Package: WARDEN (via r-universe)

July 23, 2026

Title Workflows for Health Technology Assessments in R using Discrete EveNts

Version 2.0.6

Description Toolkit to support and perform discrete event simulations with and without resource constraints in the context of health technology assessments (HTA). The package focuses on cost-effectiveness modelling and aims to be submission-ready to relevant HTA bodies in alignment with 'NICE TSD 15' <https://sheffield.ac.uk/nice-dsu/tsds/patient-level-simulation>. More details an examples can be found in the package website <https://jsanchezalv.github.io/WARDEN/>.

License GPL (>= 3)

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

BugReports https://github.com/jsanchezalv/WARDEN/issues

Suggests dplyr, ggplot2, knitr, rmarkdown, kableExtra, testthat (>= 3.0.0), survival

Imports purrr, data.table, foreach, future, doFuture, stats, utils, flexsurv, MASS, zoo, progressr, tidyr, lifecycle, Rcpp

VignetteBuilder knitr

Config/testthat/edition 3

Depends R (>= 4.1.0)

URL https://jsanchezalv.github.io/WARDEN/

LinkingTo Rcpp

NeedsCompilation yes

Author Javier Sanchez Alvarez [aut, cre], Gabriel Lemyre [ctb], Valerie Aponte Ribero [ctb]

Maintainer Javier Sanchez Alvarez <javiersanchezeco@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-07-23 10:40:02 UTC

RemoteUrl https://github.com/cran/WARDEN

RemoteRef HEAD

RemoteSha 5959b3a8e7ab49a0dce11fe21b525ffb619fcce7

Contents

`add_item` *Define or append model inputs*

Description

Build a single `{}` expression that defines inputs for a simulation.

- Named args in `...` become assignments (`name <- expr`), e.g., `add_item(a=5)`
- Unnamed args are inserted raw/unevaluated. If an unnamed arg is a `{}` block, its statements are spliced (flattened). e.g., `add_item(pick_val_v(...))`
- Works with both `|>` (native pipe) and `%>%` (magrittr): the LHS is passed to `.data` and becomes the starting block.
- input argument can be used to handle alternative `add_item2` method, e.g. `add_item(input = {a <- 5})`

Usage

```
add_item(.data = NULL, ..., input)
```

Arguments

`.data`	Optional: an existing `{}` block (or list of expressions) to start from. Receives the LHS value when using `|>` or `%>%`.
`...`	Unevaluated arguments. Named -> `name <- expr`; unnamed -> raw expr.
`input`	Optional unevaluated expression or `{}` block to splice in.

Details

Pipe chaining: when using `|>` or `%>%`, the LHS is matched to `.data` and used as the starting block. This works correctly when the LHS is a variable, an `add_item()` call, or an `input_block()` call. Any other call (e.g. a custom builder function) is captured unevaluated as a statement rather than used as the starting block; assign the result to a variable first: `blk <- my_builder(); add_item(.data = blk, x = 1)`.

Value

A single `{}` call (language object) ready for `load_inputs()`.

Examples

```
add_item(input = {fl.idfs <-  0})
add_item(input = {
 util_idfs <- if(psa_bool){rnorm(1,0.8,0.2)} else{0.8}
 util.mbc <- 0.6
 cost_idfs <- 2500})
common_inputs <- add_item(input = {
pick_val_v(
  base      = l_statics[["base"]],
```

```
  psa       = pick_psa(
    l_statics[["function"]],
    l_statics[["n"]],
    l_statics[["a"]],
    l_statics[["b"]]
  ),
  sens      = l_statics[[sens_name_used]],
  psa_ind   = psa_bool,
  sens_ind  = sensitivity_bool,
  indicator = indicators_statics,
  names_out = l_statics[["parameter_name"]],
  deploy_env = TRUE #Note this option must be active if using it at add_item2
)
}
)
```

`add_item2` *Define parameters that may be used in model calculations (uses expressions)*

Description

Define parameters that may be used in model calculations (uses expressions)

Usage

```
add_item2(.data = NULL, input)
```

Arguments

`.data`	Existing data
`input`	Items to define for the simulation as an expression (i.e., using)

Details

DEPRECATED (old description): The functions to add/modify events/inputs use named vectors or lists. If chaining together add_item2, it will just append the expressions together in the order established.

If using `pick_val_v`, note it should be used with the `deploy_env = TRUE` argument so that add_item2 process it correctly.

Value

A substituted expression to be evaluated by engine add_reactevt *Define the modifications to other events, costs, utilities, or other items affected by the occurrence of the event*

Description

Define the modifications to other events, costs, utilities, or other items affected by the occurrence of the event

Usage

```
add_reactevt(.data = NULL, name_evt, input)
```

Arguments

`.data`	Existing data for event reactions
`name_evt`	Name of the event for which reactions are defined.
`input`	Expressions that define what happens at the event, using functions as defined in the Details section

Details

There are a series of objects that can be used in this context to help define the event reactions.

The following functions may be used to define event reactions within this `add_reactevt()` function: `modify_item()` | Adds & Modifies items/flags/variables for future events (does not consider sequential) `modify_item_seq()` | Adds & Modifies items/flags/variables for future events in a sequential manner `new_event()` | Adds events to the vector of events for that patient `modify_event()` | Modifies existing events by changing their time

Apart from the items defined with add_item(), we can also use standard variables that are always defined within the simulation: `curtime` | Current event time (numeric) `prevtime` | Time of the previous event (numeric) `cur_evtlist` | Named vector of events that is yet to happen for that patient (named numeric vector) `evt` | Current event being processed (character) `i` | Patient being iterated (character) `simulation` | Simulation being iterated (numeric)

The model will run until `curtime` is set to `Inf`, so the event that terminates the model should modify `curtime` and set it to `Inf`.

The user can use `extract_from_reactions` function on the output to obtain a data.frame with all the relationships defined in the reactions in the model.

Value

A named list with the event name, and inside it the substituted expression saved for later evaluation

Examples

```
add_reactevt(name_evt = "start",input = {})
add_reactevt(name_evt = "idfs",input = {modify_item(list("fl.idfs"= 0))})
```

`add_tte` *Define events and the initial event time*

Description

Define events and the initial event time

Usage

```
add_tte(.data = NULL, arm, evts, other_inp = NULL, input)
```

Arguments

`.data`	Existing data for initial event times
`arm`	The intervention for which the events and initial event times are defined
`evts`	A vector of the names of the events
`other_inp`	A vector of other input variables that should be saved during the simulation
`input`	The definition of initial event times for the events listed in the evts argument

Details

Events need to be separately defined for each intervention.

For each event that is defined in this list, the user needs to add a reaction to the event using the `add_reactevt()` function which will determine what calculations will happen at an event.

Value

A list of initial events and event times

Examples

```
add_tte(arm="int",evts = c("start","ttot","idfs","os"),
input={
start <- 0
idfs <- draw_tte(1,'lnorm',coef1=2, coef2=0.5)
ttot <- min(draw_tte(1,'lnorm',coef1=1, coef2=4),idfs)
os <- draw_tte(1,'lnorm',coef1=0.8, coef2=0.2)
})
```

`adj_val` *Adjusted Value Calculation*

Description

This function calculates an adjusted value over a time interval with optional discounting. This is useful for instances when adding cycles may not be desirable, so one can perform "cycle-like" calculations without needing cycles, offering performance speeds. See the vignette on avoiding cycles for an example in a model.

Usage

```
adj_val(
  curtime,
  nexttime,
  by,
  expression,
  discount = NULL,
  vectorized_f = FALSE
)
```

Arguments

`curtime`	Numeric. The current time point.
`nexttime`	Numeric. The next time point. Must be greater than or equal to `curtime`.
`by`	Numeric. The step size for evaluation within the interval.
`expression`	An expression evaluated at each step. Use `.time` as the variable within the expression.
`discount`	Numeric or NULL. The discount rate to apply, or NULL for no discounting.
`vectorized_f`	boolean, FALSE by default. If TRUE, evaluates the expression once using `.time` as a vector. If FALSE, it repeatedly evaluates the expression with time as a single value (slower).

Details

The user can use the `.time` variable to select the corresponding time of the sequence being evaluated. For example, in `curtime = 0, nexttime = 4, by = 1`, `.time` would correspond to `0, 1, 2, 3`. If using `nexttime = 4.2`, `0, 1, 2, 3, 4`

Value

Numeric. The calculated adjusted value.

Note

In resource-constrained models where event timing is unpredictable (e.g., patients waiting in resource queues), use `accum_backwards = TRUE` in `run_sim()` and compute `adj_val(prevtime, curtime, ...)` at each event reaction. This ensures both time boundaries are known and the age/time adjustment is exact, avoiding prospective errors when a resource event fires earlier than expected.

Examples

```
# Define a function or vector to evaluate
bs_age <- 1
vec <- 1:8/10

# Calculate adjusted value without discounting
adj_val(0, 4, by = 1, expression = vec[floor(.time + bs_age)])
adj_val(0, 4, by = 1, expression = .time * 1.1)
#same result since .time * 1.1 can be vectorized w.r.t time
adj_val(0, 4, by = 1, expression = .time * 1.1, vectorized_f = TRUE)

# Calculate adjusted value with discounting
adj_val(0, 4, by = 1, expression = vec[floor(.time + bs_age)], discount = 0.03)
```

`ast_as_list` *Transform a substituted expression to its Abstract Syntax Tree (AST) as a list*

Description

Transform a substituted expression to its Abstract Syntax Tree (AST) as a list

Usage

```
ast_as_list(ee)
```

Arguments

ee	Substituted expression

Value

Nested list with the Abstract Syntax Tree (AST)

Examples

```
expr <- substitute({

a <- sum(5+7)

modify_item(list(afsa=ifelse(TRUE,"asda",NULL)))

modify_item_seq(list(

  o_other_q_gold1 = if(gold == 1) { utility } else { 0 },

  o_other_q_gold2 = if(gold == 2) { utility } else { 0 },

  o_other_q_gold3 = if(gold == 3) { utility } else { 0 },

  o_other_q_gold4 = if(gold == 4) { utility } else { 0 },

  o_other_q_on_dup = if(on_dup) { utility } else { 0 }

))

if(a==1){
  modify_item(list(a=list(6+b)))

  modify_event(list(e_exn = curtime + 14 / days_in_year + qexp(rnd_exn, r_exn)))
} else{
  modify_event(list(e_exn = curtime + 14 / days_in_year + qexp(rnd_exn, r_exn)))
  if(a>6){
    modify_item(list(a=8))
  }

}


if (sel_resp_incl == 1 & on_dup == 1) {

  modify_event(list(e_response = curtime, z = 6))

}

})


out <- ast_as_list(expr)
```

`ceac_des` *Calculate the cost-effectiveness acceptability curve (CEAC) for a DES model with a PSA result*

Description

Calculate the cost-effectiveness acceptability curve (CEAC) for a DES model with a PSA result

Usage

```
ceac_des(wtp, results, interventions = NULL, sensitivity_used = 1)
```

Arguments

`wtp`	Vector of length >=1 with the willingness to pay
`results`	The list object returned by `run_sim()`
`interventions`	A character vector with the names of the interventions to be used for the analysis
`sensitivity_used`	Integer signaling which sensitivity analysis to use

Value

A data frame with the CEAC results

Examples

```
res <- list(list(list(sensitivity_name = "", arm_list = c("int", "noint"
), total_lys = c(int = 9.04687362556945, noint = 9.04687362556945
), total_qalys = c(int = 6.20743830697466, noint = 6.18115138126336
), total_costs = c(int = 49921.6357486899, noint = 41225.2544659378
), total_lys_undisc = c(int = 10.8986618377039, noint = 10.8986618377039
), total_qalys_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), total_costs_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), c_default = c(int = 49921.6357486899, noint = 41225.2544659378
), c_default_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), q_default = c(int = 6.20743830697466, noint = 6.18115138126336
), q_default_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), merged_df = list(simulation = 1L, sensitivity = 1L))))

ceac_des(seq(from=10000,to=500000,by=10000),res)
```

`cond_dirichlet` *Calculate conditional dirichlet values*

Description

Calculate conditional dirichlet values

Usage

```
cond_dirichlet(alpha, i, xi, full_output = FALSE)
```

Arguments

`alpha`	mean vector
`i`	index of the known parameter (1-based index)
`xi`	known value of the i-th parameter (should be >0)
`full_output`	boolean indicating whether to return the full list of parameters

Details

Function to compute conditional dirichlet values

Value

List of length 2, one with new mu and other with covariance parameters

Examples

```
alpha <- c(2, 3, 4)
i <- 2  # Index of the known parameter
xi <- 0.5  # Known value of the second parameter

# Compute the conditional alpha parameters with full output
cond_dirichlet(alpha, i, xi, full_output = TRUE)
```

---

cond_mvn *Calculate conditional multivariate normal values*

Description

Calculate conditional multivariate normal values

Usage

```
cond_mvn(mu, Sigma, i, xi, full_output = FALSE)
```

Arguments

`mu`	mean vector
`Sigma`	covariance matrix
`i`	index of the known parameter (1-based index)
`xi`	known value of the i-th parameter
`full_output`	boolean indicating whether to return the full list of parameters

Details

Function to compute conditional multivariate normal values

Value

List of length 2, one with new mu and other with covariance parameters

Examples

```
mu <- c(1, 2, 3)
Sigma <- matrix(c(0.2, 0.05, 0.1,
                  0.05, 0.3, 0.05,
                  0.1, 0.05, 0.4), nrow = 3)

i <- 1:2  # Index of the known parameter
xi <- c(1.2,2.3)  # Known value of the first parameter

cond_mvn(mu, Sigma, i, xi,full_output = TRUE)
```

---

create_indicators *Creates a vector of indicators (0 and 1) for sensitivity/DSA analysis*

Description

Creates a vector of indicators (0 and 1) for sensitivity/DSA analysis

Usage

```
create_indicators(sens, n_sensitivity, elem, n_elem_before = 0)
```

Arguments

sens	current analysis iterator
n_sensitivity	total number of analyses to be run
elem	vector of 0s and 1s of elements to iterate through (1 = parameter is to be included in scenario/DSA)
n_elem_before	Sum of 1s (# of parameters to be included in scenario/DSA) that go before elem

Details

n_elem_before is to be used when several indicators want to be used (e.g., for patient level and common level inputs) while facilitating readibility of the code

Value

Numeric vector composed of 0 and 1, where value 1 will be used by `pick_val_v` to pick the corresponding index in its `sens` argument

Examples

```
create_indicators(10,20,c(1,1,1,1))
create_indicators(7,20,c(1,0,0,1,1,1,0,0,1,1),2)
```

disc_cycle *Cycle discounting*

Description

Cycle discounting

Usage

```
disc_cycle(
  lcldr = 0.035,
  lclprvtime = 0,
  cyclelength,
  lclcurtime,
  lclval,
  starttime = 0
)
```

Arguments

lcldr	The discount rate
lclprvtime	The time of the previous event in the simulation
cyclelength	The cycle length
lclcurtime	The time of the current event in the simulation
lclval	The value to be discounted
starttime	The start time for accrual of cycle costs (if not 0)

Details

Note this function counts both extremes of the interval, so the example below would consider 25 cycles, while disc_cycle_v leave the right interval open

Value

Double based on cycle discounting

Examples

```
disc_cycle(lcldr=0.035, lclprvtime=0, cyclelength=1/12, lclcurtime=2, lclval=500,starttime=0)
```

`disc_cycle_v` *Cycle discounting for vectors*

Description

Cycle discounting for vectors

Usage

```
disc_cycle_v(
  lcldr,
  lclprvtime,
  cyclelength,
  lclcurtime,
  lclval,
  starttime,
  max_cycles = NULL
)
```

Arguments

`lcldr`	The discount rate
`lclprvtime`	The time of the previous event in the simulation
`cyclelength`	The cycle length
`lclcurtime`	The time of the current event in the simulation
`lclval`	The value to be discounted
`starttime`	The start time for accrual of cycle costs (if not 0)
`max_cycles`	The maximum number of cycles

Details

This function per cycle discounting, i.e., considers that the cost/qaly is accrued per cycles, and performs it automatically without needing to create new events. It can accommodate changes in cycle length/value/starttime (e.g., in the case of induction and maintenance doses) within the same item.

Value

Double vector based on cycle discounting

Examples

```
disc_cycle_v(lcldr=0.03, lclprvtime=0, cyclelength=1/12, lclcurtime=2, lclval=500,starttime=0)
disc_cycle_v(
 lcldr=0.000001,
 lclprvtime=0,
 cyclelength=1/12,
 lclcurtime=2,
 lclval=500,
 starttime=0,
 max_cycles = 4)

#Here we have a change in cycle length, max number of cylces and starttime at time 2
 #(e.g., induction to maintenance)
#In the model, one would do this by redifining cycle_l, max_cycles and starttime
 #of the corresponding item at a given event time.
disc_cycle_v(lcldr=0,
 lclprvtime=c(0,1,2,2.5),
 cyclelength=c(1/12, 1/12,1/2,1/2),
 lclcurtime=c(1,2,2.5,4), lclval=c(500,500,500,500),
 starttime=c(0,0,2,2), max_cycles = c(24,24,2,2)
  )
```

---

disc_instant *Calculate instantaneous discounted costs or qalys*

Description

Calculate instantaneous discounted costs or qalys

Usage

```
disc_instant(lcldr = 0.035, lclcurtime, lclval)
```

Arguments

lcldr	The discount rate
lclcurtime	The time of the current event in the simulation
lclval	The value to be discounted

Value

Double based on discrete time discounting

Examples

```
disc_instant(lcldr=0.035, lclcurtime=3, lclval=2500)
```

disc_instant_v — *Calculate instantaneous discounted costs or qalys for vectors*

Description

Calculate instantaneous discounted costs or qalys for vectors

Usage

```
disc_instant_v(lcldr, lclcurtime, lclval)
```

Arguments

`lcldr`	The discount rate
`lclcurtime`	The time of the current event in the simulation
`lclval`	The value to be discounted

Value

Double based on discrete time discounting

Examples

```
disc_instant_v(lcldr=0.035, lclcurtime=3, lclval=2500)
```

disc_ongoing — *Calculate discounted costs and qalys between events*

Description

Calculate discounted costs and qalys between events

Usage

```
disc_ongoing(lcldr = 0.035, lclprvtime, lclcurtime, lclval)
```

Arguments

`lcldr`	The discount rate
`lclprvtime`	The time of the previous event in the simulation
`lclcurtime`	The time of the current event in the simulation
`lclval`	The value to be discounted

Value

Double based on continuous time discounting

Examples

```
disc_ongoing(lcldr=0.035,lclprvtime=0.5, lclcurtime=3, lclval=2500)
```

---

disc_ongoing_v   *Calculate discounted costs and qalys between events for vectors*

Description

Calculate discounted costs and qalys between events for vectors

Usage

```
disc_ongoing_v(lcldr, lclprvtime, lclcurtime, lclval)
```

Arguments

`lcldr`	The discount rate
`lclprvtime`	The time of the previous event in the simulation
`lclcurtime`	The time of the current event in the simulation
`lclval`	The value to be discounted

Value

Double based on continuous time discounting

Examples

```
disc_ongoing_v(lcldr=0.035,lclprvtime=0.5, lclcurtime=3, lclval=2500)
```

`discrete_resource_clone` *Clone independent discrete resources*

Description

Clone independent discrete resources

Usage

```
discrete_resource_clone(x, n = 1)
```

Arguments

`x`	discrete resource created with resource_discrete()
`n`	Number of independent clones to be generated

Value

List of independent clones of discrete resource envs (even for n = 1)

`draw_categorical` *Draw a categorical index from probabilities*

Description

Given a uniform random draw and a vector of probabilities, returns the 1-based integer index of the selected category.

Usage

```
draw_categorical(luck, probs)
```

Arguments

`luck`	Numeric between 0 and 1 (inclusive). A uniform random draw.
`probs`	Numeric vector of probabilities. Will be normalized to sum to 1 internally. All values must be >= 0.

Value

Integer: 1-based index of the selected category.

draw_tte *Draw a time to event from a list of parametric survival functions*

Description

Draw a time to event from a list of parametric survival functions

Usage

```
draw_tte(
  n_chosen,
  dist,
  coef1 = NULL,
  coef2 = NULL,
  coef3 = NULL,
  ...,
  beta_tx = 1,
  seed = NULL
)
```

Arguments

n_chosen	The number of observations to be drawn
dist	The distribution; takes values 'lnorm','norm','mvnorm','weibullPH','weibull','llogis','gompertz','gengar
coef1	First coefficient of the distribution, defined as in the coef() output on a flexsurvreg object (rate in "rpoisgamma")
coef2	Second coefficient of the distribution, defined as in the coef() output on a flexsurvreg object (theta in "rpoisgamma")
coef3	Third coefficient of the distribution, defined as in the coef() output on a flexsurvreg object (not used in "rpoisgamma")
...	Additional arguments to be used by the specific distribution (e.g., return_ind_rate if dist = "poisgamma")
beta_tx	Parameter in natural scale applied in addition to the scale/rate coefficient -e.g., a HR if used in an exponential- (not used in "rpoisgamma" nor "beta")
seed	An integer which will be used to set the seed for this draw.

Details

Other arguments relevant to each function can be called directly

Value

A vector of time to event estimates from the given parameters

Examples

```
draw_tte(n_chosen=1,dist='exp',coef1=1,beta_tx=1)
draw_tte(n_chosen=10,"poisgamma",coef1=1,coef2=1,obs_time=1,return_ind_rate=FALSE)
```

`evpi_des` *Calculate the Expected Value of Perfect Information (EVPI) for a DES model with a PSA result*

Description

Calculate the Expected Value of Perfect Information (EVPI) for a DES model with a PSA result

Usage

```
evpi_des(wtp, results, interventions = NULL, sensitivity_used = 1)
```

Arguments

`wtp`	Vector of length >=1 with the willingness to pay
`results`	The list object returned by `run_sim()`
`interventions`	A character vector with the names of the interventions to be used for the analysis
`sensitivity_used`	Integer signaling which sensitivity analysis to use

Value

A data frame with the EVPI results

Examples

```
res <- list(list(list(sensitivity_name = "", arm_list = c("int", "noint"
), total_lys = c(int = 9.04687362556945, noint = 9.04687362556945
), total_qalys = c(int = 6.20743830697466, noint = 6.18115138126336
), total_costs = c(int = 49921.6357486899, noint = 41225.2544659378
), total_lys_undisc = c(int = 10.8986618377039, noint = 10.8986618377039
), total_qalys_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), total_costs_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), c_default = c(int = 49921.6357486899, noint = 41225.2544659378
), c_default_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), q_default = c(int = 6.20743830697466, noint = 6.18115138126336
), q_default_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), merged_df = list(simulation = 1L, sensitivity = 1L))))

evpi_des(seq(from=10000,to=500000,by=10000),res)
```

extract_elements_from_list

Extracts items and events by looking into assignments, modify_event and new_event

Description

Extracts items and events by looking into assignments, modify_event and new_event

Usage

```
extract_elements_from_list(node, conditional_flag = FALSE)
```

Arguments

node	Relevant node within the nested AST list
conditional_flag	Boolean whether the statement is contained within a conditional statement

Value

A data.frame with the relevant item/event, the event where it's assigned, and whether it's contained within a conditional statement

Examples

```
expr <- substitute({

  a <- sum(5+7)

  ggplot()

  data.frame(x=1,b=2)

  list(b=5)

  a <- list(s=7)


  j <- 6
  if(TRUE){modify_event(list(j=5))}

  l <- 9

  afsa=ifelse(TRUE,"asda",NULL)


  o_exn = o_exn + 1
```

```
a = NULL

b = if(a){"CZ"}else{"AW"}

rnd_prob_exn_sev = runif(1)

exn_sev = rnd_prob_exn_sev <= p_sev

o_exn_mod = o_exn_mod + if(exn_sev) { 0 } else { 1 }

o_exn_sev = o_exn_sev + if(exn_sev) { 1 } else { 0 }

o_rec_time_without_exn = (o_exn == 0) * 1

o_rec_time_without_exn_sev = (o_exn_sev == 0) * 1

o_c_exn = if(exn_sev) { c_sev } else { c_mod }

o_other_c_exn_mod = if(exn_sev) { 0 } else { c_mod }

o_other_c_exn_sev = if(exn_sev) { c_sev } else { 0 }

o_qloss_exn = -if(exn_sev) { q_sev } else { q_mod }

o_other_qloss_exn_mod = -if(exn_sev) { 0 } else { q_mod }

o_other_qloss_exn_sev = -if(exn_sev) { q_sev } else { 0 }

o_qloss_cg_exn = -if(exn_sev) { q_cg_sev } else { q_cg_mod }

o_other_qloss_cg_exn_mod = -if(exn_sev) { 0 } else { q_cg_mod }

o_other_qloss_cg_exn_sev = -if(exn_sev) { q_cg_sev } else { 0 }

o_q = utility

o_other_q_gold1 = if(gold == 1) { utility } else { 0 }

o_other_q_gold2 = if(gold == 2) { utility } else { 0 }

o_other_q_gold3 = if(gold == 3) { utility } else { 0 }

o_other_q_gold4 = if(gold == 4) { utility } else { 0 }

o_other_q_on_dup = if(on_dup) { utility } else { 0 }

n_exn = n_exn + 1

n_exn_mod = n_exn_mod + (1 - exn_sev)

n_exn_sev = n_exn_sev + exn_sev

u_adj_exn_lt = u_adj_exn_lt + if(exn_sev) { u_adj_sev_lt } else { u_adj_mod_lt }
```

```
    utility = u_gold - u_adj_exn_lt - u_mace_lt

    o_rec_utility = utility

    rnd_exn = runif(1)


    if(a==1){
      a=list(6+b)

      modify_event(list(e_exn = curtime + 14 / days_in_year + qexp(rnd_exn, r_exn)))
    } else{
      modify_event(list(e_exn = curtime + 14 / days_in_year + qexp(rnd_exn, r_exn)))
      if(a>6){
        a=8
      }

    }


    if (sel_resp_incl == 1 & on_dup == 1) {

      modify_event(list(e_response = curtime, z = 6))

    }

  })


  out <- ast_as_list(expr)

  results <- extract_elements_from_list(out)
```

---

extract_from_reactions *Extract all items and events and their interactions from the event reactions list*

Description

Extract all items and events and their interactions from the event reactions list

Usage

```
extract_from_reactions(reactions)
```

Arguments

`reactions`	list generated through `add_reactevt`

Value

A data.frame with the relevant item/event, the event where it's assigned, and whether it's contained within a conditional statement

Examples

```
evt_react_list2 <-
  add_reactevt(name_evt = "sick",
               input = {modify_item(list(a=1+5/3))
                 assign("W", 5 + 3 / 6 )
                 x[5] <- 18
                 for(i in 1:5){
                   assign(paste0("x_",i),5+3)
                 }
                 if(j == TRUE){
                   y[["w"]] <- 612-31+3
                 }#'
                 q_default <- 0
                 c_default <- 0
                 curtime   <- Inf
                 d <- c <- k <- 67
               })

 extract_from_reactions(evt_react_list2)
```

`extract_psa_result` *Extract PSA results from a treatment*

Description

Extract PSA results from a treatment

Usage

```
extract_psa_result(x, element)
```

Arguments

`x`	The output_sim data frame from the list object returned by `run_sim()`
`element`	Variable for which PSA results are being extracted (single string)

Value

A dataframe with PSA results from the specified intervention

Examples

```
res <- list(list(list(sensitivity_name = "", arm_list = c("int", "noint"
), total_lys = c(int = 9.04687362556945, noint = 9.04687362556945
), total_qalys = c(int = 6.20743830697466, noint = 6.18115138126336
), total_costs = c(int = 49921.6357486899, noint = 41225.2544659378
), total_lys_undisc = c(int = 10.8986618377039, noint = 10.8986618377039
), total_qalys_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), total_costs_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), c_default = c(int = 49921.6357486899, noint = 41225.2544659378
), c_default_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), q_default = c(int = 6.20743830697466, noint = 6.18115138126336
), q_default_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), merged_df = list(simulation = 1L, sensitivity = 1L))))


extract_psa_result(res[[1]],"total_costs")
```

---

get_event *Get a specific event time*

Description

Get a specific event time

Usage

```
get_event(event_name, ptr, patient_id)
```

Arguments

event_name	Character string, the name of the event.
ptr	The event queue pointer. Defaults to `cur_evtlist`.
patient_id	The patient ID. Defaults to `i`.

Value

Numeric, time of event for patient

has_event — *Check if a patient has a specific event in the queue*

Description

Check if a patient has a specific event in the queue

Usage

```
has_event(event_name, ptr, patient_id, exclude_inf = FALSE)
```

Arguments

`event_name`	Character string, the name of the event.
`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`patient_id`	The patient ID. Defaults to `i`.
`exclude_inf`	Logical, whether to exclude events with Inf time. Default is FALSE.

Value

Logical, TRUE if the event exists for the patient (optionally excluding Inf), FALSE otherwise.

input_block — *Build an input block for automatic parameter selection*

Description

Creates an unevaluated `{}` expression that calls `pick_val_v()` with the correct arguments for base case, PSA, DSA, and scenario analyses. The expression is meant to be passed directly to `run_sim()` or `run_sim_parallel()` as a `*_inputs` argument.

Usage

```
input_block(
  .data = NULL,
  base,
  psa,
  sens,
  names_out,
  psa_indicators = NULL,
  sens_indicators = NULL,
  indicator_sens_binary = FALSE,
  dsa_names = NULL,
  distributions = NULL,
  covariances = NULL
)
```

Arguments

- `.data` — Optional existing `{}` block to prepend (for pipe chaining).
- `base` — A list of base case values, one entry per parameter.
- `psa` — An unevaluated expression producing PSA draws (e.g. `pick_psa(...)`), evaluated at runtime per simulation.
- `sens` — The sensitivity data object (e.g. `l_inputs`). At runtime the engine indexes it as `sens[[sens_name_used]]` to pick the active column.
- `names_out` — Character vector or list of strings of output parameter names.
- `psa_indicators` — List of 0/1 indicators controlling which parameters draw from PSA. `NULL` means all parameters draw from PSA.
- `sens_indicators` — List of indicators, one entry per parameter (may be a scalar or a vector for vector-valued parameters). In binary mode: 0 = inactive, 1 = active. In grouped mode: 0 = inactive, same integer = varied together. `NULL` means all parameters are active. Only in grouped mode can subparameters (e.g., elements of a vector parameter) be varied at different steps of a DSA.
- `indicator_sens_binary` — Logical. `TRUE` = binary (0/1) mode; each active parameter is varied independently via `create_indicators()`. `FALSE` (default) = grouped integer mode when `sens_indicators` and `distributions` are supplied; otherwise falls back to binary mode.
- `dsa_names` — Character vector of `sensitivity_names` values that are DSA directions (e.g. `c("DSA_min","DSA_max")`). Those names iterate through active parameters/groups one at a time. All other names passed to `sensitivty_names` in `run_sim()` are treated as scenarios. `NULL` (the default) means all sensitivity names are scenarios.
- `distributions` — List of distribution names (e.g. `"rnorm"`, `"mvrnorm"`, `"rdirichlet"`), required in grouped mode to determine which parameters are vectors and require conditional distributions.
- `covariances` — List of covariance matrices or scalars, required for `"mvrnorm"` / `"rdirichlet"` distributions.

Details

Supports two modes:

- **Binary** (`indicator_sens_binary = TRUE`): `sens_indicators` are 0/1 per parameter. Each non-zero parameter is varied independently via `create_indicators()`.
- **Grouped** (`indicator_sens_binary = FALSE`, the default when `sens_indicators` and `distributions` are supplied): `sens_indicators` are integer group labels. Parameters sharing the same integer are varied together; 0 means the parameter is never varied.

DSA vs scenario analyses:

Supply `dsa_names` with the subset of `sensitivity_names` that correspond to DSA directions (e.g. `c("DSA_min","DSA_max")`). Each DSA name runs one iteration per active parameter/group; other names (scenarios) apply all active parameters simultaneously in a single iteration.

When `dsa_names = NULL` (the default) every sensitivity name is treated as a scenario – all active parameters take their scenario value at once, one iteration per `sensitivity_names` entry. This is the correct default when there is no DSA.

`sens_indicators` **and zeros:**

Any parameter whose `sens_indicators` entry is `0` (or all-zero for vector-valued parameters) is **never varied**: it always uses the base or PSA value regardless of whether we are running a DSA or a scenario. This applies to both modes.

Value

A `{}` language object (same type as `add_item()`) with a `warden_block_meta` attribute used by the engine to build the iteration schedule.

See Also

`pick_val_v()`, `add_item()`, `run_sim()`

Examples

```
l_inputs <- list(
  parameter_name = list("util.sick", "util.sicker"),
  base_value     = list(0.8, 0.5),
  PSA_dist       = list("rnorm", "rbeta_mse"),
  a              = list(0.8, 0.5),
  b              = list(0.04, 0.025),
  n              = list(1, 1),
  DSA_min        = list(0.6, 0.3),
  DSA_max        = list(0.9, 0.7),
  psa_indicators = list(1, 1)
)

blk <- input_block(
  base                  = l_inputs[["base_value"]],
  psa                   = pick_psa(l_inputs[["PSA_dist"]], l_inputs[["n"]],
                                   l_inputs[["a"]], l_inputs[["b"]]),
  sens                  = l_inputs,
  names_out             = l_inputs[["parameter_name"]],
  psa_indicators        = l_inputs[["psa_indicators"]],
  indicator_sens_binary = TRUE,
  dsa_names             = c("DSA_min", "DSA_max")
)
stopifnot(is.call(blk))
stopifnot(!is.null(attr(blk, "warden_block_meta")))
```

luck_adj *Perform luck adjustment*

Description

Perform luck adjustment

Usage

```
luck_adj(prevsurv, cursurv, luck, condq = TRUE)
```

Arguments

`prevsurv`	Value of the previous survival
`cursurv`	Value of the current survival
`luck`	Luck used to be adjusted (number between 0 and 1)
`condq`	Conditional quantile approach or standard approach

Details

This function performs the luck adjustment automatically for the user, returning the adjusted luck number. Luck is interpreted in the same fashion as is standard in R (higher luck, higher time to event).

Note that if TTE is predicted using a conditional quantile function (e.g., conditional gompertz, conditional quantile weibull...) `prevsurv` and `cursurv` are the unconditional survival using the "previous" parametrization but at the previous time for `presurv` and at the current time for `cursurv`. For other distributions, `presurv` is the survival up to current time using the previous parametrization, and `cursurv` is the survival up to current time using the current parametrization.

Note that the advantage of the conditional quantile function is that it does not need the new parametrization to update the luck, which makes this approach computationally more efficient. This function can also work with vectors, which could allow to update multiple lucks in a single approach, and it can preserve names

Value

Adjusted luck number between 0 and 1

Examples

```
luck_adj(prevsurv = 0.8,
 cursurv = 0.7,
 luck = 0.5,
 condq = TRUE)

luck_adj(prevsurv = c(1,0.8,0.7),
 cursurv = c(0.7,0.6,0.5),
 luck = setNames(c(0.5,0.6,0.7),c("A","B","C")),
```

```
  condq = TRUE)

luck_adj(prevsurv = 0.8,
  cursurv = 0.7,
  luck = 0.5,
  condq = FALSE) #different results

#Unconditional approach, timepoint of change is 25,
# parameter goes from 0.02 at time 10 to 0.025 to 0.015 at time 25,
#  starting luck is 0.37
new_luck <- luck_adj(prevsurv = 1 - pweibull(q=10,3,1/0.02),
  cursurv = 1 - pweibull(q=10,3,1/0.025),
  luck = 0.37,
  condq = FALSE) #time 10 change

new_luck <- luck_adj(prevsurv = 1 - pweibull(q=25,3,1/0.025),
  cursurv = 1 - pweibull(q=25,3,1/0.015),
  luck = new_luck,
  condq = FALSE) #time 25 change

qweibull(new_luck, 3, 1/0.015) #final TTE

#Conditional quantile approach
new_luck <- luck_adj(prevsurv = 1-pweibull(q=0,3,1/0.02),
                      cursurv = 1- pweibull(q=10,3,1/0.02),
                      luck = 0.37,
                   condq = TRUE) #time 10 change, previous time is 0 so prevsurv will be 1

new_luck <- luck_adj(prevsurv = 1-pweibull(q=10,3,1/0.025),
                      cursurv = 1- pweibull(q=25,3,1/0.025),
                      luck = new_luck,
                      condq = TRUE) #time 25 change

qcond_weibull(rnd = new_luck,
                     shape = 3,
                     scale = 1/0.015,
                     lower_bound = 25) + 25 #final TTE
```

`modify_event` *Modify or add events for a patient*

Description

Modifies existing event times, or adds new events if `create_if_missing` is TRUE.

Usage

```
modify_event(events, create_if_missing = TRUE, ptr, patient_id)
```

Arguments

`events`	A named numeric vector with event names and new event times. It can also handle lists instead of named vectors (at a small computational cost).
`create_if_missing`	Logical, whether to create events if they do not exist.
`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`patient_id`	The patient ID. Defaults to `i`.

Details

The functions to add/modify events/inputs use named vectors or lists. Whenever several inputs/events are added or modified, it's recommended to group them within one function, as it reduces the computation cost. So rather than use two `modify_event` with a list of one element, it's better to group them into a single `modify_event` with a list of two elements.

This function does not evaluate sequentially.

While multiple events can be added, they must be named differently. If the same event is added multiple times at once, only the last occurrence will be kept (only one event per event type in the queue of events yet to occur). If an event occurs, then a new one with the same name can be set.

This function is intended to be used only within the `add_reactevt` function in its `input` parameter and should not be run elsewhere or it will return an error.

Value

NULL (invisible). Modifies the queue in-place.

Examples

```
add_reactevt(name_evt = "idfs",input = {modify_event(c("os"=5))})
```

---

modify_item — *Modify the value of existing items*

Description

Modify the value of existing items

Usage

```
modify_item(list_item)
```

Arguments

`list_item`	A list of items and their values or expressions

Details

DEPRECATED (old description): The functions to add/modify events/inputs use lists. Whenever several inputs/events are added or modified, it's recommended to group them within one function, as it reduces the computation cost. So rather than use two `modify_item` with a list of one element, it's better to group them into a single `modify_item` with a list of two elements.

Note that `modify_item` nor `modify_item_seq` can work on subelements (e.g., `modify_item(list(obj$item = 5))` will not work as intended, for that is better to assign directly using the expression approach, so `obj$item <- 5`).

Costs and utilities can be modified by using the construction `type_name_category`, where type is either "qaly" or "cost", name is the name (e.g., "default") and category is the category used (e.g., "instant"), so one could pass `cost_default_instant` and modify the cost. This will overwrite the value defined in the corresponding cost/utility section.

This function is intended to be used only within the `add_reactevt` function in its `input` parameter and should not be run elsewhere or it will return an error.

Value

No return value, modifies/adds item to the environment and integrates it with the main list for storage

Examples

```
add_reactevt(name_evt = "idfs",input = {modify_item(list("cost.it"=5))})
```

`modify_item_seq` *Modify the value of existing items*

Description

Modify the value of existing items

Usage

```
modify_item_seq(...)
```

Arguments

`...`	A list of items and their values or expressions. Will be evaluated sequentially (so one could have list(a= 1, b = a +2))

Details

The functions to add/modify events/inputs use lists. Whenever several inputs/events are added or modified, it's recommended to group them within one function, as it reduces the computation cost. So rather than use two `modify_item` with a list of one element, it's better to group them into a single `modify_item` with a list of two elements.

Note that `modify_item` nor `modify_item_seq` can work on subelements (e.g., `modify_item_seq(list(obj$item = 5))` will not work as intended, for that is better to assign directly using the expression approach, so `obj$item <- 5`).

Costs and utilities can be modified by using the construction `type_name_category`, where type is either "qaly" or "cost", name is the name (e.g., "default") and category is the category used (e.g., "instant"), so one could pass `cost_default_instant` and modify the cost. This will overwrite the value defined in the corresponding cost/utility section.

The function is different from modify_item in that this function evaluates sequentially the arguments within the list passed. This implies a slower performance relative to modify_item, but it can be more cleaner and convenient in certain instances.

This function is intended to be used only within the `add_reactevt` function in its `input` parameter and should not be run elsewhere or it will return an error.

Value

No return value, modifies/adds items sequentially and deploys to the environment and with the main list for storage

Examples

```
add_reactevt(name_evt = "idfs",input = {
  modify_item_seq(list(cost.idfs = 500, cost.tx = cost.idfs + 4000))
  })
```

---

new_event *Add events to the queue for a patient*

Description

Adds one or more events for a given patient to the queue.

Usage

```
new_event(events, ptr, patient_id)
```

Arguments

`events`	A named numeric vector. Names are event types, values are event times. It can also handle lists instead of named vectors (at a small computational cost).
`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`patient_id`	The patient ID. Defaults to `i`.

Details

The functions to add/modify events/inputs use named vectors or lists. Whenever several inputs/events are added or modified, it's recommended to group them within one function, as it reduces the computation cost. So rather than use two `new_event` with a list of one element, it's better to group them into a single `new_event` with a list of two elements.

While multiple events can be added, they must be named differently. If the same event is added multiple times at once, only the last occurrence will be kept (only one event per event type in the queue of events yet to occur). If an event occurs, then a new one with the same name can be set.

This function is intended to be used only within the `add_reactevt` function in its `input` parameter and should not be run elsewhere or it will return an error.

Value

NULL (invisible). Modifies the queue in-place.

Examples

```
add_reactevt(name_evt = "idfs",input = {new_event(c("ae"=5))})
```

`next_event` *Get the next events in the queue*

Description

Retrieves the next `n` events (without removing them).

Usage

```
next_event(n = 1, ptr)
```

Arguments

`n`	Number of events to retrieve. Default is 1.
`ptr`	The event queue pointer. Defaults to `cur_evtlist`.

Value

A list of events, each with `patient_id`, `event_name`, and `time`.

next_event_pt — *Get the next events in the queue for a specific patient*

Description

Retrieves the next `n` events (without removing them).

Usage

```
next_event_pt(n = 1, ptr, patient_id)
```

Arguments

`n`	Number of events to retrieve. Default is 1.
`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`patient_id`	The patient ID. Defaults to `i`.

Value

A list of events, each with `patient_id`, `event_name`, and `time`.

pcond_gompertz — *Survival Probaility function for conditional Gompertz distribution (lower bound only)*

Description

Survival Probaility function for conditional Gompertz distribution (lower bound only)

Usage

```
pcond_gompertz(time = 1, shape, rate, lower_bound = 0)
```

Arguments

`time`	Vector of times
`shape`	The shape parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
`rate`	The rate parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
`lower_bound`	The lower bound of the conditional distribution

Value

Estimate(s) from the conditional Gompertz distribution based on given parameters

Examples

```
pcond_gompertz(time=1,shape=0.05,rate=0.01,lower_bound = 50)
```

`pick_psa` *Helper function to create a list with random draws or whenever a series of functions needs to be called. Can be implemented within* `pick_val_v`.

Description

Helper function to create a list with random draws or whenever a series of functions needs to be called. Can be implemented within `pick_val_v`.

Usage

```
pick_psa(f, ...)
```

Arguments

`f`	A string or vector of strings with the function to be called, e.g., "rnorm"
`...`	parameters to be passed to the function (e.g., if "rnorm", arguments `n`, `mean`, `sd`)

Details

This function can be used to pick values for the PSA within `pick_val_v`.

The function will ignore NA items within the respective parameter (see example below). If an element in f is NA (e.g., a non PSA input) then it will return NA as its value This feature is convenient when mixing distributions with different number of arguments, e.g., `rnorm` and `rgengamma`.

While it's slightly lower than individually calling each function, it makes the code easier to read and more transparent

Value

List with length equal to `f` of parameters called

Examples

```
params <- list(
param=list("a","b"),
dist=list("rlnorm","rnorm"),
n=list(4,1),
a=list(c(1,2,3,4),1),
b=list(c(0.5,0.5,0.5,0.5),0.5),
dsa_min=list(c(1,2,3,4),2),
dsa_max=list(c(1,2,3,4),3)
)
pick_psa(params[["dist"]],params[["n"]],params[["a"]],params[["b"]])
```

```
#It works with functions that require different number of parameters
params <- list(
 param=list("a","b","c"),
 dist=list("rlnorm","rnorm","rgengamma"),
 n=list(4,1,1),
 a=list(c(1,2,3,4),1,0),
 b=list(c(0.5,0.5,0.5,0.5),0.5,1),
 c=list(NA,NA,0.2),
 dsa_min=list(c(1,2,3,4),2,1),
 dsa_max=list(c(1,2,3,4),3,3)
)

pick_psa(params[["dist"]],params[["n"]],params[["a"]],params[["b"]],params[["c"]])

#Can be combined with multiple type of functions and distributions if parameters are well located

params <- list(
param=list("a","b","c","d"),
dist=list("rlnorm","rnorm","rgengamma","draw_tte"),
n=list(4,1,1,1),
a=list(c(1,2,3,4),1,0,"norm"),
b=list(c(0.5,0.5,0.5,0.5),0.5,1,1),
c=list(NA,NA,0.2,0.5),
c=list(NA,NA,NA,NA), #NA arguments will be ignored
dsa_min=list(c(1,2,3,4),2,1,0),
dsa_max=list(c(1,2,3,4),3,3,2)
)
```

`pick_val_v` *Select which values should be applied in the corresponding loop for several values (vector or list).*

Description

Select which values should be applied in the corresponding loop for several values (vector or list).

Usage

```
pick_val_v(
  base,
  psa,
  sens,
  psa_ind = psa_bool,
  sens_ind = sens_bool,
  indicator,
  indicator_psa = NULL,
  names_out = NULL,
  indicator_sens_binary = TRUE,
```

```
    sens_iterator = NULL,
    distributions = NULL,
    covariances = NULL,
    deploy_env = TRUE
)
```

Arguments

`base`	Value if no PSA/DSA/Scenario
`psa`	Value if PSA
`sens`	Value if DSA/Scenario
`psa_ind`	Boolean whether PSA is active
`sens_ind`	Boolean whether Scenario/DSA is active
`indicator`	Indicator which checks whether the specific parameter/parameters is/are active in the DSA or Scenario loop
`indicator_psa`	Indicator which checks whether the specific parameter/parameters is/are active in the PSA loop. If NULL, it's assumed to be a vector of 1s of length equal to length(indicator)
`names_out`	Names to give the output list
`indicator_sens_binary`	Boolean, TRUE if parameters will be varied fully, FALSE if some elements of the parameters may be changed but not all
`sens_iterator`	Current iterator number of the DSA/scenario being run, e.g., 5 if it corresponds to the 5th DSA parameter being changed
`distributions`	List with length equal to length of base where the distributions are stored
`covariances`	List with length equal to length of base where the variance/covariances are stored (only relevant if multivariate normal are being used)
`deploy_env`	Boolean, if TRUE will deploy all objects in the environment where the function is called for. Must be active if using add_item (and FALSE if a list must be returned)

Details

This function can be used with vectors or lists, but will always return a list. Lists should be used when correlated variables are introduced to make sure the selector knows how to choose among those This function allows to choose between using an approach where only the full parameters are varied, and an approach where subelements of the parameters can be changed

Value

List used for the inputs

Examples

```
pick_val_v(base = list(0,0),
             psa =list(rnorm(1,0,0.1),rnorm(1,0,0.1)),
             sens = list(2,3),
             psa_ind = FALSE,
             sens_ind = TRUE,
             indicator=list(1,2),
             indicator_sens_binary = FALSE,
             sens_iterator = 2,
             distributions = list("rnorm","rnorm"),
             deploy_env = FALSE
)

pick_val_v(base = list(2,3,c(1,2)),
             psa =sapply(1:3,
                         function(x) eval(call(
                           c("rnorm","rnorm","mvrnorm")[[x]],
                           1,
                           c(2,3,list(c(1,2)))[[x]],
                           c(0.1,0.1,list(matrix(c(1,0.1,0.1,1),2,2)))[[x]]
                         ))),
             sens = list(4,5,c(1.3,2.3)),
             psa_ind = FALSE,
             sens_ind = TRUE,
             indicator=list(1,2,c(3,4)),
             names_out=c("util","util2","correlated_vector") ,
             indicator_sens_binary = FALSE,
             sens_iterator = 4,
             distributions = list("rnorm","rnorm","mvrnorm"),
             covariances = list(0.1,0.1,matrix(c(1,0.1,0.1,1),2,2)),
             deploy_env = FALSE
)
```

`pop_and_return_event` *Pop and return the next event*

Description

Removes the next event from the queue and returns its details. Not needed by user.

Usage

```
pop_and_return_event(ptr)
```

Arguments

`ptr`	The event queue pointer. Defaults to `cur_evtlist`.

Value

A named list with `patient_id`, `event_name`, and `time`.

`pop_event` *Remove the next event from the queue*

Description

Removes the next scheduled event from the queue. Not needed by user.

Usage

```
pop_event(ptr)
```

Arguments

`ptr`	The event queue pointer. Defaults to `cur_evtlist`.

Value

NULL (invisible). Modifies the queue in-place.

`print.warden_results` *Print method for warden_results*

Description

Displays a summary of the structure of a WARDEN simulation results object.

Usage

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

Arguments

`x`	A `warden_results` object returned by `run_sim()` or `run_sim_parallel()`.
`...`	Additional arguments (ignored).

Value

Invisibly returns `x`.

qbeta_mse — *Draw from a beta distribution based on mean and se (quantile)*

Description

Draw from a beta distribution based on mean and se (quantile)

Usage

```
qbeta_mse(q, mean_v, se)
```

Arguments

q	Quantiles to be used
mean_v	A vector of the mean values
se	A vector of the standard errors of the means

Value

A single estimate from the beta distribution based on given parameters

Examples

```
qbeta_mse(q=0.5,mean_v=0.8,se=0.2)
```

qcategorical — *Quantile function for categorical distribution*

Description

Alias for `draw_categorical()` using standard quantile-function naming.

Usage

```
qcategorical(p, probs)
```

Arguments

p	Numeric between 0 and 1 (inclusive). Probability (same as `luck` in `draw_categorical()`).
probs	Numeric vector of probabilities.

Value

Integer: 1-based index.

qcond_exp — *Conditional quantile function for exponential distribution*

Description

Conditional quantile function for exponential distribution

Usage

```
qcond_exp(rnd, rate)
```

Arguments

`rnd`	Vector of quantiles
`rate`	The rate parameter Note taht the conditional quantile for an exponential is independent of time due to constant hazard

Value

Estimate(s) from the conditional exponential distribution based on given parameters

Examples

```
qcond_exp(rnd = 0.5,rate = 3)
```

qcond_gamma — *Conditional quantile function for gamma distribution*

Description

Conditional quantile function for gamma distribution

Usage

```
qcond_gamma(rnd, shape, rate, lower_bound, s_obs)
```

Arguments

`rnd`	Vector of quantiles
`shape`	The shape parameter
`rate`	The rate parameter
`lower_bound`	The lower bound to be used (current time)
`s_obs`	is the survival observed up to lower_bound time, normally defined from time 0 as 1 - pgamma(q = lower_bound, rate, shape) but may be different if parametrization has changed previously

Value

Estimate(s) from the conditional gamma distribution based on given parameters

Examples

```
qcond_gamma(rnd = 0.5, shape = 1.06178, rate = 0.01108,lower_bound = 1, s_obs=0.8)
```

---

qcond_gompertz — *Quantile function for conditional Gompertz distribution (lower bound only)*

Description

Quantile function for conditional Gompertz distribution (lower bound only)

Usage

```
qcond_gompertz(rnd, shape, rate, lower_bound = as.numeric(c(0)))
```

Arguments

rnd	Vector of quantiles
shape	The shape parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
rate	The rate parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
lower_bound	The lower bound of the conditional distribution

Value

Estimate(s) from the conditional Gompertz distribution based on given parameters

Examples

```
qcond_gompertz(rnd=0.5,shape=0.05,rate=0.01,lower_bound = 50)
```

qcond_llogis — *Conditional quantile function for loglogistic distribution*

Description

Conditional quantile function for loglogistic distribution

Usage

```
qcond_llogis(rnd, shape, scale, lower_bound = as.numeric(c(0)))
```

Arguments

`rnd`	Vector of quantiles
`shape`	The shape parameter
`scale`	The scale parameter
`lower_bound`	The lower bound to be used (current time)

Value

Estimate(s) from the conditional loglogistic distribution based on given parameters

Examples

```
qcond_llogis(rnd = 0.5,shape = 1,scale = 1,lower_bound = 1)
```

qcond_lnorm — *Conditional quantile function for lognormal distribution*

Description

Conditional quantile function for lognormal distribution

Usage

```
qcond_lnorm(rnd, meanlog, sdlog, lower_bound, s_obs)
```

Arguments

`rnd`	Vector of quantiles
`meanlog`	The meanlog parameter
`sdlog`	The sdlog parameter
`lower_bound`	The lower bound to be used (current time)
`s_obs`	is the survival observed up to lower_bound time, normally defined from time 0 as 1 - plnorm(q = lower_bound, meanlog, sdlog) but may be different if parametrization has changed previously

Value

Estimate(s) from the conditional lognormal distribution based on given parameters

Examples

```
qcond_lnorm(rnd = 0.5, meanlog = 1,sdlog = 1,lower_bound = 1, s_obs=0.8)
```

---

qcond_norm — *Conditional quantile function for normal distribution*

Description

Conditional quantile function for normal distribution

Usage

```
qcond_norm(rnd, mean, sd, lower_bound, s_obs)
```

Arguments

rnd	Vector of quantiles
mean	The mean parameter
sd	The sd parameter
lower_bound	The lower bound to be used (current time)
s_obs	is the survival observed up to lower_bound time, normally defined from time 0 as 1 - pnorm(q = lower_bound, mean, sd) but may be different if parametrization has changed previously

Value

Estimate(s) from the conditional normal distribution based on given parameters

Examples

```
qcond_norm(rnd = 0.5, mean = 1,sd = 1,lower_bound = 1, s_obs=0.8)
```

qcond_weibull — *Conditional quantile function for weibull distribution*

Description

Conditional quantile function for weibull distribution

Usage

```
qcond_weibull(rnd, shape, scale, lower_bound = as.numeric(c(0)))
```

Arguments

rnd	Vector of quantiles
shape	The shape parameter as in R stats package weibull
scale	The scale parameter as in R stats package weibull
lower_bound	The lower bound to be used (current time)

Value

Estimate(s) from the conditional weibull distribution based on given parameters

Examples

```
qcond_weibull(rnd = 0.5,shape = 3,scale = 66.66,lower_bound = 50)
```

qcond_weibullPH — *Conditional quantile function for WeibullPH (flexsurv)*

Description

Conditional quantile function for WeibullPH (flexsurv)

Usage

```
qcond_weibullPH(rnd, shape, scale, lower_bound = as.numeric(c(0)))
```

Arguments

rnd	Vector of quantiles (between 0 and 1)
shape	Shape parameter of WeibullPH
scale	Scale (rate) parameter of WeibullPH (i.e., as in hazard = scale * t^(shape - 1))
lower_bound	Lower bound (current time)

Value

Estimate(s) from the conditional weibullPH distribution based on given parameters

Examples

```
qcond_weibullPH(rnd = 0.5, shape = 2, scale = 0.01, lower_bound = 5)
```

---

qgamma_mse *Use quantiles from a gamma distribution based on mean and se*

Description

Use quantiles from a gamma distribution based on mean and se

Usage

```
qgamma_mse(q = 1, mean_v, se, seed = NULL)
```

Arguments

q	Quantile to draw
mean_v	A vector of the mean values
se	A vector of the standard errors of the means
seed	An integer which will be used to set the seed for this draw.

Value

A single estimate from the gamma distribution based on given parameters

Examples

```
qgamma_mse(q=0.5,mean_v=0.8,se=0.2)
```

`qtimecov` *Draw time-to-event with time-dependent covariates and luck adjustment*

Description

Simulate a time-to-event (TTE) from a parametric distribution with parameters varying over time. User provides parameter functions and distribution name. The function uses internal survival and conditional quantile functions, plus luck adjustment to simulate the event time. See the vignette on avoiding cycles for an example in a model.

Usage

```
qtimecov(
  luck,
  a_fun,
  b_fun = NULL,
  dist = "exp",
  dt = 0.1,
  max_time = 100,
  start_time = 0
)
```

Arguments

`luck`	Numeric between 0 and 1. Initial random quantile (luck).
`a_fun`	Function of time .time returning the first distribution parameter (e.g., rate, shape, meanlog).
`b_fun`	Function of time .time returning the second distribution parameter (e.g., scale, sdlog). Defaults to a function returning NA.
`dist`	Character string specifying the distribution. Supported: "exp", "gamma", "lnorm", "norm", "weibull", "llogis", "gompertz".
`dt`	Numeric. Time step increment to update parameters and survival. Default 0.1.
`max_time`	Numeric. Max allowed event time to prevent infinite loops. Default 100.
`start_time`	Numeric. Time to use as a starting point of reference (e.g., curtime).

Details

The objective of this function is to avoid the user to have cycle events with the only scope of updating some variables that depend on time and re-evaluate a TTE. The idea is that this function should only be called at start and when an event impacts a variable (e.g., stroke event impacting death TTE), in which case it would need to be called again at that point. In that case, the user would need to call e.g., `a <- qtimecov` with `max_time = curtime` arguments, and then call it again with no max_time, and `luck = a$luck, start_time=a$tte` (so there is no need to add curtime to the resulting time).

It's recommended to play with `dt` argument to balance running time and precision of the estimates. For example, if we know we only update the equation annually (not continuously), then we could just set `dt = 1`, which would make computations faster.

Value

List with simulated time-to-event and final luck value.

Examples

```
param_fun_factory <- function(p0, p1, p2, p3) {
  function(.time) p0 + p1*.time + p2*.time^2 + p3*(floor(.time) + 1)
}

set.seed(42)

# 1. Exponential Example
rate_exp <- param_fun_factory(0.1, 0, 0, 0)
qtimecov(
  luck = runif(1),
  a_fun = rate_exp,
  dist = "exp"
)


# 2. Gamma Example
shape_gamma <- param_fun_factory(2, 0, 0, 0)
rate_gamma <- param_fun_factory(0.2, 0, 0, 0)
qtimecov(
  luck = runif(1),
  a_fun = shape_gamma,
  b_fun = rate_gamma,
  dist = "gamma"
)


# 3. Lognormal Example
meanlog_lnorm <- param_fun_factory(log(10) - 0.5*0.5^2, 0, 0, 0)
sdlog_lnorm <- param_fun_factory(0.5, 0, 0, 0)
qtimecov(
  luck = runif(1),
  a_fun = meanlog_lnorm,
  b_fun = sdlog_lnorm,
  dist = "lnorm"
)


# 4. Normal Example
mean_norm <- param_fun_factory(10, 0, 0, 0)
sd_norm <- param_fun_factory(2, 0, 0, 0)
qtimecov(
  luck = runif(1),
```

```
  a_fun = mean_norm,
  b_fun = sd_norm,
  dist = "norm"
)


# 5. Weibull Example
shape_weibull <- param_fun_factory(2, 0, 0, 0)
scale_weibull <- param_fun_factory(10, 0, 0, 0)
qtimecov(
  luck = runif(1),
  a_fun = shape_weibull,
  b_fun = scale_weibull,
  dist = "weibull"
)


# 6. Loglogistic Example
shape_llogis <- param_fun_factory(2.5, 0, 0, 0)
scale_llogis <- param_fun_factory(7.6, 0, 0, 0)
qtimecov(
  luck = runif(1),
  a_fun = shape_llogis,
  b_fun = scale_llogis,
  dist = "llogis"
)


# 7. Gompertz Example
shape_gomp <- param_fun_factory(0.01, 0, 0, 0)
rate_gomp <- param_fun_factory(0.091, 0, 0, 0)
qtimecov(
  luck = runif(1),
  a_fun = shape_gomp,
  b_fun = rate_gomp,
  dist = "gompertz"
)

#Time varying example, with change at time 8
rate_exp <- function(.time) 0.1 + 0.01*.time * 0.00001*.time^2
rate_exp2 <- function(.time) 0.2 + 0.02*.time
time_change <- 8
init_luck <- 0.95

a <- qtimecov(luck = init_luck,a_fun = rate_exp,dist = "exp", dt = 0.005,
                      max_time = time_change)
qtimecov(luck = a$luck,a_fun = rate_exp2,dist = "exp", dt = 0.005, start_time=a$tte)


#An example of how it would work in the model, this would also work with time varying covariates!
rate_exp <- function(.time) 0.1
rate_exp2 <- function(.time) 0.2
rate_exp3 <- function(.time) 0.3
```

```
time_change <- 10 #evt 1
time_change2 <- 15 #evt2
init_luck <- 0.95
#at start, we would just draw TTE
qtimecov(luck = init_luck,a_fun = rate_exp,dist = "exp", dt = 0.005)

#at event in which rate changes (at time 10) we need to do this:
a <- qtimecov(luck = init_luck,a_fun = rate_exp,dist = "exp", dt = 0.005,
                      max_time = time_change)
new_luck <- a$luck
qtimecov(luck = new_luck,a_fun = rate_exp2,dist = "exp", dt = 0.005, start_time=a$tte)

#at second  event in which rate changes again (at time 15) we need to do this:
a <- qtimecov(luck = new_luck,a_fun = rate_exp2,dist = "exp", dt = 0.005,
                      max_time = time_change2, start_time=a$tte)
new_luck <- a$luck
#final TTE is
qtimecov(luck = new_luck,a_fun = rate_exp3,dist = "exp", dt = 0.005, start_time=a$tte)
```

`queue_create` *Create a New Event Queue*

Description

Initializes a new event queue with the specified priority order of event names.

Usage

```
queue_create(priority_order)
```

Arguments

`priority_order` A character vector of event names sorted by decreasing importance.

Value

An external pointer to the new event queue.

`queue_empty` *Check if the event queue is empty*

Description

Check if the event queue is empty

Usage

```
queue_empty(ptr, exclude_inf = FALSE)
```

Arguments

`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`exclude_inf`	Logical, whether to exclude events with Inf time. Default is FALSE.

Value

Logical, TRUE if the queue is empty, FALSE otherwise.

`queue_size` *Get the Size of the Event Queue*

Description

Get the Size of the Event Queue

Usage

```
queue_size(ptr, exclude_inf = FALSE)
```

Arguments

`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`exclude_inf`	Logical, whether to exclude events with Inf time. Default is FALSE.

Value

An integer indicating the number of events in the queue.

`random_stream` *Creates an environment (similar to R6 class) of random uniform numbers to be drawn from*

Description

Creates an environment (similar to R6 class) of random uniform numbers to be drawn from

Usage

```
random_stream(stream_size = 100, strict = FALSE)
```

Arguments

`stream_size`	Length of the vector of random uniform values to initialize
`strict`	Logical (default `FALSE`). When `TRUE`, `draw_n()` throws an error if the stream is exhausted instead of regenerating with a warning.

Details

This function creates an environment object that behaves similar to an R6 class but offers more speed vs. an R6 class.

The object is always initialized (see example below) to a specific vector of random uniform values. The user can then call the object with `obj$draw_number(n)`, where n is an integer, and will return the first n elements of the created vector of uniform values. It will automatically remove those indexes from the vector, so the next time the user calls `obj$draw_n()` it will already consider the next index.

The user can also access the latest elements drawn by accessing `obj$random_n` (useful for when performing a luck adjustment), the current stream still to be drawn using `obj$stream` and the original size (when created) using `obj$stream_size`.

If performing luck adjustment, the user can always modify the random value by using `obj$random_n <- luck_adj(...)` (only valid if used with the expression approach, not with `modify_item`)

Value

Self (environment) behaving similar to R6 class

Examples

```
stream_1 <- random_stream(1000)
number_1 <- stream_1$draw_n() #extract 1st index from the vector created
identical(number_1,stream_1$random_n) #same value
number_2 <- stream_1$draw_n() #gets 1st index (considers previous)
identical(number_2,stream_1$random_n) #same value
```

---

rbeta_mse — *Draw from a beta distribution based on mean and se*

Description

Draw from a beta distribution based on mean and se

Usage

```
rbeta_mse(n = 1, mean_v, se, seed = NULL)
```

Arguments

`n`	Number of draws (must be >= 1)
`mean_v`	A vector of the mean values
`se`	A vector of the standard errors of the means
`seed`	An integer which will be used to set the seed for this draw.

Value

A single estimate from the beta distribution based on given parameters

Examples

```
rbeta_mse(n=1,mean_v=0.8,se=0.2)
```

`rcond_gompertz` *Draw from a conditional Gompertz distribution (lower bound only)*

Description

Draw from a conditional Gompertz distribution (lower bound only)

Usage

```
rcond_gompertz(n = 1, shape, rate, lower_bound = 0, seed = NULL)
```

Arguments

`n`	The number of observations to be drawn
`shape`	The shape parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
`rate`	The rate parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
`lower_bound`	The lower bound of the conditional distribution
`seed`	An integer which will be used to set the seed for this draw.

Value

Estimate(s) from the conditional Gompertz distribution based on given parameters

Examples

```
rcond_gompertz(1,shape=0.05,rate=0.01,lower_bound = 50)
```

`rcond_gompertz_lu` *Draw from a conditional Gompertz distribution (lower and upper bound)*

Description

Draw from a conditional Gompertz distribution (lower and upper bound)

Usage

```
rcond_gompertz_lu(
  n,
  shape,
  rate,
  lower_bound = 0,
  upper_bound = Inf,
  seed = NULL
)
```

Arguments

n	The number of observations to be drawn
shape	The shape parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
rate	The rate parameter of the Gompertz distribution, defined as in the coef() output on a flexsurvreg object
lower_bound	The lower bound of the conditional distribution
upper_bound	The upper bound of the conditional distribution
seed	An integer which will be used to set the seed for this draw.

Value

Estimate(s) from the Conditional Gompertz distribution based on given parameters

Examples

```
rcond_gompertz_lu(1,shape=0.05,rate=0.01,lower_bound = 50)
```

rdirichlet — *Draw from a dirichlet distribution based on number of counts in transition. Adapted from brms::rdirichlet*

Description

Draw from a dirichlet distribution based on number of counts in transition. Adapted from brms::rdirichlet

Usage

```
rdirichlet(n = 1, alpha, seed = NULL)
```

Arguments

`n`	Number of draws (must be >= 1). If n>1, it will return a list of matrices.
`alpha`	A matrix of alphas (transition counts)
`seed`	An integer which will be used to set the seed for this draw.

Value

A transition matrix. If n>1, it will return a list of matrices.

Examples

```
rdirichlet(n=1,alpha= matrix(c(1251, 0, 350, 731),2,2))
rdirichlet(n=2,alpha= matrix(c(1251, 0, 350, 731),2,2))
```

rdirichlet_prob — *Draw from a dirichlet distribution based on mean transition probabilities and standard errors*

Description

Draw from a dirichlet distribution based on mean transition probabilities and standard errors

Usage

```
rdirichlet_prob(n = 1, alpha, se, seed = NULL)
```

Arguments

`n`	Number of draws (must be >= 1). If n>1, it will return a list of matrices.
`alpha`	A matrix of transition probabilities
`se`	A matrix of standard errors
`seed`	An integer which will be used to set the seed for this draw.

Value

A transition matrix. If n>1, it will return a list of matrices.

Examples

```
rdirichlet_prob(n=1,alpha= matrix(c(0.7,0.3,0,0.1,0.7,0.2,0.1,0.2,0.7),3,3),
se=matrix(c(0.7,0.3,0,0.1,0.7,0.2,0.1,0.2,0.7)/10,3,3))

rdirichlet_prob(n=2,alpha= matrix(c(0.7,0.3,0,0.1,0.7,0.2,0.1,0.2,0.7),3,3),
se=matrix(c(0.7,0.3,0,0.1,0.7,0.2,0.1,0.2,0.7)/10,3,3))
```

`release` *Release a discrete resource*

Description

Frees the resource for the current patient (`i`) and, if the patient was actually using the resource and a `resume_event` is supplied, schedules that event for the next patient in the queue.

Usage

```
release(resource, resume_event = NULL, amount = NULL)
```

Arguments

`resource`	A `resource_discrete` object.
`resume_event`	Character string. Name of the event to schedule for the next queued patient, or `NULL` (default) to skip re-triggering.
`amount`	Integer or `NULL` (default). Units to release. `NULL` means 1. Must exactly match the amount used when seizing (indivisible units).

Value

Invisibly, `TRUE` if the patient was using the resource, `FALSE` otherwise.

release_all — *Release multiple discrete resources*

Description

Frees the current patient (`i`) from each resource. Applies an all_or_none policy: only acts if the patient is using ALL listed resources. Removes the patient from both the using list and the queue. When `amounts` is `NULL` (default), releases all entries and purges queue entries for the patient; when `amounts` is specified, performs an exact indivisible release without purging the queue. Schedules resume events only for resources where the patient was actually using (i.e., where capacity was freed).

Usage

```
release_all(resources, resume_event = NULL, amounts = NULL)
```

Arguments

`resources`	A list of `resource_discrete` objects.
`resume_event`	`NULL` (no re-triggering), a single string (same event for all resources' queues), or a character vector of length `length(resources)` (per-resource events, matched positionally).
`amounts`	Integer vector of units per resource, or `NULL` (default). `NULL` releases all entries and purges queue entries for the patient. Specified amounts perform exact indivisible releases without purging.

Value

Invisibly `NULL`.

release_all_if_using — *Release multiple discrete resources (using only)*

Description

Frees the current patient (`i`) from each resource only if they are currently using it. Applies an all_or_none policy: only acts if the patient is using ALL listed resources. Does not remove the patient from any queue. Use this when a patient transitions state but should keep their queue position.

Usage

```
release_all_if_using(resources, resume_event = NULL, amounts = NULL)
```

Arguments

`resources`	A list of `resource_discrete` objects.
`resume_event`	`NULL` (no re-triggering), a single string (same event for all resources' queues), or a character vector of length `length(resources)` (per-resource events, matched positionally).
`amounts`	Integer vector of units per resource, or `NULL` (default). `NULL` releases all entries and purges queue entries for the patient. Specified amounts perform exact indivisible releases without purging.

Value

Invisibly `NULL`.

`release_if_using` *Release a discrete resource only if using*

Description

Releases the resource for the current patient (`i`) only if they are currently using it. Does nothing if the patient is not using the resource. Never removes queue entries.

Usage

```
release_if_using(resource, resume_event = NULL, amount = NULL)
```

Arguments

`resource`	A `resource_discrete` object.
`resume_event`	Character string. Name of the event to schedule for the next queued patient, or `NULL` (default) to skip re-triggering.
`amount`	Integer or `NULL` (default). Units to release. `NULL` means 1. Must exactly match the amount used when seizing (indivisible units).

Value

Invisibly `NULL`.

`remove_event` *Remove events for a patient*

Description

Removes one or more events from the queue for the given patient.

Usage

```
remove_event(events, ptr, patient_id)
```

Arguments

`events`	A character vector of event names to remove. It can also handle lists instead of named vectors (at a small computational cost).
`ptr`	The event queue pointer. Defaults to `cur_evtlist`.
`patient_id`	The patient ID. Defaults to `i`.

Value

NULL (invisible). Modifies the queue in-place.

`replicate_profiles` *Replicate profiles data.frame*

Description

Replicate profiles data.frame

Usage

```
replicate_profiles(
  profiles,
  replications,
  probabilities = NULL,
  replacement = TRUE,
  seed_used = NULL
)
```

Arguments

`profiles`	data.frame of profiles
`replications`	integer, final number of observations
`probabilities`	vector of probabilities with the same length as the number of rows of profiles. Does not need to add up to 1 (are reweighted)
`replacement`	Boolean whether replacement is used
`seed_used`	Integer with the seed to be used for consistent results

Value

Resampled data.frame of profiles

Examples

```
replicate_profiles(profiles=data.frame(id=1:100,age=rnorm(100,60,5)),
replications=200,probabilities=rep(1,100))
```

`resource_discrete` *Create a discrete resource*

Description

Creates a discrete resource management system for discrete event simulations. This system manages a fixed number of identical resource units that can be blocked (used) by patients and maintains a priority queue for waiting patients.

Usage

```
resource_discrete(
  n,
  discipline = "FIFO",
  max_queue = Inf,
  allow_multiple_queue = TRUE
)
```

Arguments

`n` Integer >= 0. Total capacity of the resource.

`discipline` `"FIFO"` (default) or `"LIFO"`. Queue discipline applied within the same priority level.

`max_queue` Non-negative integer or `Inf` (default). Maximum number of patients that can wait in the queue. Patients that arrive when the queue is full are rejected (`attempt_block()` returns `NA`).

`allow_multiple_queue` Logical (default `TRUE`). If `FALSE`, a patient that already has a queued entry will be rejected (`attempt_block()` returns `NA`) instead of being allowed to queue again.

Value

An environment of class `"resource_discrete"` with the following methods:

`size()` Total capacity.

`queue_size()` Number of patients currently in the queue.

`n_free()` Number of free resource units.

n_using() Number of units currently in use.

utilization() Fraction of capacity in use (0-1).

patients_using() Vector of patient IDs currently using the resource.

patients_using_times() Vector of start times for patients using the resource.

queue_start_times() Vector of queue start times in queue order.

queue_priorities() Vector of priorities in queue order.

queue_info(n) Data frame with patient_id, priority, start_time for the queue.

is_patient_in_queue(patient_id) Check if patient is in queue. Defaults to i from the calling environment.

is_patient_using(patient_id) Check if patient is using the resource. Defaults to i from the calling environment.

attempt_block(patient_id, priority, start_time, amount) Attempt to block amount units. Returns TRUE (acquired), FALSE (queued), or NA (rejected). Defaults patient_id to i and start_time to curtime from the calling environment.

attempt_free(patient_id, remove_all, amount) Free the resource for a patient. Defaults patient_id to i.

attempt_free_if_using(patient_id, remove_all) Free only if patient is currently using. Defaults patient_id to i.

next_patient_in_line(n) Get next n patients in queue.

modify_priority(patient_id, new_priority) Modify patient priority in queue. When the patient has k active queue entries, complexity is O(k log n), not O(log n).

add_resource(n) Add n resource units.

remove_resource(n, current_time) Remove n resource units. Defaults current_time to curtime.

queue_wait_time(patient_id) Final queue wait time, set on successful dequeue. Returns NA if never queued or still waiting. Defaults patient_id to i.

queue_wait_time_current(patient_id, current_time) Current elapsed queue wait: 0 at entry, grows each event while waiting, returns final wait on acquisition, NA if never queued. Defaults to i / curtime.

had_to_queue(patient_id) Returns 1L if patient ever queued, 0L otherwise. Defaults patient_id to i.

time_in_use(patient_id, current_time) Time since patient acquired resource. Returns NA if not currently using. Defaults to i / curtime.

total_patients_blocked() Count of unique patients that ever successfully acquired.

total_patients_queued() Count of unique patients that ever entered the queue.

batch_seize(patient_ids, priority, start_time, amount_each) Seize for multiple patients in a single C++ call. Returns an integer vector (1 = acquired, 0 = queued, -1 = rejected).

Examples

```
# Create a resource with 3 units
beds <- resource_discrete(3)
```

```
# Check initial state
beds$size()       # 3
beds$n_free()     # 3
beds$queue_size() # 0

# Block resources (reads i and curtime from calling environment)
i <- 101L; curtime <- 0.0
beds$attempt_block()

# Or explicitly
beds$attempt_block(patient_id = 102L, priority = 1L, start_time = 1.0)

# LIFO resource and limited queue
stack <- resource_discrete(2, discipline = "LIFO", max_queue = 5)
```

`rgamma_mse` *Draw from a gamma distribution based on mean and se*

Description

Draw from a gamma distribution based on mean and se

Usage

```
rgamma_mse(n = 1, mean_v, se, seed = NULL)
```

Arguments

`n`	Number of draws (must be >= 1)
`mean_v`	A vector of the mean values
`se`	A vector of the standard errors of the means
`seed`	An integer which will be used to set the seed for this draw.

Value

A single estimate from the gamma distribution based on given parameters

Examples

```
rgamma_mse(n=1,mean_v=0.8,se=0.2)
```

`rpoisgamma` *Draw time to event (tte) from a Poisson or Poisson-Gamma (PG) Mixture/Negative Binomial (NB) Process*

Description

Draw time to event (tte) from a Poisson or Poisson-Gamma (PG) Mixture/Negative Binomial (NB) Process

Usage

```
rpoisgamma(
  n,
  rate,
  theta = NULL,
  obs_time = 1,
  t_reps,
  seed = NULL,
  return_ind_rate = FALSE,
  return_df = FALSE
)
```

Arguments

`n`	The number of observations to be drawn
`rate`	rate of the event (in terms of events per observation-time)
`theta`	Optional. When omitted, the function simulates times for a Poisson process. Represents the shape of the gamma mixture distribution. Estimated and reported as theta in negative binomial regression analyses in r.
`obs_time`	period over which events are observable
`t_reps`	Optional. Number of TBEs to be generated to capture events within the observation window. When omitted, the function sets t_reps to the 99.99th quantile of the Poisson (if no theta is provided) or negative binomial (if theta is provided). Thus, the risk of missing possible events in the observation window is 0.01%.
`seed`	An integer which will be used to set the seed for this draw.
`return_ind_rate`	A boolean that indicates whether an additional vector with the rate parameters used per observation is used. It will alter the structure of the results to two lists, one storing tte with name tte, and the other with name ind_rate
`return_df`	A boolean that indicates whether a data.table object should be returned

Details

Function to simulate event times from a Poisson or Poisson-Gamma (PG) Mixture/Negative Binomial (NB) Process Event times are determined by sampling times between events (TBEs) from an exponential distribution, and cumulating these to derive the event times. Events occurring within the set observation time window are retained and returned. For times for a Poisson process, the provided rate is assumed constant. For a PG or NB, the individual rates are sampled from a Gamma distribution with shape = theta and scale = rate/theta.

Value

Estimate(s) from the time to event based on poisson/Poisson-Gamma (PG) Mixture/Negative Binomial (NB) distribution based on given parameters

Examples

```
rpoisgamma(1,rate=1,obs_time=1,theta=1)
```

---

rpoisgamma_rcpp — *Draw time to event (tte) from a Poisson or Poisson-Gamma (PG) Mixture/Negative Binomial (NB) Process using C++*

Description

Draw time to event (tte) from a Poisson or Poisson-Gamma (PG) Mixture/Negative Binomial (NB) Process using C++

Usage

```
rpoisgamma_rcpp(
  n,
  rate,
  theta = NULL,
  obs_time = 1,
  t_reps = NULL,
  seed = NULL,
  return_ind_rate = FALSE,
  return_df = FALSE
)
```

Arguments

n	The number of observations to be drawn
rate	rate of the event (events per unit time)
theta	Optional. If provided, Poisson-Gamma (NB). Represents gamma shape.
obs_time	period over which events are observable

t_reps	Optional. Number of TBEs to be generated to capture events within the observation window.
seed	Optional integer seed for reproducibility.
return_ind_rate	Logical: include individual rate vector in output when theta provided.
return_df	Logical: return a data.frame with event-level rows (if TRUE).

Value

If return_df=TRUE: a data.frame (or NULL if no events). Else: list with tte and optionally ind_rate.

Examples

```
rpoisgamma_rcpp(1, rate = 1, obs_time = 1, theta = 1)
```

---

run_sim *Run the simulation*

Description

Run the simulation

Usage

```
run_sim(
  arm_list = c("int", "noint"),
  sensitivity_inputs = NULL,
  common_all_inputs = NULL,
  common_pt_inputs = NULL,
  unique_pt_inputs = NULL,
  init_event_list = NULL,
  evt_react_list = evt_react_list,
  util_ongoing_list = NULL,
  util_instant_list = NULL,
  util_cycle_list = NULL,
  cost_ongoing_list = NULL,
  cost_instant_list = NULL,
  cost_cycle_list = NULL,
  other_ongoing_list = NULL,
  other_instant_list = NULL,
  npats = 500,
  n_sim = 1,
  psa_bool = NULL,
  sensitivity_bool = FALSE,
```

```
  sensitivity_names = NULL,
  n_sensitivity = 1,
  input_out = character(),
  ipd = 1,
  constrained = FALSE,
  timed_freq = NULL,
  debug = FALSE,
  accum_backwards = FALSE,
  continue_on_error = FALSE,
  seed = NULL
)
```

Arguments

`arm_list` A vector of the names of the interventions evaluated in the simulation

`sensitivity_inputs` A list of sensitivity inputs that do not change within a sensitivity in a similar fashion to common_all_inputs, etc

`common_all_inputs` A list of inputs common across patients that do not change within a simulation

`common_pt_inputs` A list of inputs that change across patients but are not affected by the intervention

`unique_pt_inputs` A list of inputs that change across each intervention

`init_event_list` A list of initial events and event times. If no initial events are given, a "Start" event at time 0 is created automatically

`evt_react_list` A list of event reactions

`util_ongoing_list` Vector of QALY named variables that are accrued at an ongoing basis (discounted using drq)

`util_instant_list` Vector of QALY named variables that are accrued instantaneously at an event (discounted using drq)

`util_cycle_list` Vector of QALY named variables that are accrued in cycles (discounted using drq)

`cost_ongoing_list` Vector of cost named variables that are accrued at an ongoing basis (discounted using drc)

`cost_instant_list` Vector of cost named variables that are accrued instantaneously at an event (discounted using drc)

`cost_cycle_list` Vector of cost named variables that are accrued in cycles (discounted using drc)

other_ongoing_list
: Vector of other named variables that are accrued at an ongoing basis (discounted using drq)

other_instant_list
: Vector of other named variables that are accrued instantaneously at an event (discounted using drq)

npats
: The number of patients to be simulated (it will simulate npats * length(arm_list))

n_sim
: The number of simulations to run per sensitivity

psa_bool
: A boolean to determine if PSA should be conducted. If n_sim > 1 and psa_bool = FALSE, the differences between simulations will be due to sampling

sensitivity_bool
: A boolean to determine if Scenarios/DSA should be conducted.

sensitivity_names
: A vector of scenario/DSA names that can be used to select the right sensitivity (e.g., c("Scenario_1", "Scenario_2")). The parameter "sens_name_used" is created from it which corresponds to the one being used for each iteration.

n_sensitivity
: Number of sensitivity analysis (DSA or Scenarios) to run. It will be interacted with sensitivity_names argument if not null (n_sensitivityitivity = n_sensitivity * length(sensitivity_names)). For DSA, it should be as many parameters as there are. For scenario, it should be 1.

input_out
: A vector of variables to be returned in the output data frame

ipd
: Integer taking value 1 for full IPD data returned, and 2 IPD data but aggregating events (returning last value for numeric/character/factor variables. For other objects (e.g., matrices), the IPD will still be returned as the aggregation rule is not clear). Other values mean no IPD data returned (removes non-numerical or length>1 items)

constrained
: Boolean, FALSE by default, which runs the simulation with patients not interacting with each other, TRUE if resources are shared within an arm (allows constrained resources)

timed_freq
: If NULL, it does not produce any timed outputs. Otherwise should be a number (e.g., every 1 year)

debug
: If TRUE, will generate a log file

accum_backwards
: If TRUE, the ongoing accumulators will count backwards (i.e., the current value is applied until the previous update). If FALSE, the current value is applied between the current event and the next time it is updated. Recommended for resource-constrained models where queue waits make future event times unpredictable. With TRUE, ongoing values are computed at each event for the interval just completed, avoiding prospective errors from `adj_val()`.

continue_on_error
: If TRUE, on error it will attempt to continue by skipping the current simulation

seed
: Starting seed to be used for the whole analysis. If null, it's set to 1 by default.

Details

This function is slightly different from run_sim_parallel. run_sim_parallel only runs multiple-core at the simulation level. run_sim uses only-single core. run_sim can be more efficient if using only one simulation (e.g., deterministic), while run_sim_parallel will be more efficient if the number of simulations is >1 (e.g., PSA).

Event ties are processed in the order declared within the init_event_list argument (evts argument within the first sublist of that object). To do so, the program automatically adds a sequence from to 0 to the (number of events - 1) times 1e-10 to add to the event times when selecting the event with minimum time. This time has been selected as it's relatively small yet not so small as to be ignored by which.min (see .Machine for more details)

A list of protected objects that should not be used by the user as input names or in the global environment to avoid the risk of overwriting them is as follows: c("arm", "arm_list", "categories_for_export", "cur_evtlist", "curtime", "evt", "i", "prevtime", "sens", "simulation", "sens_name_used","list_env","uc_lists","npats","ipd").

The engine uses the L'Ecuyer-CMRG for the random number generator. Note that the random seeds are set to be unique in their category (i.e., at patient level, patient-arm level, etc.)

If no drc or drq parameters are passed within sensitivity or common_all input lists, these are assigned a default value 0.03 for discounting costs, QALYs and others.

Ongoing items will look backward to the last time updated when performing the discounting and accumulation. This means that the user does not necessarily need to keep updating the value, but only add it when the value changes looking forward (e.g., o_q = utility at event 1, at event 2 utility does not change, but at event 3 it does, so we want to make sure to add o_q = utility at event 3 before updating utility. The program will automatically look back until event 1). Note that in previous versions of the package backward was the default, and now this has switched to forward.

The requirement to use modify_item if using accum_backwards = TRUE, is no longer the case thanks to a new method using active bindings, so it can be used normally.

It is important to note that the QALYs and Costs (ongoing or instant or per cycle) used should be of length 1. If they were of length > 1, the model would expand the data, so instead of having each event as a row, the event would have N rows (equal to the length of the costs/qalys to discount passed). This means more processing of the results data would be needed in order for it to provide the correct results.

If the cycle lists are used, then it is expected the user will declare as well the name of the variable pasted with cycle_l and cycle_starttime (e.g., c_default_cycle_l and c_default_cycle_starttime) to ensure the discounting can be computed using cycles, with cycle_l being the cycle length, and cycle_starttime being the starting time in which the variable started counting. Optionally, max_cycles must also be added (if no maximum number of cycles, it should be set equal to NA).

debug = TRUE will export a log file with the timestamp up the error in the main working directory. Note that using this mode without modify_item or modify_item_seq may lead to inaccuracies if assignments are done in non-standard ways, as the AST may not catch all the relevant assignments (e.g., an assigment like assign(paste("x_",i),5) in a loop will not be identified).

continue_on_error will skip the current simulation (so it won't continue for the rest of patient-arms) if TRUE. Note that this will make the progress bar not correct, as a set of patients that were expected to be run is not.

Value

A list of data frames with the simulation results

Examples

```
common_all_inputs <-add_item(
util.sick = 0.8,
util.sicker = 0.5,
cost.sick = 3000,
cost.sicker = 7000,
cost.int = 1000,
coef_noint = log(0.2),
HR_int = 0.8,
drc = 0.035, #different values than what's assumed by default
drq = 0.035,
random_seed_sicker_i = sample.int(100000,5,replace = FALSE)
)

common_pt_inputs <- add_item(death= max(0.0000001,rnorm(n=1, mean=12, sd=3)))

unique_pt_inputs <- add_item(fl.sick = 1,
                             q_default = util.sick,
                             c_default = cost.sick + if(arm=="int"){cost.int}else{0})

init_event_list <-
add_tte(arm=c("noint","int"), evts = c("sick","sicker","death") ,input={
  sick <- 0
  sicker <- draw_tte(1,dist="exp",
   coef1=coef_noint, beta_tx = ifelse(arm=="int",HR_int,1),
    seed = random_seed_sicker_i[i])

})

evt_react_list <-
add_reactevt(name_evt = "sick",
             input = {}) |>
  add_reactevt(name_evt = "sicker",
               input = {
                 q_default <- util.sicker
                 c_default <- cost.sicker + if(arm=="int"){cost.int}else{0}
                 fl.sick <- 0
               }) |>
  add_reactevt(name_evt = "death",
               input = {
                 q_default <- 0
                 c_default <- 0
                 curtime <- Inf
               })

util_ongoing <- "q_default"
cost_ongoing <- "c_default"


run_sim(arm_list=c("int","noint"),
common_all_inputs = common_all_inputs,
common_pt_inputs = common_pt_inputs,
```

```
unique_pt_inputs = unique_pt_inputs,
init_event_list = init_event_list,
evt_react_list = evt_react_list,
util_ongoing_list = util_ongoing,
cost_ongoing_list = cost_ongoing,
npats = 2,
n_sim = 1,
psa_bool = FALSE,
ipd = 1)
```

`run_sim_parallel` *Run simulations in parallel mode (at the simulation level)*

Description

Run simulations in parallel mode (at the simulation level)

Usage

```
run_sim_parallel(
  arm_list = c("int", "noint"),
  sensitivity_inputs = NULL,
  common_all_inputs = NULL,
  common_pt_inputs = NULL,
  unique_pt_inputs = NULL,
  init_event_list = NULL,
  evt_react_list = evt_react_list,
  util_ongoing_list = NULL,
  util_instant_list = NULL,
  util_cycle_list = NULL,
  cost_ongoing_list = NULL,
  cost_instant_list = NULL,
  cost_cycle_list = NULL,
  other_ongoing_list = NULL,
  other_instant_list = NULL,
  npats = 500,
  n_sim = 1,
  psa_bool = NULL,
  sensitivity_bool = FALSE,
  sensitivity_names = NULL,
  n_sensitivity = 1,
  ncores = 1,
  input_out = character(),
  ipd = 1,
  constrained = FALSE,
  timed_freq = NULL,
  debug = FALSE,
```

```
  accum_backwards = FALSE,
  continue_on_error = FALSE,
  seed = NULL
)
```

Arguments

`arm_list` A vector of the names of the interventions evaluated in the simulation

`sensitivity_inputs` A list of sensitivity inputs that do not change within a sensitivity in a similar fashion to common_all_inputs, etc

`common_all_inputs` A list of inputs common across patients that do not change within a simulation

`common_pt_inputs` A list of inputs that change across patients but are not affected by the intervention

`unique_pt_inputs` A list of inputs that change across each intervention

`init_event_list` A list of initial events and event times. If no initial events are given, a "Start" event at time 0 is created automatically

`evt_react_list` A list of event reactions

`util_ongoing_list` Vector of QALY named variables that are accrued at an ongoing basis (discounted using drq)

`util_instant_list` Vector of QALY named variables that are accrued instantaneously at an event (discounted using drq)

`util_cycle_list` Vector of QALY named variables that are accrued in cycles (discounted using drq)

`cost_ongoing_list` Vector of cost named variables that are accrued at an ongoing basis (discounted using drc)

`cost_instant_list` Vector of cost named variables that are accrued instantaneously at an event (discounted using drc)

`cost_cycle_list` Vector of cost named variables that are accrued in cycles (discounted using drc)

`other_ongoing_list` Vector of other named variables that are accrued at an ongoing basis (discounted using drq)

`other_instant_list` Vector of other named variables that are accrued instantaneously at an event (discounted using drq)

`npats` The number of patients to be simulated (it will simulate npats * length(arm_list))

`n_sim`	The number of simulations to run per sensitivity
`psa_bool`	A boolean to determine if PSA should be conducted. If n_sim > 1 and psa_bool = FALSE, the differences between simulations will be due to sampling
`sensitivity_bool`	A boolean to determine if Scenarios/DSA should be conducted.
`sensitivity_names`	A vector of scenario/DSA names that can be used to select the right sensitivity (e.g., c("Scenario_1", "Scenario_2")). The parameter "sens_name_used" is created from it which corresponds to the one being used for each iteration.
`n_sensitivity`	Number of sensitivity analysis (DSA or Scenarios) to run. It will be interacted with sensitivity_names argument if not null (n_sensitivityitivity = n_sensitivity * length(sensitivity_names)). For DSA, it should be as many parameters as there are. For scenario, it should be 1.
`ncores`	The number of cores to use for parallel computing
`input_out`	A vector of variables to be returned in the output data frame
`ipd`	Integer taking value 0 if no IPD data returned, 1 for full IPD data returned, and 2 IPD data but aggregating events
`constrained`	Boolean, FALSE by default, which runs the simulation with patients not interacting with each other, TRUE if resources are shared within an arm (allows constrained resources)
`timed_freq`	If NULL, it does not produce any timed outputs. Otherwise should be a number (e.g., every 1 year)
`debug`	If TRUE, will generate a log file
`accum_backwards`	If TRUE, the ongoing accumulators will count backwards (i.e., the current value is applied until the previous update). If FALSE, the current value is applied between the current event and the next time it is updated. Recommended for resource-constrained models where queue waits make future event times unpredictable. With TRUE, ongoing values are computed at each event for the interval just completed, avoiding prospective errors from `adj_val()`.
`continue_on_error`	If TRUE, on error at patient stage will attempt to continue to the next simulation (only works if n_sim and/or n_sensitivity are > 1, not at the patient level)
`seed`	Starting seed to be used for the whole analysis. If null, it's set to 1 by default.

Details

This function is slightly different from `run_sim`. `run_sim` allows to run single-core. `run_sim_parallel` allows to use multiple-core at the simulation level, making it more efficient for a large number of simulations relative to `run_sim` (e.g., for PSA).

Event ties are processed in the order declared within the `init_event_list` argument (`evts` argument within the first sublist of that object). To do so, the program automatically adds a sequence from to 0 to the (number of events - 1) times 1e-10 to add to the event times when selecting the event with minimum time. This time has been selected as it's relatively small yet not so small as to be ignored by which.min (see .Machine for more details)

A list of protected objects that should not be used by the user as input names or in the global environment to avoid the risk of overwriting them is as follows: c("arm", "arm_list", "categories_for_export", "cur_evtlist", "curtime", "evt", "i", "prevtime", "sens", "simulation", "sens_name_used","list_env","uc_lists","npats","ipd").

The engine uses the L'Ecuyer-CMRG for the random number generator. Note that if ncores > 1, then results per simulation will only be exactly replicable if using run_sim_parallel (as seeds are automatically transformed to be seven integer seeds -i.e, L'Ecuyer-CMRG seeds-) Note that the random seeds are set to be unique in their category (i.e., at patient level, patient-arm level, etc.)

If no `drc` or `drq` parameters are passed within `sensitivity` or `common_all` input lists, these are assigned a default value 0.03 for discounting costs, QALYs and others.

Ongoing items will look backward to the last time updated when performing the discounting and accumulation. This means that the user does not necessarily need to keep updating the value, but only add it when the value changes looking forward (e.g., o_q = utility at event 1, at event 2 utility does not change, but at event 3 it does, so we want to make sure to add o_q = utility at event 3 before updating utility. The program will automatically look back until event 1). Note that in previous versions of the package backward was the default, and now this has switched to forward.

The requirement to use `modify_item` if using `accum_backwards = TRUE`, is no longer the case thanks to a new method using active bindings, so it can be used normally.

If the `cycle` lists are used, then it is expected the user will declare as well the name of the variable pasted with `cycle_l` and `cycle_starttime` (e.g., c_default_cycle_l and c_default_cycle_starttime) to ensure the discounting can be computed using cycles, with cycle_l being the cycle length, and cycle_starttime being the starting time in which the variable started counting. Optionally, `max_cycles` must also be added (if no maximum number of cycles, it should be set equal to NA).

`debug = TRUE` will export a log file with the timestamp up the error in the main working directory. Note that using this mode without modify_item or modify_item_seq may lead to inaccuracies if assignments are done in non-standard ways, as the AST may not catch all the relevant assignments (e.g., an assigment like assign(paste("x_",i),5) in a loop will not be identified).

If `continue_on_error` is set to FALSE, it will only export analysis level inputs due to the parallel engine (use single-engine for those inputs) `continue_on_error` will skip the current simulation (so it won't continue for the rest of patient-arms) if TRUE. Note that this will make the progress bar not correct, as a set of patients that were expected to be run is not.

Value

A list of lists with the analysis results

Examples

```
common_all_inputs <-add_item(
util.sick = 0.8,
util.sicker = 0.5,
cost.sick = 3000,
cost.sicker = 7000,
cost.int = 1000,
coef_noint = log(0.2),
HR_int = 0.8,
drc = 0.035, #different values than what's assumed by default
drq = 0.035,
random_seed_sicker_i = sample.int(100000,5,replace = FALSE)
```

```
)

common_pt_inputs <- add_item(death= max(0.0000001,rnorm(n=1, mean=12, sd=3)))

unique_pt_inputs <- add_item(fl.sick = 1,
                             q_default = util.sick,
                             c_default = cost.sick + if(arm=="int"){cost.int}else{0})

init_event_list <-
add_tte(arm=c("noint","int"), evts = c("sick","sicker","death") ,input={
  sick <- 0
  sicker <- draw_tte(1,dist="exp",
   coef1=coef_noint, beta_tx = ifelse(arm=="int",HR_int,1),
   seed = random_seed_sicker_i[i])

})

evt_react_list <-
add_reactevt(name_evt = "sick",
             input = {}) |>
  add_reactevt(name_evt = "sicker",
               input = {
                 q_default <- util.sicker
                 c_default <- cost.sicker + if(arm=="int"){cost.int}else{0}
                 fl.sick <- 0
               }) |>
  add_reactevt(name_evt = "death",
               input = {
                 q_default <- 0
                 c_default <- 0
                 curtime <- Inf
               })

util_ongoing <- "q_default"
cost_ongoing <- "c_default"


run_sim_parallel(arm_list=c("int","noint"),
common_all_inputs = common_all_inputs,
common_pt_inputs = common_pt_inputs,
unique_pt_inputs = unique_pt_inputs,
init_event_list = init_event_list,
evt_react_list = evt_react_list,
util_ongoing_list = util_ongoing,
cost_ongoing_list = cost_ongoing,
npats = 2,
n_sim = 1,
psa_bool = FALSE,
ipd = 1,
ncores = 1)
```

scale_remaining_time *Scale remaining time to an event by a hazard ratio*

Description

Computes `curtime + (event_time - curtime) / hr`. Use inside event reactions to shrink or expand the remaining time to an event.

Usage

```
scale_remaining_time(event_name, hr)
```

Arguments

`event_name`	Character. Name of the event whose time to scale.
`hr`	Numeric greater than 0. Hazard ratio to apply. Values below 1 extend time; values above 1 shorten it.

Value

Numeric: the new absolute event time.

seize *Seize a discrete resource*

Description

Convenience wrapper around `resource$attempt_block()`. Reads `i` and `curtime` from the calling environment automatically.

Usage

```
seize(resource, amount = 1L, priority = 1L, ...)
```

Arguments

`resource`	A `resource_discrete` object.
`amount`	Integer. Number of resource units to seize (default `1L`).
`priority`	Integer. Queue priority for the patient (default `1L`). Lower values are higher priority.
`...`	Not used. Passing `seize_all()`-specific arguments here (e.g., `accum_queue`, `force_unblock`, `policy`) will raise an error.

Value

`TRUE` if acquired, `FALSE` if queued, `NA` if rejected (queue full, only when `max_queue` is set on the resource).

`seize_all` *Seize multiple discrete resources atomically*

Description

Attempts to acquire a list of resources in a single C++ call, avoiding per-resource R-to-C++ round trips.

Usage

```
seize_all(
  resources,
  policy = "all_or_none",
  amounts = NULL,
  priorities = NULL,
  force_unblock = FALSE,
  accum_queue = TRUE
)
```

Arguments

`resources`	A list of `resource_discrete` objects.
`policy`	`"all_or_none"` (default) – acquires all only if all are available; queues for the first bottleneck without holding any others. `"sequential"` – acquires in list order, holding partial acquisitions; user is responsible for avoiding deadlocks.
`amounts`	Integer vector of units per resource (default `1L` for each).
`priorities`	Integer vector of priorities per resource (default `1L`).
`force_unblock`	Logical (default `FALSE`). When `TRUE` and all resources have sufficient capacity but the patient is not first in queue on some resources (deadlock), forcibly moves the patient to the front of those queues and acquires all resources. Has no effect when capacity is genuinely insufficient.
`accum_queue`	Logical (default `TRUE`). When `FALSE` and the patient already has a queue entry on a bottleneck resource from a previous failed attempt, skips adding a new entry (existing entry serves). Prevents phantom queue accumulation on retries.

Value

`TRUE` if all acquired, `FALSE` if queued for a bottleneck resource, `NA` if rejected.

sens_iterator — *Create an iterator based on sens of the current iteration within a scenario (DSA)*

Description

Create an iterator based on sens of the current iteration within a scenario (DSA)

Usage

```
sens_iterator(sens, n_sensitivity)
```

Arguments

`sens`	current analysis iterator
`n_sensitivity`	total number of analyses to be run

Details

In a situation like a DSA, where two (low and high) scenarios are run, sens will go from 1 to n_sensitivity*2. However, this is not ideal as the parameter selector may depend on knowing the parameter order (i.e., 1, 2, 3...), which means resetting the counter back to 1 once sens reaches n_sensitivity (or any multiple of n_sensitivity) is needed.

Value

Integer iterator based on the number of sensitivity analyses being run and the total iterator

Examples

```
sens_iterator(5,20)
sens_iterator(25,20)
```

shared_decr — *Decrement a shared counter*

Description

Subtracts `delta` from the value of a `shared_input` object and returns the new value.

Usage

```
shared_decr(obj, delta = 1)
```

Arguments

`obj`	A `shared_input` object (constrained / shared mode).
`delta`	Numeric. Amount to subtract (default `1`).

Value

The new value after decrementing.

`shared_incr` *Increment a shared counter*

Description

Adds `delta` to the value of a `shared_input` object and returns the new value.

Usage

```
shared_incr(obj, delta = 1)
```

Arguments

`obj`	A `shared_input` object (constrained / shared mode).
`delta`	Numeric. Amount to add (default `1`).

Value

The new value after incrementing.

`shared_input` *Shared input object*

Description

Constructor for a lightweight "shared or immutable" value holder.

Usage

```
shared_input(expr, constrained = NULL)
```

Arguments

`expr`	A value or expression to initialize the shared input with. The expression is evaluated immediately.
`constrained`	Logical. If `TRUE`, creates a shared environment-backed object. If `FALSE`, creates an immutable copy-on-modify object. If `NULL` (default), the function looks up `constrained` in the calling environment; only an explicit `TRUE` enables shared mode.

Details

`shared_input()` produces a simple object that wraps a value with controlled mutability semantics. It can operate in two distinct modes:

- **Immutable (non-shared)**: every modification produces a fresh, independent copy of the object (safe for parallel or functional code).
- **Shared (constrained)**: the object's value is stored in a common environment shared across all aliases (by-reference semantics). This allows coordinated updates across multiple handles.

The mode is determined either by the explicit argument `constrained`, or by inheriting the value of a `constrained` variable in the parent frame.

- In **immutable mode**, each wrapper stores its value in closures (`make_val()`) and is fully copy-on-modify. No references are shared.
- In **shared mode**, all wrappers produced by `$modify()` or direct aliasing point to the same underlying environment (`state`). This means updating one updates all aliases until a `$clone()` or `$reset()` breaks the link.

The underlying `state` environments are internal. Users should rely only on the public methods above.

Note: if the stored value itself is a reference type (e.g., environment, external pointer, R6 object), those internal references remain shared regardless of mode, following normal R semantics.

Value

An object of class `shared_input_val` (immutable mode) or `shared_input_env` (shared mode), both inheriting from class `"shared_input"`. Each instance exposes the following user methods:

$value() Returns the current stored value.

$modify(new_v) In immutable mode: returns a new independent wrapper with updated value. In shared mode: updates the shared value by reference and returns a new wrapper pointing to the same shared state.

$clone() Returns a deep copy (independent wrapper and independent internal state). Subsequent modifications on clones do not affect the original object or its aliases.

$reset() Returns a new wrapper whose value is restored to the original initialization value. In both modes this creates an independent fresh state.

$fork(n) Creates `n` independent deep clones as a list. Useful for generating multiple isolated copies quickly.

Examples

```
# --- Immutable (default) mode ---
a <- shared_input(5)
a$value()                # 5
a2 <- a$modify(a$value() + 7)
a$value()                # 5
a2$value()               # 12
```

```
# Cloning and resetting
a3 <- a2$clone()
a4 <- a2$reset()
a3$value(); a4$value()    # 12, 5

# Forking
forks <- a$fork(3)
vapply(forks, function(x) x$value(), numeric(1))

# --- Shared (constrained) mode ---
constrained <- TRUE
b1 <- shared_input(10)
b2 <- b1        # alias (same state)
b1$modify(11)
b1$value(); b2$value()  # both 11

b3 <- b1$clone()
b1$modify(99)
b1$value(); b3$value()  # 99, 11

# Reset breaks sharing
b4 <- b1$reset()
b4$value()              # 10
```

---

summary_results_det *Deterministic results for a specific treatment*

Description

Deterministic results for a specific treatment

Usage

```
summary_results_det(
  out = results[[1]][[1]],
  arm = NULL,
  wtp = 50000,
  sens = 1,
  sim = 1
)
```

Arguments

out	The list object returned by `run_sim()`, or a subset of it (e.g., `results[[1]][[1]]`). When the full results object or a simulation list is passed, `sens` and `sim` are used to select the appropriate element.
arm	The reference treatment for calculation of incremental outcomes
wtp	Willingness to pay to have INMB

sens	Integer indicating which sensitivity analysis to use when out is the full results object or a simulation list. Defaults to 1.
sim	Integer indicating which simulation to use when out is the full results object or a simulation list. Defaults to 1.

Value

A dataframe with absolute costs, LYs, QALYs, and ICER and ICUR for each intervention

Examples

```
res <- list(list(list(sensitivity_name = "", arm_list = c("int", "noint"
), total_lys = c(int = 9.04687362556945, noint = 9.04687362556945
), total_qalys = c(int = 6.20743830697466, noint = 6.18115138126336
), total_costs = c(int = 49921.6357486899, noint = 41225.2544659378
), total_lys_undisc = c(int = 10.8986618377039, noint = 10.8986618377039
), total_qalys_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), total_costs_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), c_default = c(int = 49921.6357486899, noint = 41225.2544659378
), c_default_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), q_default = c(int = 6.20743830697466, noint = 6.18115138126336
), q_default_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), merged_df = list(simulation = 1L, sensitivity = 1L))))


summary_results_det(res[[1]][[1]], arm="int")
summary_results_det(res, arm="int", sens=1, sim=1)
```

---

summary_results_sens *Summary of sensitivity outputs for a treatment*

Description

Summary of sensitivity outputs for a treatment

Usage

```
summary_results_sens(out = results, arm = NULL, wtp = 50000)
```

Arguments

out	The list object returned by run_sim()
arm	The reference treatment for calculation of incremental outcomes
wtp	Willingness to pay to have INMB

Value

A data frame with each sensitivity output per arm

Examples

```
res <- list(list(list(sensitivity_name = "", arm_list = c("int", "noint"
), total_lys = c(int = 9.04687362556945, noint = 9.04687362556945
), total_qalys = c(int = 6.20743830697466, noint = 6.18115138126336
), total_costs = c(int = 49921.6357486899, noint = 41225.2544659378
), total_lys_undisc = c(int = 10.8986618377039, noint = 10.8986618377039
), total_qalys_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), total_costs_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), c_default = c(int = 49921.6357486899, noint = 41225.2544659378
), c_default_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), q_default = c(int = 6.20743830697466, noint = 6.18115138126336
), q_default_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), merged_df = list(simulation = 1L, sensitivity = 1L))))


summary_results_sens(res,arm="int")
```

---

summary_results_sim *Summary of PSA outputs for a treatment*

Description

Summary of PSA outputs for a treatment

Usage

```
summary_results_sim(out = results[[1]], arm = NULL, wtp = 50000, sens = 1)
```

Arguments

out	The list object returned by run_sim(), or a subset of it (e.g., results[[1]]). When the full results object is passed, sens is used to select the appropriate sensitivity analysis.
arm	The reference treatment for calculation of incremental outcomes
wtp	Willingness to pay to have INMB
sens	Integer indicating which sensitivity analysis to use when out is the full results object. Defaults to 1.

Value

A data frame with mean and 95% CI of absolute costs, LYs, QALYs, ICER and ICUR for each intervention from the PSA samples

Examples

```
res <- list(list(list(sensitivity_name = "", arm_list = c("int", "noint"
), total_lys = c(int = 9.04687362556945, noint = 9.04687362556945
), total_qalys = c(int = 6.20743830697466, noint = 6.18115138126336
), total_costs = c(int = 49921.6357486899, noint = 41225.2544659378
), total_lys_undisc = c(int = 10.8986618377039, noint = 10.8986618377039
), total_qalys_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), total_costs_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), c_default = c(int = 49921.6357486899, noint = 41225.2544659378
), c_default_undisc = c(int = 59831.3573929783, noint = 49293.1025437205
), q_default = c(int = 6.20743830697466, noint = 6.18115138126336
), q_default_undisc = c(int = 7.50117621700097, noint = 7.47414569286751
), merged_df = list(simulation = 1L, sensitivity = 1L))))


summary_results_sim(res[[1]], arm="int")
summary_results_sim(res, arm="int", sens=1)
```

---

tte.df *Example TTE IPD data*

Description

An example of TTE IPD data for the example_ipd file

Usage

```
tte.df
```

Format

`tte.df`:

A data frame with 1000 rows and 8 columns:

USUBJID Patient ID

ARMCD, ARM Arm code and variables

PARAMCD, PARAM Parameter

AVAL, AVALCD Values of interest

CNSR Censored observation?

Source

Simulated through FlexsurvPlus package using sim_adtte(seed = 821, rho = 0, beta_1a = log(0.6), beta_1b = log(0.6), beta_pd = log(0.2)) validate_model *Validate model inputs before running*

Description

Performs static checks on the model inputs without running the simulation. Catches common configuration errors early.

Usage

```
validate_model(
  arm_list = c("int", "noint"),
  init_event_list = NULL,
  evt_react_list = NULL,
  util_ongoing_list = NULL,
  util_instant_list = NULL,
  util_cycle_list = NULL,
  cost_ongoing_list = NULL,
  cost_instant_list = NULL,
  cost_cycle_list = NULL,
  other_ongoing_list = NULL,
  other_instant_list = NULL,
  npats = 500,
  n_sim = 1,
  psa_bool = NULL,
  seed = NULL
)
```

Arguments

`arm_list` Character vector of intervention names.

`init_event_list` List of initial events and event times (output of `add_tte()`).

`evt_react_list` List of event reactions (output of `add_reactevt()`).

`util_ongoing_list` Character vector of ongoing QALY variable names.

`util_instant_list` Character vector of instant QALY variable names.

`util_cycle_list` Character vector of cycle QALY variable names.

`cost_ongoing_list` Character vector of ongoing cost variable names.

`cost_instant_list` Character vector of instant cost variable names.

`cost_cycle_list` Character vector of cycle cost variable names.

other_ongoing_list
: Character vector of other ongoing variable names.

other_instant_list
: Character vector of other instant variable names.

npats — Integer. Number of patients per simulation.

n_sim — Integer. Number of simulations.

psa_bool — Logical. Whether PSA mode is intended.

seed — Numeric. Random seed.

Value

Invisibly `TRUE` if all checks pass. Prints a summary of checks.

---

waning_hr — *Compute effective hazard ratio with treatment waning*

Description

Returns the effective hazard ratio at a given time on treatment, accounting for waning of treatment effect.

Usage

```
waning_hr(base_hr, time_on_tx, start, duration, type = "linear")
```

Arguments

`base_hr`	Numeric. Initial treatment HR (< 1 means benefit).
`time_on_tx`	Numeric. Time the patient has been on treatment.
`start`	Numeric. Time at which waning begins (HR = base_hr before this).
`duration`	Numeric. Duration of the waning period (HR reaches 1 at start + duration).
`type`	Character. Waning function: `"linear"` (default), `"exponential"`, or `"instant"`.

Value

Numeric scalar: effective HR at `time_on_tx`.

Index