years.

Costs

The `costs` `data.table` contains the following columns:

category The cost category.

dr The discount rate.

sample A randomly sampled parameter set from the probabilistic sensitivity analysis (PSA)

strategy_id The treatment strategy ID.

grp_id An optional column denoting a subgroup. If not included, it is assumed that a single subgroup is being analyzed.

costs Costs.

Quality-adjusted life-years

The `qalys` `data.table` contains the following columns:

dr The discount rate.

sample A randomly sampled parameter set from the probabilistic sensitivity analysis (PSA)

strategy_id The treatment strategy ID.

grp_id An optional column denoting a subgroup. If not included, it is assumed that a single subgroup is being analyzed.

qalys Quality-adjusted life-years

`cea` *Cost-effectiveness analysis*

Description

Conduct cost-effectiveness analysis (CEA) given output of an economic model; that is, summarize a probabilistic sensitivity analysis (PSA), possibly by subgroup.

- `cea()` computes the probability that each treatment is most cost-effective, output for a cost-effectiveness acceptability frontier, the expected value of perfect information, and the net monetary benefit for each treatment.
- `cea_pw()` conducts pairwise CEA by comparing strategies to a comparator. Computed quantities include the incremental cost-effectiveness ratio, the incremental net monetary benefit, output for a cost-effectiveness plane, and output for a cost-effectiveness acceptability curve.

Usage

```
cea(x, ...)

cea_pw(x, ...)

## Default S3 method:
cea(x, k = seq(0, 2e+05, 500), sample, strategy, grp = NULL, e, c, ...)

## Default S3 method:
cea_pw(
  x,
  k = seq(0, 2e+05, 500),
  comparator,
  sample,
  strategy,
  grp = NULL,
  e,
  c,
  ...
)

## S3 method for class 'ce'
cea(x, k = seq(0, 2e+05, 500), dr_qalys, dr_costs, ...)

## S3 method for class 'ce'
cea_pw(x, k = seq(0, 2e+05, 500), comparator, dr_qalys, dr_costs, ...)
```

Arguments

`x`	An object of simulation output characterizing the probability distribution of clinical effectiveness and costs. If the default method is used, then `x` must be

	a `data.frame` or `data.table` containing columns of mean costs and clinical effectiveness where each row denotes a randomly sampled parameter set and treatment strategy.
`...`	Further arguments passed to or from other methods. Currently unused.
`k`	Vector of willingness to pay values.
`sample`	Character name of column from `x` denoting a randomly sampled parameter set.
`strategy`	Character name of column from `x` denoting treatment strategy.
`grp`	Character name of column from `x` denoting subgroup. If `NULL`, then it is assumed that there is only one group.
`e`	Character name of column from `x` denoting clinical effectiveness.
`c`	Character name of column from `x` denoting costs.
`comparator`	Name of the comparator strategy in `x`.
`dr_qalys`	Discount rate for quality-adjusted life-years (QALYs).
`dr_costs`	Discount rate for costs.

Value

`cea()` returns a list of four `data.table` elements.

summary A `data.table` of the mean, 2.5% quantile, and 97.5% quantile by strategy and group for clinical effectiveness and costs.

mce The probability that each strategy is the most effective treatment for each group for the range of specified willingness to pay values. In addition, the column `best` denotes the optimal strategy (i.e., the strategy with the highest expected net monetary benefit), which can be used to plot the cost-effectiveness acceptability frontier (CEAF).

evpi The expected value of perfect information (EVPI) by group for the range of specified willingness to pay values. The EVPI is computed by subtracting the expected net monetary benefit given current information (i.e., the strategy with the highest expected net monetary benefit) from the expected net monetary benefit given perfect information.

nmb The mean, 2.5% quantile, and 97.5% quantile of net monetary benefits for the range of specified willingness to pay values.

`cea_pw` also returns a list of four `data.table` elements:

summary A data.table of the mean, 2.5% quantile, and 97.5% quantile by strategy and group for incremental clinical effectiveness and costs.

delta Incremental effectiveness and incremental cost for each simulated parameter set by strategy and group. Can be used to plot a cost-effectiveness plane.

ceac Values needed to plot a cost-effectiveness acceptability curve by group. The CEAC plots the probability that each strategy is more cost-effective than the comparator for the specified willingness to pay values.

inmb The mean, 2.5% quantile, and 97.5% quantile of incremental net monetary benefits for the range of specified willingness to pay values.

Examples

```
library("data.table")
library("ggplot2")
theme_set(theme_bw())

# Simulation output
n_samples <- 30

sim <- data.table(sample = rep(seq(n_samples), 4),
                  c = c(rlnorm(n_samples, 5, .1), rlnorm(n_samples, 5, .1),
                        rlnorm(n_samples, 11, .1), rlnorm(n_samples, 11, .1)),
                  e = c(rnorm(n_samples, 8, .2), rnorm(n_samples, 8.5, .1),
                        rnorm(n_samples, 11, .6), rnorm(n_samples, 11.5, .6)),
                  strategy_id = rep(1:2, each = n_samples * 2),
                  grp_id = rep(rep(1:2, each = n_samples), 2)
)

# Cost-effectiveness analysis
cea_out <- cea(sim, k = seq(0, 200000, 500), sample = "sample",
               strategy = "strategy_id", grp = "grp_id",
               e = "e", c = "c")
names(cea_out)

## Some sample output
## The probability that each strategy is the most cost-effective
## in each group with a willingness to pay of 20,000
cea_out$mce[k == 20000]

# Pairwise cost-effectiveness analysis
cea_pw_out <-  cea_pw(sim,  k = seq(0, 200000, 500), comparator = 1,
                      sample = "sample", strategy = "strategy_id",
                      grp = "grp_id", e = "e", c = "c")
names(cea_pw_out)

## Some sample output
## The cost-effectiveness acceptability curve
head(cea_pw_out$ceac[k >= 20000])

# Summarize the incremental cost-effectiveness ratio
labs <- list(strategy_id = c("Strategy 1" = 1, "Strategy 2" = 2),
             grp_id = c("Group 1" = 1, "Group 2" = 2))
format(icer(cea_pw_out, labels = labs))

# Plots
plot_ceplane(cea_pw_out, label = labs)
plot_ceac(cea_out, label = labs)
plot_ceac(cea_pw_out, label = labs)
plot_ceaf(cea_out, label = labs)
plot_evpi(cea_out, label = labs)
```

`CohortDtstm` *Cohort discrete time state transition model*

Description

Simulate outcomes from a cohort discrete time state transition model.

Format

An R6::R6Class object.

Public fields

`trans_model` The model for health state transitions. Must be an object of class `CohortDtstmTrans`.

`utility_model` The model for health state utility. Must be an object of class `StateVals`.

`cost_models` The models used to predict costs by health state. Must be a list of objects of class `StateVals`, where each element of the list represents a different cost category.

`stateprobs_` An object of class `stateprobs` simulated using `$sim_stateprobs()`.

`qalys_` An object of class `qalys` simulated using `$sim_qalys()`.

`costs_` An object of class `costs` simulated using `$sim_costs()`.

Methods

Public methods:

- `CohortDtstm$new()`
- `CohortDtstm$sim_stateprobs()`
- `CohortDtstm$sim_qalys()`
- `CohortDtstm$sim_costs()`
- `CohortDtstm$summarize()`
- `CohortDtstm$clone()`

Method `new()`**:** Create a new `CohortDtstm` object.

Usage:

```
CohortDtstm$new(trans_model = NULL, utility_model = NULL, cost_models = NULL)
```

Arguments:

`trans_model` The `trans_model` field.

`utility_model` The `utility_model` field.

`cost_models` The `cost_models` field.

Returns: A new `CohortDtstm` object.

Method `sim_stateprobs()`**:** Simulate health state probabilities using `CohortDtstmTrans$sim_stateprobs()`.

Usage:

```
CohortDtstm$sim_stateprobs(n_cycles)
```

Arguments:

`n_cycles` The number of model cycles to simulate the model for.

Returns: An instance of `self` with simulated output of class stateprobs stored in `stateprobs_`.

Method `sim_qalys()`**:** Simulate quality-adjusted life-years (QALYs) as a function of `stateprobs_` and `utility_model`. See `sim_qalys()` for details.

Usage:

```
CohortDtstm$sim_qalys(
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right"),
  lys = TRUE
)
```

Arguments:

`dr` Discount rate.

`integrate_method` Method used to integrate state values when computing costs or QALYs. Options are `trapz` for the trapezoid rule, `riemann_left` for a left Riemann sum, and `riemann_right` for a right Riemann sum.

`lys` If `TRUE`, then life-years are simulated in addition to QALYs.

Returns: An instance of `self` with simulated output of class qalys stored in `qalys_`.

Method `sim_costs()`**:** Simulate costs as a function of `stateprobs_` and `cost_models`. See `sim_costs()` for details.

Usage:

```
CohortDtstm$sim_costs(
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right")
)
```

Arguments:

`dr` Discount rate.

`integrate_method` Method used to integrate state values when computing costs or QALYs. Options are `trapz` for the trapezoid rule, `riemann_left` for a left Riemann sum, and `riemann_right` for a right Riemann sum.

Returns: An instance of `self` with simulated output of class costs stored in `costs_`.

Method `summarize()`**:** Summarize costs and QALYs so that cost-effectiveness analysis can be performed. See `summarize_ce()`.

Usage:

```
CohortDtstm$summarize(by_grp = FALSE)
```

Arguments:

`by_grp` If `TRUE`, then costs and QALYs are computed by subgroup. If `FALSE`, then costs and QALYs are aggregated across all patients (and subgroups).

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CohortDtstm$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Incerti and Jansen (2021). See Section 2.1 for a description of a cohort DTSTM and details on simulating costs and QALYs from state probabilities. An example in oncology is provided in Section 4.3.

See Also

`CohortDtstm` objects can be created from model objects as documented in `create_CohortDtstm()`. The `CohortDtstmTrans` documentation describes the class for the transition model and the `StateVals` documentation describes the class for the cost and utility models. A `CohortDtstmTrans` object is typically created using `create_CohortDtstmTrans()`.

There are currently three relevant vignettes. `vignette("markov-cohort")` details a relatively simple Markov model and `vignette("markov-inhomogeneous-cohort")` describes a more complex time inhomogeneous model in which transition probabilities vary in every model cycle. The `vignette("mlogit")` shows how a transition model can be parameterized using a multinomial logistic regression model when transition data is collected at evenly spaced intervals.

Examples

```
library("data.table")
library("ggplot2")
theme_set(theme_bw())
set.seed(102)

# NOTE: This example replicates the "Simple Markov cohort model"
# vignette using a different approach. Here, we explicitly construct
# the transition probabilities "by hand". In the vignette, the transition
# probabilities are defined using expressions (i.e., by using
# `define_model()`). The `define_model()` approach does (more or less) what
# is done here under the hood.

# (0) Model setup
hesim_dat <- hesim_data(
  strategies = data.table(
    strategy_id = 1:2,
    strategy_name = c("Monotherapy", "Combination therapy")
  ),
  patients <- data.table(patient_id = 1),
  states = data.table(
    state_id = 1:3,
    state_name = c("State A", "State B", "State C")
  )
)
n_states <- nrow(hesim_dat$states) + 1
labs <- get_labels(hesim_dat)

# (1) Parameters
n_samples <- 10 # Number of samples for PSA

## Transition matrix
### Input data (one transition matrix for each parameter sample,
```

```
###             treatment strategy, patient, and time interval)
p_id <- tpmatrix_id(expand(hesim_dat, times = c(0, 2)), n_samples)
N <- nrow(p_id)

### Transition matrices (one for each row in p_id)
p <- array(NA, dim = c(n_states, n_states, nrow(p_id)))

#### Baseline risk
trans_mono <- rbind(
  c(1251, 350, 116, 17),
  c(0, 731, 512, 15),
  c(0, 0, 1312, 437),
  c(0, 0, 0, 469)
)
mono_ind <- which(p_id$strategy_id == 1 | p_id$time_id == 2)
p[,, mono_ind] <- rdirichlet_mat(n = 2, trans_mono)

#### Apply relative risks
combo_ind <- setdiff(1:nrow(p_id), mono_ind)
lrr_se <- (log(.710) - log(.365))/(2 * qnorm(.975))
rr <- rlnorm(n_samples, meanlog = log(.509), sdlog = lrr_se)
rr_indices <- list( # Indices of transition matrix to apply RR to
  c(1, 2), c(1, 3), c(1, 4),
  c(2, 3), c(2, 4),
  c(3, 4)
)
rr_mat <- matrix(rr, nrow = n_samples, ncol = length(rr_indices))
p[,, combo_ind] <- apply_rr(p[, , mono_ind],
                            rr = rr_mat,
                            index = rr_indices)
tp <- tparams_transprobs(p, p_id)

## Utility
utility_tbl <- stateval_tbl(
  data.table(
    state_id = 1:3,
    est = c(1, 1, 1)
  ),
  dist = "fixed"
)

## Costs
drugcost_tbl <- stateval_tbl(
  data.table(
    strategy_id = c(1, 1, 2, 2),
    time_start = c(0, 2, 0, 2),
    est = c(2278, 2278, 2278 + 2086.50, 2278)
  ),
  dist = "fixed"
)

dmedcost_tbl <- stateval_tbl(
  data.table(
```

```
    state_id = 1:3,
    mean = c(A = 1701, B = 1774, C = 6948),
    se = c(A = 1701, B = 1774, C = 6948)
  ),
  dist = "gamma"
)

cmedcost_tbl <- stateval_tbl(
  data.table(
    state_id = 1:3,
    mean = c(A = 1055, B = 1278, C = 2059),
    se = c(A = 1055, B = 1278, C = 2059)
  ),
  dist = "gamma"
)

# (2) Simulation
## Constructing the economic model
### Transition probabilities
transmod <- CohortDtstmTrans$new(params = tp)

### Utility
utilitymod <- create_StateVals(utility_tbl,
                               hesim_data = hesim_dat,
                               n = n_samples)

### Costs
drugcostmod <- create_StateVals(drugcost_tbl,
                                hesim_data = hesim_dat,
                                n = n_samples)
dmedcostmod <- create_StateVals(dmedcost_tbl,
                                hesim_data = hesim_dat,
                                n = n_samples)
cmedcostmod <- create_StateVals(cmedcost_tbl,
                                hesim_data = hesim_dat,
                                n = n_samples)
costmods <- list(drug = drugcostmod,
                 direct_medical = dmedcostmod,
                 community_medical = cmedcostmod)

### Economic model
econmod <- CohortDtstm$new(trans_model = transmod,
                           utility_model = utilitymod,
                           cost_models = costmods)

## Simulating outcomes
econmod$sim_stateprobs(n_cycles = 20)
autoplot(econmod$stateprobs_, ci = TRUE, ci_style = "ribbon",
         labels = labs)
econmod$sim_qalys(dr = 0, integrate_method = "riemann_right")
econmod$sim_costs(dr = 0.06, integrate_method = "riemann_right")

# (3) Decision analysis
```

```
ce_sim <- econmod$summarize()
wtp <- seq(0, 25000, 500)
cea_pw_out <- cea_pw(ce_sim, comparator = 1, dr_qalys = 0, dr_costs = .06,
                     k = wtp)
format(icer(cea_pw_out))
```

`CohortDtstmTrans` *Transitions for a cohort discrete time state transition model*

Description

Simulate health state transitions in a cohort discrete time state transition model.

Format

An R6::R6Class object.

Public fields

`params` Parameters for simulating health state transitions. Supports objects of class `tparams_transprobs` or `params_mlogit_list`.

`input_data` An object of class `input_mats`.

`cycle_length` The length of a model cycle in terms of years. The default is `1` meaning that model cycles are 1 year long.

`absorbing` A numeric vector denoting the states that are absorbing states; i.e., states that cannot be transitioned from. Each element should correspond to a `state_id`, which should, in turn, be the index of the health state.

Active bindings

`start_stateprobs` A non-negative vector with length equal to the number of health states containing the probability that the cohort is in each health state at the start of the simulation. For example, if there were three states and the cohort began the simulation in state 1, then `start_stateprobs = c(1, 0, 0)`. Automatically normalized to sum to 1. If `NULL`, then a vector with the first element equal to 1 and all remaining elements equal to 0.

`trans_mat` A transition matrix describing the states and transitions in a discrete-time multi-state model. Only required if the model is parameterized using multinomial logistic regression. The `(i,j)` element represents a transition from state `i` to state `j`. Each possible transition from row `i` should be based on a separate multinomial logistic regression and ordered from `0` to `K - 1` where `K` is the number of possible transitions. Transitions that are not possible should be `NA`. and the reference category for each row should be `0`.

Methods

Public methods:

- `CohortDtstmTrans$new()`
- `CohortDtstmTrans$sim_stateprobs()`
- `CohortDtstmTrans$clone()`

Method `new()`**:** Create a new `CohortDtstmTrans` object.

Usage:

```
CohortDtstmTrans$new(
  params,
  input_data = NULL,
  trans_mat = NULL,
  start_stateprobs = NULL,
  cycle_length = 1,
  absorbing = NULL
)
```

Arguments:

`params` The `params` field.

`input_data` The `input_data` field.

`trans_mat` The `trans_mat` field.

`start_stateprobs` The `start_stateprobs` field.

`cycle_length` The `cycle_length` field.

`absorbing` The `absorbing` field. If `NULL`, then the constructor will determine which states are absorbing automatically; non `NULL` values will override this behavior.

Returns: A new `CohortDtstmTrans` object.

Method `sim_stateprobs()`**:** Simulate probability of being in each health state during each model cycle.

Usage:

```
CohortDtstmTrans$sim_stateprobs(n_cycles)
```

Arguments:

`n_cycles` The number of model cycles to simulate the model for.

Returns: An object of class `stateprobs`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CohortDtstmTrans$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`create_CohortDtstmTrans()` creates a `CohortDtstmTrans` object from either a fitted statistical model or a parameter object. A complete economic model can be implemented with the `CohortDtstm` class.

Examples

```
library("msm")
library("data.table")
set.seed(101)

# We consider two examples that have the same treatment strategies and patients.
# One model is parameterized by fitting a multi-state model with the "msm"
# package; in the second model, the parameters are entered "manually" with
# a "params_mlogit_list" object.

# MODEL SETUP
strategies <- data.table(
  strategy_id = c(1, 2, 3),
  strategy_name = c("SOC", "New 1", "New 2")
)
patients <- data.table(patient_id = 1:2)
hesim_dat <- hesim_data(
  strategies = strategies,
  patients = patients
)

# EXAMPLE #1: msm
## Fit multi-state model with panel data via msm
qinit <- rbind(
  c(0, 0.28163, 0.01239),
  c(0, 0, 0.10204),
  c(0, 0, 0)
)
fit <- msm(state_id ~ time, subject = patient_id,
           data = onc3p[patient_id %in% sample(patient_id, 100)],
           covariates = list("1-2" =~ strategy_name),
           qmatrix = qinit)

## Simulation model
transmod_data <- expand(hesim_dat)
transmod <- create_CohortDtstmTrans(fit,
                                    input_data = transmod_data,
                                    cycle_length = 1/2,
                                    fixedpars = 2,
                                    n = 2)
transmod$sim_stateprobs(n_cycles = 2)

# EXAMPLE #2: params_mlogit_list
## Input data
transmod_data[, intercept := 1]
transmod_data[, new1 := ifelse(strategy_name == "New 1", 1, 0)]
transmod_data[, new2 := ifelse(strategy_name == "New 2", 1, 0)]

## Parameters
n <- 10
transmod_params <- params_mlogit_list(
```

```
    ## Transitions from stable state (stable -> progression, stable -> death)
    stable = params_mlogit(
      coefs = list(
        progression = data.frame(
          intercept = rnorm(n, -0.65, .1),
          new1 = rnorm(n, log(.8), .02),
          new2 = rnorm(n, log(.7, .02))
        ),
        death = data.frame(
          intercept = rnorm(n, -3.75, .1),
          new1 = rep(0, n),
          new2 = rep(0, n)
        )
      )
    ),

    ## Transition from progression state (progression -> death)
    progression = params_mlogit(
      coefs = list(
        death = data.frame(
          intercept = rnorm(n, 2.45, .1),
          new1 = rep(0, n),
          new2 = rep(0, n)
        )
      )
    )
  )
  transmod_params

  ## Simulation model
  tmat <- rbind(c(0, 1, 2),
                c(NA, 0, 1),
                c(NA, NA, NA))
  transmod <- create_CohortDtstmTrans(transmod_params,
                                      input_data = transmod_data,
                                      trans_mat = tmat, cycle_length = 1)
  transmod$sim_stateprobs(n_cycles = 2)
```

`costs` *Costs object*

Description

An object of class `costs` returned from methods `$sim_costs()` in model classes that store simulated costs.

Components

A `costs` object inherits from `data.table` and contains the following columns:

sample	A random sample from the PSA.
strategy_id	The treatment strategy ID.
patient_id	The patient ID.
grp_id	The subgroup ID.
state_id	The health state ID.
dr	The rate used to discount costs.
category	The cost category (e.g., drug costs, medical costs, etc).
costs	The simulated cost values.

`create_CohortDtstm` *Create* `CohortDtstm` *object*

Description

A generic function for creating an object of class `CohortDtstm`.

Usage

```
create_CohortDtstm(object, ...)

## S3 method for class 'model_def'
create_CohortDtstm(
  object,
  input_data,
  cost_args = NULL,
  utility_args = NULL,
  ...
)
```

Arguments

`object`	An object of the appropriate class.
`...`	Further arguments passed to `CohortDtstmTrans$new()` in `CohortDtstmTrans`.
`input_data`	An object of class `expanded_hesim_data`.
`cost_args`	A list of further arguments passed to `StateVals$new()` in `StateVals` when initiating cost models.
`utility_args`	A list of further arguments passed to `StateVals$new()` in `StateVals` when initiating the utility model.

`create_CohortDtstmTrans`

Create `CohortDtstmTrans` *object*

Description

A generic function for creating an object of class `CohortDtstmTrans`.

Usage

```
create_CohortDtstmTrans(object, ...)

## S3 method for class 'multinom_list'
create_CohortDtstmTrans(
  object,
  input_data,
  trans_mat,
  n = 1000,
  uncertainty = c("normal", "none"),
  ...
)

## S3 method for class 'msm'
create_CohortDtstmTrans(
  object,
  input_data,
  cycle_length,
  n = 1000,
  uncertainty = c("normal", "none"),
  ...
)

## S3 method for class 'params_mlogit_list'
create_CohortDtstmTrans(object, input_data, trans_mat, ...)
```

Arguments

`object`	An object of the appropriate class containing either a fitted statistical model or model parameters.
`...`	Further arguments passed to `CohortDtstmTrans$new()` in `CohortDtstmTrans`.
`input_data`	An object of class `expanded_hesim_data` returned by `expand.hesim_data()`
`trans_mat`	A transition matrix describing the states and transitions in a discrete-time multi-state model. See `CohortDtstmTrans`.
`n`	Number of random observations to draw. Not used if `uncertainty = "none"`.

`uncertainty`	Method determining how parameter uncertainty should be handled. If `"normal"`, then parameters are randomly drawn from their multivariate normal distribution. If `"none"`, then only point estimates are returned.
`cycle_length`	The length of a model cycle in terms of years. The default is 1 meaning that model cycles are 1 year long.

Details

Disease models may either be created from a fitted statistical model or from a parameter object. In the case of the former, `input_data` is a data frame like object that is used to look for variables from the statistical model that are required for simulation. In this sense, `input_data` is very similar to the `newdata` argument in most `predict()` methods (e.g., see `predict.lm()`). In other words, variables used in the `formula` of the statistical model must also be in `input_data`.

In the case of the latter, the columns of `input_data` must be named in a manner that is consistent with the parameter object. In the typical case (e.g., with `params_surv` or `params_mlogit`), the parameter object contains coefficients from a regression model, usually stored as matrix where rows index parameter samples (i.e., for a probabilistic sensitivity analysis) and columns index model terms. In such instances, there must be one column from `input_data` with the same name as each model term in the coefficient matrix; that is, the columns in `input_data` are matched with the columns of the coefficient matrices by name. If there are model terms in the coefficient matrices that are not contained in `input_data`, then an error will be thrown.

See Also

See `CohortDtstmTrans` for examples.

`create_IndivCtstmTrans`

Create `IndivCtstmTrans` *object*

Description

A generic function for creating an object of class `IndivCtstmTrans`.

Usage

```
create_IndivCtstmTrans(object, ...)

## S3 method for class 'flexsurvreg_list'
create_IndivCtstmTrans(
  object,
  input_data,
  trans_mat,
  clock = c("reset", "forward"),
  n = 1000,
  uncertainty = c("normal", "none"),
  ...
```

```
)

## S3 method for class 'flexsurvreg'
create_IndivCtstmTrans(
  object,
  input_data,
  trans_mat,
  clock = c("reset", "forward"),
  n = 1000,
  uncertainty = c("normal", "none"),
  ...
)

## S3 method for class 'params_surv'
create_IndivCtstmTrans(
  object,
  input_data,
  trans_mat,
  clock = c("reset", "forward", "mix"),
  reset_states = NULL,
  ...
)

## S3 method for class 'params_surv_list'
create_IndivCtstmTrans(
  object,
  input_data,
  trans_mat,
  clock = c("reset", "forward", "mix"),
  reset_states = NULL,
  ...
)
```

Arguments

`object`	An object of the appropriate class containing either a fitted multi-state model or parameters of a multi-state model.
`...`	Further arguments passed to `IndivCtstmTrans$new()` in `IndivCtstmTrans`.
`input_data`	An object of class `expanded_hesim_data` returned by `expand.hesim_data`.
`trans_mat`	The transition matrix describing the states and transitions in a multi-state model in the format from the `mstate` package. See `IndivCtstmTrans`.
`clock`	"reset" for a clock-reset model, "forward" for a clock-forward model, and "mix" for a mixture of clock-reset and clock-forward models. See the field `clock` in `IndivCtstmTrans`.
`n`	Number of random observations to draw. Not used if `uncertainty = "none"`.
`uncertainty`	Method determining how parameter uncertainty should be handled. If `"normal"`, then parameters are randomly drawn from their multivariate normal distribution. If `"none"`, then only point estimates are returned.

`reset_states`	A vector denoting the states in which time resets. See the field `reset_states` in `IndivCtstmTrans`.

Details

Disease models may either be created from a fitted statistical model or from a parameter object. In the case of the former, `input_data` is a data frame like object that is used to look for variables from the statistical model that are required for simulation. In this sense, `input_data` is very similar to the `newdata` argument in most `predict()` methods (e.g., see `predict.lm()`). In other words, variables used in the `formula` of the statistical model must also be in `input_data`.

In the case of the latter, the columns of `input_data` must be named in a manner that is consistent with the parameter object. In the typical case (e.g., with `params_surv` or `params_mlogit`), the parameter object contains coefficients from a regression model, usually stored as matrix where rows index parameter samples (i.e., for a probabilistic sensitivity analysis) and columns index model terms. In such instances, there must be one column from `input_data` with the same name as each model term in the coefficient matrix; that is, the columns in `input_data` are matched with the columns of the coefficient matrices by name. If there are model terms in the coefficient matrices that are not contained in `input_data`, then an error will be thrown.

Value

Returns an `R6Class` object of class `IndivCtstmTrans`.

See Also

See `IndivCtstmTrans` and `IndivCtstm` for examples.

`create_params` *Create a parameter object from a fitted model*

Description

`create_params` is a generic function for creating an object containing parameters from a fitted statistical model. If `uncertainty != "none"`, then random samples from suitable probability distributions are returned.

Usage

```
create_params(object, ...)

## S3 method for class 'lm'
create_params(object, n = 1000, uncertainty = c("normal", "none"), ...)

## S3 method for class 'multinom'
create_params(object, n = 1000, uncertainty = c("normal", "none"), ...)

## S3 method for class 'multinom_list'
```

```
create_params(object, n = 1000, uncertainty = c("normal", "none"), ...)

## S3 method for class 'flexsurvreg'
create_params(object, n = 1000, uncertainty = c("normal", "none"), ...)

## S3 method for class 'flexsurvreg_list'
create_params(object, n = 1000, uncertainty = c("normal", "none"), ...)

## S3 method for class 'partsurvfit'
create_params(
  object,
  n = 1000,
  uncertainty = c("normal", "bootstrap", "none"),
  max_errors = 0,
  silent = FALSE,
  ...
)
```

Arguments

`object`	A statistical model to randomly sample parameters from.
`...`	Currently unused.
`n`	Number of random observations to draw. Not used if `uncertainty` = `"none"`.
`uncertainty`	Method determining how parameter uncertainty should be handled. If `"normal"`, then parameters are randomly drawn from their multivariate normal distribution. If `"bootstrap"`, then parameters are bootstrapped using `bootstrap`. If `"none"`, then only point estimates are returned.
`max_errors`	Maximum number of errors that are allowed when fitting statistical models during the bootstrap procedure. This argument may be useful if, for instance, the model fails to converge during some bootstrap replications. Default is 0.
`silent`	Logical indicating whether error messages should be suppressed. Passed to the `silent` argument of `try()`.

Value

An object prefixed by `params_`. Mapping between `create_params` and the classes of the returned objects are:

- `create_params.lm` -> `params_lm`
- `create_params.multinom` -> `params_mlogit`
- `create_params.multinom_list` -> `params_mlogit_list`
- `create_params.flexsurvreg` -> `params_surv`
- `create_params.flexsurvreg_list` -> `params_surv_list`
- `create_params.partsurvfit` -> `params_surv_list`

See Also

These methods are typically used alongside `create_input_mats()` to create model objects as a function of input data and a fitted statistical model. For instance, `create_PsmCurves()` creates the survival model for a partitioned survival model, `create_IndivCtstmTrans()` creates the transition model for an individual continuous time state transition model, `create_CohortDtstmTrans()` creates the transition model for a cohort discrete time state transition model, and `create_StateVals()` creates a health state values model.

Examples

```
# create_params.lm
fit <- lm(costs ~ female, data = psm4_exdata$costs$medical)
n <- 5
params_lm <- create_params(fit, n = n)
head(params_lm$coefs)
head(params_lm$sigma)

# create_params.flexsurvreg
library("flexsurv")
fit <- flexsurvreg(formula = Surv(futime, fustat) ~ 1,
                   data = ovarian, dist = "weibull")
n <- 5
params_surv_wei <- create_params(fit, n = n)
print(params_surv_wei$dist)
head(params_surv_wei$coefs)
```

`create_PsmCurves` *Create* `PsmCurves` *object*

Description

A generic function for creating a `PsmCurves` object.

Usage

```
create_PsmCurves(object, ...)

## S3 method for class 'flexsurvreg_list'
create_PsmCurves(
  object,
  input_data,
  n = 1000,
  uncertainty = c("normal", "bootstrap", "none"),
  est_data = NULL,
  ...
)

## S3 method for class 'params_surv_list'
create_PsmCurves(object, input_data, ...)
```

Arguments

`object`	An object of the appropriate class containing either fitted survival models or parameters of survival models.
`...`	Further arguments passed to or from other methods. Passed to `create_params.partsurvfit()` when `object` is of class `flexsurvreg_list`.
`input_data`	An object of class `expanded_hesim_data` returned by `expand.hesim_data()`. Must be expanded by the data tables `"strategies"` and `"patients"`.
`n`	Number of random observations to draw. Not used if `uncertainty = "none"`.
`uncertainty`	Method determining how parameter uncertainty should be handled. If `"normal"`, then parameters are randomly drawn from their multivariate normal distribution. If `"bootstrap"`, then parameters are bootstrapped using `bootstrap`. If `"none"`, then only point estimates are returned.
`est_data`	A `data.table` or `data.frame` of estimation data used to fit survival models during bootstrap replications.

Details

Disease models may either be created from a fitted statistical model or from a parameter object. In the case of the former, `input_data` is a data frame like object that is used to look for variables from the statistical model that are required for simulation. In this sense, `input_data` is very similar to the `newdata` argument in most `predict()` methods (e.g., see `predict.lm()`). In other words, variables used in the `formula` of the statistical model must also be in `input_data`.

In the case of the latter, the columns of `input_data` must be named in a manner that is consistent with the parameter object. In the typical case (e.g., with `params_surv` or `params_mlogit`), the parameter object contains coefficients from a regression model, usually stored as matrix where rows index parameter samples (i.e., for a probabilistic sensitivity analysis) and columns index model terms. In such instances, there must be one column from `input_data` with the same name as each model term in the coefficient matrix; that is, the columns in `input_data` are matched with the columns of the coefficient matrices by name. If there are model terms in the coefficient matrices that are not contained in `input_data`, then an error will be thrown.

Value

Returns an `R6Class` object of class `PsmCurves`.

See Also

See `PsmCurves` and `Psm` for examples. `PsmCurves` provides an example in which a model is parameterized both with (via `create_PsmCurves.flexsurvreg_list()`) and without (via `create_PsmCurves.params_surv_list` access to patient-level data. The `Psm` example shows how state probabilities, costs, and utilities can be computed from predicted survival curves.

`create_StateVals` *Create a* `StateVals` *object*

Description

`create_StateVals()` is a generic function for creating an object of class `StateVals` from a fitted statistical model or a `stateval_tbl` object.

Usage

```
create_StateVals(object, ...)

## S3 method for class 'lm'
create_StateVals(
  object,
  input_data = NULL,
  n = 1000,
  uncertainty = c("normal", "none"),
  ...
)

## S3 method for class 'stateval_tbl'
create_StateVals(object, hesim_data = NULL, n = 1000, ...)
```

Arguments

`object`	A model object of the appropriate class.
`...`	Further arguments (`time_reset` and `method`) passed to `StateVals$new()`.
`input_data`	An object of class `expanded_hesim_data`. Must be expanded by treatment strategies, patients, and health states.
`n`	Number of random observations of the parameters to draw when parameters are fit using a statistical model.
`uncertainty`	Method determining how parameter uncertainty should be handled. See documentation in `create_params()`.
`hesim_data`	A `hesim_data` object. Only required when `object` is of class `stateval_tbl`. See "details".

Details

If `object` is a `stateval_tbl`, then a `hesim_data` object is used to specify treatment strategies, patients, and/or health states not included as columns in the table, or, to match patients in the table to groups. Not required if the table includes one row for each treatment strategy, patient, and health state combination. Patients are matched to groups by specifying both a `patient_id` and a `grp_var` column in the `patients` table.

Value

A `StateVals` object.

See Also

See `StateVals` for documentation of the class and additional examples. An example use case for `create_StateVals.stateval_tbl()` is provided in the `stateval_tbl()` documentation.

Examples

```
set.seed(10)

# EXAMPLE FOR `create_statevals.lm()`
## Simple example comparing two treatment strategies where
## medical costs vary by sex and health state

## Setup model
hesim_dat <- hesim_data(
  strategies = data.frame(strategy_id = c(1, 2)),
  patients = data.frame(
    patient_id = c(1, 2),
    female = c(1, 0)
  ),
  states = data.frame(
    state_id = c(1, 2, 3),
    state_name = c("state1", "state2", "state3")
  )
)

## Fit model
medcost_estimation_data <- psm4_exdata$costs$medical
medcost_estimation_data$time5 <- rbinom(nrow(medcost_estimation_data),
                                        1, .5) # Illustrative time dummy
medcost_fit <- lm(costs ~ female + state_name + time5,
                  data = medcost_estimation_data)

## Create medical cost model
### Allow medical costs to vary across time in addition to by patient and
### health state
medcost_times <- time_intervals(
  data.frame(time_start = c(0, 3, 5),
            time5 = c(0, 0, 1)) # Time dummy corresponds to time > 5
)
medcost_input_data <- expand(hesim_dat,
                             by = c("strategies", "patients", "states"),
                             times = medcost_times)
medcost_model <- create_StateVals(medcost_fit, medcost_input_data,
                                  n = 1)

## Explore predictions from medical cost model
### We can assess predictions at multiple time points
medcost_model$sim(t = c(1, 6), type = "predict")
```

`create_trans_dt` *Create a data table of health state transitions*

Description

Create a data table of health state transitions from a transition matrix describing the states and transitions in a multi-state model suitable for use with `hesim_data`.

Usage

```
create_trans_dt(trans_mat)
```

Arguments

`trans_mat`	A transition matrix in the format from the `mstate` package. See `IndivCtstmTrans`.

Value

Returns a `data.table` in tidy format with three columns:

transition_id Health state transition ID.

from The starting health state.

to The health state that will be transitions to.

Examples

```
tmat <- rbind(c(NA, 1, 2),
              c(NA, NA, 3),
              c(NA, NA, NA))
create_trans_dt(tmat)
```

`define_model` *Define and evaluate model expression*

Description

A model expression is defined by specifying random number generation functions for a probabilistic sensitivity analysis (PSA) and transformations of the sampled parameters as a function of `input_data`. The unevaluated expressions are evaluated with `eval_model()` and used to generate the model inputs needed to create an economic model.

Usage

```
define_model(tparams_def, rng_def, params = NULL, n_states = NULL)

eval_model(x, input_data)
```

Arguments

`tparams_def`	A tparams_def object or a list of tparams_def objects. A list might be considered if time intervals specified with the `times` argument in `define_tparams()` vary across parameters. Parameters for a transition probability matrix (`tpmatrix`), utilities (`utility`), and/or cost categories (`costs`) are returned as a named list (see `define_tparams()` for more details).
`rng_def`	A rng_def object used to randomly draw samples of the parameters from suitable probability distributions.
`params`	Either (i) a list containing the values of parameters for random number generation or (ii) parameter samples that have already been randomly generated using `eval_rng()`. In case (ii), `rng_def` should be `NULL`.
`n_states`	The number of health states (inclusive of all health states including the the death state) in the model. If `tpmatrix` is an element returned by `tparams_def`, then it will be equal to the number of states in the transition probability matrix; otherwise it must be specified as an argument.
`x`	An object of class `model_def` created with `define_model()`.
`input_data`	An object of class expanded_hesim_data expanded by patients and treatment strategies.

Details

`eval_model()` evaluates the expressions in an object of class `model_def` returned by `define_model()` and is, in turn, used within functions that instantiate economic models (e.g., `create_CohortDtstm()`). The direct output of `eval_model()` can also be useful for understanding and debugging model definitions, but it is not used directly for simulation.

Economic models are constructed as a function of input data and parameters:

1. *Input data*: Objects of class expanded_hesim_data consisting of the treatment strategies and patient population.
2. *Parameters*: The underlying parameter estimates from the literature are first stored in a list (`params` argument). Random number generation is then used to sample the parameters from suitable probability distributions for the PSA (`rng_def` argument). Finally, the sampled parameters are transformed as a function of the input data into values (e.g., elements of a transition probability matrix) used for the simulation (`tparams_def` argument). The `params` argument can be omitted if the underlying parameters values are defined inside a `define_rng()` block.

Value

`define_model()` returns an object of class `model_def`, which is a list containing the arguments to the function. `eval_model()` returns a list containing ID variables identifying parameter samples, treatment strategies, patient cohorts, and time intervals; the values of parameters of the transition probability matrix, utilities, and/or cost categories; the number of health states; and the number of random number generation samples for the PSA.

See Also

`define_tparams()`, `define_rng()`

Examples

```
# Data
library("data.table")
strategies <- data.table(strategy_id = 1:2,
                         strategy_name = c("Monotherapy", "Combination therapy"))
patients <- data.table(patient_id = 1)
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients)
data <- expand(hesim_dat)

# Model parameters
rng_def <- define_rng({
  alpha <- matrix(c(1251, 350, 116, 17,
                    0, 731, 512, 15,
                    0, 0, 1312, 437,
                    0, 0, 0, 469),
                  nrow = 4, byrow = TRUE)
  rownames(alpha) <- colnames(alpha) <- c("A", "B", "C", "D")
  lrr_mean <- log(.509)
  lrr_se <- (log(.710) - log(.365))/(2 * qnorm(.975))

  list(
    p_mono = dirichlet_rng(alpha),
    rr_comb = lognormal_rng(lrr_mean, lrr_se),
    u = 1,
    c_zido = 2278,
    c_lam = 2086.50,
    c_med = gamma_rng(mean = c(A = 2756, B = 3052, C = 9007),
                      sd = c(A = 2756, B = 3052, C = 9007))
  )
}, n = 2)

tparams_def <- define_tparams({
  rr = ifelse(strategy_name == "Monotherapy", 1, rr_comb)
  list(
    tpmatrix = tpmatrix(
      C, p_mono$A_B * rr, p_mono$A_C * rr, p_mono$A_D * rr,
      0, C, p_mono$B_C * rr, p_mono$B_D * rr,
      0, 0, C, p_mono$C_D * rr,
      0, 0, 0, 1),
    utility = u,
    costs = list(
      drug = ifelse(strategy_name == "Monotherapy",
                    c_zido, c_zido + c_lam),
      medical = c_med
    )
  )
})

# Simulation
## Define the economic model
```

```
model_def <- define_model(
  tparams_def = tparams_def,
  rng_def = rng_def)

### Evaluate the model expression to generate model inputs
### This can be useful for understanding the output of a model expression
eval_model(model_def, data)

## Create an economic model with a factory function
econmod <- create_CohortDtstm(model_def, data)
```

`define_rng` *Define and evaluate random number generation expressions*

Description

Random number generation expressions are used to randomly sample model parameters from suitable distributions for probabilistic sensitivity analysis. These functions are typically used when evaluating an object of class `model_def` defined using `define_model()`.

Usage

```
define_rng(expr, n = 1, ...)

eval_rng(x, params = NULL, check = TRUE)
```

Arguments

`expr`	An expression used to randomly draw variates for each parameter of interest in the model. Braces should be used so that the result of the last expression within the braces is evaluated. The expression must return a list where each element is either a `vector` or tabular object (`matrix`, `data.frame`, or `data.table`). The length of the vector must either be or `n` and the number of rows in the tabular object must be `n`.
`n`	Number of samples of the parameters to draw.
`...`	Additional arguments to pass to the environment used to evaluate `expr`.
`x`	An object of class `rng_def` created with `define_rng()`.
`params`	A list containing the values of parameters for random number generation. Each element of the list should either be a `vector`, `matrix`, `data.frame`, or `data.table`
`check`	Whether to check the returned output so that (i) it returns a list and (ii) each element has the correct length or number of rows. Default is `TRUE`, meaning that any output can be returned. This is always `TRUE` when used inside `define_model()`.

Details

`hesim` contains a number of random number generation functions that return parameter samples in convenient formats and do not typically require the number of samples, `n`, as arguments (see `rng_distributions`). The random number generation expressions are evaluated using `eval_rng()` and used within `expr` in `define_rng()`. If a multivariate object is returned by `eval_rng()`, then the rows are random samples and columns are distinct parameters (e.g., costs for each health state, elements of a transition probability matrix).

Value

`define_rng()` returns an object of class `rng_def`, which is a list containing the unevaluated random number generation expressions passed to `expr`, `n`, and any additional arguments passed to `...` . `eval_rng()` evaluates the `rng_def` object and returns an `eval_rng` object containing the evaluated expression.

See Also

Parameters can be conveniently sampled from probability distributions using a number of random number generation functions (see `rng_distributions`). An economic model can be created with `create_CohortDtstm()` by using `define_rng()` (or a previously evaluated `eval_rng` object) alongside `define_tparams()` to define a model with `define_model()`. It can be useful to summarize an evaluated expression with `summary.eval_rng()`.

Examples

```
params <- list(
  alpha = matrix(c(75, 25, 33, 67), byrow = TRUE, ncol = 2),
  inptcost_mean = c(A = 900, B = 1500, C = 2000),
  outptcost_mean = matrix(c(300, 600, 800,
                             400, 700, 700),
                           ncol = 3, byrow = TRUE)
)
rng_def <- define_rng({
  aecost_mean <- c(500, 800, 1000) # Local object not
                                   # not returned by eval_rng()
  list( # Sampled values of parameters returned by eval_rng()
    p = dirichlet_rng(alpha), # Default column names
    inptcost = gamma_rng(mean = inptcost_mean, # Column names based on
                         sd = inptcost_mean),  # named vector
    outptcost = outptcost_mean, # No column names because
                                # outptcost_mean has none.
    aecost = gamma_rng(mean = aecost_mean, # Explicit naming of columns
                       sd = aecost_mean,
                       names = aecost_colnames)
  )
}, n = 2, aecost_colnames = c("A", "B", "C")) # Add aecost_colnames to environment
params_sample <- eval_rng(x = rng_def, params)
summary(params_sample)
params_sample
```

`define_tparams` *Define and evaluate transformed parameter expressions*

Description

Transformed parameter expressions are used to transform the parameter values sampled with `eval_rng()` as a function of input data (treatment strategies and patients) and time intervals. These functions are used when evaluating an object of class `model_def` defined using `define_model()`. The transformed parameters are ultimately converted into tparams objects and used to simulate outcomes with an economic model.

Usage

```
define_tparams(expr, times = NULL, ...)

eval_tparams(x, input_data, rng_params)
```

Arguments

`expr`	Expressions used to transform parameters. As with `define_rng()`, braces should be used so that the result of the last expression within the braces is evaluated. The expression must return a named list with the following possible elements: • *tpmatrix*: The transition probability matrix used to simulate transition probabilities in the economic model. This should either be the output of `tpmatrix()` or a 3-dimensional array as in `tparams_transprobs()`. • *utility*: The utility values to attach to states and used to simulate quality-adjusted life-years in the economic model. Either a vector (in which case utility is the same in each health state) or a `data.table`/`data.frame`/`matrix` with a column for each (non-death) health state. • *costs*: A named list of costs for each category used to simulate costs in the economic model. Each element of the list must be in the same format as `utility`.
`times`	Distinct times denoting the stopping time of time intervals.
`...`	Additional arguments to pass to the environment used to evaluate `expr`.
`x`	An object of class `tparams_def`.
`input_data`	An object of class expanded_hesim_data (as in `eval_model()`) expanded by the distinct times in `times`.
`rng_params`	Random samples of the parameters returned by `eval_rng()`.

Details

`define_tparams()` is evaluated when creating economic models as a function of `model_def` objects defined with `define_model()`. Operations are "vectorized" in the sense that they are performed for each unique combination of `input_data` and `params`. `expr` is evaluated in an environment including each variable from `input_data`, all elements of `rng_params`, and a variable `time` containing the values from `times`. The `time` variable can be used to create models where parameters vary as a function of time. `eval_tparams()` is not exported and is only meant for use within `eval_model()`.

Value

`define_tparams()` returns an object of class `tparams_def`, which is a list containing the unevaluated "transformation" expressions passed to `expr`, `times`, and any additional arguments passed to `...` . `eval_tparams()` evaluates the `tparams_def` object and should return a list of transformed parameter objects.

See Also

`define_model()`, `define_rng()`

`disprog` *Disease progression object*

Description

An object of class `disprog` returned from methods `$sim_disease()` in model classes. It contains simulated trajectories through a multi-state model.

Components

A `disprog` object inherits from `data.table` and contains the following columns:

sample A random sample from the PSA.

strategy_id The treatment strategy ID.

patient_id The patient ID.

from The health state ID transitioned from.

to The health state ID transitioned to.

final An indicator equal to 1 if a patient is in their final health state during the simulation and 0 otherwise.

time_start The time at the start of the interval.

time_stop The time at the end of the interval.

The object also contains `size` and `absorbing` attributes. The `size` attribute is a numeric vector with the elements `n_samples`, `n_strategies`, `n_patients`, and `n_states` denoting the number of samples, treatment strategies, patients, and health states The `absorbing` attribute is a numeric vector containing the absorbing health states; i.e., the health states that cannot be transitioned from. Operationally, an absorbing state is a row in a transition matrix (as in the `trans_mat` field of the `IndivCtstmTrans` class) with all `NA`s.

See Also

A disease progression object can be simulated with either the `IndivCtstm` or `IndivCtstmTrans` classes.

`expand.hesim_data` *Expand hesim_data*

Description

Create a data table in long format from all combinations of specified tables from an object of class hesim_data and optionally time intervals. See "Details" for an explanation of how the expansion is done.

Usage

```
## S3 method for class 'hesim_data'
expand(object, by = c("strategies", "patients"), times = NULL)
```

Arguments

`object`	An object of class `hesim_data`.
`by`	A character vector of the names of the data tables in `hesim_data` to expand by.
`times`	Either a numeric vector of distinct times denoting the start of time intervals or a time_intervals object.

Details

This function is similar to `expand.grid()`, but works for data frames or data tables. Specifically, it creates a `data.table` from all combinations of the supplied tables in `object` and optionally the start of times intervals in `times`. The supplied tables are determined using the by argument. The resulting dataset is sorted by prioritizing ID variables as follows: (i) `strategy_id`, (ii) `patient_id`, (iii) the health-related ID variable (either `state_id` or `transition_id`, and (iv) the time intervals from `times`.

Value

An object of class `expanded_hesim_data`, which is a `data.table` with an "id_vars" attribute containing the names of the ID variables in the data table and, if `times` is not `NULL`, a `time_intervals` object derived from `times`.

Examples

```
strategies <- data.frame(strategy_id = c(1, 2))
patients <- data.frame(patient_id = seq(1, 3), age = c(65, 50, 75),
                          gender = c("Female", "Female", "Male"))
states <- data.frame(state_id =  seq(1, 3),
                     state_var = c(2, 1, 9))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients,
                        states = states)
expand(hesim_dat, by = c("strategies", "patients"))
expand(hesim_dat, by = c("strategies", "patients"),
       times = c(0, 2, 10))
```

`expmat` *Matrix exponential*

Description

This is a wrapper around `msm::MatrixExp()` that computes the exponential of multiple square matrices.

Usage

```
expmat(x, t = 1, ...)
```

Arguments

`x`	An array of matrices.
`t`	An optional scaling factor for `x`.
`...`	Arguments to pass to `msm::MatrixExp()`.

Details

This function is most useful when exponentiating transition intensity matrices to produce transition probability matrices. To create transition probability matrices for discrete time state transition models with annual cycles, set `t=1`. An array of matrices is returned which can be used to create the `value` element of a `tparams_transprobs` object. See `qmatrix()` for an example.

Value

Returns an array of exponentiated matrices. If `length(t)` > 1, then `length(t)` arrays are returned for each element in `x`.

See Also

`qmatrix.msm()`, `qmatrix.data.table()`

`fast_rgengamma` *Random generation for generalized gamma distribution*

Description

Draw random samples from a generalized gamma distribution using the parameterization from `flexsurv`. Written in C++ for speed. Equivalent to `flexsurv::rgengamma`.

Usage

```
fast_rgengamma(n, mu = 0, sigma = 1, Q)
```

Arguments

`n`	Number of random observations to draw.
`mu`	Vector of location parameters. and columns correspond to rates during specified time intervals.
`sigma`	Vector of scale parameters as described in `flexsurv`.
`Q`	Vector of shape parameters.

Value

A vector of random samples from the generalized gamma distribution. The length of the sample is determined by n. The numerical arguments other than n are recycled so that the number of samples is equal to n.

Examples

```
n <- 1000
m <- 2 ; s <- 1.7; q <- 1
ptm <- proc.time()
r1 <- fast_rgengamma(n, mu = m, sigma = s, Q = q)
proc.time() - ptm
ptm <- proc.time()
library("flexsurv")
r2 <- flexsurv::rgengamma(n, mu = m, sigma = s, Q = q)
proc.time() - ptm
summary(r1)
summary(r2)
```

`flexsurvreg_list` *List of* `flexsurvreg` *objects*

Description

Combine `flexsurvreg` objects into a list.

Usage

```
flexsurvreg_list(...)
```

Arguments

`...`	Objects of class `flexsurvreg`, which can be named.

Value

An object of class `flexsurvreg_list`.

Examples

```
library("flexsurv")
fit1 <- flexsurv::flexsurvreg(formula = Surv(futime, fustat) ~ 1, data = ovarian, dist = "weibull")
fit2 <- flexsurv::flexsurvreg(formula = Surv(futime, fustat) ~ 1, data = ovarian, dist = "exp")
fsreg_list <- flexsurvreg_list(wei = fit1, exp = fit2)
class(fsreg_list)
```

`get_labels` *Get value labels*

Description

Get value labels for the ID variables in a `hesim_data` object and create a list of named vectors that can be passed to formatting and plotting functions. This lets users create nice labels for treatment strategies, subgroups, health states, and/or transitions when presenting results.

Usage

```
get_labels(
  object,
  strategy = "strategy_name",
  grp = "grp_name",
  state = "state_name",
  transition = "transition_name",
  death_label = "Death"
)
```

Arguments

`object`	An object of class `hesim_data` created with `hesim_data()`.
`strategy`	The name of the column in the `strategy` element of `object` containing labels for `strategy_id`.
`grp`	The name of the column in the `patient` element of `object` containing labels for `grp_id`.
`state`	The name of the column in the `state` element of `object` containing labels for `state_id`.
`transition`	The name of the column in the `transition` element of `object` containing labels for `transition_id`.
`death_label`	The label to use for the death health state. By default a label named "Death" will be concatenated to the labels for the non-death health states. The death state can be omitted from labels for the health states by setting `death_label = NULL`.

Value

A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the ID variables.

See Also

`hesim_data()`, `set_labels()`

Examples

```
library("data.table")
strategies <- data.table(
  strategy_id = c(1, 2),
  strategy_name = c("Strategy 1", "Strategy 2")
)
patients <- data.table(
  patient_id = seq(1, 4),
  age = c(50, 55, 60, 65),
  grp_id = c(1, 1, 2, 2),
  grp_name = rep(c("Age 50-59", "Age 60-69"), each = 2)
)
states <- data.table(
  state_id =  seq(1, 2),
  state_name = c("State 1", "State 2")
)
hesim_dat <- hesim_data(
  strategies = strategies,
  patients = patients,
  states = states
)
labs <- get_labels(hesim_dat)
labs

# Pass to set_labels()
d <- data.table(strategy_id = c(1, 1, 2, 2),
                grp_id = c(1, 2, 1, 2))
set_labels(d, labs, new_name = c("strategy_name", "grp_name"))
d
```

`hesim_data` *Data for health economic simulation modeling*

Description

A list of tables required for health economic simulation modeling. This object is used to setup models by defining the treatment strategies, target population, and model structure.

Usage

```
hesim_data(strategies, patients, states = NULL, transitions = NULL)
```

Arguments

`strategies`	A table of treatment strategies. Must contain the column `strategy_id` denoting a unique strategy. Other columns are variables describing the characteristics of a treatment strategy.
`patients`	A table of patients. Must contain the column `patient_id` denoting a unique patient. The number of rows should be equal to the number of patients in the model. The table may also include columns for `grp_id` for subgroups and `patient_wt` specifying the weight to apply to each patient (within a subgroup). If `grp_id` is `NULL`, then it is assumed that there is only one subgroup. If `patient_wt` is `NULL`. then each patient is given the same weight. Weights cannot be used in individual-level models because each patient should be weighted equally; that is, weights can only be specified in cohort models. Weights within subgroups are normalized to sum to one. Other columns are variables describing the characteristics of a patient.
`states`	A table of health states. Must contain the column `state_id`, which denotes a unique health state. The number of rows should be equal to the number of health states in the model. Other columns can describe the characteristics of a health state.
`transitions`	A table of health state transitions. Must contain the column `transition_id`, which denotes a unique transition; `from`, which denotes the starting health state; and `to` which denotes the state that will be transitioned to.

Value

Returns an object of class `hesim_data`, which is a list of data tables for health economic simulation modeling.

Note

Each table must either be a `data.frame` or `data.table`. All ID variables within each table must be numeric vectors of integers and should be of the form 1,2,...N where N is the number of unique values of the ID variable.

See Also

`expand.hesim_data()`, `get_labels()`

Examples

```
strategies <- data.frame(strategy_id = c(1, 2))
patients <- data.frame(patient_id = seq(1, 3), age = c(65, 50, 75),
                       gender = c("Female", "Female", "Male"))
states <- data.frame(state_id =  seq(1, 3),
                     state_var = c(2, 1, 9))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients,
                        states = states)
```

---

icer *Incremental cost-effectiveness ratio*

Description

Generate a tidy table of incremental cost-effectiveness ratios (ICERs) given output from `cea_pw()` with `icer()` and format for pretty printing with `format.icer()`.

Usage

```
icer(x, prob = 0.95, k = 50000, labels = NULL, ...)

## S3 method for class 'icer'
format(
  x,
  digits_qalys = 2,
  digits_costs = 0,
  pivot_from = "strategy",
  drop_grp = TRUE,
  pretty_names = TRUE,
  ...
)
```

Arguments

`x`	An object of class `cea_pw` returned by `cea_pw()`.
`prob`	A numeric scalar in the interval `(0,1)` giving the confidence interval. Default is 0.95 for a 95 percent interval.
`k`	Willingness to pay per quality-adjusted life-year.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.
`...`	Further arguments passed to and from methods. Currently unused.
`digits_qalys`	Number of digits to use to report QALYs.
`digits_costs`	Number of digits to use to report costs.
`pivot_from`	Character vector denoting a column or columns used to "widen" the data. Should either be `"strategy"`, `"grp"`, `"outcome"`, or some combination of the three. There will be one column for each value of the variables in `pivot_from`. Default is to widen so there is a column for each treatment strategy. Set to `NULL` if you do not want to widen the table.
`drop_grp`	If `TRUE`, then the group column will be removed if there is only one subgroup; other it will be kept. If `FALSE`, then the `grp` column is never removed.
`pretty_names`	Logical. If `TRUE`, then the columns `strategy`, `grp`, `outcome`, and `value` are renamed (if they exist) to `Strategy`, `Group`, `Outcome`, and `Value`.

Details

Note that `icer()` will report negative ICERs; however, `format()` will correctly note whether a treatment strategy is dominated by or dominates the reference treatment.

Value

`icer()` returns an object of class `icer` that is a tidy `data.table` with the following columns:

strategy	The treatment strategy.
grp	The subgroup.
outcome	The outcome metric.
estimate	The point estimate computed as the average across the PSA samples.
lower	The lower limit of the confidence interval.
upper	The upper limit of the confidence interval.

`format.icer()` formats the table according to the arguments passed.

See Also

`cea_pw()`

`incr_effect` *Incremental treatment effect*

Description

Computes incremental effect for all treatment strategies on outcome variables from a probabilistic sensitivity analysis relative to a comparator.

Usage

```
incr_effect(x, comparator, sample, strategy, grp = NULL, outcomes)
```

Arguments

`x`	A `data.frame` or `data.table` containing simulation output with information on outcome variables for each randomly sampled parameter set from a PSA. Each row should denote a randomly sampled parameter set and treatment strategy.
`comparator`	The comparator strategy. If the strategy column is a character variable, then must be a character; if the strategy column is an integer variable, then must be an integer.
`sample`	Character name of column denoting a randomly sampled parameter set.
`strategy`	Character name of column denoting treatment strategy.
`grp`	Character name of column denoting subgroup. If `NULL`, then it is assumed that there is only one group.
`outcomes`	Name of columns to compute incremental changes for.

Value

A `data.table` containing the differences in the values of each variable specified in outcomes between each treatment strategy and the comparator.

Examples

```
# simulation output
n_samples <- 100
sim <- data.frame(sample = rep(seq(n_samples), 4),
             c = c(rlnorm(n_samples, 5, .1), rlnorm(n_samples, 5, .1),
                    rlnorm(n_samples, 11, .1), rlnorm(n_samples, 11, .1)),
             e = c(rnorm(n_samples, 8, .2), rnorm(n_samples, 8.5, .1),
                   rnorm(n_samples, 11, .6), rnorm(n_samples, 11.5, .6)),
             strategy = rep(paste0("Strategy ", seq(1, 2)),
                            each = n_samples * 2),
             grp = rep(rep(c("Group 1", "Group 2"),
                            each = n_samples), 2)
)
# calculate incremental effect of Strategy 2 relative to Strategy 1 by group
ie <- incr_effect(sim, comparator = "Strategy 1", sample = "sample",
                        strategy = "strategy", grp = "grp", outcomes = c("c", "e"))
head(ie)
```

`IndivCtstm` *Individual-level continuous time state transition model*

Description

Simulate outcomes from an individual-level continuous time state transition model (CTSTM). The class supports "clock-reset" (i.e., semi-Markov), "clock-forward" (i.e., Markov), and mixtures of clock-reset and clock-forward multi-state models as described in `IndivCtstmTrans`.

Format

An R6::R6Class object.

Public fields

`trans_model` The model for health state transitions. Must be an object of class `IndivCtstmTrans`.

`utility_model` The model for health state utility. Must be an object of class `StateVals`.

`cost_models` The models used to predict costs by health state. Must be a list of objects of class `StateVals`, where each element of the list represents a different cost category.

`disprog_` An object of class `disprog`.

`stateprobs_` An object of class `stateprobs` simulated using `$sim_stateprobs()`.

`qalys_` An object of class `qalys` simulated using `$sim_qalys()`.

`costs_` An object of class `costs` simulated using `$sim_costs()`.

Methods

Public methods:

- `IndivCtstm$new()`
- `IndivCtstm$sim_disease()`
- `IndivCtstm$sim_stateprobs()`
- `IndivCtstm$sim_qalys()`
- `IndivCtstm$sim_costs()`
- `IndivCtstm$summarize()`
- `IndivCtstm$clone()`

Method `new()`**:** Create a new `IndivCtstm` object.

Usage:

```
IndivCtstm$new(trans_model = NULL, utility_model = NULL, cost_models = NULL)
```

Arguments:

`trans_model` The `trans_model` field.

`utility_model` The `utility_model` field.

`cost_models` The `cost_models` field.

Returns: A new `IndivCtstm` object.

Method `sim_disease()`**:** Simulate disease progression (i.e., individual trajectories through a multi-state model) using `IndivCtstmTrans$sim_disease()`.

Usage:

```
IndivCtstm$sim_disease(max_t = 100, max_age = 100, progress = NULL)
```

Arguments:

`max_t` A scalar or vector denoting the length of time to simulate the model. If a vector, must be equal to the number of simulated patients.

`max_age` A scalar or vector denoting the maximum age to simulate each patient until. If a vector, must be equal to the number of simulated patients.

`progress` An integer, specifying the PSA iteration (i.e., sample) that should be printed every `progress` PSA iterations. For example, if `progress = 2`, then every second PSA iteration is printed. Default is `NULL`, in which case no output is printed.

Returns: An instance of self with simulated output stored in `disprog_`.

Method `sim_stateprobs()`**:** Simulate health state probabilities as a function of time using the simulation output stored in `disprog`.

Usage:

```
IndivCtstm$sim_stateprobs(t)
```

Arguments:

`t` A numeric vector of times.

Returns: An instance of `self` with simulated output of class `stateprobs` stored in `stateprobs_`.

Method `sim_qalys()`**:** Simulate quality-adjusted life-years (QALYs) as a function of `disprog_` and `utility_model`.

Usage:

```
IndivCtstm$sim_qalys(
  dr = 0.03,
  type = c("predict", "random"),
  lys = TRUE,
  by_patient = FALSE
)
```

Arguments:

`dr` Discount rate.

`type` `"predict"` for mean values or `"random"` for random samples as in `$sim()` in `StateVals`.

`lys` If `TRUE`, then life-years are simulated in addition to QALYs.

`by_patient` If `TRUE`, then QALYs and/or costs are computed at the patient level. If `FALSE`, then they are averaged across patients by health state.

Returns: An instance of `self` with simulated output of class qalys stored in `qalys_`.

Method `sim_costs()`**:** Simulate costs as a function of `disprog_` and `cost_models`.

Usage:

```
IndivCtstm$sim_costs(
  dr = 0.03,
  type = c("predict", "random"),
  by_patient = FALSE,
  max_t = Inf
)
```

Arguments:

`dr` Discount rate.

`type` `"predict"` for mean values or `"random"` for random samples as in `$sim()` in `StateVals`.

`by_patient` If `TRUE`, then QALYs and/or costs are computed at the patient level. If `FALSE`, then they are averaged across patients by health state.

`max_t` Maximum time duration to compute costs once a patient has entered a (new) health state. By default, equal to `Inf`, so that costs are computed over the entire duration that a patient is in a given health state. If time varies by each cost category, then time can also be passed as a numeric vector of length equal to the number of cost categories (e.g., `c(1, 2, Inf, 3)` for a model with four cost categories).

Returns: An instance of `self` with simulated output of class costs stored in `costs_`.

Method `summarize()`**:** Summarize costs and QALYs so that cost-effectiveness analysis can be performed. See `summarize_ce()`.

Usage:

```
IndivCtstm$summarize(by_grp = FALSE)
```

Arguments:

`by_grp` If `TRUE`, then costs and QALYs are computed by subgroup. If `FALSE`, then costs and QALYs are aggregated across all patients (and subgroups).

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`IndivCtstm$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

Incerti and Jansen (2021). See Section 2.2 for a mathematical description of an individual-level CTSTM and Section 4.1 for an example in oncology.

See Also

The `IndivCtstmTrans` documentation describes the class for the transition model and the `StateVals` documentation describes the class for the cost and utility models. An `IndivCtstmTrans` object is typically created using `create_IndivCtstmTrans()`.

There are currently two relevant vignettes. `vignette("mstate")` shows how to parameterize `IndivCtstmTrans` objects in cases where patient-level data is available by fitting a multi-state models. `vignette("markov-inhomogeneous-indiv")` shows how the time inhomogeneous Markov cohort model in `vignette("markov-inhomogeneous-cohort")` can be developed as an individual patient simulation; in doing so, it shows how `IndivCtstm` models can be used even without access to patient-level data.

Examples

```
library("flexsurv")

# Treatment strategies, target population, and model structure
strategies <- data.frame(strategy_id = c(1, 2))
patients <- data.frame(patient_id = seq(1, 3),
                       age = c(45, 50, 60),
                       female = c(0, 0, 1))
states <- data.frame(state_id = c(1, 2))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients,
                        states = states)

# Parameter estimation
## Multi-state model
tmat <- rbind(c(NA, 1, 2),
              c(3, NA, 4),
              c(NA, NA, NA))
fits <- vector(length = max(tmat, na.rm = TRUE), mode = "list")
surv_dat <- data.frame(mstate3_exdata$transitions)
for (i in 1:length(fits)){
  fits[[i]] <- flexsurvreg(Surv(years, status) ~ factor(strategy_id),
                           data = surv_dat,
                           subset = (trans == i),
                           dist = "weibull")
}
fits <- flexsurvreg_list(fits)
```

```
## Utility
utility_tbl <- stateval_tbl(data.frame(state_id = states$state_id,
                                       mean = mstate3_exdata$utility$mean,
                                       se = mstate3_exdata$utility$se),
                            dist = "beta")
## Costs
drugcost_tbl <- stateval_tbl(data.frame(strategy_id = strategies$strategy_id,
                                        est = mstate3_exdata$costs$drugs$costs),
                             dist = "fixed")
medcost_tbl <- stateval_tbl(data.frame(state_id = states$state_id,
                                       mean = mstate3_exdata$costs$medical$mean,
                                       se = mstate3_exdata$costs$medical$se),
                            dist = "gamma")

# Economic model
n_samples = 2

## Construct model
### Transitions
transmod_data <- expand(hesim_dat)
transmod <- create_IndivCtstmTrans(fits, input_data = transmod_data,
                                   trans_mat = tmat,
                                   n = n_samples)

### Utility
utilitymod <- create_StateVals(utility_tbl, n = n_samples, hesim_data = hesim_dat)

### Costs
drugcostmod <- create_StateVals(drugcost_tbl, n = n_samples, hesim_data = hesim_dat)
medcostmod <- create_StateVals(medcost_tbl, n = n_samples, hesim_data = hesim_dat)
costmods <- list(drugs = drugcostmod,
                 medical = medcostmod)

### Combine
ictstm <- IndivCtstm$new(trans_model = transmod,
                         utility_model = utilitymod,
                         cost_models = costmods)


## Simulate outcomes
head(ictstm$sim_disease()$disprog_)
head(ictstm$sim_stateprobs(t = c(0, 5, 10))$stateprobs_[t == 5])
ictstm$sim_qalys(dr = .03)
ictstm$sim_costs(dr = .03)

### Summarize cost-effectiveness
ce <- ictstm$summarize()
head(ce)
format(summary(ce), pivot_from = "strategy")
```

`IndivCtstmTrans` *Transitions for an individual-level continuous time state transition model*

Description

Simulate health state transitions in an individual-level continuous time state transition model using parameters from a multi-state model.

Format

An R6::R6Class object.

Super class

`hesim::CtstmTrans` -> `IndivCtstmTrans`

Public fields

`params` An object of class `params_surv` or `params_surv_list`.

`input_data` Input data used to simulate health state transitions by sample from the probabilistic sensitivity analysis (PSA), treatment strategy and patient. Must be an object of class `input_mats`. If `params` contains parameters from a list of models (i.e., of class `params_surv_list`), then `input_data` must contain a unique row for each treatment strategy and patient; if `params` contains parameters from a joint model (i.e., of class `params_surv`), then `input_data` must contain a unique row for each treatment strategy, patient, and transition.

`trans_mat` A transition matrix describing the states and transitions in a multi-state model in the format from the `mstate` package. See the documentation for the argument `"trans"` in `mstate::msprep`.

`start_state` A scalar or vector denoting the starting health state. Default is the first health state. If a vector, must be equal to the number of simulated patients.

`start_age` A scalar or vector denoting the starting age of each patient in the simulation. Default is 38. If a vector, must be equal to the number of simulated patients.

`death_state` The death state in `trans_mat`. Used with `max_age` in `sim_disease` as patients transition to this state upon reaching maximum age. By default, it is set to the final absorbing state (i.e., a row in `trans_mat` with all NAs).

`clock` "reset" for a clock-reset model, "forward" for a clock-forward model, and "mix" for a mixture of clock-reset and clock-forward models. A clock-reset model is a semi-Markov model in which transition rates depend on time since entering a state. A clock-forward model is a Markov model in which transition rates depend on time since entering the initial state. If `"mix"` is used, then `reset_states` must be specified.

`reset_states` A vector denoting the states in which time resets. Hazard functions are always a function of elapsed time since either the start of the model or from when time was previously reset. Only used if `clock = "mix"`.

Methods

Public methods:

- `IndivCtstmTrans$new()`
- `IndivCtstmTrans$sim_disease()`
- `IndivCtstmTrans$sim_stateprobs()`
- `IndivCtstmTrans$check()`
- `IndivCtstmTrans$clone()`

Method `new()`**:** Create a new `IndivCtstmTrans` object.

Usage:

```
IndivCtstmTrans$new(
  params,
  input_data,
  trans_mat,
  start_state = 1,
  start_age = 38,
  death_state = NULL,
  clock = c("reset", "forward", "mix"),
  reset_states = NULL
)
```

Arguments:

`params` The `params` field.

`input_data` The `input_data` field.

`trans_mat` The `trans_mat` field.

`start_state` The `start_state` field.

`start_age` The `start_age` field.

`death_state` The `death_state` field.

`clock` The `clock` field.

`reset_states` The `reset_states` field.

Returns: A new `IndivCtstmTrans` object.

Method `sim_disease()`**:** Simulate disease progression (i.e., individual trajectories through a multi-state model using an individual patient simulation).

Usage:

```
IndivCtstmTrans$sim_disease(max_t = 100, max_age = 100, progress = NULL)
```

Arguments:

`max_t` A scalar or vector denoting the length of time to simulate the model. If a vector, must be equal to the number of simulated patients.

`max_age` A scalar or vector denoting the maximum age to simulate each patient until. If a vector, must be equal to the number of simulated patients.

`progress` An integer, specifying the PSA iteration (i.e., sample) that should be printed every progress PSA iterations. For example, if progress = 2, then every second PSA iteration is printed. Default is NULL, in which case no output is printed.

Returns: An object of class `disprog`.

Method `sim_stateprobs()`**:** Simulate health state probabilities from a disprog object.

Usage:

`IndivCtstmTrans$sim_stateprobs(t, disprog = NULL, ...)`

Arguments:

`t` A numeric vector of times.

`disprog` A disprog object. If `NULL`, then this will be simulated prior to computing state probabilities using `IndivCtstm$sim_disease()`.

`...` Additional arguments to pass to `IndivCtstm$sim_disease()` if `disprog = NULL`.

Returns: An object of class `stateprobs`.

Method `check()`**:** Input validation for class. Checks that fields are the correct type.

Usage:

`IndivCtstmTrans$check()`

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`IndivCtstmTrans$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

`IndivCtstmTrans` objects are conveniently created from either fitted models or parameter objects with `create_IndivCtstmTrans()`. A complete economic model can be implemented with the `IndivCtstm` class.

Examples

```
library("flexsurv")

# Simulation data
strategies <- data.frame(strategy_id = c(1, 2, 3))
patients <- data.frame(patient_id = seq(1, 3),
                       age = c(45, 50, 60),
                       female = c(0, 0, 1))

# Multi-state model with transition specific models
tmat <- rbind(c(NA, 1, 2),
              c(NA, NA, 3),
              c(NA, NA, NA))
fits <- vector(length = max(tmat, na.rm = TRUE), mode = "list")
for (i in 1:length(fits)){
  fits[[i]] <- flexsurvreg(Surv(years, status) ~ 1,
                           data = bosms3[bosms3$trans == i, ],
                           dist = "exp")
}
```

```
fits <- flexsurvreg_list(fits)

# Simulation model
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients)
fits_data <- expand(hesim_dat)
transmod <- create_IndivCtstmTrans(fits, input_data = fits_data,
                                   trans_mat = tmat,
                                   n = 2)
head(transmod$hazard(c(1, 2, 3)))
head(transmod$cumhazard(c(1, 2, 3)))

## Simulate disease progression and state probabilities together
transmod$sim_stateprobs(t = c(0, 5, 10))[t == 5]

## Simulate disease progression and state probabilities separately
disprog <- transmod$sim_disease(max_t = 10)
transmod$sim_stateprobs(t = c(0, 5, 10), disprog = disprog)[t == 5]
```

`input_mats` *Input matrices for a statistical model*

Description

An object of class `input_mats` contains input matrices for simulating a statistical model. Consists of (i) input matrices, `X`, and (ii) metadata used to index each matrix in `X`.

Once created, an `input_mats` object can be converted to a `data.table` with `as.data.table()`, which is a helpful way to check that the object is as expected. The `print()` method summarizes the object and prints it to the console.

More details are provided under "Details" below.

Usage

```
input_mats(X, ...)

## S3 method for class 'input_mats'
as.data.table(x, ...)

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

Arguments

`X`	A list of input matrices for predicting the values of each parameter in a statistical model. May also be a list of lists of input matrices when a list of separate models is fit (e.g., with `flexsurvreg_list()`).
`...`	For `input_mats()`, arguments to pass to `id_attributes()`. For `print()`, arguments to pass to `print.data.table()`.
`x`	An `input_mats` object.

Details

`input_mats` objects are used with `params` objects to simulate disease models, cost models, and/or utility models. Each column of `$X` contains variables from the `params` object and a given row corresponds to a combination of the ID variables. An input matrix must always have rows for the treatment strategies (`strategy_id`) and patients (`patient_id`); it may optionally have rows for health variables (`state_id` or `transition_id`) and time intervals (`time_id`). The rows must be sorted by prioritizing (i) `strategy_id`, (ii) `patient_id`, (iii) the health related ID variable (either `state_id` or `transition_id`) and (iv) `time_id`.

While `input_mats` objects can be created directly with `input_mats()`, it is rarely a good idea to do so. They are typically created as the `input_data` field when creating model objects (e.g., with `create_IndivCtstmTrans()`, `create_CohortDtstmTrans()`, and `create_PsmCurves()`). Internally, these functions create the input matrices using `create_input_mats()` methods, which ensure that they are in the correct format. Users may also use `create_input_mats()` methods, but there is not usually a good reason to do so.

`as.data.table.input_mats()` will convert input matrices into a single `data.table()` that column binds the ID variables and the unique combinations of variables contained in the elements of `$X`. `print.input_mats()` prints a call to `as.data.table()` and provides additional information about the ID variables.

See Also

See `IndivCtstmTrans()` and `PsmCurves()` for examples in which the `input_data` field of an instance of a model class is an `input_mats` object.

Examples

```
library("data.table")

# Input matrices are typically created as part of model objects
# Let's illustrate with a partitioned survival model (PSM)

## Model setup
strategies <- data.frame(strategy_id = c(1, 2),
                         new_strategy = c(0, 1))
patients <- data.frame(patient_id = seq(1, 3),
                       age = c(45, 47, 60),
                       female = c(1, 0, 0),
                       group = factor(c("Good", "Medium", "Poor")))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients)

## Create survival models for PSM
### Parameters
n <- 2
survmod_params <- params_surv_list(
  # Progression free survival (PFS)
  pfs = params_surv(
    coefs = list(
      rate = data.frame(intercept = rnorm(n, log(1/5), 1),
                        new_strategy = rnorm(n, log(.8), 1))
```

```
      ),
      dist = "exp"
    ),

    # Overall survival (OS)
    os = params_surv(
      coefs = list(
        rate = data.frame(intercept = rnorm(n, log(1/10), 1))
      ),
      dist = "exp"
    )
  )

### Input data
survmod_input_data <- expand(hesim_dat)[, intercept := 1]

### Model object
survmod <- create_PsmCurves(survmod_params, input_data = survmod_input_data)

## Inspect input data
survmod$input_data # Print "input_mats" object to console
as.data.table(survmod$input_data) # Convert "input_mats" object to data.table
```

`mom_beta` *Method of moments for beta distribution*

Description

Compute the parameters `shape1` and `shape2` of the beta distribution using method of moments given the mean and standard deviation of the random variable of interest.

Usage

```
mom_beta(mean, sd)
```

Arguments

`mean`	Mean of the random variable.
`sd`	Standard deviation of the random variable.

Details

If μ is the mean and σ is the standard deviation of the random variable, then the method of moments estimates of the parameters `shape1` = $\alpha > 0$ and `shape2` = $\beta > 0$ are:

$$\alpha = \mu \left(\frac{\mu(1-\mu)}{\sigma^2} - 1 \right)$$

and

$$\beta = (1-\mu) \left(\frac{\mu(1-\mu)}{\sigma^2} - 1 \right)$$

Value

A list containing the parameters `shape1` and `shape2`.

Examples

```
mom_beta(mean = .8, sd = .1)
# The function is vectorized.
mom_beta(mean = c(.6, .8), sd = c(.08, .1))
```

`mom_gamma` *Method of moments for gamma distribution*

Description

Compute the shape and scale (or rate) parameters of the gamma distribution using method of moments for the random variable of interest.

Usage

```
mom_gamma(mean, sd, scale = TRUE)
```

Arguments

`mean`	Mean of the random variable.
`sd`	Standard deviation of the random variable.
`scale`	Logical. If TRUE (default), then the scale parameter is returned; otherwise, the rate parameter is returned.

Details

If μ is the mean and σ is the standard deviation of the random variable, then the method of moments estimates of the parameters `shape` = $\alpha > 0$ and `scale` = $\theta > 0$ are:

$$\theta = \frac{\sigma^2}{\mu}$$

and

$$\alpha = \frac{\mu}{\theta}$$

The inverse of the scale parameter, $\beta = 1/\theta$, is the rate parameter.

Value

If `scale = TRUE`, then a list containing the parameters `shape` and `scale`; otherwise, if `scale = FALSE`, then a list containing the parameters `shape` and `rate`.

Examples

```
mom_gamma(mean = 10000, sd = 2000)
# The function is vectorized.
mom_gamma(mean = c(8000, 10000), sd = c(1500, 2000))
```

`mstate3_exdata` *Example data for a reversible 3-state multi-state model*

Description

Example multi-state data for parameterizing a continuous time state transition model. Costs and utility are also included to facilitate cost-effectiveness analysis.

Usage

```
mstate3_exdata
```

Format

A list containing the following elements:

- transitions A data frame containing the times at which patient transitions between health states based on the prothr dataset from the mstate package.
- costs A list of data frames. The first data frame contains summary medical cost estimates and the second data frame contains drug cost data.
- utility A data frame of summary utility estimates.

Transitions data

The data frame has the following columns:

strategy_id Treatment strategy identification number.

patient_id Patient identification number.

age Patient age (in years).

female 1 if a patient is female; 0 if male.

from Starting state.

to Receiving state.

trans Transition number.

Tstart Starting time.

Tstop Transition time.

years Elapsed years between `Tstart` and `Tstop`.

status Status variable; 1=transition, 0=censored.

Cost data

The cost list contains two data frames. The first data frame contains data on the drug costs associated with each treatment strategy.

strategy_id The treatment strategy identification number.

costs Annualized drug costs.

The second data frame contains summary data on medical costs by health state, and contains the following columns:

state_id The health state identification number.

mean Mean costs.

se Standard error of medical costs.

Utility data

The data frame has the following columns:

state_id The health state identification number.

mean Mean utility

se Standard error of utility

`multinom3_exdata` *Example data for a 3-state multinomial model*

Description

Example discrete time health state transitions data simulated using multinomial logistic regression. Costs and utility are also included to facilitate cost-effectiveness analysis.

Usage

```
multinom3_exdata
```

Format

A list containing the following elements:

- transitions A data frame containing patient transitions between health states at discrete time intervals (i.e., on a yearly basis).
- costs A list of data frames. The first data frame contains drug cost data and the second contains summary medical cost estimates.
- utility A data frame of summary utility estimates.

Transitions data

The data frame has the following columns:

patient_id	Patient identification number.
strategy_id	Treatment strategy identification number.
strategy_name	Treatment strategy name.
age	Patient age (in years).
age_cat	A factor variable with 3 age groups: (i) age less than 40, (ii) age at least 40 and less than 60, and (iii) age at least 60.
female	1 if a patient is female; 0 if male.
year	The year since the start of data collection with the first year equal to 1.
state_from	State making a transition from.
state_to	State making a transition to.
year_cat	Factor variable for year with 3 categories: (i) year 3 and below, (ii) year between 3 and 6, and (iii) year 7 and above.

Cost data

The cost list contains two data frames. The first data frame contains data on the drug costs associated with each treatment strategy.

strategy_id	The treatment strategy identification number.
strategy_name	The treatment strategy name.
costs	Annualized drug costs.

The second data frame contains summary data on medical costs by health state, and contains the following columns:

state_id	The health state identification number.
state_name	The name of the health state.
mean	Mean medical costs.
se	Standard error of medical costs.

Utility data

The data frame has the following columns:

state_id	The health state identification number.
state_name	The name of the health state.
mean	Mean utility
se	Standard error of utility.

`multinom_list` *List of* `multinom` *objects*

Description

Combine `multinom` objects into a list.

Usage

```
multinom_list(...)
```

Arguments

`...`	Objects of class `multinom`, which can be named.

Value

An object of class `multinom_list`.

Examples

```
library("nnet")
library("data.table")
trans_data <- data.table(multinom3_exdata$transitions)
dat_healthy <- trans_data[state_from == "Healthy"]
fit_healthy <- multinom(state_to ~ strategy_name + female + age_cat + year_cat,
                        data = dat_healthy)
dat_sick <- trans_data[state_from == "Sick"]
dat_sick$state_to <- droplevels(dat_sick$state_to)
fit_sick <- multinom(state_to ~ strategy_name + female + age_cat + year_cat,
                     data = dat_sick)
fits <- multinom_list(healthy = fit_healthy, sick = fit_sick)
class(fits)
```

`onc3` *Multi-state oncology data for 3-state model*

Description

Simulated 3-state dataset in oncology with three health states (Stable, Progression, and Death) and three possible transitions (Stable -> Progression, Stable -> Death, and Progression -> Death).

Usage

```
onc3
```

Format

A `data.table` with the following columns:

from Health state making a transition from.

to Health state making a transition to.

strategy_name Standard of care (SOC), new treatment 1 (New 1), or new treatment 2 (New 2).

female 1 if a patient is female; 0 if male.

age Patient age (in years).

patient_id Patient identification number.

time_start Starting time.

time_stop Stopping time.

status Status indicator: 1=transition, 0=censored.

transition_id Integer denoting transition: 1 = Stable -> Progression, 2 = Stable -> Death, 3 = Progression -> Death.

strategy_id Strategy identification number.

time Elapsed years between `time_start` and `time_stop`.

See Also

onc3p

Examples

```
head(onc3)
```

`onc3p` *Multi-state panel oncology data for 3-state model*

Description

The same dataset as onc3 converted into a panel data format in which health states are recorded at a finite series of times.

Usage

```
onc3p
```

Format

A `data.table` with the following columns:

state The name of the health state (Stable, Progression, and Death).

strategy_name Standard of care (SOC), new treatment 1 (New 1), or new treatment 2 (New 2).

female 1 if a patient is female; 0 if male.

age Patient age (in years).

patient_id Patient identification number.

time Time that `state` was recorded.

strategy_id Strategy identification number.

state_id The health state identification number.

See Also

onc3

Examples

```
head(onc3p)
```

`params` *Parameter object*

Description

Objects prefixed by "params_" are lists containing the parameters of a statistical model used for simulation modeling. The parameters are used to simulate outcomes as a function of covariates contained in input matrices (input_mats).

See Also

`tparams`

---

params_lm *Parameters of a linear model*

Description

Create a list containing the parameters of a fitted linear regression model.

Usage

```
params_lm(coefs, sigma = 1)
```

Arguments

`coefs`	Samples of the coefficients under sampling uncertainty. Must be a matrix or any object coercible to a matrix such as `data.frame` or `data.table`.
`sigma`	A vector of samples of the standard error of the regression model. Default value is 1 for all samples. Only used if the model is used to randomly simulate values (rather than to predict means).

Details

Fitted linear models are used to predict values, y, as a function of covariates, x,

$$y = x^T \beta + \epsilon.$$

Predicted means are given by $x^T \hat{\beta}$ where $\hat{\beta}$ is the vector of estimated regression coefficients. Random samples are obtained by sampling the error term from a normal distribution, $\epsilon \sim N(0, \hat{\sigma}^2)$.

Value

An object of class `params_lm`, which is a list containing `coefs`, `sigma`, and `n_samples`. `n_samples` is equal to the number of rows in `coefs`. The `coefs` element is always converted into a matrix.

See Also

This parameter object is useful for modeling health state values when values can vary across patients and/or health states as a function of covariates. In many cases it will, however, be simpler, and more flexible to use a `stateval_tbl`. For an example use case see the documentation for `create_StateVals.lm()`.

Examples

```
library("MASS")
n <- 2
params <- params_lm(
  coefs = mvrnorm(n, mu = c(.5,.6),
                  Sigma = matrix(c(.05, .01, .01, .05), nrow = 2)),
  sigma <- rgamma(n, shape = .5, rate = 4)
```

```
)
summary(params)
params
```

`params_mlogit` *Parameters of a multinomial logit model*

Description

Store the parameters of a fitted multinomial logistic regression model. The model is used to predict probabilities of K classes, which represent the probability of transitioning to particular health state in a discrete time state transition model. Can be used as an element of a `params_mlogit_list` to parameterize a `CohortDtstmTrans` object.

Usage

```
params_mlogit(coefs)
```

Arguments

`coefs`	A 3D array of stacked matrices containing samples of the regression coefficients under sampling uncertainty. May also be a list of objects (e.g., data frames) that can be coerced into matrices with `as.matrix()`. Each matrix must have the same number of columns and the number of matrices must be equal to $K - 1$.

Details

Multinomial logit models are used to predict the probability of membership for subject i in each of K classes as a function of covariates:

$$Pr(y_i = c) = \frac{e^{\beta_c x_i}}{\sum_{k=1}^{K} e^{\beta_k x_i}}$$

Value

An object of class `params_mlogit`, which is a list containing `coefs` and `n_samples`, where `n_samples` is equal to the number of rows in each element of `coefs`. The `coefs` element is always converted into a 3D array of stacked matrices.

See Also

`summary.params_mlogit()`, `params_mlogit_list()`, `CohortDtstmTrans`

Examples

```
# Consider a sick-sicker model and model transitions from the sick state

## We can instantiate from a list of data frames
params <- params_mlogit(
  coefs = list(
    ### Transition from sick to sicker
    sicker = data.frame(
      intercept = c(-0.33, -.2, -.15),
      treat = c(log(.75), log(.8), log(.9))
    ),

   ### Transition from sick to death
    death = data.frame(
      intercept = c(-1, -1.2, -.5),
      treat = c(log(.6), log(.65), log(.55))
    )
  )
)
summary(params)
params

## We can also instantiate from an array
coefs_sicker <- data.frame(
  intercept = c(-0.33, -.2, -.15),
  treat = c(log(.75), log(.8), log(.9))
)
coefs_death <- data.frame(
  intercept = c(-1, -1.2, -.5),
  treat = c(log(.6), log(.65), log(.55))
)

params2 <- params_mlogit(
  coefs <- array(
    data = c(as.matrix(coefs_sicker),
             as.matrix(coefs_death)),
    dim = c(3, 2, 2),
    dimnames = list(NULL, c("intercept", "treat"), c("sicker", "death"))
  )
)
params2
```

`params_mlogit_list` *Parameters of a list of multinomial logit models*

Description

Create a list containing the parameters of multiple fitted multinomial logit models. Can be used to parameterize state transitions in a discrete time transition model by passing to the `params` field of a `CohortDtstmTrans` object.

Usage

```
params_mlogit_list(...)
```

Arguments

`...`	Objects of class `params_mlogit`, which can be named.

Value

An object of class `params_mlogit_list`, which is a list containing `params_mlogit` objects.

See Also

`summary.params_mlogit_list()`, `params_mlogit()`, `CohortDtstmTrans`

Examples

```
# Consider a sick-sicker model

params <- params_mlogit_list(
  ## Transitions from sick state (sick -> sicker, sick -> death)
  sick = params_mlogit(
    coefs = list(
      sicker = data.frame(
        intercept = c(-0.33, -.2),
        treat = c(log(.75), log(.8))
      ),
      death = data.frame(
        intercept = c(-1, -1.2),
        treat = c(log(.6), log(.65))
      )
    )
  ),

  ## Transitions from sicker state (sicker -> death)
  sicker = params_mlogit(
    coefs = list(
      death = data.frame(
        intercept = c(-1.5, -1.4),
        treat = c(log(.5), log(.55))
      )
    )
  )
)
summary(params)
params
```

`params_surv` *Parameters of a survival model*

Description

Create a list containing the parameters of a single fitted parametric or flexible parametric survival model.

Usage

```
params_surv(coefs, dist, aux = NULL)
```

Arguments

`coefs`	A list of length equal to the number of parameters in the survival distribution. Each element of the list is a matrix of samples of the regression coefficients under sampling uncertainty used to predict a given parameter. All parameters are expressed on the real line (e.g., after log transformation if they are defined as positive). Each element of the list may also be an object coercible to a matrix such as a `data.frame` or `data.table`.
`dist`	Character vector denoting the parametric distribution. See "Details".
`aux`	Auxiliary arguments used with splines, fractional polynomial, or piecewise exponential models. See "Details".

Details

Survival is modeled as a function of L parameters α_l. Letting $F(t)$ be the cumulative distribution function, survival at time t is given by

$$1 - F(t|\alpha_1(x_1), \ldots, \alpha_L(x_L)).$$

The parameters are modeled as a function of covariates, x_l, with an inverse transformation function $g^{-1}()$,

$$\alpha_l = g^{-1}(x_l^T \beta_l).$$

$g^{-1}()$ is typically $exp()$ if a parameter is strictly positive and the identity function if the parameter space is unrestricted.

The types of distributions that can be specified are:

- `exponential` or `exp` Exponential distribution. `coef` must contain the `rate` parameter on the log scale and the same parameterization as in `stats::Exponential`.
- `weibull` or `weibull.quiet` Weibull distribution. The first element of `coef` is the `shape` parameter (on the log scale) and the second element is the `scale` parameter (also on the log scale). The parameterization is that same as in `stats::Weibull`.
- `weibullPH` Weibull distribution with a proportional hazards parameterization. The first element of `coef` is the `shape` parameter (on the log scale) and the second element is the `scale` parameter (also on the log scale). The parameterization is that same as in `flexsurv::WeibullPH`.

- `gamma` Gamma distribution. The first element of `coef` is the `shape` parameter (on the log scale) and the second element is the `rate` parameter (also on the log scale). The parameterization is that same as in `stats::GammaDist`.
- `lnorm` Lognormal distribution. The first element of `coef` is the `meanlog` parameter (i.e., the mean of survival on the log scale) and the second element is the `sdlog` parameter (i.e., the standard deviation of survival on the log scale). The parameterization is that same as in `stats::Lognormal`. The coefficients predicting the `meanlog` parameter are untransformed whereas the coefficients predicting the `sdlog` parameter are defined on the log scale.
- `gompertz` Gompertz distribution. The first element of `coef` is the `shape` parameter and the second element is the `rate` parameter (on the log scale). The parameterization is that same as in `flexsurv::Gompertz`.
- `llogis` Log-logistic distribution. The first element of `coef` is the `shape` parameter (on the log scale) and the second element is the `scale` parameter (also on the log scale). The parameterization is that same as in `flexsurv::Llogis`.
- `gengamma` Generalized gamma distribution. The first element of `coef` is the location parameter `mu`, the second element is the scale parameter `sigma` (on the log scale), and the third element is the shape parameter `Q`. The parameterization is that same as in `flexsurv::GenGamma`.
- `survspline` Survival splines. Each element of `coef` is a parameter of the spline model (i.e. `gamma_0`, `gamma_1`, `...`) with length equal to the number of knots (including the boundary knots). See below for details on the auxiliary arguments. The parameterization is that same as in `flexsurv::Survspline`.
- `fracpoly` Fractional polynomials. Each element of `coef` is a parameter of the fractional polynomial model (i.e. `gamma_0`, `gamma_1`, `...`) with length equal to the number of powers plus 1. See below for details on the auxiliary arguments (i.e., `powers`).
- `pwexp` Piecewise exponential distribution. Each element of `coef` is rate parameter for a distinct time interval. The times at which the rates change should be specified with the auxiliary argument `time` (see below for more details).
- `fixed` A fixed survival time. Can be used for "non-random" number generation. `coef` should contain a single parameter, `est`, of the fixed survival times.

Auxiliary arguments for spline models should be specified as a list containing the elements:

`knots` A numeric vector of knots.

`scale` The survival outcome to be modeled as a spline function. Options are `"log_cumhazard"` for the log cumulative hazard; `"log_hazard"` for the log hazard rate; `"log_cumodds"` for the log cumulative odds; and `"inv_normal"` for the inverse normal distribution function.

`timescale` If `"log"` (the default), then survival is modeled as a spline function of log time; if `"identity"`, then it is modeled as a spline function of time.

Auxiliary arguments for fractional polynomial models should be specified as a list containing the elements:

`powers` A vector of the powers of the fractional polynomial with each element chosen from the following set: -2. -1, -0.5, 0, 0.5, 1, 2, 3.

Auxiliary arguments for piecewise exponential models should be specified as a list containing the element:

`time` A vector equal to the number of rate parameters giving the times at which the rate changes.

Furthermore, when splines (with `scale = "log_hazard"`) or fractional polynomials are used, numerical methods must be used to compute the cumulative hazard and for random number generation. The following additional auxiliary arguments can therefore be specified:

`cumhaz_method` Numerical method used to compute cumulative hazard (i.e., to integrate the hazard function). Always used for fractional polynomials but only used for splines if `scale = "log_hazard"`. Options are `"quad"` for adaptive quadrature and `"riemann"` for Riemann sum.

`random_method` Method used to randomly draw from an arbitrary survival function. Options are `"invcdf"` for the inverse CDF and `"discrete"` for a discrete time approximation that randomly samples events from a Bernoulli distribution at discrete times.

`step` Step size for computation of cumulative hazard with numerical integration. Only required when using `"riemann"` to compute the cumulative hazard or using `"discrete"` for random number generation.

Value

An object of class `params_surv`, which is a list containing `coefs`, `dist`, and `n_samples`. `n_samples` is equal to the number of rows in each element of `coefs`, which must be the same. The `coefs` element is always converted into a list of matrices. The list may also contain aux if a spline, fractional polynomial, or piecewise exponential model is used.

Examples

```
n <- 10
params <- params_surv(
  coefs = list(
    shape = data.frame(
      intercept = rnorm(n, .5, .23)
    ),
    scale = data.frame(
      intercept = rnorm(n, 12.39, 1.49),
      age = rnorm(n, -.09, .023)
   )
  ),
  dist = "weibull"
)
summary(params)
params
```

`params_surv_list` *Parameters of a list of survival models*

Description

Create a list containing the parameters of multiple fitted parametric survival models.

Usage

```
params_surv_list(...)
```

Arguments

`...`	Objects of class `params_surv`, which can be named.

Value

An object of class `params_surv_list`, which is a list containing `params_surv` objects.

See Also

`create_params()`

Examples

```
n <- 5
params <- params_surv_list(
  # Model for progression free survival
  pfs = params_surv(
    coefs = list(
      rate = data.frame(intercept = rnorm(n, log(.5), .5),
                        new_strategy = rnorm(n, log(.8), .1))
   ),
    dist = "exp"
  ),

  # Model for overall survival
  os = params_surv(
    coefs = list(
      rate = data.frame(intercept = rnorm(n, log(.3) , .5))
    ),
    dist = "exp"
  )
)
summary(params)
params
```

`plot_ceac` *Plot cost-effectiveness acceptability curve*

Description

Plot a cost-effectiveness curve from either the output of `cea()` or `cea_pw()` using `ggplot2`. The former compares all treatment strategies simultaneously and uses the probabilistic sensitivity analysis (PSA) to compute the probability that each strategy is the most cost-effective at a given willingness to pay value, while the latter uses the PSA to compute the probability that each treatment is cost-effective relative to a comparator.

Usage

```
plot_ceac(x, ...)

## S3 method for class 'cea_pw'
plot_ceac(x, labels = NULL, ...)

## S3 method for class 'cea'
plot_ceac(x, labels = NULL, ...)
```

Arguments

`x`	An object of the appropriate class.
`...`	Further arguments passed to and from methods. Currently unused.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.

Details

See the `cea()` documentation for an example. If there are multiple subgroups, then a faceted plot is produced with one plot for each subgroup.

`plot_ceaf` *Plot cost-effectiveness acceptability frontier*

Description

Plot a cost-effectiveness acceptability frontier (CEAF) from the output of `cea` using `ggplot2`. The CEAF plots the probability that the optimal treatment strategy (i.e., the strategy with the highest expected net monetary benefit) is cost-effective.

Usage

```
plot_ceaf(x, labels = NULL)
```

Arguments

`x`	A cea object produced by `cea`.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.

Details

See the `cea()` documentation for an example. If there are multiple subgroups, then a faceted plot is produced with one plot for each subgroup.

Value

A `ggplot` object.

`plot_ceplane` *Plot cost-effectiveness plane*

Description

Plot a cost-effectiveness plane from the output of `cea_pw()` using `ggplot2`. Each point is a random draw of incremental costs (y-axis) and incremental QALYs (x-axis) from a probabilistic sensitivity analysis.

Usage

```
plot_ceplane(x, k = 50000, labels = NULL)
```

Arguments

`x`	A `cea_pw` object produced by `cea_pw()`.
`k`	Willingness to pay per QALY.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.

Details

See the `cea_pw()` documentation for an example. If there are multiple subgroups, then a faceted plot is produced with one plot for each subgroup.

Value

A `ggplot` object.

`plot_evpi` *Plot expected value of perfect information*

Description

Plot the expected value of perfect information (EVPI) from the output of `cea()` using `ggplot2`. Intuitively, the EVPI provides an estimate of the amount that a decision maker would be willing to pay to collect additional data and completely eliminate uncertainty.

Usage

```
plot_evpi(x, labels = NULL)
```

Arguments

`x`	A `cea` object produced by `cea()`.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.

Details

See the `cea()` documentation for an example. If there are multiple subgroups, then a faceted plot is produced with one plot for each subgroup.

Value

A `ggplot` object.

`Psm` *N-state partitioned survival model*

Description

Simulate outcomes from an N-state partitioned survival model.

Format

An R6::R6Class object.

Public fields

`survival_models` The survival models used to predict survival curves. Must be an object of class `PsmCurves`.

`utility_model` The model for health state utility. Must be an object of class `StateVals`.

`cost_models` The models used to predict costs by health state. Must be a list of objects of class `StateVals`, where each element of the list represents a different cost category.

`n_states` Number of states in the partitioned survival model.

`t_` A numeric vector of times at which survival curves were predicted. Determined by the argument `t` in `$sim_curves()`.

`survival_` An object of class survival simulated using `sim_survival()`.

`stateprobs_` An object of class stateprobs simulated using `$sim_stateprobs()`.

`qalys_` An object of class qalys simulated using `$sim_qalys()`.

`costs_` An object of class costs simulated using `$sim_costs()`.

Methods

Public methods:

- `Psm$new()`
- `Psm$sim_survival()`
- `Psm$sim_stateprobs()`
- `Psm$sim_qalys()`
- `Psm$sim_costs()`
- `Psm$summarize()`
- `Psm$clone()`

Method `new()`**:** Create a new `Psm` object.

Usage:

```
Psm$new(survival_models = NULL, utility_model = NULL, cost_models = NULL)
```

Arguments:

`survival_models` The `survival_models` field.

`utility_model` The `utility_model` field.

`cost_models` The `cost_models` field.

Details: `n_states` is set equal to the number of survival models plus one.

Returns: A new `Psm` object.

Method `sim_survival()`**:** Simulate survival curves as a function of time using `PsmCurves$survival()`.

Usage:

```
Psm$sim_survival(t)
```

Arguments:

`t` A numeric vector of times. The first element must be `0`.

Returns: An instance of `self` with simulated output from `PsmCurves$survival()` stored in `survival_`.

Method `sim_stateprobs()`**:** Simulate health state probabilities from `survival_` using a partitioned survival analysis.

Usage:

```
Psm$sim_stateprobs()
```

Returns: An instance of `self` with simulated output of class stateprobs stored in `stateprobs_`.

Method `sim_qalys()`**:** Simulate quality-adjusted life-years (QALYs) as a function of `stateprobs_` and `utility_model`. See `sim_qalys()` for details.

Usage:

```
Psm$sim_qalys(
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right"),
  lys = TRUE
)
```

Arguments:

`dr` Discount rate.

`integrate_method` Method used to integrate state values when computing costs or QALYs. Options are `trapz` for the trapezoid rule, `riemann_left` for a left Riemann sum, and `riemann_right` for a right Riemann sum.

`lys` If `TRUE`, then life-years are simulated in addition to QALYs.

Returns: An instance of `self` with simulated output of class qalys stored in `qalys_`.

Method `sim_costs()`**:** Simulate costs as a function of `stateprobs_` and `cost_models`. See `sim_costs()` for details.

Usage:

```
Psm$sim_costs(
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right")
)
```

Arguments:

`dr` Discount rate.

`integrate_method` Method used to integrate state values when computing costs or QALYs. Options are `trapz` for the trapezoid rule, `riemann_left` for a left Riemann sum, and `riemann_right` for a right Riemann sum.

Returns: An instance of `self` with simulated output of class costs stored in `costs_`.

Method `summarize()`**:** Summarize costs and QALYs so that cost-effectiveness analysis can be performed. See `summarize_ce()`.

Usage:

```
Psm$summarize(by_grp = FALSE)
```

Arguments:

`by_grp` If `TRUE`, then costs and QALYs are computed by subgroup. If `FALSE`, then costs and QALYs are aggregated across all patients (and subgroups).

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`Psm$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

Incerti and Jansen (2021). See Section 2.3 for a mathematical description of a PSM and Section 4.2 for an example in oncology. The mathematical approach used to simulate costs and QALYs from state probabilities is described in Section 2.1.

See Also

The `PsmCurves` documentation describes the class for the survival models and the `StateVals` documentation describes the class for the cost and utility models. A `PsmCurves` object is typically created using `create_PsmCurves()`. The `PsmCurves` documentation provides an example in which the model is parameterized from parameter objects (i.e., without having the patient-level data required to fit a model with R). A longer example is provided in `vignette("psm")`.

Examples

```
library("flexsurv")
library("ggplot2")
theme_set(theme_bw())

# Model setup
strategies <- data.frame(strategy_id = c(1, 2, 3),
                         strategy_name = paste0("Strategy ", 1:3))
patients <- data.frame(patient_id = seq(1, 3),
                       age = c(45, 50, 60),
                       female = c(0, 0, 1))
states <- data.frame(state_id =  seq(1, 3),
                     state_name = paste0("State ", seq(1, 3)))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients,
                        states = states)
labs <- c(
  get_labels(hesim_dat),
  list(curve = c("Endpoint 1" = 1,
                 "Endpoint 2" = 2,
                 "Endpoint 3" = 3))
)
n_samples <- 2

# Survival models
surv_est_data <- psm4_exdata$survival
fit1 <- flexsurvreg(Surv(endpoint1_time, endpoint1_status) ~ factor(strategy_id),
                    data = surv_est_data, dist = "exp")
fit2 <- flexsurvreg(Surv(endpoint2_time, endpoint2_status) ~ factor(strategy_id),
                    data = surv_est_data, dist = "exp")
```

```
fit3 <- flexsurvreg(Surv(endpoint3_time, endpoint3_status) ~ factor(strategy_id),
                    data = surv_est_data, dist = "exp")
fits <- flexsurvreg_list(fit1, fit2, fit3)

surv_input_data <- expand(hesim_dat, by = c("strategies", "patients"))
psm_curves <- create_PsmCurves(fits, input_data = surv_input_data,
                               uncertainty = "bootstrap", est_data = surv_est_data,
                               n = n_samples)

# Cost model(s)
cost_input_data <- expand(hesim_dat, by = c("strategies", "patients", "states"))
fit_costs_medical <- lm(costs ~ female + state_name,
                        data = psm4_exdata$costs$medical)
psm_costs_medical <- create_StateVals(fit_costs_medical,
                                      input_data = cost_input_data,
                                      n = n_samples)

# Utility model
utility_tbl <- stateval_tbl(tbl = data.frame(state_id = states$state_id,
                                             min = psm4_exdata$utility$lower,
                                             max = psm4_exdata$utility$upper),
                            dist = "unif")
psm_utility <- create_StateVals(utility_tbl, n = n_samples,
                                hesim_data = hesim_dat)

# Partitioned survival decision model
psm <- Psm$new(survival_models = psm_curves,
               utility_model = psm_utility,
               cost_models = list(medical = psm_costs_medical))
psm$sim_survival(t = seq(0, 5, 1/12))
autoplot(psm$survival_, labels = labs, ci = FALSE, ci_style = "ribbon")
psm$sim_stateprobs()
autoplot(psm$stateprobs_, labels = labs)
psm$sim_costs(dr = .03)
head(psm$costs_)
head(psm$sim_qalys(dr = .03)$qalys_)
```

`psm4_exdata` *Example data for a 4-state partitioned survival model*

Description

A collection of example datasets containing simulated survival, costs, and utility data for a 4-state partitioned survival model.

Usage

```
psm4_exdata
```

Format

A list containing the following elements:

- Survival A data frame containing patient information and time to 3 separate survival endpoints.
- CostsA list of data frames. The first data frame contains medical cost data and the second data frame contains drug cost data.

Survival data

The survival data frame contains a list of 3 survival curves, each containing the following columns.

female An indicator variable equal to 1 if the patient is female and 0 otherwise.

age The age of the patient in years.

strategy_id The id of the treatment strategy used.

endpoint1_time Follow up time with right censored data to survival endpoint 1.

endpoint1_status A status indicator for survival endpoint 1 equal to 0 if alive and 1 if dead.

endpoint2_time Follow up time with right censored data to survival endpoint 2.

endpoint2_status A status indicator for survival endpoint 2 equal to 0 if alive and 1 if dead.

endpoint3_time Follow up time with right censored data to survival endpoint 3.

endpoint3_status A status indicator for survival endpoint 3 equal to 0 if alive and 1 if dead.

Cost data

The cost list contains two data frames.The first data frame contains data on the medical costs by patient and health state, and contains the following columns:

patient_id An integer denoting the id of the patient.

female An indicator variable equal to 1 if the patient is female and 0 otherwise.

state_name A categorical variable denoting the three possible health states.

costs Annualized medical costs.

The second data frame contains data on the drug costs associated with each treatment strategy.

strategy_id The id of each treatment strategy.

costs Annualized drug costs.

`PsmCurves` *Partitioned survival curves*

Description

Summarize N-1 survival curves for an N-state partitioned survival model.

Format

An R6::R6Class object.

Public fields

`params` An object of class `params_surv_list`.

`input_data` An object of class `input_mats`. Each row in `X` must be a unique treatment strategy and patient.

Methods

Public methods:

- `PsmCurves$new()`
- `PsmCurves$hazard()`
- `PsmCurves$cumhazard()`
- `PsmCurves$survival()`
- `PsmCurves$rmst()`
- `PsmCurves$quantile()`
- `PsmCurves$check()`
- `PsmCurves$clone()`

Method `new()`**:** Create a new `PsmCurves` object.

Usage:

```
PsmCurves$new(params, input_data)
```

Arguments:

`params` The `params` field.

`input_data` The `input_data` field.

Returns: A new `PsmCurves` object.

Method `hazard()`**:** Predict the hazard function for each survival curve as a function of time.

Usage:

```
PsmCurves$hazard(t)
```

Arguments:

`t` A numeric vector of times.

Returns: A `data.table` with columns `sample`, `strategy_id`, `patient_id`, `grp_id`, `curve` (the curve number), `t`, and `hazard`.

Method `cumhazard()`**:** Predict the cumulative hazard function for each survival curve as a function of time.

Usage:

```
PsmCurves$cumhazard(t)
```

Arguments:

`t` A numeric vector of times.

Returns: A `data.table` with columns `sample`, `strategy_id`, `patient_id`, `grp_id`, `curve`, `t`, and `cumhazard`.

Method `survival()`**:** Predict survival probabilities for each survival curve as a function of time.

Usage:

```
PsmCurves$survival(t)
```

Arguments:

`t` A numeric vector of times.

Returns: An object of class `survival`.

Method `rmst()`**:** Predict the restricted mean survival time up until time points `t` for each survival curve.

Usage:

```
PsmCurves$rmst(t, dr = 0)
```

Arguments:

`t` A numeric vector of times.

`dr` Discount rate.

Returns: A `data.table` with columns `sample`, `strategy_id`, `patient_id`, `grp_id`, `curve`, `t`, and `rmst`.

Method `quantile()`**:** Predict quantiles of the survival distribution for each survival curve.

Usage:

```
PsmCurves$quantile(p)
```

Arguments:

`p` A numeric vector of probabilities for computing quantiles.

Returns: A `data.table` with columns `sample`, `strategy_id`, `patient_id`, `grp_id`, `curve`, `p` and `quantile`.

Method `check()`**:** Input validation for class. Checks that fields are the correct type.

Usage:

```
PsmCurves$check()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PsmCurves$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`PsmCurves` are conveniently created from either fitted models or parameter objects with `create_PsmCurves()`. A complete economic model can be implemented with the `Psm` class. A longer example is provided in `vignette("psm")`.

Examples

```
library("flexsurv")
N_SAMPLES <- 5 # Number of parameter samples for PSA

# Consider a 3-state model where there is a
# progression-free survival (PFS) and an
# overall survival (OS) endpoint

# (0) Model setup
hesim_dat <- hesim_data(
  strategies = data.frame(
    strategy_id = c(1, 2),
    strategy_name = c("SOC", "New 1")
  ),
  patients = data.frame(
    patient_id = 1
  )
)

# (1) Parameterize survival models
## (1.1) If patient-level data is available,
## we can fit survival models

### (1.1.1) Data for estimation (for simplicity, only use 2 strategies)
surv_est_data <- as_pfs_os(
  onc3[strategy_name != "New 2"],
  patient_vars = c("patient_id", "strategy_name")
)
surv_est_data$strategy_name <- droplevels(surv_est_data$strategy_name)

### (1.1.2) Fit models
fit_pfs <- flexsurvreg(Surv(pfs_time, pfs_status) ~ strategy_name,
                       data = surv_est_data, dist = "exp")
fit_os <- flexsurvreg(Surv(os_time, os_status) ~ strategy_name,
                      data = surv_est_data, dist = "exp")
fits <- flexsurvreg_list(pfs = fit_pfs, os = fit_os)

## (1.2) If patient-level data is NOT available,
## we can construct the parameter objects "manually"

### (1.2.1) Baseline hazard:
### Assume that we know the (log) rate parameters for both PFS and OS
### for SOC (i.e., the intercept) and their standard error
logint_pfs_est <- -1.7470900
logint_pfs_se <-  0.03866223
logint_os_est <- -2.7487675
```

```
logint_os_se <- 0.04845015

### (1.2.2) Relative treatment effect:
### Assume we know the log hazard ratios (and their standard errors)
### for comparing the new interventions to the SOC
loghr_pfs_est_new1 <- -0.1772028
loghr_pfs_se_new1 <- 0.05420119
loghr_os_est_new1 <- -0.1603632
loghr_os_se_new1 <- 0.06948962

### (1.2.3) Create "params_surv_list" object by combining the baseline hazard
### and relative treatment effects
params <- params_surv_list(
  #### Model for PFS
  pfs = params_surv(
    coefs = list(
      rate = data.frame( # coefficients predict log rate
        intercept = rnorm(N_SAMPLES, logint_pfs_est, logint_pfs_se),
        new1 = rnorm(N_SAMPLES, loghr_pfs_est_new1, loghr_pfs_se_new1)
      )
    ),
    dist = "exp"
  ),

  #### Model for OS
  os = params_surv(
    coefs = list(
      rate = data.frame(
        intercept = rnorm(N_SAMPLES, logint_os_est, logint_os_se),
        new1 = rnorm(N_SAMPLES, loghr_os_est_new1, loghr_os_se_new1)
      )
    ),
    dist = "exp"
  )
)

#### The print (and summary) methods for the "params_surv_list" object will
#### summarize each of the model terms, which is a good way to check
#### if it's been setup correctly
params

# (2) Simulation
## (2.1) Construct the model
### (2.1.1) Case where patient-level data was available
### Use create_PsmCurves.params_flexsurvreg_list() method
surv_input_data <- expand(hesim_dat, by = c("strategies", "patients"))
psm_curves1 <- create_PsmCurves(fits, input_data = surv_input_data,
                                n = N_SAMPLES,
                                uncertainty = "normal",
                                est_data = surv_est_data)

### (2.1.2) Case where patient-level data was NOT available
### Use create_PsmCurves.params_surv_list() method
```

```
surv_input_data$intercept <- 1
surv_input_data$new1 <- ifelse(surv_input_data$strategy_name == "New 1",
                               1, 0)
psm_curves2 <- create_PsmCurves(params, input_data = surv_input_data)

## (2.2) Summarize survival models
## There are minor discrepancies between the case where models were fit
## with flexsurvreg() and the case where the "params_surv_list" object
## was constructed manually due to differences in the random draws
## of the parameter samples. These differences are decreasing in the size
## of N_SAMPLES
times <- seq(0, 10, 1/12) # Monthly times

### Quantiles
head(psm_curves1$quantile(p = c(.25, .5, .75)))
head(psm_curves2$quantile(p = c(.25, .5, .75)))

### Survival curves
head(psm_curves1$survival(t = times))
head(psm_curves2$survival(t = times))

### Restricted mean survival
head(psm_curves1$rmst(t = c(2, 5)))
head(psm_curves2$rmst(t = c(2, 5)))
```

`qalys` *Quality-adjusted life-years object*

Description

An object of class `qalys` returned from methods `$sim_qalys()` in model classes that store simulated quality-adjusted life-years (QALYs).

Components

A `qalys` object inherits from `data.table` and contains the following columns:

sample A random sample from the PSA.

strategy_id The treatment strategy ID.

patient_id The patient ID.

grp_id The subgroup ID.

state_id The health state ID.

dr The rate used to discount QALYs.

category A single category always equal to "qalys".

qalys The simulated values of QALYs.

If the argument `lys = TRUE`, then the `data.table` also contains a column `lys` containing simulated life-years.

`qmatrix.matrix` *Transition intensity matrix from tabular object*

Description

Creates transition intensity matrices where elements represent the instantaneous risk of moving between health states.

Usage

```
## S3 method for class 'matrix'
qmatrix(x, trans_mat, ...)

## S3 method for class 'data.table'
qmatrix(x, trans_mat, ...)

## S3 method for class 'data.frame'
qmatrix(x, trans_mat, ...)
```

Arguments

`x`	A two-dimensional tabular object containing elements of the transition intensity matrix. A column represents a transition from state r to state s. Each row represents elements of a different transition intensity matrix. See "Details" for more information.
`trans_mat`	Just as in `IndivCtstmTrans`, a transition matrix describing the states and transitions in a multi-state model.
`...`	Further arguments passed to or from other methods. Currently unused.

Details

The object `x` must only contain non-zero and non-diagonal elements of a transition intensity matrix. The diagonal elements are automatically computed as the negative sum of the other rows.

Value

An array of transition intensity matrices with the third dimension equal to the number of rows in `x`.

See Also

`qmatrix.msm()`

Examples

```
# 3 state irreversible model
tmat <- rbind(c(NA, 1, 2),
              c(NA, NA, 3),
              c(NA, NA, NA))
q12 <- c(.8, .7)
q13 <- c(.2, .3)
q23 <- c(1.1, 1.2)
q <- data.frame(q12, q13, q23)
qmat <- qmatrix(q, trans_mat = tmat)
print(qmat)

# Matrix exponential of each matrix in array
expmat(qmat)
```

`qmatrix.msm` *Transition intensity matrix from* `msm` *object*

Description

Draw transition intensity matrices for a probabilistic sensitivity analysis from a fitted `msm` object.

Usage

```
## S3 method for class 'msm'
qmatrix(x, newdata = NULL, uncertainty = c("normal", "none"), n = 1000, ...)
```

Arguments

`x`	A `msm::msm` object.
`newdata`	A data frame to look for variables with which to predict. A separate transition intensity matrix is predicted based on each row in `newdata`. Can be `NULL` if no covariates are included in the fitted `msm` object.
`uncertainty`	Method used to draw transition intensity matrices. If `"none"`, then point estimates are used. If `"normal"`, then samples are drawn from the multivariate normal distribution of the regression coefficients.
`n`	Number of random observations of the parameters to draw.
`...`	Further arguments passed to or from other methods. Currently unused.

Value

An array of transition intensity matrices with the third dimension equal to the number of rows in `newdata`.

See Also

`qmatrix.matrix()`

Examples

```
library("msm")
set.seed(101)
 qinit <- rbind(
   c(0, 0.28163, 0.01239),
   c(0, 0, 0.10204),
   c(0, 0, 0)
 )
fit <- msm(state_id ~ time, subject = patient_id,
           data = onc3p[patient_id %in% sample(patient_id, 100)],
           covariates = list("1-2" =~ age + strategy_name),
           qmatrix = qinit)
qmatrix(fit, newdata = data.frame(age = 55, strategy_name = "New 1"),
        uncertainty = "none")
qmatrix(fit, newdata = data.frame(age = 55, strategy_name = "New 1"),
        uncertainty = "normal",  n = 3)
```

`rcat` *Random generation for categorical distribution*

Description

Draw random samples from a categorical distribution given a matrix of probabilities. `rcat` is vectorized and written in C++ for speed.

Usage

```
rcat(n, prob)
```

Arguments

`n`	Number of random observations to draw.
`prob`	A matrix of probabilities where rows correspond to observations and columns correspond to categories.

Value

A vector of random samples from the categorical distribution. The length of the sample is determined by n. The numerical arguments other than n are recycled so that the number of samples is equal to n.

Examples

```
p <- c(.2, .5, .3)
n <- 10000
pmat <- matrix(rep(p, n), nrow = n, ncol = length(p), byrow = TRUE)
```

```
# rcat
set.seed(100)
ptm <- proc.time()
samp1 <- rcat(n, pmat)
proc.time() - ptm
prop.table(table(samp1))

# rmultinom from base R
set.seed(100)
ptm <- proc.time()
samp2 <- t(apply(pmat, 1, rmultinom, n = 1, size = 1))
samp2 <- apply(samp2, 1, function(x) which(x == 1))
proc.time() - ptm
prop.table(table(samp2))
```

`rdirichlet_mat` *Random generation for multiple Dirichlet distributions*

Description

Draw random samples from multiple Dirichlet distributions for use in transition probability matrices.

Usage

```
rdirichlet_mat(
  n,
  alpha,
  output = c("array", "matrix", "data.frame", "data.table")
)
```

Arguments

`n`	Number of samples to draw.
`alpha`	A matrix where each row is a separate vector of shape parameters.
`output`	The class of the object returned by the function. Either an `array`, `matrix`, `data.frame`, or `data.table`.

Details

This function is meant for representing the distribution of transition probabilities in a transition matrix. The `(i,j)` element of `alpha` is a transition from state `i` to state `j`. It is vectorized and written in `C++` for speed.

Value

If `output = "array"`, then an array of matrices is returned where each row of each matrix is a sample from the Dirichlet distribution. If `output` results in a two dimensional object (i.e., a `matrix`, `data.frame`, or `data.table`, then each row contains all elements of the sampled matrix from the Dirichlet distribution ordered rowwise; that is, each matrix is flattened. In these cases, the number of rows must be less than or equal to the number of columns.

Examples

```
alpha <- matrix(c(100, 200, 500, 50, 70, 75), ncol = 3, nrow = 2, byrow = TRUE)
samp <- rdirichlet_mat(100, alpha)
print(samp[, , 1:2])
```

`rng_distributions` *Random number generation distributions*

Description

A collection of functions for randomly generating deviates from probability distributions with `define_rng()`.

Usage

```
beta_rng(
  shape1 = 1,
  shape2 = 1,
  mean = NULL,
  sd = NULL,
  names = NULL,
  n = parent.frame()$n
)

dirichlet_rng(alpha, names = NULL, n = parent.frame()$n)

fixed(est, names = NULL, n = parent.frame()$n)

custom(x, names = NULL, n = parent.frame()$n)

gamma_rng(mean, sd, names = NULL, n = parent.frame()$n)

lognormal_rng(meanlog, sdlog, names = NULL, n = parent.frame()$n)

multi_normal_rng(mu, Sigma, names = NULL, n = parent.frame()$n, ...)

normal_rng(mean, sd, names = NULL, n = parent.frame()$n)

uniform_rng(min, max, names = NULL, n = parent.frame()$n)
```

Arguments

`shape1, shape2`	Non-negative parameters of the Beta distribution.
`mean, sd`	Mean and standard deviation of the random variable.
`names`	Names for columns if an object with multiple columns is returned by the function.
`n`	The number of random samples of the parameters to draw. Default is the value of `n` in the environment in which the function is called, which can be useful when used inside `define_rng` because it means that a value does not need to be explicitly passed to `n`.
`alpha`	A matrix where each row is a separate vector of shape parameters.
`est`	A vector of estimates of the variable of interest.
`x`	A numeric `vector`, `matrix`, `data.frame`, or `data.table` containing random samples of the variable of interest from a suitable probability distribution. This would typically be a posterior distribution from a Bayesian analysis.
`meanlog, sdlog`	Mean and standard deviation of the distribution on the log scale.
`mu, Sigma`	`mu` is a vector giving the means of the variables and `Sigma` is a positive-definite symmetric matrix specifying the covariance matrix of the variables.
`...`	Additional arguments to pass to underlying random number generation functions. See "details".
`min, max`	Lower and upper limits of the distribution. Must be finite.

Details

These functions are not exported and are meant for use with `define_rng()`. They consequently assume that the number of samples to draw, `n`, is defined in the parent environment. Convenience random number generation functions include:

`beta_rng()` If `mean` and `sd` are both not `NULL`, then parameters of the beta distribution are derived using the methods of moments with `mom_beta()`. Beta variates are generated with `stats::rbeta()`.

`custom()` Use previously sampled values from a custom probability distribution. There are three possibilities: (i) if `n` is equal to the number previously sampled values (say `n_samples`), then `x` is returned as is; (ii) if `n` < `n_samples`, then samples from `x` are sampled without replacement; and (iii) if `n` > `n_samples`, then samples from `x` are sampled with replacement and a warning is provided.

`dirichlet_rng()` Dirichlet variates for each row in the matrix are generated with `rdirichlet_mat()`. The sampled values are stored in a `data.table` where there is a column for each element of `alpha` (with elements ordered rowwise).

`fixed()` This function should be used when values of the variable of interest are fixed (i.e., they are known with certainty). If `length(est)` > 1, an `n` by `length(est)` `data.table` is returned meaning that each element of `est` is repeated `n` times; otherwise (if `length(est)` == 1), a vector is returned where `est` is repeated `n` times.

`gamma_rng()` The parameters of the gamma distribution are derived using the methods of moments with `mom_gamma()` and gamma variates are generated with `stats::rgamma()`.

`lognormal_rng()` Lognormal variates are generated with `stats::rlnorm()`.

`multi_normal_rng()` Multivariate normal variates are generated with `MASS::mvrnorm()`.

`normal_rng()` Normal variates are generated with `stats::rnorm()`.

`uniform_rng()` Uniform variates are generated with `stats::runif()`.

Value

Functions either return a vector of length `n` or an `n` by `k` `data.table`. Multivariate distributions always return a `data.table`. If a univariate distribution is used, then a `data.table` is returned if each parameter is specified as a vector with length greater than 1; otherwise, if parameters are scalars, then a vector is returned. In the `data.table` case, `k` is equal to the length of the parameter vectors entered as arguments. For example, if the probability distribution contained `mean` as an argument and `mean` were of length 3, then an `n` by 3 matrix would be returned. The length of all parameter vectors must be the same. For instance, if the vector `mean` were of length 3 then all additional parameters (e.g., `sd`) must also be of length 3.

If a `data.table` is returned by a distribution, then its column names are set according to the following hierarchy:

1. With the `names` argument if it is not `NULL`
2. With the names of the parameter vectors if they are named vectors. If there are multiple parameter vector arguments, then the names of the first parameter vector with non `NULL` names is used. For instance, if `mean` and `sd` are both arguments to a random number generation function and `mean` is a named vector, then the names from the vector `mean` are used.
3. As `v1`, ..., `vk` if the `names` argument is `NULL` and there are no named parameter vectors.

See Also

`define_rng()`

`rpwexp` *Random generation for piecewise exponential distribution*

Description

Draw random samples from an exponential distribution with piecewise rates. `rpwexp` is vectorized and written in C++ for speed.

Usage

```
rpwexp(n, rate = 1, time = 0)
```

Arguments

`n`	Number of random observations to draw.
`rate`	A matrix of rates where rows correspond to observations and columns correspond to rates during specified time intervals.
`time`	A vector equal to the number of columns in `rate` giving the times at which the rate changes

Value

A vector of random samples from the piecewise exponential distribution. The length of the sample is determined by n. The numerical arguments other than n are recycled so that the number of samples is equal to n.

Examples

```
rate <- c(.6, 1.2, 1.3)
n <- 100000
ratemat <- matrix(rep(rate, n/2), nrow = n,
                  ncol = 3, byrow = TRUE)
t <- c(0, 10, 15)
ptm <- proc.time()
samp <- rpwexp(n, ratemat, t)
proc.time() - ptm
summary(samp)
```

`set_labels` *Set value labels*

Description

Update existing variables or create new ones that replace existing values with more informative labels as in `factor()`. All modifications are performed by reference (see `data.table::set()` for more information about assignment by reference).

Usage

```
set_labels(x, labels, new_names = NULL, as_factor = TRUE)
```

Arguments

`x`	A `data.table`.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.
`new_names`	A character vector of the same length as `labels` where each element denotes the name of a new variable to create for the corresponding element in `labels`. If `NULL`, then the variables in `labels` are modified and no new variables are created; otherwise, the existing variables are not modified and new variables are created instead.
`as_factor`	If `TRUE` factor variables are created; otherwise character vectors are created.

Value

`x` is modified by reference and returned invisibly.

See Also

`get_labels()`

Examples

```
library("data.table")
labs <- list("strategy_id" = c("s1" = 1,
                               "s2" = 2),
             "grp_id" = c("g1" = 1,
                          "g2" = 2))
d1 <- data.table(strategy_id = 1:2, grp_id = 1:2)
d2 <- copy(d1); d3 <- copy(d2)
set_labels(d2, labels = labs)
set_labels(d3, labels = labs, new_names = c("strategy_name", "grp_name"))
d1
d2
d3
```

---

sim_ev *Expected values from state probabilities*

Description

Simulate expected values as a function of simulated state occupancy probabilities, with simulation of costs and quality-adjusted life-years (QALYs) as particular use cases.

Usage

```
## S3 method for class 'stateprobs'
sim_ev(
  object,
  models = NULL,
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right"),
  value_name = "value",
  outcome_name = "outcome",
  ...
)

sim_qalys(
  object,
  model,
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right"),
  lys = TRUE
)

sim_costs(
```

```
  object,
  models,
  dr = 0.03,
  integrate_method = c("trapz", "riemann_left", "riemann_right")
)
```

Arguments

`object`	A `stateprobs` object.
`dr`	Discount rate.
`integrate_method`	Method used to integrate state values when computing costs or QALYs. Options are `trapz` (the default) for the trapezoid rule, `riemann_left` left for a left Riemann sum, and `riemann_right` right for a right Riemann sum.
`value_name`	Name of the column containing values of the outcome. Default is `"value"`.
`outcome_name`	Name of the column indicating the outcome corresponding to each model. Only used if `models` is a list. Default is `"outcome"`.
`...`	Currently unused.
`model, models`	An object or list of objects of class `StateVals` used to model state values. When using `sim_qalys()`, this should be a single model for utility. With `sim_costs()`, a list of models should be used with one model for each cost category. Finally, with `sim_ev()`, this may either be a single model or a list of models. May also be `NULL`, in which case length of stay is computed based on the state probabilities contained in `object`.
`lys`	If `TRUE`, then life-years are simulated in addition to QALYs.

Details

Expected values in cohort models (i.e., those implemented with the `CohortDtstm` and `Psm` classes) are mean outcomes for patients comprising the cohort. The method used to simulate expected values depends on the `$method` field in the `StateVals` object(s) stored in `model(s)`. If `$method = "starting"`, then state values represent a one-time value that occurs at time 0.

The more common use case is `$method = "wlos"`, or a "weighted length of stay". That is, expected values for each health state can be thought of as state values weighted by the time a patient spends in each state (and discounted by a discount factor that depends on the discount rate `dr`). The precise computation proceeds in four steps. In the first step, the probability of being in each health state at each discrete time point is simulated (this is the output contained in the `stateprobs` object). Second, a `StateVals` model is used to predict state values at each time point. Third an expected value at each time point is computed by multiplying the state probability, the state value, and the discount factor. Fourth, the expected values at each time point are summed across all time points.

The summation in the fourth step can be thought of as a discrete approximation of an integral. In particular, the limits of integration can be partitioned into time intervals, with each interval containing a start and an end. The `integrate_method` argument determines the approach used for this approximation:

1. A left Riemann sum (`integrate_method = "riemann_left"`) uses expected values at the start of each time interval.

2. A right Riemann sum (`integrate_method = "riemann_right"`) uses expected values at the end of each time interval.
3. The trapezoid rule (`integrate_method = "trapz"`) averages expected values at the start and end of each time interval. (This will generally be the most accurate and is recommended.)

Mathematical details are provided in the reference within the "References" section below.

Value

`sim_ev()` returns a `data.table` with the following columns:

sample A random sample from the PSA.

strategy_id The treatment strategy ID.

patient_id The patient ID.

grp_id The subgroup ID.

state_id The health state ID.

dr The rate used to discount costs.

outcome The outcome corresponding to each model in `models`. Only included if `models` is a list.

value The expected value.

The names of the `outcome` and `value` columns may be changed with the `value_name` and `outcome_name` arguments. `sim_costs()` and `sim_qalys()` return similar objects, that are of class `costs` and `qalys`, respectively.

Note

The ID variables in the state value models in `models` must be consistent with the ID variables contained in `object`. In particular, the `models` should predict state values for each non-absorbing health state in `object`; that is, the number of health states modeled with the `models` should equal the number of health states in `object` less the number of absorbing states.

The absorbing states are saved as an attribute named `absorbing` to `stateprobs` objects. When simulating state probabilities with a `CohortDtstmTrans` object, the absorbing state is determined by the `absorbing` field in the class; in a `Psm` (or with `sim_stateprobs.survival()`), the absorbing state is always equal to the final health state.

References

Incerti and Jansen (2021). See Section 2.1 for mathematical details.

See Also

State probabilities can be simulated using the `$sim_stateprobs()` methods from either the `CohortDtstmTrans` (or `CohortDtstm`) or `Psm` classes. State probabilities can also be computed directly from survival curves with the generic method `sim_stateprobs.survival()`.

Costs and QALYs are typically computed within the `R6` model classes using the `$sim_costs()` and `$sim_qalys()` methods. For instance, see the documentation and examples for the `CohortDtstm` and `Psm` classes. The `sim_qalys()` and `sim_costs()` functions are exported to give users additional flexibility when creating their own modeling pipelines. `sim_ev()` may be useful for computing outcomes other than costs or QALYs.

`costs` and `qalys` objects can be passed to `summarize_ce()` to create a cost-effectiveness object for performing a cost-effectiveness analysis with `cea()`. Although note that typically the `$summarize()` method belonging to the `CohortDtstm` or `Psm` classes would be used instead.

Use the `IndivCtstm` class to simulate costs and QALYs with an individual continuous-time state transition model.

Examples

```
# We need (i) a state probability object and (ii) a model for state values
## We should start by setting up our decision problem
hesim_dat <-  hesim_data(strategies = data.frame(strategy_id = 1:2),
                         patients = data.frame(patient_id = 1:3),
                         states = data.frame(state_id = 1))
input_data <- expand(hesim_dat, by = c("strategies", "patients"))

## (i) Simulate a state probability object
tpmat_id <- tpmatrix_id(input_data, n_samples = 2)
p_12 <- ifelse(tpmat_id$strategy_id == 1, .15, .1)
tpmat <- tpmatrix(
  C, p_12,
  0, 1
)
transmod <- CohortDtstmTrans$new(params = tparams_transprobs(tpmat, tpmat_id))
stprobs <- transmod$sim_stateprobs(n_cycles = 3)

## Construct model for state values
outcome_tbl <- stateval_tbl(
  data.frame(
    state_id = 1,
    est = 5000
  ),
  dist = "fixed"
)
outmod <- create_StateVals(outcome_tbl, n = 2, hesim_data = hesim_dat)

# We can then simulate expected values
## The generic expected values function
sim_ev(stprobs, models = outmod)

## We can also pass a list of models
sim_ev(stprobs, models = list(`Outcome 1` = outmod))

## Suppose the outcome were a cost category. Then we might
## prefer the following:
sim_costs(stprobs, models = list(drug = outmod))

## Length of stay is computed if there is no state value model
sim_ev(stprobs)
```

---

sim_stateprobs.survival

Simulate state probabilities from survival curves

Description

Simulate health state probabilities from a `survival` object using partitioned survival analysis.

Usage

```
## S3 method for class 'survival'
sim_stateprobs(x, ...)
```

Arguments

`x`	An object of class `survival`.
`...`	Further arguments passed to or from other methods.

Details

In an N-state partitioned survival model there are $N-1$ survival curves and $S_n(t)$ is the cumulative survival function denoting the probability of survival to health state n or a lower indexed state beyond time t. The probability that a patient is in health state 1 at time t is simply $S_1(t)$. State membership in health states $2, \ldots, N-1$ is calculated as $S_n(t) - S_{n-1}(t)$. Finally, the probability of being in the final health state N (i.e., the death state) is $1 - S_{N-1}(t)$, or one minus the overall survival curve.

In some cases, the survival curves may cross. `hesim` will issue a warning but the function will still run. Probabilities will be set to 0 in a health state if the prior survival curve lies above the curve for state n; that is, if $S_n(t) < S_{n-1}(t)$, then the probability of being in state n is set to 0 and $S_n(t)$ is adjusted to equal $S_{n-1}(t)$. The probability of being in the final health state is also adjusted if necessary to ensure that probabilities sum to 1.

Value

A `stateprobs` object.

See Also

`survival`

Examples

```
library("data.table")
library("survival")

# This example shows how to simulate a partitioned survival model by
# manually constructing a "survival" object. We will consider a case in which
# Cox proportional hazards models from the survival package---which are not
```

```
# integrated with hesim---are used for parameter estimation. We will use
# point estimates in the example, but bootstrapping, Bayesian modeling,
# or other techniques could be used to draw samples for a probabilistic
# sensitivity analysis.

# (0) We first setup our model per usual by defining the treatment strategies,
# target population, and health states
hesim_dat <- hesim_data(
  strategies = data.table(strategy_id = 1:3,
                          strategy_name = c("SOC", "New 1", "New 2")),
  patients = data.table(patient_id = 1:2,
                        female = c(0, 1),
                        grp_id = 1),
  states = data.table(state_id = 1:2,
                      state_name = c("Stable", "Progression"))
)

# (1) Next we will estimate Cox models with survival::coxph(). We illustrate
# by predicting progression free survival (PFS) and overall survival (OS)
## Fit models
onc3_pfs_os <- as_pfs_os(onc3, patient_vars = c("patient_id", "female",
                                                "strategy_name"))
fit_pfs <- coxph(Surv(pfs_time, pfs_status) ~ strategy_name + female,
                 data = onc3_pfs_os)
fit_os <- coxph(Surv(os_time, pfs_status) ~ strategy_name + female,
                data = onc3_pfs_os)

## Predict survival on input data
surv_input_data <- expand(hesim_dat)
times <- seq(0, 14, 1/12)
predict_survival <- function(object, newdata, times) {
  surv <- summary(survfit(object, newdata = newdata, se.fit = FALSE),
                  t = times)
  pred <- newdata[rep(seq_len(nrow(newdata)), each = length(times)), ]
  pred[, sample := 1] # Point estimates only in this example
  pred[, time := rep(surv$time, times = nrow(newdata))]
  pred[, survival := c(surv$surv)]
  return(pred[, ])
}
pfs <- predict_survival(fit_pfs, newdata = surv_input_data, times = times)
os <- predict_survival(fit_os, newdata = surv_input_data, times = times)
surv <- rbind(
  as.data.table(pfs)[, curve := 1L],
  as.data.table(os)[, curve := 2L]
)

## Convert predictions to a survival object
surv <- survival(surv, t = "time")
## Not run: autoplot(surv)

# (2) We can then compute state probabilities from the survival object
stprobs <- sim_stateprobs(surv)
```

```
# (3) Finally, we can use the state probabilities to compute QALYs and costs
## A dummy utility model to illustrate
utility_tbl <- stateval_tbl(
  data.table(state_id = 1:2,
             est = c(1, 1)
  ),
  dist = "fixed"
)
utilitymod <- create_StateVals(utility_tbl,
                               hesim_data = hesim_dat,
                               n = 1)

## Instantiate Psm class and compute QALYs
psm <- Psm$new(utility_model = utilitymod)
psm$stateprobs_ <- stprobs
psm$sim_qalys()
psm$qalys_
```

`stateprobs` *State probability object*

Description

An object of class `stateprobs` returned by `sim_stateprobs()` or from `$sim_stateprobs()` methods in model classes.

Components

A `stateprobs` object inherits from `data.table` and contains the following columns:

sample A random sample from the PSA.

strategy_id The treatment strategy ID.

patient_id The patient ID.

grp_id The subgroup ID.

state_id The health state ID.

t The time at which a state probability is computed.

prob The probability of being in a given health state.

When simulating individual-level models, the `patient_id` column is not included as state probabilities are computed by averaging across patients.

In cohort models, the object also contains `size` and `absorbing` attributes. The `size` attribute is a numeric vector with the elements `n_samples`, `n_strategies`, `n_patients`, `n_states`, and `n_times` denoting the number of samples, treatment strategies, patients, health states, and times. The `absorbing` attribute is a numeric vector containing the absorbing health states (see the `absorbing` field of the `CohortDtstmTrans` class for more details).

`StateVals` *Model for state values*

Description

Simulate values (i.e., utility or costs) associated with health states in a state transition or partitioned survival model.

Public fields

`params` Parameters for simulating state values. Currently supports objects of class `tparams_mean` or `params_lm`.

`input_data` An object of class input_mats. Only used for `params_lm` objects.

`method` The method used to simulate costs and quality-adjusted life-years (QALYs) as a function of state values. If `wlos`, then costs and QALYs are simulated by weighting state values by the length of stay in a health state. If `starting`, then state values represent a one-time value that occurs when a patient enters a health state. When `starting` is used in a cohort model, the state values only accrue at time 0; in contrast, in an individual-level model, state values accrue each time a patient enters a new state and are discounted based on time of entrance into that state.

`time_reset` If `FALSE` then time intervals are based on time since the start of the simulation. If `TRUE`, then time intervals reset each time a patient enters a new health state. This is relevant if, for example, costs vary over time within health states. Only used if `method` = `wlos`.

Methods

Public methods:

- `StateVals$new()`
- `StateVals$sim()`
- `StateVals$check()`
- `StateVals$clone()`

Method `new()`**:** Create a new `StateVals` object.

Usage:

```
StateVals$new(
  params,
  input_data = NULL,
  method = c("wlos", "starting"),
  time_reset = FALSE
)
```

Arguments:

`params` The `params` field.

`input_data` The `input_data` field.

`method` The `method` field.

`time_reset` The `time_reset` field.

Returns: A new `StateVals` object.

Method `sim()`**:** Simulate state values with either predicted means or random samples by treatment strategy, patient, health state, and time `t`.

Usage:

```
StateVals$sim(t, type = c("predict", "random"))
```

Arguments:

`t` A numeric vector of times.

`type` `"predict"` for mean values or `"random"` for random samples.

Returns: A `data.table` of simulated state values with columns for `sample`, `strategy_id`, `patient_id`, `state_id`, `time`, and `value`.

Method `check()`**:** Input validation for class. Checks that fields are the correct type.

Usage:

```
StateVals$check()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
StateVals$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Simple sick-sicker example where drug costs vary by treatment strategy
# and over time. Prior to time = 5, costs are $10,000 for treatment strategy
# 1 and $5,000 for treatment strategy 2. After time = 5, costs are $2,000
# for both treatment strategies

## Setup the model
hesim_dat <- hesim_data(
  strategies = data.frame(strategy_id = c(1, 2)),
  patients = data.frame(patient_id = 1:3),
  states = data.frame(state_id = c(1, 2), # Non-death states
                      state_name = c("sick", "sicker"))
)

## Drug costs vary by health state and time interval
drugcost_tbl <- stateval_tbl(
  data.frame(
    strategy_id = c(1, 1, 2, 2),
    time_start = c(0, 5, 0, 5),
    est = c(10000, 2000, 5000, 2000)
  ),
  dist = "fixed"
)
drugcost_tbl
```

```
## Create drug cost model
drugcostmod <- create_StateVals(drugcost_tbl, n = 1, hesim_data = hesim_dat)

## Explore predictions from the drug cost model
drugcostmod$sim(t = c(2, 6), type = "predict")
```

`stateval_tbl` *Table to store state value parameters*

Description

Create a table for storing parameter estimates used to simulate costs or utility in an economic model by treatment strategy, patient, health state, and (optionally) time interval.

Usage

```
stateval_tbl(
  tbl,
  dist = c("norm", "beta", "gamma", "lnorm", "unif", "fixed", "custom"),
  hesim_data = NULL,
  grp_var = NULL
)
```

Arguments

`tbl`	A `data.frame` or `data.table` for storing parameter values. See "Details" for specifics.
`dist`	Probability distribution used to sample parameters for a probabilistic sensitivity analysis (PSA).
`hesim_data`	A `hesim_data` object. This argument is deprecated and should be passed to `create_StateVals.stateval_tbl()` instead.
`grp_var`	The name of the variable used to group patients.

Details

`tbl` is a tabular object containing columns for treatment strategies (`strategy_id`), patients (`patient_id`), health states (`state_id`), and/or the start of time intervals (`time_start`). The table must contain at least one column named `strategy_id`, `patient_id`, or `state_id`, but does not need to contain all of them. Each row denotes a unique treatment strategy, patient, health state, and/or time interval pair. `tbl` may also contain a column with the name specified in `grp_var` (rather than `patient_id`) so that state values are assigned to groups of patients.

`tbl` must also contain columns summarizing the state values for each row, which depend on the probability distribution selected with `dist`. Available distributions include the normal (`norm`), beta (`beta`), gamma (`gamma`), lognormal (`lnorm`), and uniform (`unif`) distributions. In addition, the option `fixed` can be used if estimates are known with certainty and `custom` can be used if parameter values for a PSA have been previously sampled from an arbitrary probability distribution. The columns in `tbl` that must be included, by distribution, are:

norm `mean` and `sd`

beta `mean` and `se` or `shape1` and `shape2`

gamma `mean` and `se`, `shape` and `rate`, or `shape` and `scale`

lnorm `meanlog` or `sdlog`

unif `min` and `max`

fixed `est`

custom `sample` and `value`

Note that if `dist = "custom"`, then `tbl` must include a column named `sample` (an integer vector denoting a unique random draw) and `value` (denoting the value of the randomly sampled parameter). In this case, there is a unique row in `tbl` for each random draw (`sample`) and each combination of strategies, patients, health states, and/or time intervals. Again, `tbl` must contain at least one column named `strategy_id`, `patient_id` (or `grp_var`), or `state_id`, but does not need to contain them all.

Value

An object of class `stateval_tbl`, which is a `data.table` of parameter values with attributes for `dist` and `grp_var`.

See Also

`create_StateVals()`, `StateVals`

Examples

```
strategies <- data.frame(strategy_id = c(1, 2))
patients <- data.frame(patient_id = seq(1, 3),
                       grp = c(1, 1, 2),
                       age = c(45, 50, 60),
                       female = c(0, 0, 1))
states <- data.frame(state_id = c(1, 2))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients,
                        states = states)

# Utility varies by health state and patient group
utility_tbl <- stateval_tbl(data.frame(state_id = rep(states$state_id, 2),
                                       grp = rep(rep(c(1, 2)), each = nrow(states)),
                                       mean = c(.8, .7, .75, .55),
                                       se = c(.18, .12, .10, .06)),
                            dist = "beta",
                            grp_var = "grp")
print(utility_tbl)
utilmod <- create_StateVals(utility_tbl, n = 2, hesim_data = hesim_dat)
```

```
# Costs vary by treatment strategy
cost_tbl <- stateval_tbl(data.frame(strategy_id = strategies$strategy_id,
                                    mean = c(5000, 3000),
                                    se = c(200, 100)),
                         dist = "gamma")
print(cost_tbl)
costmod <- create_StateVals(cost_tbl, n = 2, hesim_data = hesim_dat)
```

`summary.ce` *Summary method for cost-effectiveness object*

Description

Summarize a `ce` object by producing confidence intervals for quality-adjusted life-years (QALYs) and each cost category with `summary.ce()` and format for pretty printing with `format.summary.ce()`.

Usage

```
## S3 method for class 'ce'
summary(object, prob = 0.95, labels = NULL, ...)

## S3 method for class 'summary.ce'
format(
  x,
  digits_qalys = 2,
  digits_costs = 0,
  dr_qalys = NULL,
  dr_costs = NULL,
  pivot_from = "strategy",
  drop_grp = TRUE,
  pretty_names = TRUE,
  ...
)
```

Arguments

`object`	A `ce` object.
`prob`	A numeric scalar in the interval `(0,1)` giving the confidence interval. Default is 0.95 for a 95 percent interval.
`labels`	A list of named vectors containing the values and labels of variables. The elements of each vector are the values of a variable and the names are the labels. The names of the list are the names of the variables. See the output returned by `get_labels()` for an example.
`...`	Further arguments passed to or from other methods. Currently unused.
`x`	A `summary.ce` object.

`digits_qalys`	Number of digits to use to report QALYs.
`digits_costs`	Number of digits to use to report costs.
`dr_qalys`	Discount rate to subset to for quality-adjusted life-years (QALYs).
`dr_costs`	Discount rate to subset to for costs.
`pivot_from`	Character vector denoting a column or columns used to "widen" the data. Should either be `"strategy"`, `"grp"`, `"outcome"`, or some combination of the three. There will be one column for each value of the variables in `pivot_from`. Default is to widen so there is a column for each treatment strategy. Set to `NULL` if you do not want to widen the table.
`drop_grp`	If `TRUE`, then the group column will be removed if there is only one subgroup; other it will be kept. If `FALSE`, then the `grp` column is never removed.
`pretty_names`	Logical. If `TRUE`, then the columns `strategy`, `grp`, `outcome`, `dr`, and `value` are renamed (if they exist) to `Strategy`, `Group`, `Outcome`, `Discount rate`, and `Value`.

Details

For an example, see `IndivCtstm`.

Value

`summary.ce()` returns an object of class `summary.ce` that is a tidy `data.table` with the following columns:

dr The discount rate.

strategy The treatment strategy.

grp The patient subgroup.

type Either `"QALYs"` or `"Costs"`.

category Category is always `"QALYs"` when `type` == `"QALYs"`; otherwise, it is the cost category.

estimate The point estimate computed as the average across the PSA samples.

lower The lower limit of the confidence interval.

upper The upper limit of the confidence interval.

`format.summary.ce()` formats the table according to the arguments passed.

`summary.eval_rng` *Summarize* `eval_rng` *object*

Description

Summarize the model parameters randomly sampled for probabilistic sensitivity analysis with `eval_rng()`.

Usage

```
## S3 method for class 'eval_rng'
summary(object, probs = c(0.025, 0.975), sep = "_", ...)

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

Arguments

`object, x`	An `eval_rng` object.
`probs`	A numeric vector of probabilities with values in `[0,1]` used to compute quantiles with `stats::quantile()`.
`sep`	When a list element returned by `eval_rng` is a tabular object, the parameter name is created by concatenating the name of the list element with the columns of the tabular object. The `sep` argument determines the character string used to separate the terms.
`...`	For the print method, arguments to pass to `summary.eval_rng()`.

Value

`summary.eval_rng()` returns a `data.table` with columns for (i) the name of the parameter (`param`), (ii) the mean of the parameter samples (`mean`), (iii) the standard deviation of the parameter samples (`sd`), and (iv) quantiles of the parameter samples corresponding to the `probs` argument. `print.eval_rng()` prints the output of `summary.eval_rng()` to the console.

See Also

See `eval_rng()` for an example.

`summary.params` *Summarize parameter objects*

Description

Summarize the coefficients of a parameter object by computing the mean, standard deviation, and quantiles for each model term. This is a convenient way to check whether a parameter object has been specified correctly and sampling distributions of the coefficients are as expected.

Usage

```
## S3 method for class 'params_lm'
summary(object, probs = c(0.025, 0.975), ...)

## S3 method for class 'params_mlogit'
summary(object, probs = c(0.025, 0.975), ...)

## S3 method for class 'params_mlogit_list'
summary(object, probs = c(0.025, 0.975), ...)

## S3 method for class 'params_surv'
summary(object, probs = c(0.025, 0.975), ...)

## S3 method for class 'params_surv_list'
summary(object, probs = c(0.025, 0.975), ...)
```

Arguments

`object`	An object of the appropriate class.
`probs`	A numeric vector of probabilities with values in `[0,1]` used to compute quantiles. By default, the 2.5th and 97.5th percentiles are computed.
`...`	Additional arguments affecting the summary. Currently unused.

Value

A `data.table` that always contains the following columns:

term The regression term.

mean The mean value of the regression term.

sd The standard deviation of the values of the regression term.

In addition, the `probs` argument determines the quantiles that are computed. By default, the columns `2.5%` and `97.5%` are returned corresponding to the 2.5th and 97.5th percentiles.

Finally, the following columns may also be present:

parameter The name of the parameter of interest. This is relevant for any parametric model in which the underlying probability distribution has multiple parameters. For instance, both `params_surv` and `params_surv_list` store regression coefficients that are used to model the underlying parameters of the survival distribution (e.g., shape and scale for a Weibull model). Similarly, there are two parameters (`mean` and `sd`) for `params_lm` objects.

model The name of the statistical model. This is used for a `params_surv_list` object, where each list element represents a separate model. In a state transition model, each model is a unique health state transition and in a partitioned survival model, there is a separate model for each curve.

to The health state that is being transitioned to. In `params_mlogit` and `params_mlogit_list` objects, there are coefficients for each health state that can be transitioned to.

from The health state that is being transitions from. This is used for a `params_mlogit_list` objects where a different multinomial logistic regression is used for each state that can be transitioned from.

See Also

For examples, see the the underlying parameter object functions: `params_lm()`, `params_surv()`, `params_surv_list()`, `params_mlogit()`, and `params_mlogit_list()`.

`summary.tparams_mean` *Summarize* `tparams_mean` *object*

Description

The `summary()` method summarizes a `tparams_mean` object containing predicted means; summary statistics are computed for each combination of the ID variables. The `print()` method summarizes the object using `summary.tparams_mean()` and prints it to the console.

Usage

```
## S3 method for class 'tparams_mean'
summary(object, probs = c(0.025, 0.975), ...)

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

Arguments

`object, x`	A `tparams_mean` object.
`probs`	A numeric vector of probabilities with values in `[0,1]` used to compute quantiles. By default, the 2.5th and 97.5th percentiles are computed.
`...`	Currently unused.

Value

A `data.table` with columns for (i) the ID variables, (ii) the mean of each parameter across parameter samples (`mean`), (iii) the standard deviation of the parameter samples (`sd`), and (iv) quantiles of the parameter samples corresponding to the `probs` argument.

See Also

See `tparams_mean` for an example use of the summary and print methods.

`summary.tparams_transprobs`

Summarize `tparams_transprobs` *object*

Description

The `summary()` method summarizes a `tparams_transprobs` object containing predicted transition probabilities; summary statistics are computed for each possible transition by the relevant ID variables.

Usage

```
## S3 method for class 'tparams_transprobs'
summary(object, probs = NULL, unflatten = FALSE, ...)
```

Arguments

`object`	A `tparams_transprobs` object.
`probs`	A numeric vector of probabilities with values in `[0,1]` used to compute quantiles. Computing quantiles can be slow when `object` is large, so the default is `NULL`, meaning that no quantiles are computed.
`unflatten`	If `FALSE`, then each column containing a summary statistic is a vector and the generated table contains one row (for each set of ID variables) for each possible transition; if `TRUE`, then each column stores a list of `matrix` objects containing transition probability matrices formed by "unflattening" the one-dimensional vectors. See "Value" below for additional details.
`...`	Additional arguments affecting the summary. Currently unused.

Value

If `unflatten = "FALSE"` (the default), then a `data.table` is returned with columns for (i) the health state that is being transitioned from (`from`), (ii) the health state that is being transitioned to (`to`) (iii) the mean of each parameter across parameter samples (`mean`), (iv) the standard deviation of the parameter samples (`sd`), and (v) quantiles of the parameter samples corresponding to the `probs` argument.

If, on the other hand, `unflatten = "TRUE"`, then the parameters are unflattened to form transition probability matrices; that is, the `mean`, `sd`, and quantile columns are (lists of) matrices.

In both cases, the ID variables are also returned as columns.

See Also

See `tparams_transprobs` for an example use of the summary method.

`summary.tpmatrix` *Summarize transition probability matrix*

Description

Summarize a `tpmatrix` object storing transition probability matrices. Summary statistics are computed for each possible transition.

Usage

```
## S3 method for class 'tpmatrix'
summary(object, id = NULL, probs = NULL, unflatten = FALSE, ...)
```

Arguments

`object`	A `tpmatrix` object.
`id`	A `tpmatrix_id` object for which columns contain the ID variables for each row in `object`. If not `NULL`, then transition probability matrices are summarized by the ID variables in `id`.
`probs`	A numeric vector of probabilities with values in `[0,1]` used to compute quantiles. Computing quantiles can be slow when `object` is large, so the default is `NULL`, meaning that no quantiles are computed.
`unflatten`	If `FALSE`, then each column containing a summary statistic is a vector and the generated table contains one row (for each set of ID variables) for each possible transition; if `TRUE`, then each column stores a list of `matrix` objects containing transition probability matrices formed by "unflattening" the one-dimensional vectors. See "Value" below for additional details.
`...`	Additional arguments affecting the summary. Currently unused.

Value

If `unflatten = "FALSE"` (the default), then a `data.table` is returned with columns for (i) the health state that is being transitioned from (`from`), (ii) the health state that is being transitioned to (`to`) (iii) the mean of each parameter across parameter samples (`mean`), (iv) the standard deviation of the parameter samples (`sd`), and (v) quantiles of the parameter samples corresponding to the `probs` argument.

If, on the other hand, `unflatten = "TRUE"`, then the parameters are unflattened to form transition probability matrices; that is, the `mean`, `sd`, and quantile columns are (lists of) matrices.

In both cases, if `id` is not `NULL`, then the ID variables are also returned as columns.

Examples

```
library("data.table")
hesim_dat <-  hesim_data(strategies = data.table(strategy_id = 1:2),
                                patients = data.table(patient_id = 1:3))
input_data <- expand(hesim_dat, by = c("strategies", "patients"))

# Summarize across all rows in "input_data"
p_12 <- ifelse(input_data$strategy_id == 1, .8, .6)
p <- tpmatrix(
  C, p_12,
  0, 1
)

## Summary where each column is a vector
summary(p)
summary(p, probs = c(.025, .975))

## Summary where each column is a matrix
ps <- summary(p, probs = .5, unflatten = TRUE)
ps
ps$mean

# Summarize by ID variables
tpmat_id <- tpmatrix_id(input_data, n_samples = 2)
p_12 <- ifelse(tpmat_id$strategy_id == 1, .8, .6)
p <- tpmatrix(
  C, p_12,
  0, 1
)

## Summary where each column is a vector
summary(p, id = tpmat_id)

## Summary where each column is a matrix
ps <- summary(p, id = tpmat_id, unflatten = TRUE)
ps
ps$mean
```

`survival` *Survival object*

Description

An object of class `survival` stores survival probabilities. It is typically returned by `Psm$sim_survival()` or `PsmCurves$survival()`; however, it can also be constructed "manually" from existing data using the `survival()` function as described below. The latter option is useful if survival modeling has been performed by an `R` package other than those that integrate with `hesim` (currently `flexsurv`). In this case a simulation model can still be developed by using `sim_stateprobs.survival()` to compute simulated state probabilities and then simulating quality-adjusted life-years and costs in a typical fashion.

Usage

```
survival(
  data,
  sample = "sample",
  strategy_id = "strategy_id",
  patient_id = "patient_id",
  grp_id = "grp_id",
  curve = "curve",
  t = "t",
  survival = "survival"
)
```

Arguments

`data`	A tabular object that can be coerced to a `data.table` with `as.data.table()`.
`sample`	The name of the column corresponding to `sample`.
`strategy_id`	The name of the column corresponding to `strategy_id`.
`patient_id`	The name of the column corresponding to `patient_id`.
`grp_id`	The name of the column corresponding to `grp_id`.
`curve`	The name of the column corresponding to `curve`.
`t`	The name of the column corresponding to `t`.
`survival`	The name of the column corresponding to `survival`.

Value

An object of class `survival` that inherits from `data.table` and contains the following columns:

sample A random sample from the PSA.

strategy_id The treatment strategy ID.

patient_id The patient ID.

grp_id The subgroup ID.

curve One of the `N-1` survival curves in an N-state partitioned survival model. Each curve corresponds to unique endpoint.

t The time at which a survival probability is computed.

survival The probability of surviving to time `t`.

The object also contains a `size` attribute that contains the elements `n_samples`, `n_strategies`, `n_patients`, `n_states`, and `n_times` denoting the number of samples, treatment strategies, patients, health states, and times.

See Also

`survival` objects are returned by methods in the `Psm` and `PsmCurves` classes. An example in which a `survival` object is constructed "manually" (presumably from a preexisting survival model fit using software other than `flexsurv`) is provided in the documentation to `sim_stateprobs.survival()`.

`surv_quantile` *Survival quantiles*

Description

Compute quantiles from survival curves.

Usage

```
surv_quantile(x, probs = 0.5, t, surv_cols, by)
```

Arguments

`x`	A `data.table` or `data.frame`.
`probs`	A numeric vector of probabilities with values in `[0,1]`.
`t`	A character scalar of the name of the time column.
`surv_cols`	A character vector of the names of columns containing survival curves.
`by`	A character vector of the names of columns to group by.

Value

A `data.table` of quantiles of each survival curve in `surv_cols` by each group in `by`.

Examples

```
library("data.table")
t <- seq(0, 10, by = .01)
surv1 <- seq(1, .3, length.out = length(t))
surv2 <- seq(1, .2, length.out = length(t))
strategies <- c("Strategy 1", "Strategy 2")
surv <- data.table(strategy = rep(strategies, each = length(t)),
                   t = rep(t, 2),
                   surv = c(surv1, surv2))
surv_quantile(surv, probs = c(.4, .5), t = "t",
              surv_cols = "surv", by = "strategy")
```

`time_intervals` *Time intervals*

Description

Create a table of time intervals given a vector or data frame of unique times. This would typically be passed to id_attributes.

Usage

```
time_intervals(times)
```

Arguments

`times`	Either a vector of starting times for each interval or a `data.frame` with at least one column named `time_start`.

Value

An object of class `time_intervals` that inherits from `data.table` in the same format as `time_intervals` as described in id_attributes.

See Also

id_attributes

Examples

```
time_intervals(c(0, 3, 5))
time_intervals(data.frame(time_start = c(0, 3, 5),
                          time_cat = c("Time <= 3", "3 < Time <= 5",
                                       "Time > 5")))
```

`tparams` *Transformed parameter object*

Description

Objects prefixed by "tparams_" are lists containing transformed parameters used to simulate outcomes. The parameters have presumably already been transformed as a function of input data and consequently do not need to be used alongside input matrices. In other words, transformed parameters are parameters that have already been predicted as a function of covariates.

See Also

params

`tparams_mean` *Predicted means*

Description

Create a list containing means predicted from a statistical model.

Usage

```
tparams_mean(value, ...)
```

Arguments

`value`	Matrix of samples from the distribution of the mean. Columns denote random samples and rows denote means for different observations.
`...`	Arguments to pass to id_attributes. Each row in `value` must be a prediction for a `strategy_id`, `patient_id`, `state_id`, and optionally `time_id` combination.

Value

An object of class `tparams_mean`, which is a list containing `value`, `n_samples`, and the ID attributes passed to id_attributes.

Note

The `tparams_mean()` constructor would not normally be used by users; instead, a `tparams_mean` object is typically created automatically as part of the `StateVals` class with `create_StateVals()`.

See Also

A `tparams_mean` object is a type of transformed parameter object and is a supported class type of the `params` field of the `StateVals` class. See the documentation for `create_StateVals()` and `stateval_tbl()` for examples of how to create`StateVals` objects. Predicted means can be summarized across parameter samples using `summary.tparams_mean()`.

Examples

```
# Setup model
hesim_dat <- hesim_data(
  strategies = data.frame(strategy_id = c(1, 2)),
  patients = data.frame(patient_id = c(1, 2)),
  states = data.frame(
    state_id = c(1, 2, 3),
    state_name = c("state1", "state2", "state3")
  )
)

# Cost model
```

```
cost_tbl <- stateval_tbl(
  data.frame(strategy_id = hesim_dat$strategies$strategy_id,
             mean = c(5000, 3000),
             se = c(200, 100)
            ),
  dist = "gamma"
)
costmod <- create_StateVals(cost_tbl, n = 2, hesim_data = hesim_dat)

# The 'params' field of the `StateVals` class is a tparams_mean object
class(costmod$params)
costmod$params
summary(costmod$params)
```

`tparams_transprobs` *Transition probabilities*

Description

Create a list containing predicted transition probabilities at discrete times. Since the transition probabilities have presumably already been predicted based on covariate values, no input data is required for simulation. The class can be instantiated from either an `array`, a `data.table`, a `data.frame`, or a `tpmatrix`. This is the object in `hesim` used to specify the transition probabilities required to simulate Markov chains with the `CohortDtstmTrans` class.

Usage

```
tparams_transprobs(object, ...)

## S3 method for class 'array'
tparams_transprobs(
  object,
  tpmatrix_id = NULL,
  times = NULL,
  grp_id = NULL,
  patient_wt = NULL,
  ...
)

## S3 method for class 'data.table'
tparams_transprobs(object, ...)

## S3 method for class 'data.frame'
tparams_transprobs(object, ...)

## S3 method for class 'tpmatrix'
tparams_transprobs(object, tpmatrix_id, ...)
```

Arguments

`object`	An object of the appropriate class.
`...`	Further arguments passed to or from other methods. Currently unused.
`tpmatrix_id`	An object of class `tpmatrix_id` (or an equivalent `data.table` with the same ID columns as returned by `tpmatrix_id()`).
`times`	An optional numeric vector of distinct times to pass to time_intervals representing time intervals indexed by the 4th dimension of the array. May either be the start or the end of intervals. This argument is not required if there is only one time interval.
`grp_id`	An optional numeric vector of integers denoting the subgroups. Must be the same length as the 3rd dimension of the array.
`patient_wt`	An optional numeric vector denoting the weight to apply to each patient within a subgroup. Must be the same length as the 3rd dimension of the array.

Details

The format of `object` depends on its class:

array Either a 3D or a 6D array is possible.

  - If a 3D array, then each slice is a square transition probability matrix. In this case `tpmatrix_id` is required and each matrix slice corresponds to the same numbered row in `tpmatrix_id`. The number of matrix slices must equal the number of rows in `tpmatrix_id`.
  - If a 6D array, then the dimensions of the array should be indexed as follows: 1st (`sample`), 2nd (`strategy_id`), 3rd (`patient_id`), 4th (`time_id`), 5th (rows of transition matrix), and 6th (columns of transition matrix). In other words, an index of `[s, k, i, t]` represents the transition matrix for the `s`th sample, `k`th treatment strategy, `i`th patient, and `t`th time interval.

data.table Must contain the following:

  - ID columns for the parameter sample (`sample`), treatment strategy (`strategy_id`), and patient (`patient_id`). If the number of time intervals is greater than 1 it must also contain the column `time_start` denoting the starting time of a time interval. A column `patient_wt` may also be used to denote the weight to apply to each patient.
  - Columns for each element of the transition probability matrix. They should be prefixed with "prob_" and ordered rowwise. For example, the following columns would be used for a 2x2 transition probability matrix: `prob_1` (1st row, 1st column), `prob_2` (1st row, 2nd column), `prob_3` (2nd row, 1st column), and `prob_4` (2nd row, 2nd column).

data.frame Same as `data.table`.

tpmatrix An object of class `tpmatrix`.

A `tparams_transprobs` object is also instantiated when creating a cohort discrete time state transition model using `define_model()`.

Value

An object of class `tparams_transprobs`, which is a list containing `value` and relevant ID attributes. The element `value` is an array of predicted transition probability matrices from the probability distribution of the underlying statistical model. Each matrix in `value` is a prediction for a `sample`, `strategy_id`, `patient_id`, and optionally `time_id` combination.

See Also

A `tparams_transprobs` object is used to store the "parameters" of the transition component of a cohort discrete time state transition model (cDTSTM). You can create such an object with `CohortDtstmTran$new()`.

`tpmatrix()` and `tpmatrix_id()` provide a convenient way to construct a `tparams_transprobs` object in a flexible way. `define_model()` is, in turn, a convenient way to construct a `tpmatrix` object using mathematical expressions; in this case, an entire cDTSTM can be instantiated from a model definition using `create_CohortDtstm.model_def()`. Detailed examples are provided in `vignette("markov-cohort")` and `vignette("markov-inhomogeneous-cohort")`

The output of a `tparams_transprobs` object is rather verbose. It can be helpful to check the output by converting it to a `data.table` (containing both the ID variables and flattened transition probability matrices) with `as.data.table.tparams_transprobs()`. Transition probabilities can also be summarized (across parameter samples) using `summary.tparams_transprobs()`.

Examples

```
hesim_dat <- hesim_data(strategies = data.frame(strategy_id = 1:2),
                        patients = data.frame(patient_id = 1:3))
input_data <- expand(hesim_dat, by = c("strategies", "patients"))

# tpmatrix objects provide a convenient way to construct
# tparams_transprobs() objects
tpmat_id <- tpmatrix_id(input_data, n_samples = 2)
p_12 <- runif(nrow(tpmat_id), .6, .7) +
  .05 * (tpmat_id$strategy_id == 2)
tpmat <- tpmatrix(
  C, p_12,
  0, 1
)
tprobs <- tparams_transprobs(tpmat, tpmat_id)
names(tprobs) # Names of list elements

# Convert to data.table in wide format
as.data.table(tprobs)

# Convert to data.table in long format
as.data.table(tprobs, long = TRUE)

# Summary where each column is a vector
summary(tprobs)
summary(tprobs, probs = c(.025, .975))

# Summary where each column is a matrix
ps <- summary(tprobs, id = tpmat_id, unflatten = TRUE)
ps
ps$mean
```

`tpmatrix` *Transition probability matrix*

Description

`tpmatrix()` both defines and evaluates a transition probability matrix in which elements are expressions. It can be used within `define_tparams()` to create a transition probability matrix or directly to create a `tparams_transprobs()` object. These are, in turn, ultimately used to create a CohortDtstmTrans object for simulating health state transitions.

Usage

```
tpmatrix(..., complement = NULL, states = NULL, prefix = "", sep = "_")
```

Arguments

`...`	Named values of expressions defining elements of the matrix. Each element of `...` should either be a vector or a 2-dimensional tabular object such as a data frame. See "Details" and the examples below.
`complement`	Either a character vector or a numeric vector denoting the transitions (i.e., the columns of the tabular object formed from `...`) that are complementary (see "Details" below). If a character vector, each element should be the name of a column in the tabular object; if a numeric vector, each element should be the index of a column in the tabular object.
`states, prefix, sep`	Arguments passed to `tpmatrix_names()` for naming the columns. If `states = NULL` (the default), then the states are named `s1`, ..., `sh` where `h` is the number of health states.

Details

A `tpmatrix` is a 2-dimensional tabular object that stores flattened square transition probability matrices in each row. Each transition probability matrix is filled rowwise. The complementary probability (equal to 1 minus the sum of the probabilities of all other elements in a row of a transition probability matrix) can be conveniently referred to as `C` or specified with the `complement` argument. There can only be one complement for each row in a transition probability matrix.

Value

Returns a `tpmatrix` object that inherits from `data.table` where each column is an element of the transition probability matrix with elements ordered rowwise.

See Also

A `tpmatrix` is useful because it provides a convenient way to construct a `tparams_transprobs` object, which is the object in `hesim` used to specify the transition probabilities required to simulate

Markov chains with the `CohortDtstmTrans` class. See the `tparams_transprobs` documentation for more details.

The `summary.tpmatrix()` method can be used to summarize a `tpmatrix` across parameter samples.

Examples

```
p_12 <- c(.7, .6)
tpmatrix(
  C, p_12,
  0, 1
)

tpmatrix(
  C, p_12,
  C, 1
)

# Pass matrix
pmat <- matrix(c(.5, .5, .3, .7), byrow = TRUE, ncol = 4)
tpmatrix(pmat)

# Pass vectors and data frames
p1 <- data.frame(
  p_12 = c(.7, .6),
  p_13 = c(.1, .2)
)

p2 <- data.frame(
  p_21 = 0,
  p_22 = c(.4, .45),
  p_23 = c(.6, .55)
)

p3 <- data.frame(
  p_31 = c(0, 0),
  p_32 = c(0, 0),
  p_33 = c(1, 1)
)

tpmatrix(
  C, p1,
  p2,
  p3
)

# Use the 'complement' argument
pmat <- data.frame(s1_s1 = 0, s1_s2 = .5, s2_s1 = .3, s2_s2 = 0)
tpmatrix(pmat, complement = c("s1_s1", "s2_s2"))
tpmatrix(pmat, complement = c(1, 4)) # Can also pass integers

# Can control column names
```

```
tpmatrix(pmat, complement = c(1, 4),
         states = c("state1", "state2"), sep = ".")
```

`tpmatrix_id` *Transition probability matrix IDs*

Description

Creates ID variables for each row returned by `tpmatrix()`. This function is most conveniently used along with `tpmatrix()` to construct a `tparams_transprobs()` object.

Usage

```
tpmatrix_id(object, n_samples)
```

Arguments

`object`	An object of class `expanded_hesim_data` returned by `expand.hesim_data()`. This dataset must be expanded by treatment strategies, patients, and optionally time intervals.
`n_samples`	The number of parameters samples used for the probabilistic sensitivity analysis (PSA).

Value

Returns a `tpmatrix_id` object that inherits from `data.table` with the same columns in `object` repeated `n_samples` times. That is, to facilitate creation of a `tparams_transprobs()` object, there is one row for each parameter sample, treatment strategy, patient, and optionally time interval.

See Also

`tpmatrix()`, `tparams_transprobs()`, `expand.hesim_data()`

Examples

```
strategies <- data.frame(strategy_id = c(1, 2))
patients <- data.frame(patient_id = seq(1, 3), age = c(65, 50, 75),
                        gender = c("Female", "Female", "Male"))
hesim_dat <- hesim_data(strategies = strategies,
                        patients = patients)
input_data <- expand(hesim_dat, by = c("strategies", "patients"))
tpmatrix_id(input_data, n_samples = 2)
```

`tpmatrix_names` *Names for elements of a transition probability matrix*

Description

Create names for all elements of a transition probability matrix given names for the health states. This is useful for flattening a transition probability matrix (rowwise) into a vector and naming the resulting vector. The name of an element of the flattened vector representing a transition from the ith state to the jth state is of the form `paste0(prefix, states[i], sep, states[j])`.

Usage

```
tpmatrix_names(states, prefix = "p_", sep = "_")
```

Arguments

`states`	A character vector of the names of health states in the transition matrix.
`prefix`	A prefix that precedes the described transitions between states used to name a transition. For example, if `prefix` = `"p_"` (and `sep` = `"_"`), then a transition between state `i` and state `j` will be of the form `"p_states[i]_states[j]"`; similarly, if `prefix` = `""`, then the same transition will be named `"states[i]_states[j]"`.
`sep`	A character string to separate the terms representing state `i` and state `j`. For instance, if `sep` = `"."`, the resulting name will be of the form `"states[i].states[j]"`.

Value

A character vector containing a name for each element of the transition probability matrix encompassing all possible transitions.

See Also

See `tpmatrix()`, which uses `tpmatrix_names()` to name the columns of the returned object.

Examples

```
tpmatrix_names(LETTERS[1:4])
tpmatrix_names(LETTERS[1:4], prefix = "")
tpmatrix_names(LETTERS[1:4], prefix = "", sep = ".")
```

Index

∗ **datasets**
  mstate3_exdata, 61
  multinom3_exdata, 62
  onc3, 64
  onc3p, 65
  psm4_exdata, 81

apply_rr, 4
as.data.table(), *115*
as.data.table.input_mats (input_mats), 57
as.data.table.tparams_transprobs, 5
as.data.table.tparams_transprobs(), *121*
as_array3, 6
as_pfs_os, 8
as_tbl2 (as_array3), 6
autoplot.stateprobs, 9
autoplot.survival, 10

beta_rng (rng_distributions), 92
bootstrap, *29*, *31*
Braces, *37*
braces, *39*

ce, 11, *107*
cea, 12, *75*
cea(), *12*, *13*, *74–77*, *99*
cea_pw (cea), 12
cea_pw(), *12*, *47*, *48*, *74*, *76*
CohortDtstm, 15, *21*, *24*, *97–99*
CohortDtstmTrans, *15*, *17*, 20, *21*, *24–26*, *68–70*, *98*, *102*, *119*, *122*, *123*
costs, *15*, *16*, 23, *49*, *51*, *78*, *79*, *98*, *99*
create_CohortDtstm, 24
create_CohortDtstm(), *17*, *35*, *38*
create_CohortDtstm.model_def(), *121*
create_CohortDtstmTrans, 25
create_CohortDtstmTrans(), *17*, *21*, *30*, *58*
create_IndivCtstmTrans, 26
create_IndivCtstmTrans(), *30*, *52*, *56*, *58*
create_input_mats(), *30*, *58*
create_params, 28
create_params(), *32*, *74*
create_params.partsurvfit(), *31*
create_PsmCurves, 30
create_PsmCurves(), *30*, *58*, *80*, *85*
create_StateVals, 32
create_StateVals(), *30*, *106*, *118*
create_StateVals.lm(), *67*
create_StateVals.stateval_tbl(), *33*, *105*
create_trans_dt, 34
custom (rng_distributions), 92

data.table, *34*, *57*, *109*, *110*, *112*, *113*
data.table::set(), *95*
define_model, 34
define_model(), *37–40*, *120*, *121*
define_rng, 37
define_rng(), *35*, *39*, *40*, *92–94*
define_tparams, 39
define_tparams(), *35*, *38*, *40*, *122*
dirichlet_rng (rng_distributions), 92
disprog, 40, *49*, *56*

eval_model (define_model), 34
eval_model(), *39*, *40*
eval_rng, *109*
eval_rng (define_rng), 37
eval_rng(), *39*, *109*
eval_tparams (define_tparams), 39
eval_tparams(), *40*
expand.grid(), *41*
expand.hesim_data, *27*, 41
expand.hesim_data(), *25*, *31*, *46*, *124*
expanded_hesim_data, *24*, *32*, *35*, *39*
expmat, 42

factor(), *95*

fast_rgengamma, 42
fixed *(*rng_distributions*)*, 92
flexsurv::GenGamma, *72*
flexsurv::Gompertz, *72*
flexsurv::Llogis, *72*
flexsurv::Survspline, *72*
flexsurv::WeibullPH, *71*
flexsurvreg, *43*
flexsurvreg_list, *31*, 43
flexsurvreg_list(), *57*
format.icer *(*icer*)*, 47
format.summary.ce *(*summary.ce*)*, 107
formula, *26*, *28*, *31*

gamma_rng *(*rng_distributions*)*, 92
get_labels, 44
get_labels(), *9, 10*, *46, 47*, *75–77*, *95, 96*, *107*
ggplot2, *74–77*

health state values, *67*
hesim::CtstmTrans, *54*
hesim_data, *32*, *34*, *41*, 45, *105*
hesim_data(), *44, 45*

icer, 47
ID, *35*
id_attributes, *116–118*
id_attributes(), *57*
incr_effect, 48
IndivCtstm, *28*, *40*, 49, *56*, *99*, *108*
IndivCtstmTrans, *26–28*, *34*, *40*, *49*, *52*, 54, *88*
IndivCtstmTrans(), *58*
input_mats, *20*, *54*, *57*, 57, *66*, *83*, *103*
input_mats(), *58*

lognormal_rng *(*rng_distributions*)*, 92

MASS::mvrnorm(), *94*
metadata, *57*
mom_beta, 59
mom_beta(), *93*
mom_gamma, 60
mom_gamma(), *93*
msm::MatrixExp(), *42*
msm::msm, *89*
mstate, *27*, *34*, *54*, *61*
mstate3_exdata, 61
mstate::msprep, *54*
multi_normal_rng *(*rng_distributions*)*, 92
multinom, *64*
multinom3_exdata, 62
multinom_list, 64

normal_rng *(*rng_distributions*)*, 92

onc3, 64, *65, 66*
onc3p, *65*, 65

params, *58*, 66, *117*
params_lm, *29*, 67, *103*, *111*
params_lm(), *111*
params_mlogit, *26*, *28, 29*, *31*, 68, *70*, *111*
params_mlogit(), *70*, *111*
params_mlogit_list, *20*, *29*, *68*, 69, *111*
params_mlogit_list(), *68*, *111*
params_surv, *26*, *28, 29*, *31*, *54*, 71, *74*, *111*
params_surv(), *111*
params_surv_list, *29*, *54*, 73, *83*, *111*
params_surv_list(), *111*
plot_ceac, 74
plot_ceaf, 75
plot_ceplane, 76
plot_evpi, 77
predict(), *26*, *28*, *31*
predict.lm(), *26*, *28*, *31*
print.data.table(), *57*
print.eval_rng *(*summary.eval_rng*)*, 109
print.input_mats *(*input_mats*)*, 57
print.params_mlogit *(*params_mlogit*)*, 68
print.params_surv *(*params_surv*)*, 71
print.tparams_mean *(*summary.tparams_mean*)*, 111
print.tparams_transprobs *(*tparams_transprobs*)*, 119
prothr, *61*
Psm, *10, 11*, *31*, 77, *85*, *97–99*, *115*
psm4_exdata, 81
PsmCurves, *30, 31*, *78*, *80*, 83, *115*
PsmCurves(), *58*

qalys, *15, 16*, *49*, *51*, *78, 79*, 87, *98, 99*
qmatrix(), *42*
qmatrix.data.frame *(*qmatrix.matrix*)*, 88
qmatrix.data.table *(*qmatrix.matrix*)*, 88
qmatrix.data.table(), *42*
qmatrix.matrix, 88

qmatrix.msm, 89
qmatrix.msm(), *42*, *88*

R6::R6Class, *15*, *20*, *49*, *54*, *77*, *83*
R6Class, *28*
rcat, 90
rdirichlet_mat, 91
rdirichlet_mat(), *93*
rng_def, *35*
rng_distributions, *38*, 92
rpwexp, 94

set_labels, 95
set_labels(), *45*
sim_costs (sim_ev), 96
sim_ev, 96
sim_qalys (sim_ev), 96
sim_stateprobs(), *102*
sim_stateprobs.survival, 100
sim_stateprobs.survival(), *98*, *114*, *115*
stateprobs, *9*, *15*, *16*, *21*, *49*, *50*, *56*, *78*, *79*, *97*, *98*, *100*, 102
stateval_tbl, *32*, *67*, 105
stateval_tbl(), *33*, *118*
StateVals, *15*, *17*, *24*, *32*, *33*, *49*, *51*, *52*, *78*, *80*, *97*, 103, *106*, *118*
StateVals$new(), *32*
stats::Exponential, *71*
stats::GammaDist, *72*
stats::Lognormal, *72*
stats::quantile(), *109*
stats::rbeta(), *93*
stats::rgamma(), *93*
stats::rlnorm(), *94*
stats::rnorm(), *94*
stats::runif(), *94*
stats::Weibull, *71*
summarize_ce(), *16*, *51*, *79*, *99*
summary.ce, 107
summary.eval_rng, 109
summary.params, 110
summary.params_lm (summary.params), 110
summary.params_mlogit (summary.params), 110
summary.params_mlogit(), *68*
summary.params_mlogit_list (summary.params), 110
summary.params_mlogit_list(), *70*
summary.params_surv (summary.params), 110
summary.params_surv_list (summary.params), 110
summary.tparams_mean, 111
summary.tparams_mean(), *118*
summary.tparams_transprobs, 112
summary.tparams_transprobs(), *121*
summary.tpmatrix, 113
summary.tpmatrix(), *123*
surv_quantile, 116
survival, *10*, *78*, *84*, *100*, 114

time_intervals, *41*, 116, *120*
tparams, *39*, *66*, 117
tparams_def, *35*
tparams_mean, *103*, *111*, *112*, 118
tparams_transprobs, *5*, *20*, *42*, *112*, *113*, 119, *122*, *123*
tparams_transprobs(), *6*, *39*, *122*, *124*
tpmatrix, *7*, *113*, *119–121*, 122
tpmatrix(), *39*, *121*, *124*
tpmatrix_id, *113*, *120*, 124
tpmatrix_id(), *121*
tpmatrix_names, 125
tpmatrix_names(), *5*, *7*, *122*
transformed parameter, *118*
try(), *29*

uniform_rng (rng_distributions), 92