

Package ‘TrialSimulator’

September 4, 2026

Type Package

Title Clinical Trial Simulator

Version 1.35.8

Description Simulate phase II and/or phase III clinical trials. It supports various types of end-points and adaptive strategies. Tools for carrying out graphical testing procedure and combination test under group sequential design are also provided.

License MIT + file LICENSE

Encoding UTF-8

Imports base64enc, cli, dplyr, emmeans, ggplot2, gMCPLite, htmltools, mvtnorm, R6, Rcpp, rlang, rpact, rstudioapi, survival, utils

LinkingTo Rcpp

RoxygenNote 7.3.3

Suggests data.table, DoseFinding, gsDesign, kableExtra, knitr, mirai, PWEALL, PWEXP, rmarkdown, simdata, survminer, testthat (>= 3.0.0)

VignetteBuilder knitr

URL <https://zhangh12.github.io/TrialSimulator/>

BugReports <https://github.com/zhangh12/TrialSimulator/issues>

Depends R (>= 4.1.0)

Config/testthat/edition 3

NeedsCompilation yes

Author Han Zhang [cre, aut]

Maintainer Han Zhang <zhangh.ustc@gmail.com>

Repository CRAN

Date/Publication 2026-09-04 17:00:02 UTC

Contents

add_arms	3
arm	4
Arms	5
calendarTime	8
controller	9
Controllers	10
CorrelatedPfsAndOs2	12
CorrelatedPfsAndOs3	14
CorrelatedPfsAndOs4	15
crossover	16
doNothing	17
DynamicRNGFunction	18
endpoint	19
Endpoints	22
enrollment	26
eventNumber	27
expandRegimen	28
fitCoxph	29
fitFarringtonManning	30
fitLinear	31
fitLogistic	32
fitLogrank	33
getAdaptiveDesignOutput	34
getConditionalPowerOutput	35
getDoseRangingOutput	35
getFixedDesignOutput	35
GraphicalTesting	36
GroupSequentialTest	44
listener	48
Listeners	49
milestone	52
Milestones	53
PiecewiseConstantExponentialRNG	55
plot.milestone_time_summary	56
plot.three_state_model	57
qPiecewiseExponential	57
rconst	59
regimen	60
Regimens	61
remove_arms	64
resize	65
set_duration	66
solveMixtureExponentialDistribution	66
solvePiecewiseConstantExponentialDistribution	68
solveThreeStateModel	69
StaggeredRecruiter	70

stop_followup	72
summarizeDataFrame	73
summarizeMilestoneTime	74
trial	75
Trials	78
update_accrual_rate	101
update_generator	102
update_milestone	103
update_sample_ratio	104
weibullDropout	105

add_arms	<i>Adding One or More Arms to a Trial</i>
----------	---

Description

add one or more arms to a trial. This function can be used in two scenarios: (1) adding arms right after a trial is created (i.e., `trial(...)`). `sample_ratio` and arms added through ... must be of same length; (2) adding arms to an existing trial in action functions of milestones.

This is a user-friendly wrapper of the member function of trial, i.e., `Trials$add_arms()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

The trial captures an independent deep copy of every arm it registers. Subsequent changes to the original arm object do not affect the trial, and adaptations within the trial (e.g., `update_generator()`) do not modify the original arm. Complete the configuration of an arm before registering it. The copy covers the arm and endpoint objects themselves; a mutable environment or R6 object captured by a generator function is shared by design of R closures and is not isolated.

Usage

```
add_arms(trial, sample_ratio, ...)
```

Arguments

trial	a trial object returned by <code>trial()</code> .
sample_ratio	integer vector. Sample ratio for permuted block randomization. It will be appended to existing sample ratio in the trial.
...	one or more objects returned from <code>arm()</code> . Randomization is carried out with updated sample ratio of newly added arm. Note that we can run <code>Trials\$add_arm(sample_ratio1, arm1)</code> followed by <code>add_arms(trial, sample_ratio2, arm2)</code> . We would expected similar result with <code>add_arms(trial, c(sample_ratio1, sample_ratio2), arm1, arm2)</code> . Note that these two methods won't return exactly the same trial because randomization queue are generated twice in the first approach but only once in the second approach. But statistically, they are equivalent and of the same distribution.

Value

no return value, called for its side effect of updating `trial`.

arm	<i>Define an Arm</i>
-----	----------------------

Description

Define an arm in a trial. This is a user-friendly wrapper for the class constructor `Arms$new()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
arm(name, ...)
```

Arguments

name	character. Name of arm, which is the arm's label in generated trial data, i.e., the one retrieved by calling <code>Trials\$get_locked_data()</code> in action functions. It must not contain '@' or ';', which are reserved for encoding the treatment switching history (regimen_trajectory) of trials with a regimen.
...	subset condition that is compatible with <code>dplyr::filter</code> . This can be used to specify inclusion criteria of an arm. By default it is not specified, i.e. all data generated by the generator will be used as trial data. More than one conditions can be specified in

Value

an object of R6 class `Arms`, representing an arm of a trial.

Examples

```
risk <- data.frame(
  end_time = c(1, 10, 26.0, 52.0),
  piecewise_risk = c(1, 1.01, 0.381, 0.150) * exp(-3.01)
)

pfs <- endpoint(name = 'pfs', type='tte',
  generator = PiecewiseConstantExponentialRNG,
  risk = risk, endpoint_name = 'pfs')

orr <- endpoint(
  name = 'orr', type = 'non-tte',
  readout = c(orr = 2), generator = rbinom,
  size = 1, prob = .4)

placebo <- arm(name = 'pbo')
```

```

placebo$add_endpoints(pfs, orr)

## try to generate some data from the arm
## it is NOT a recommended way to use the package in simulation
head(placebo$get_endpoints()[[1]]$get_generator()(n = 1e3))

## get name of endpoints in the arm
## for illustration only, NOT recommended
placebo$get_endpoints()[[2]]$get_name()

## run it in console to get summary report
## It is the recommended way to view an arm
placebo

```

Arms

Class of Arm

Description

Create a class of arm.

Public methods in this R6 class are used in developing this package. Thus, we have to export the whole R6 class which exposures all public methods. However, only the public methods in the list below are useful to end users.

- `$add_endpoints()` add one or multiple endpoints to the arm.
- `$print()` print a summary report of the arm based on example data of its endpoints.

In addition, `$generate_data()` and `$get_endpoints()` are for exploratory purpose only. They appear in examples and vignettes to help users understand how this class works, but they are not needed at all in formal simulation once users know how to use this package.

Internal machinery. The remaining public methods (`$get_name()`, `$get_endpoints_name()`, `$has_endpoint()` and `$update_endpoint_generator()`) are public only because they are invoked on an arm object by other components of the package (trials), which R6 cannot grant through private members. Users should not call them directly.

Value

an R6Class generator object; use `arm()` to create an arm.

Methods

Public methods:

- [Arms\\$new\(\)](#)
- [Arms\\$add_endpoints\(\)](#)
- [Arms\\$get_name\(\)](#)
- [Arms\\$has_endpoint\(\)](#)

- `Arms$get_endpoints()`
- `Arms$get_endpoints_name()`
- `Arms$update_endpoint_generator()`
- `Arms$generate_data()`
- `Arms$print()`
- `Arms$clone()`

Method `new()`: initialize an arm

Usage:

```
Arms$new(name, ...)
```

Arguments:

`name` name of arm, which is the arm's label in generated data. It must not contain '@' or ';', which are reserved for encoding the treatment switching history (regimen_trajectory) of trials with a regimen.

`...` subset condition that is compatible with `dplyr::filter`. This can be used to specify inclusion criteria of an arm. By default it is not specified, i.e. all data generated by the generator will be used as trial data. More than one conditions can be specified in `...`

Method `add_endpoints()`: add one or multiple endpoints to the arm.

Usage:

```
Arms$add_endpoints(...)
```

Arguments:

`...` one or more objects returned from `endpoint()`.

Examples:

```
a <- arm(name = 'trt')
x <- endpoint(name = 'x', type = 'tte',
              generator = rexp) # median = log(2)/1 = 0.7
y <- endpoint(name = 'y', type = 'non-tte', readout = c(y = 0),
              generator = rnorm, sd = 1.4, mean = 0.7)
```

```
a$add_endpoints(y, x)
```

```
## run it in console to see the summary report
```

```
a
```

```
print(a) # use the print method
```

Method `get_name()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return name of arm.

Usage:

```
Arms$get_name()
```

Method `has_endpoint()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

check if the arm has any endpoint. Return TRUE or FALSE.

Usage:

```
Arms$has_endpoint()
```

Method `get_endpoints()`: return a list of endpoints in the arm.

Usage:

```
Arms$get_endpoints()
```

Method `get_endpoints_name()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return name of endpoints registered to the arm.

Usage:

```
Arms$get_endpoints_name()
```

Method `update_endpoint_generator()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

update generator of an endpoint object

Usage:

```
Arms$update_endpoint_generator(endpoint_name, generator, ...)
```

Arguments:

`endpoint_name` character. A vector of endpoint names whose generator is updated.

`generator` a random number generation (RNG) function. See `generator` of `endpoint()`.

... optional arguments for generator.

Method `generate_data()`: generate arm data.

Usage:

```
Arms$generate_data(n_patients_in_arm)
```

Arguments:

`n_patients_in_arm` integer. Number of patients randomized to the arm.

Method `print()`: print an arm.

Usage:

```
Arms$print(categorical_vars = NULL)
```

Arguments:

`categorical_vars` character vector of categorical variables. This can be used to specify variables with limited distinct (numeric) values as categorical variables in summary report.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Instead of using Arms$new(), please use arm(), a user-friendly
# wrapper. See examples in ?arm

## -----
## Method `Arms$add_endpoints`
## -----

a <- arm(name = 'trt')
x <- endpoint(name = 'x', type = 'tte',
              generator = rexp) # median = log(2)/1 = 0.7
y <- endpoint(name = 'y', type = 'non-tte', readout = c(y = 0),
              generator = rnorm, sd = 1.4, mean = 0.7)

a$add_endpoints(y, x)

## run it in console to see the summary report
a

print(a) # use the print method
```

calendarTime

Triggering Condition by Calendar Time

Description

Define a condition to trigger trial milestone by calendar time. The milestone will be triggered when a trial has been running for at least the specified duration since the first patient is enrolled. It can be used combined with conditions specified by [enrollment](#) and [eventNumber](#).

Refer to the [vignette](#) to learn how to define milestones when performing simulation using TrialSimulator.

Usage

```
calendarTime(time)
```

Arguments

time numeric. Calendar time to trigger a milestone of a trial.

Value

an object of class ‘Condition’

Examples

```
milestone(name = 'end of trial', when = calendarTime(time = 12))
```

controller*Define a Controller*

Description

Define a controller of a trial. This is a user-friendly wrapper for the class constructor `Controller$new()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

`TrialSimulator` uses a controller to coordinate a trial object and a listener object to run simulations, in which the trial object defines endpoints, arms, and other settings of a trial, while the listener object monitors trials to triggered pre-defined milestones and execute action functions. See vignettes of this package for more examples.

Usage

```
controller(trial, listener)
```

Arguments

<code>trial</code>	an object returned from <code>trial()</code> .
<code>listener</code>	an object returned from <code>listener()</code> .

Value

an object of R6 class `Controllers`, executing simulation of a trial.

Examples

```
# a minimum, meaningful, and executable example,
# where a randomized trial with two arms is simulated and analyzed.

control <- arm(name = 'control arm')
active <- arm(name = 'active arm')

pfs_in_control <- endpoint(name = 'PFS', type = 'tte',
                           generator = rexp, rate = log(2) / 5)
control$add_endpoints(pfs_in_control)

pfs_in_active <- endpoint(name = 'PFS', type = 'tte',
                          generator = rexp, rate = log(2) / 6)
active$add_endpoints(pfs_in_active)

accrual_rate <- data.frame(end_time = c(10, Inf),
                           piecewise_rate = c(30, 50))

trial <- trial(name = 'trial',
               n_patients = 1000,
               duration = 40,
               enroller = StaggeredRecruiter,
               accrual_rate = accrual_rate,
```

```

        dropout = rweibull, shape = 2, scale = 38)

trial$add_arms(sample_ratio = c(1, 1), control, active)

action_at_final <- function(trial){
  locked_data <- trial$get_locked_data('final analysis')
  fitLogrank(Surv(PFS, PFS_event) ~ arm, placebo = 'control arm',
             data = locked_data, alternative = 'less')
  invisible(NULL)
}

final <- milestone(name = 'final analysis',
                  action = action_at_final,
                  when = calendarTime(time = 40))

listener <- listener()
listener$add_milestones(final)

controller <- controller(trial, listener)
controller$run(n = 1)

```

Controllers

Class of Controller

Description

Create a class of controller to run a trial.

Public methods in this R6 class are used in developing this package. Thus, we have to export the whole R6 class which exposures all public methods. However, only the public methods in the list below are useful to end users.

- `$run()` run trial simulation, sequentially or in parallel. It cannot be called twice on a controller unless `$reset()` is called in between.
- `$get_output()` return a data frame of all outputs saved during simulation.
- `$reset()` reset the trial and listener registered to the controller before starting a new simulation with `$run()`.

Value

an R6Class generator object; use `controller()` to create a controller.

Methods

Public methods:

- `Controllers$new()`
- `Controllers$reset()`

- `Controllers$get_output()`
- `Controllers$run()`
- `Controllers$clone()`

Method `new()`: initialize a controller of the trial

Usage:

```
Controllers$new(trial, listener)
```

Arguments:

`trial` a trial object returned from `trial()`.

`listener` a listener object returned from `listener()`.

Method `reset()`: reset the trial and listener registered to the controller so that a new simulation can be started with `controller$run()`. The trial and the milestones are restored to their as-designed version.

Usage:

```
Controllers$reset()
```

Method `get_output()`: return a data frame of all current outputs saved by calling `save()`.

Usage:

```
Controllers$get_output(cols = NULL, simplify = TRUE, tidy = FALSE)
```

Arguments:

`cols` character vector. Columns to be returned from the data frame of simulation outputs. If `NULL`, all columns are returned.

`simplify` logical. Return vector rather than a data frame of one column when `length(cols) == 1` and `simplify == TRUE`.

`tidy` logical. `TrialSimulator` automatically records a set of standard outputs at milestones, even when `doNothing` is used as action functions. These includes time of triggering milestones, number of observed events for time-to-event endpoints, and number of non-missing readouts for non-TTE endpoints (see vignette('actionFunctions')). This usually mean a large number of columns in outputs. If users have no intent to summarize a trial on these columns, setting `tidy = TRUE` can eliminate these columns from `get_output()`. This is useful to reduced the size of output data frame when a large number of replicates are done for simulation. Note that currently we use regex `"^n_events_<.*?>_<.*?>$"` and `"^milestone_time_<.*?>$"` to match columns to be eliminated. If users plan to use `tidy = TRUE`, caution is needed when naming custom outputs in `save()`. Default `FALSE`.

Method `run()`: run trial simulation. It cannot be called again on the same controller unless `reset()` is called first.

Usage:

```
Controllers$run(
  n = 1,
  n_workers = 1,
  plot_event = TRUE,
  silent = FALSE,
  tidy = FALSE
)
```

Arguments:

`n` integer. Number of replicates of simulation. `n = 1` by default. Simulation results can be accessed by `controller$get_output()`.

`n_workers` integer. Number of parallel workers. When `n_workers = 1` (default), replicates are run sequentially. When `n_workers > 1`, replicates are distributed across parallel workers using the `mirai` package, which must be installed separately. Each worker receives a serialized copy of the trial and listener objects and runs its share of replicates independently. If any replicate encounters an error, execution stops and already-collected results are preserved in `$get_output()`. To debug, manually set seed in `trial()` and `n_workers = 1` in `run()` for reproduced results. Note that optimal `n_workers` may not be `parallel::detectCores()`. For example, Macbook with M1/M2/M3 chips may have performance cores and efficiency cores. To achieve the best parallel performance, one may want to use the performance cores only. For a M1 laptop with 4 performance cores, `n_workers = 3` may give the best performance.

`plot_event` logical. Create event plot if TRUE. Forced to FALSE when `n > 1` or `n_workers > 1`.

`silent` logical. TRUE if muting all messages during a trial. Note that warning messages are still displayed. When `silent = TRUE` and replicates are run sequentially (`n_workers = 1`), a progress bar is displayed automatically if the simulation is expected to take more than 1 minute.

`tidy` logical. If TRUE, the per-arm event count table (output column `n_events_<milestone>_<arms>`) is not saved at milestones; the per-endpoint totals and milestone times are still saved. Saving that table is the most expensive part of the standard outputs, so `tidy = TRUE` is recommended for a large number of replicates unless the per-arm counts are needed in the summary. This differs from `tidy` in `$get_output()`, which removes all standard columns from the returned data frame after the fact. Default FALSE.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
##
```

CorrelatedPfsAndOs2 *Generate Correlated PFS and OS Using Gumbel Copula*

Description

Generate correlated PFS and OS endpoints using the Gumbel copula. Marginally, both PFS and OS follow exponential distributions. This function can be used as custom generator in the function `endpoint()`.

Note that the Gumbel copula is applied to the survival functions of OS and time-to-progression (TTP). PFS is defined as `min(TTP, OS)`, which also follows an exponential distribution.

For more information, refer to [this vignette](#).

Usage

```
CorrelatedPfsAndOs2(
  n,
  median_pfs,
  median_os,
  kendall,
  pfs_name = "pfs",
  os_name = "os"
)
```

Arguments

n	integer. Number of observations.
median_pfs	numeric. Median of PFS.
median_os	numeric. Median of OS.
kendall	numeric. Kendall's tau between observed, uncensored PFS and OS. Must be non-negative and usually away from zero. Note that this argument is not the Kendall's tau between TTP and OS.
pfs_name	column name of PFS in returned data frame. It must be consistent with name in the function endpoint().
os_name	column name of OS in returned data frame. It must be consistent with name in the function endpoint().

Value

A data frame of n rows and four columns, including PFS, OS and their event indicators. The event indicators are all 1s. The column names are <pfs_name>, <pfs_name>_event, <os_name>, and <os_name>_event.

Examples

```
pfs_and_os <- endpoint(name = c('PFS', 'Os'),
  type = c('tte', 'tte'),
  generator = CorrelatedPfsAndOs2,
  median_pfs = 5,
  median_os = 11,
  kendall = .6,
  pfs_name = 'PFS',
  os_name = 'Os')

pfs_and_os # run it in console to see summary report

## for validation purpose only
## not the recommended way to use TrialSimulator
dat <- pfs_and_os$test_generator(n = 1e4)
cor(dat[, 1:2], method = 'kendall') ## close to 0.6
```

CorrelatedPfsAndOs3	<i>Generate Correlated PFS and OS Using the Three-States Illness-Death Model</i>
---------------------	--

Description

Generate correlated PFS and OS endpoints using the three-states model. This function can be used as custom generator in the function `endpoint()`.

Usage

```
CorrelatedPfsAndOs3(n, h01, h02, h12, pfs_name = "pfs", os_name = "os")
```

Arguments

<code>n</code>	integer. Number of observations.
<code>h01</code>	constant transition hazard from state "initial" to state "progression".
<code>h02</code>	constant transition hazard from state "initial" to state "death".
<code>h12</code>	constant transition hazard from state "progression" to state "death".
<code>pfs_name</code>	column name of PFS in returned data frame. It must be consistent with name in the function <code>endpoint()</code> .
<code>os_name</code>	column name of OS in returned data frame. It must be consistent with name in the function <code>endpoint()</code> .

Value

A data frame of `n` rows and four columns, including PFS, OS and their event indicators. The event indicators are all 1s. The column names are `<pfs_name>`, `<pfs_name>_event`, `<os_name>`, and `<os_name>_event`.

Examples

```
## use as function (if you don't use TrialSimulator for simulation)
pfs_and_os_trt <- CorrelatedPfsAndOs3(1e4, 0.06, 0.30, 0.30, 'PFS', 'OS')
pfs_and_os_pbo <- CorrelatedPfsAndOs3(1e4, 0.10, 0.40, 0.30, 'PFS', 'OS')

## use as generator (if you use TrialSimulator for simulation)

pfs_and_os <- endpoint(name = c('PFS', 'os'),
                      type = c('tte', 'tte'),
                      generator = CorrelatedPfsAndOs3,
                      h01 = .06, h02 = .30, h12 = .30,
                      pfs_name = 'PFS', os_name = 'os')

pfs_and_os # run it in console to see summary report
```

CorrelatedPfsAndOs4 *Generate Correlated PFS, OS and Objective Response*

Description

Generate correlated PFS, OS and objective response using the four-states model. It can be used as custom generator of `endpoint()`.

Usage

```
CorrelatedPfsAndOs4(
  n,
  transition_probability,
  duration,
  death_name = "death",
  progression_name = "progression",
  response_name = "response"
)
```

Arguments

<code>n</code>	integer. Number of observations.
<code>transition_probability</code>	a 4x4 matrix defining transition probabilities between stable (initial state, 1), response (2), progression (3) and death (absorbing, 4).
<code>duration</code>	integer. Duration of trial. Set it to a sufficient large integer in practice to cover the duration of the trial (potentially be extended).
<code>death_name</code>	column name of OS in returned data frame. It must be consistent with name in the function <code>endpoint()</code> .
<code>progression_name</code>	column name of PFS in returned data frame. It must be consistent with name in the function <code>endpoint()</code> .
<code>response_name</code>	column name of objective response in returned data frame. It must be consistent with name in the function <code>endpoint()</code> .

Value

A data frame of `n` rows and 6 columns (response, progression, death, and their event indicators with 1 means event and 0 means censored at duration). The column names are `<death_name>`, `<death_name>_event`, `<progression_name>`, `<progression_name>_event`, `<response_name>` and `<response_name>_event`.

Note that it returns time-to-response for each patients with status of censoring at pre-set duration. If a binary indicator of response at a time point is needed as an endpoint, we may write a wrapper function to convert the column `<response_name>` to binary and remove the column `<response_name>_event` from return value.

Examples

```
m <- matrix(c(0.99, 0.0035, 0.0055, 0.0010,
              0, 0.9900, 0.0052, 0.0048,
              0, 0, 0.9960, 0.0040,
              0, 0, 0, 1),
            nrow = 4, byrow = TRUE)

## use as function (if you don't use TrialSimulator for simulation)

dat <- CorrelatedPfsAndOs4(1e4, m, 365 * 3)

## use as generator (if you use TrialSimulator for simulation)

ep <- endpoint(name = c('pfs', 'os', 'or'),
              type = c('tte', 'tte', 'tte'), ## OR is TTE, not binary
              generator = CorrelatedPfsAndOs4,
              transition_probability = m,
              duration = 365 * 3,
              death_name = 'os', ## rename output from generator to match with "name"
              progression_name = 'pfs',
              response_name = 'or')

ep # run it in console to see summary report
```

crossover

Crossover at a Milestone

Description

Apply a milestone-triggered crossover to patients who are still in the trial.

Unlike a regimen registered via `add_regimen()` (which is applied once, at enrollment), `crossover()` is meant to be called *inside* a milestone's action function. At the earliest crossover (calendar) time $T = \text{trial}\$get_current_time() + \text{delay}$, eligible patients may switch to a new treatment, and only their *post-switch* endpoint values are altered; already-observed data is left intact. The crossover triplet is stacked onto the trial's regimen, so it is also re-applied to patients enrolled later.

Eligibility (the pool passed to `what()`) consists of patients with at least one endpoint still "open" (unobserved, dropout-/duration-aware) at T ; fully-observed patients are excluded. `when()` must return a switch time with $\text{enroll_time} + \text{switch_time} \geq T$ (a crossover cannot predate its opening) or an error is raised. `how()` may only change post-switch outcomes; returning a changed value for a pre-switch/locked cell raises an error.

Two helper columns are injected into `patient_data` for the triplet functions: `earliest_crossover_calendar_time` ($= T$) and `earliest_crossover_time_from_enrollment` ($= \max(T - \text{enroll_time}, 0)$), so `when()` can write e.g. `switch_time = pmax(pfs, earliest_crossover_time_from_enrollment)`.

Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and `TrialSimulator` has no way to track it.

This is a user-friendly wrapper of the member function of `trial`, i.e., `Trials$crossover()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
crossover(trial, what, how, when = NULL, delay = 0, ...)
```

Arguments

trial	a trial object returned by trial().
what	a function selecting which eligible patients crossover, returning one row per crossing-over patient with their new_treatment. Values of new_treatment must not contain '@' or ';', which are reserved for encoding regimen_trajectory. See regimen().
how	a function returning the modified post-switch endpoint values for crossing-over patients. See regimen().
when	(optional) a function returning switch_time from enrollment. If NULL (default), patients switch at T, i.e. switch_time = earliest_crossover_time_from_enrollment.
delay	numeric. Time after the milestone before the crossover opens, so T = trial\$get_current_time() + delay. Default 0 (opens at the milestone).
...	(optional) named arguments routed to what, when, and/or how.

Value

no return value, called for its side effect of updating trial.

doNothing

*An Action Function that Does Nothing***Description**

This is an action function that does nothing when the corresponding milestone is triggered. When the listener is monitoring a trial and determining the time to trigger a milestone, data is automatically locked with other necessary data manipulations (censoring, truncation, etc.) are executed. If the users have no intent to modify the trial adaptively at the milestone, e.g., adding (add_arms()) or removing (remove_arms()) arm(s), changing sampling ratio(s) (update_sample_ratio()), modifying trial duration (set_duration()), carrying out statistical testing, or saving intermediate results (save(), etc.), then this function can be used to set the argument action when creating a new milestone. Note that the triggering time and number of observations/events of endpoints at a milestone with action = doNothing is still recorded in output automatically.

Usage

```
doNothing(trial, ...)
```

Arguments

trial	an object returned from trial().
...	(optional) arguments. This is for capturing redundant arguments in milestone() only.

Value

This function returns NULL. Actually, nothing is done in this function.

DynamicRNGFunction	<i>A wrapper of random number generator.</i>
--------------------	--

Description

This function may be useful to advanced users of TrialSimulator. It creates a wrapper function of a random number generator, while fixing a subset or all of arguments. This function is design to prevent inadvertent changing to arguments of random number generator. See examples below.

Usage

```
DynamicRNGFunction(fn, ...)
```

Arguments

<code>fn</code>	random number generator, e.g., <code>rnorm</code> , <code>rchisq</code> , etc. It can be user-defined random number generator as well, e.g., <code>PiecewiseConstantExponentialRNG</code> .
<code>...</code>	arguments for <code>fn</code> . Specifying invalid arguments can trigger error and be stopped. There are three exceptions. (1) <code>rng</code> can be passed through <code>...</code> to give true name of <code>fn</code> . This could be necessary as it may be hard to parse it accurately in <code>DynamicRNGFunction</code> , or simply for a more informative purpose in some scenarios. (2) <code>var_name</code> can be passed through <code>...</code> to specify the name of generated variable. (3) <code>simplify</code> can be set to <code>FALSE</code> to convert a vector into a one-column data frame in returned object. This happens for built-in random number generators, e.g., <code>rnorm</code> , <code>rbinom</code> , etc. These three arguments will not be passed into <code>fn</code> .

Value

a function to generate random number based on `fn` and arguments in `...`. Specified arguments will be fixed and cannot be changed when invoking `DynamicRNGFunction(fn, ...)`. For example, if `foo <- DynamicRNGFunction(rnorm, sd = 2)`, then `foo(n = 100)` will always generate data from normal distribution of variance 4. `foo(n = 100, sd = 1)` will trigger an error. However, if an argument is not specified in `DynamicRNGFunction`, then it can be specified later. For example, `foo(n = 100, mean = -1)` will generate data from $N(-1, 4)$.

Examples

```
# example code
dfunc <- DynamicRNGFunction(rnorm, sd = 3.2)
x <- dfunc(1e3) # mean 0 and sd 3.2
hist(x)

y <- dfunc(1e3, mean = 3.5) # mean can be changed
```

```
mean(y)

try(z <- dfunc(1e3, sd = 1)) # error because sd is fixed in dfunc
```

endpoint	<i>Define Endpoints</i>
----------	-------------------------

Description

Define one or multiple endpoints. This is a user-friendly wrapper for the class constructor `Endpoints$new`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Note that it is users' responsibility to assure that the units of readout of non-tte endpoints, dropout time, and trial duration are consistent.

Usage

```
endpoint(name, type, readout = NULL, generator, ...)
```

Arguments

name	character vector. Name(s) of endpoint(s)
type	character vector. Type(s) of endpoint(s) in name. It supports "tte" for time-to-event endpoints, and "non-tte" for all other types of endpoints (e.g., continuous, binary, categorical, or repeated measurement. <code>TrialSimulator</code> will do some verification if an endpoint is of type "tte". However, no special manipulation is done for non-tte endpoints. "baseline" can be used for a non-tte endpoint that is observed at randomization (e.g., a baseline covariate, biomarker, or subgroup indicator). Its readout is 0 by definition and must not be specified in readout; doing so triggers an error.
readout	numeric vector named by non-tte endpoint(s). readout should be specified for every non-tte endpoint. For example, <code>c(endpoint1 = 6, endpoint2 = 3)</code> , which means that it takes 6 and 3 unit time to get readouts of endpoint1 and endpoint2 of a patient since being randomized. For readouts of a longitudinal endpoint being collected at baseline (baseline) and 2 (ep1), 4 (ep2) unit time, its readout can be set as <code>c(baseline = 0, ep1 = 2, ep2 = 4)</code> . Error message will be prompted if readout is not named or is not specified for all non-tte endpoint, or it is specified for any tte endpoints. If all endpoints are tte, readout should be its default value <code>NULL</code> . Endpoints of type "baseline" must be omitted from readout, as their readout is 0.
generator	a RNG function. Its first argument must be <code>n</code> , number of patients. It must return a data frame of <code>n</code> rows. It supports all univariate random number generators, like those in <code>stats</code> , e.g., <code>stats::rnorm</code> , <code>stats::rexp</code> , etc. that with <code>n</code> as the first argument for number of observations. generator could be any custom functions as long as (1) its first argument is <code>n</code> ; and (2) it returns a

vector of length n or a data frame of n rows. Custom random number generator can return data of more than one endpoint. This is useful when users need to simulate correlated endpoints (e.g., longitudinal endpoints, or PFS/OS). The column names of returned data frame should match to the argument name exactly, but order does not matter. If an endpoint is of type "tte", the custom generator should also return a column as event indicator. The column name of event indicator is <endpoint name>_event. For example, if "pfs" is "tte", then custom generator should return at least two columns "pfs" and "pfs_event". Usually pfs_event can be all 1s if no censoring. For other generators, e.g., `TrialSimulator::PiecewiseConstantExponentialRNG` and `TrialSimulator::CorrelatedPfsAndOs4`, the event indicators could take values 0/1 due to the nature of their algorithms. Censoring can also be specified later in `trial()` through its argument `dropout`. See `?Trials`. Note that if covariates, e.g., biomarker, subgroup, are needed in generating and analyzing trial data, they can and should be defined as endpoints as well.

... (optional) arguments of generator.

Value

an object of R6 class `Endpoints`, representing one or more endpoints.

Examples

```
set.seed(12345)
## Example 1. Generate a time-to-event endpoint.
## Two columns are returned, one for time, one for event (1/0, 0 for
## A built-in RNG function is used to handle piecewise constant exponential
## distribution
risk <- data.frame(
  end_time = c(1, 10, 26.0, 52.0),
  piecewise_risk = c(1, 1.01, 0.381, 0.150) * exp(-3.01)
)

pfs <- endpoint(name = 'pfs', type='tte',
               generator = PiecewiseConstantExponentialRNG,
               risk = risk, endpoint_name = 'pfs')

# run it in R console to display a summary report
# event indicator takes values 0/1
pfs

## Example 2. Generate continuous and binary endpoints using R's built-in
## RNG functions, e.g. rnorm, rexp, rbinom, etc.
ep1 <- endpoint(
  name = 'cd4', type = 'non-tte', generator = rnorm, readout = c(cd4=1),
  mean = 1.2)
ep2 <- endpoint(
  name = 'resp_time', type = 'non-tte', generator = rexp, readout = c(resp_time=0),
  rate = 4.5)
ep3 <- endpoint(
  name = 'orr', type = 'non-tte', readout = c(orr=3), generator = rbinom,
```

```

        size = 1, prob = .4)

ep1 # run it in R console. Mean and sd should be comparable to (1.2, 1.0)

ep2 # run it in R console. Median should be comparable to  $\log(2)/4.5 = 0.154$ 

ep3 # run it in R console. Mean and sd should be comparable to 0.4 and 0.49

## Example3: delayed effect
## Use piecewise constant exponential random number generator
## Baseline hazards are piecewise constant
## Hazard ratios are piecewise constant, resulting a delayed effect.
## Note that this example is for explaining the concept of "endpoint".
## Generating endpoint data manually is not the recommended way to use this package.

run <- TRUE

if (!requireNamespace("survminer", quietly = TRUE)) {
  run <- FALSE
  message("Please install 'survminer' to run this example.")
}

if (!requireNamespace("survival", quietly = TRUE)) {
  run <- FALSE
  message("Please install 'survival' to run this example.")
}

if(run){
  risk1 <- risk
  ep1 <- endpoint(
    name = 'pfs', type='tte',
    generator = PiecewiseConstantExponentialRNG,
    risk=risk1, endpoint_name = 'pfs')

  risk2 <- risk1
  risk2$hazard_ratio <- c(1, 1, .6, .4)
  ep2 <- endpoint(
    name = 'pfs', type='tte',
    generator = PiecewiseConstantExponentialRNG,
    risk=risk2, endpoint_name = 'pfs')

  n <- 1000
  tte <- rbind(ep1$get_generator()(n), ep2$get_generator()(n))
  arm <- rep(0:1, each = n)
  dat <- data.frame(tte, arm)
  sfit <- survival::survfit(
    survival::Surv(time = pfs, event = pfs_event) ~ arm, dat)

  survminer::ggsurvplot(sfit,
    data = dat,
    pval = TRUE, # Show p-value
    conf.int = TRUE, # Show confidence intervals

```

```

risk.table = TRUE, # Add risk table
palette = c("blue", "red"))

## print summary reports for endpoint objects in console
ep1
ep2

}

## Example 4: generate correlated pfs and os
## See vignette('simulatePfsAndOsIdm') and vignette('simulatePfsAndOsGumbel')

```

Endpoints

*Class of Endpoint***Description**

Create a class of endpoint to specify its name, type, readout time (optional) and assign a random number generator.

Public methods in this R6 class are used in developing this package. Thus, I have to export the whole R6 class which exposures all public methods. However, **none of the public methods is useful to end users except for the one below.**

- `$print()` print a summary report of endpoint(s) based on example data from the generator.

In addition, `$test_generator()`, `$get_generator()` and `$get_name()` are for exploratory purpose only. They appear in examples and vignettes to help users understand how this class works, but they are not needed at all in formal simulation once users know how to use this package.

Internal machinery. The remaining public methods (`$get_uid()` and `$update_generator()`) are public only because they are invoked on an endpoint object by other components of the package (arms), which R6 cannot grant through private members. Users should not call them directly.

Value

an R6Class generator object; use `endpoint()` to create endpoints.

Methods**Public methods:**

- `Endpoints$new()`
- `Endpoints$test_generator()`
- `Endpoints$get_generator()`
- `Endpoints$update_generator()`
- `Endpoints$get_uid()`
- `Endpoints$get_name()`

- `Endpoints$print()`
- `Endpoints$clone()`

Method `new()`: initialize an endpoint.

Usage:

```
Endpoints$new(name, type, readout = NULL, generator, ...)
```

Arguments:

`name` character vector. Name(s) of endpoint(s)

`type` character vector. Type(s) of endpoint(s). It supports "tte" for time-to-event endpoints, and "non-tte" for all other types of endpoints (e.g., continuous, binary, categorical, or repeated measurement). `TrialSimulator` will do some verification if an endpoint is of type "tte". However, no special manipulation is done for non-tte endpoints. Note: `Endpoints$new()` only recognizes "tte" and "non-tte". The "baseline" type accepted by `endpoint()` is a non-tte endpoint observed at randomization; `endpoint()` converts it to "non-tte" with `readout 0` before constructing the object, so it never reaches here.

`readout` a named numeric vector with `name` to be non-tte endpoint(s). `readout` must be specified for every non-tte endpoint. For example, `c(endpoint1 = 6, endpoint2 = 3)`, which means that it takes 6 and 3 unit time to get readout of `endpoint1` and `endpoint2` of a patient since being randomized. Error message would be prompted if `readout` is not named or `readout` is not specified for some non-tte endpoint. If all endpoints are tte, `readout` should be `NULL` as default.

`generator` a random number generation (RNG) function. It supports all built-in random number generators in `stats`, e.g., `stats::rnorm`, `stats::rexp`, etc. that with `n` as the argument for number of observations and returns a vector. A custom RNG function is also supported. `generator` could be any functions as long as (1) its first argument is `n`; and (2) it returns a vector of length `n` (univariate endpoint) or a data frame of `n` rows (multiple endpoints), i.e., custom RNG can return data of more than one endpoint. This is useful when users need to simulate correlated endpoints or longitudinal data. The column names of returned data frame should match to `name` exactly, although order of columns does not matter. If an endpoint is of type "tte", the custom generator should also return a column as its event indicator. For example, if "pfs" is "tte", then custom generator should return at least two columns "pfs" and "pfs_event". Usually `pfs_event` can be all 1s if no censoring. Some RNG functions, e.g., `TrialSimulator::PiecewiseConstantExponentialRNG()` and `TrialSimulator::CorrelatedPfsAndOs4()`, simulate TTE endpoint data with censoring simultaneously, thus 0 exists in the columns of event indicators. Users can implement censorship in their own RNG. Censoring can also be specified later when defining a trial object through argument `dropout`. See `?trial`. Note that if covariates, e.g., biomarker, subgroup, are needed in generating and analyzing trial data, they can and should be defined as endpoints in `endpoint()` as well.

`...` optional arguments for `generator`.

Method `test_generator()`: test random number generator of the endpoints. It returns an example dataset of an endpoint object. Note that users of `TrialSimulator` does not need to call this function to generate trial data; instead, the package will call this function at milestone automatically. Users may see example in vignette where this function is called. However, it is for illustration purpose only. In practice, this function may be used for debugging if users suspect some issues in custom generator, otherwise, this function should never be called in formal simulation.

Usage:

```
Endpoints$test_generator(n = 1000)
```

Arguments:

n integer. Number of random numbers generated from the generator.

Method get_generator(): return random number generator of an endpoint

Usage:

```
Endpoints$get_generator()
```

Method update_generator(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

update endpoint generator

Usage:

```
Endpoints$update_generator(generator, ...)
```

Arguments:

generator a random number generation (RNG) function. See generator of endpoint().
... optional arguments for generator.

Method get_uid(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return uid

Usage:

```
Endpoints$get_uid()
```

Method get_name(): return endpoints' name

Usage:

```
Endpoints$get_name()
```

Method print(): print an endpoint object

Usage:

```
Endpoints$print(categorical_vars = NULL)
```

Arguments:

categorical_vars a character vector of endpoints. This can be used to force variables with limited distinct values as categorical variables in summary report. For example, a numeric endpoint may take integer values 0, 1, 2. Instead of computing mean and standard derivation in the summary report, put this endpoint in categorical_vars can force it be a categorical variable and a barplot is generated in summary report instead.

Examples:

```
rng <- function(n){
  data.frame(x = sample(1:3, n, replace = TRUE),
            y = sample(1:3, n, replace = TRUE)
  )
}
ep <- endpoint(name = c('x', 'y'),
```

```

        type = c('non-tte', 'non-tte'),
        readout = c(x = 0, y = 0),
        generator = rng)

## x and y as continuous endpoints, thus mean and sd are reported
ep

## force y to be categorical to create barplot of it
print(ep, categorical_vars = 'y')
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```

# Instead of using Endpoints$new(), please use endpoint(), a user-friendly
# wrapper to define endpoints. See examples in ?endpoint.
```

```

## -----
## Method `Endpoints$print`
## -----
```

```

rng <- function(n){
  data.frame(x = sample(1:3, n, replace = TRUE),
            y = sample(1:3, n, replace = TRUE)
  )
}
```

```

ep <- endpoint(name = c('x', 'y'),
              type = c('non-tte', 'non-tte'),
              readout = c(x = 0, y = 0),
              generator = rng)
```

```

## x and y as continuous endpoints, thus mean and sd are reported
ep
```

```

## force y to be categorical to create barplot of it
print(ep, categorical_vars = 'y')
```

enrollment

Triggering Condition by Number of Randomized Patients

Description

Define a condition to trigger trial milestone by the number of randomized patients. The milestone will be triggered when a trial has enrolled at least the specified number of patients. It can be used combined with conditions specified by [calendarTime](#) and [eventNumber](#).

Refer to the [vignette](#) to learn how to define milestones when performing simulation using TrialSimulator.

Usage

```
enrollment(n, ..., arms = NULL, min_treatment_duration = 0)
```

Arguments

n	integer. Number of randomized patients.
...	subset conditions compatible with <code>dplyr::filter</code> . Number of randomized patients will be counted on subset of trial data only.
arms	vector of character. Name of arms on which the number of patients is counted. If NULL, use all arms that are not yet removed from the trial by the time of calculation.
min_treatment_duration	numeric. Zero or positive value. minimum treatment duration of enrolled patients. Default is 0, i.e., looking for triggering time based on number of enrolled patients in population specified by ... and arms. If positive, it means that milestone is triggered when a specific number of enrolled patients have received treatment for at least min_treatment_duration duration. It is users' responsibility to assure that the unit of min_treatment_duration are consistent with readout of non-tte endpoints, dropout time, and trial duration.

Value

an object of class 'Condition'

Examples

```
## ensure sufficient sample size of whole trial
enrollment(n = 100)

## ensure sufficient sample size in sub-group of interest
enrollment(n = 100, biomarker1 == 'positive' & biomarker2 == 'high')

## ensure sufficient sample size in high dose + placebo
enrollment(n = 1000, arms = c('high dose', 'placebo'))

## ensure sufficient treatment duration
```

```
enrollment(n = 500, min_treatment_duration = 2)
```

eventNumber	<i>Triggering Condition by Number of Events or Non-missing Observations of an Endpoint</i>
-------------	--

Description

Define a condition to trigger trial milestone by the number of events of a time-to-event endpoint or the number of non-missing observations of a non-time-to-event endpoint. The milestone will be triggered when a trial has observed at least the specified number of endpoint events (or non-missing observations). It can be used combined with conditions specified by [calendarTime](#) and [enrollment](#).

Number of events for a time-to-event endpoint can vary at different milestones as more patients are randomized into a trial, or more events onset over time.

Number of non-missing observations for a non-time-to-event endpoint can vary at different milestones as more patients are randomized into a trial, or more patients have been treated until their readout time (thus, NA turns to a value).

Both numbers are affected by dropout.

Refer to the [vignette](#) to learn how to define milestones when performing simulation using TrialSimulator.

Usage

```
eventNumber(endpoint, n, ..., arms = NULL)
```

Arguments

endpoint	character. Name of an endpoint. It should be something that is specified in the argument name in <code>endpoint()</code> .
n	integer. Targeted number of events or non-missing observations, depending on the type of endpoint.
...	subset conditions compatible with <code>dplyr::filter</code> . Number of events/observations will be counted on subset of trial data only.
arms	vector of character. Name of arms on which the number of events/observations is counted. If NULL, use all arms that are not yet removed from the trial (using <code>remove_arms()</code>) by the time of calculation.

Value

an object of class 'Condition'

expandRegimen	<i>Expand regimen trajectory into long format</i>
---------------	---

Description

Expands the compact regimen_trajectory column in locked data (returned by `trial$get_locked_data()`) into a long-format data frame with one row per regimen segment per patient.

The regimen_trajectory column stores each patient's treatment history as a semicolon-separated string of "name\@time" entries, e.g. "placebo\@0;low dose\@8.5". `expandRegimen` parses this into two additional columns:

- regimen — name of the treatment regimen
- switch_time_from_enrollment — time from enrollment at which the patient switched to this regimen

The regimen_trajectory column is dropped from the result.

Usage

```
expandRegimen(data)
```

Arguments

data a data frame returned by `trial$get_locked_data()`.

Value

a long-format data frame: one row per regimen segment per patient, with regimen and switch_time_from_enrollment appended and regimen_trajectory removed.

Examples

```
## Not run:
locked <- trial$get_locked_data('final')
long   <- expandRegimen(locked)

## End(Not run)
```

fitCoxph

*Fit Cox Proportional Hazard Ratio model***Description**

Fit Cox proportional hazards model on an time-to-event endpoint.

Refer to [this vignette](#) for more information and examples.

Usage

```
fitCoxph(formula, placebo, data, alternative, scale, ..., tidy = TRUE)
```

Arguments

formula	An object of class formula that can be used with <code>survival::coxph</code> . The data frame data must consist a column arm and a column of the endpoint specified in formula. Covariates can be adjusted. Interactions between arm and covariates are allowed in formula, but arm must has a term of main effect, and only estimate of that main effect is tested.
placebo	Character. String indicating the placebo in <code>data\$arm</code> .
data	Data frame. Usually it is a data snapshot locked at a milestone.
alternative	a character string specifying the alternative hypothesis, must be one of "greater" or "less", i.e., one-sided test is enforced. No default value. "greater" means superiority of treatment over placebo is established by an hazard ratio greater than 1.
scale	character. The type of estimate in the output. Must be one of "log hazard ratio" or "hazard ratio". No default value.
...	(optional) subset conditions compatible with <code>dplyr::filter</code> . <code>coxph</code> will be fitted on this subset only. This argument can be useful to create a subset of data for analysis when a trial consists of more than two arms. By default, it is not specified, all data will be used to fit the model. More than one condition can be specified in ..., e.g., <code>fitCoxph(formula, 'pbo', data, 'less', 'log hazard ratio', arm %in% c('pbo', 'low dose'), x > 0.5)</code> , which is equivalent to: <code>fitCoxph(formula, 'pbo', data, 'less', 'log hazard ratio', arm %in% c('pbo', 'low dose') & x > 0.5)</code> . Note that if more than one treatment arm are present in the data after applying filter in ..., models are fitted and tested for placebo verse each of the treatment arms.
tidy	logical. FALSE if more information are returned. Default: TRUE.

Value

a data frame with columns:

arm name of the treatment arm.

placebo name of the placebo arm.

`estimate` estimate of main effect of arm, depending on scale.

`p` one-sided p-value for log hazard ratio (treated vs placebo).

`info` the number of events of the endpoint in the subset.

`z` the z statistics of log hazard ratios.

`fitFarringtonManning` *Farrington-Manning test for rate difference*

Description

Test rate difference by comparing it to a pre-specified value using the Farrington-Manning test.

Refer to [this vignette](#) for more information and examples.

Usage

```
fitFarringtonManning(endpoint, placebo, data, alternative, ..., delta = 0)
```

Arguments

<code>endpoint</code>	Character. Name of the endpoint in data.
<code>placebo</code>	Character. String indicating the placebo in <code>data\$arm</code> .
<code>data</code>	Data frame. Usually it is a locked data set.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "greater" or "less", i.e., one-sided test is enforced. No default value. "greater" means superiority of treatment over placebo is established by rate difference greater than 'delta'.
<code>...</code>	Subset conditions compatible with <code>dplyr::filter</code> . <code>glm</code> will be fitted on this subset only. This argument can be useful to create a subset of data for analysis when a trial consists of more than two arms. By default, it is not specified, all data will be used to fit the model. More than one condition can be specified in <code>...</code> , e.g., <code>fitFarringtonManning('remission', 'pbo', data, delta, arm %in% c('pbo', 'low dose'), cfb > 0.5)</code> , which is equivalent to: <code>fitFarringtonManning('remission', 'pbo', data, delta, arm %in% c('pbo', 'low dose') & cfb > 0.5)</code> . Note that if more than one treatment arm are present in the data after applying filter in <code>...</code> , models are fitted for placebo verse each of the treatment arms.
<code>delta</code>	the rate difference between a treatment arm and placebo under the null. 0 by default.

Value

a data frame with columns:

- arm name of the treatment arm.
- placebo name of the placebo arm.
- estimate estimate of rate difference.
- p one-sided p-value for log odds ratio (treated vs placebo).
- info sample size in the subset with NA being removed.
- z the z statistics of log odds ratio (treated vs placebo).

References

Farrington, Conor P., and Godfrey Manning. "Test statistics and sample size formulae for comparative binomial trials with null hypothesis of non-zero risk difference or non-unity relative risk." *Statistics in medicine* 9.12 (1990): 1447-1454.

fitLinear	<i>Fit linear regression model</i>
-----------	------------------------------------

Description

Fit linear regression model on a continuous endpoint.
 Refer to [this vignette](#) for more information and examples.

Usage

```
fitLinear(formula, placebo, data, alternative, ...)
```

Arguments

formula	an object of class formula. Must include arm and endpoint in data. Covariates can be adjusted.
placebo	Character. String indicating the placebo arm in data\$arm.
data	Data frame. Usually it is a locked data set.
alternative	a character string specifying the alternative hypothesis, must be one of "greater" or "less", i.e., one-sided test is enforced. No default value. "greater" means superiority of treatment over placebo is established by a greater mean in treated arm.
...	Subset conditions compatible with <code>dplyr::filter</code> . glm will be fitted on this subset only. This argument can be useful to create a subset of data for analysis when a trial consists of more than two arms. By default, it is not specified, all data will be used to fit the model. More than one condition can be specified in ..., e.g., <code>fitLinear(cfb ~ arm, 'pbo', data, 'greater', arm %in% c('pbo', 'low dose'), cfb > 0.5)</code> , which is equivalent to: <code>fitLinear(cfb ~</code>

arm, 'pbo', data, 'greater', arm %in% c('pbo', 'low dose') & cfb > 0.5).
 Note that if more than one treatment arm are present in the data after applying filter in ..., models are fitted and tested for placebo verse each of the treatment arms.

Value

a data frame with columns:

arm name of the treatment arm.

placebo name of the placebo arm.

estimate estimate of average treatment effect of arm.

p one-sided p-value for between-arm difference (treated vs placebo).

info sample size used in model with NA being removed.

z z statistics of between-arm difference (treated vs placebo).

fitLogistic

Fit logistic regression model

Description

Fit logistic regression model on an binary endpoint.

Refer to [this vignette](#) for more information and examples.

Usage

```
fitLogistic(formula, placebo, data, alternative, scale, ...)
```

Arguments

formula	An object of class formula. Must include arm and endpoint in data. Covariates can be adjusted.
placebo	Character. String indicating the placebo in data\$arm.
data	Data frame. Usually it is a locked data set.
alternative	a character string specifying the alternative hypothesis, must be one of "greater" or "less", i.e., one-sided test is enforced. No default value. "greater" means superiority of treatment over placebo is established by an odds ratio greater than 1.
scale	character. The type of estimate in the output. Must be one of "coefficient", "log odds ratio", "odds ratio", "risk ratio", or "risk difference". No default value.

... Subset conditions compatible with `dplyr::filter`. `glm` will be fitted on this subset only. This argument can be useful to create a subset of data for analysis when a trial consists of more than two arms. By default, it is not specified, all data will be used to fit the model. More than one condition can be specified in ..., e.g., `fitLogistic(remission ~ arm, 'pbo', data, 'greater', 'odds ratio', arm %in% c('pbo', 'low dose'), cfb > 0.5)`, which is equivalent to: `fitLogistic(remission ~ arm, 'pbo', data, 'greater', 'odds ratio', arm %in% c('pbo', 'low dose') & cfb > 0.5)`. Note that if more than one treatment arm are present in the data after applying filter in ..., models are fitted for placebo verse each of the treatment arms.

Value

a data frame with columns:

- arm name of the treatment arm.
- placebo name of the placebo arm.
- estimate estimate depending on scale.
- p one-sided p-value for log odds ratio (treated vs placebo).
- info sample size used in model with NA being removed.
- z z statistics of log odds ratio (treated vs placebo).

fitLogrank	<i>Carry out log rank test</i>
------------	--------------------------------

Description

Compute log rank test statistic on an endpoint.
 Refer to [this vignette](#) for more information and examples.

Usage

```
fitLogrank(formula, placebo, data, alternative, ..., tidy = TRUE)
```

Arguments

formula	An object of class formula that can be used with <code>survival::survdiff</code> . Must consist arm and endpoint in data. No covariate is allowed. Stratification variables are supported and can be added using <code>strata(...)</code> .
placebo	character. String of placebo in <code>data\$arm</code> .
data	data frame. Usually it is a locked data.
alternative	a character string specifying the alternative hypothesis, must be one of "greater" or "less", i.e., one-sided test is enforced. No default value. "greater" means superiority of treatment over placebo is established by an hazard ratio greater than 1.

... subset condition that is compatible with `dplyr::filter`. The log rank test (`survival::survdiff`) is carried out on this subset only. This argument could be useful to create a subset of data for analysis when a trial consists of more than two arms. By default it is not specified, all data will be used to fit the model. More than one conditions can be specified in ..., e.g., `fitLogrank(formula, data, arm %in% c('pbo', 'low dose'), x > 0.5)`, which is equivalent to `fitLogrank(formula, data, arm %in% c('pbo', 'low dose') & x > 0.5)`. Note that if more than one treatment arm are present in the data after applying filter in ..., models are fitted for placebo verse each of the treatment arms.

tidy logical. FALSE if more information are returned. Default TRUE.

Value

a data frame with columns:

arm name of the treatment arm.

placebo name of the placebo arm.

p one-sided p-value for log-rank test (treated vs placebo).

info the number of events of the endpoint in the subset.

z the z statistic of the log rank test, with the sign of the log hazard ratio of treatment vs placebo.

If the statistic is undefined because its variance is zero (e.g., no informative event comparison in the subset), a simulation placeholder $z = 0$ with the corresponding $p = 0.5$ is returned with a warning rather than an error, so that a few such replicates in a large simulation do not require error handling in action functions. The placeholder carries no evidence either way and is not a standardized normal statistic.

getAdaptiveDesignOutput

Get simulation output in the vignette adaptiveDesign.Rmd

Description

Internal function that retrieves precomputed simulation results. Not meant for use by package users.

Usage

```
getAdaptiveDesignOutput()
```

Value

A data frame containing simulation results of 1000 replicates.

`getConditionalPowerOutput`*Get simulation output in the vignette conditionalPower.Rmd*

Description

Internal function that retrieves precomputed simulation results. Not meant for use by package users.

Usage`getConditionalPowerOutput()`**Value**

A data frame containing simulation results of 1000 replicates.

`getDoseRangingOutput` *Get simulation output in the vignette doseRanging.Rmd*

Description

Internal function that retrieves precomputed simulation results. Not meant for use by package users.

Usage`getDoseRangingOutput()`**Value**

A data frame containing simulation results of 10 replicates.

`getFixedDesignOutput` *Get simulation output in the vignette fixedDesign.Rmd*

Description

Internal function that retrieves precomputed simulation results. Not meant for use by package users.

Usage`getFixedDesignOutput()`**Value**

A data frame containing simulation results of 1000 replicates.

GraphicalTesting	<i>Class of GraphicalTesting</i>
------------------	----------------------------------

Description

Perform graphical testing under group sequential design for one or multiple endpoints. See Maurer & Bretz (2013).

Methods

Public methods:

- `GraphicalTesting$new()`
- `GraphicalTesting$reset()`
- `GraphicalTesting$is_valid_hid()`
- `GraphicalTesting$get_hypothesis_name()`
- `GraphicalTesting$get_weight()`
- `GraphicalTesting$set_weight()`
- `GraphicalTesting$get_alpha()`
- `GraphicalTesting$set_alpha()`
- `GraphicalTesting$get_hypotheses_ids()`
- `GraphicalTesting$get_number_hypotheses()`
- `GraphicalTesting$get_hids_not_in_graph()`
- `GraphicalTesting$get_testable_hypotheses()`
- `GraphicalTesting$has_testable_hypotheses()`
- `GraphicalTesting$is_in_graph()`
- `GraphicalTesting$is_testable()`
- `GraphicalTesting$get_hid()`
- `GraphicalTesting$reject_a_hypothesis()`
- `GraphicalTesting$set_trajectory()`
- `GraphicalTesting$get_trajectory()`
- `GraphicalTesting$test_hypotheses()`
- `GraphicalTesting$test()`
- `GraphicalTesting$get_current_testing_results()`
- `GraphicalTesting$get_current_decision()`
- `GraphicalTesting$print()`
- `GraphicalTesting$clone()`

Method `new()`: Initialize an object for graphical testing procedure. Group sequential design is also supported.

Usage:

```
GraphicalTesting$new(
  alpha,
  transition,
  alpha_spending,
  planned_max_info,
  hypotheses = NULL,
  silent = FALSE
)
```

Arguments:

alpha initial alpha allocated to each of the hypotheses.

transition matrix of transition weights. Its diagonals should be all 0s. The row sums should be 1s (for better power) or 0s (if no outbound edge from a node).

alpha_spending character vector of same length of alpha. Currently it supports 'asP', 'asOF', and 'asUser'.

planned_max_info vector of integers. Maximum numbers of events (tte endpoints) or patients (non-tte endpoints) at the final analysis of each hypothesis when planning a trial. The actual numbers could be different, which can be specified elsewhere.

hypotheses vector of characters. Names of hypotheses.

silent TRUE if muting all messages and not generating plots.

Method `reset()`: reset an object of class `GraphicalTesting` to original status so that it can be reused.

Usage:

```
GraphicalTesting$reset()
```

Method `is_valid_hid()`: determine if index of a hypothesis is valid

Usage:

```
GraphicalTesting$is_valid_hid(hid)
```

Arguments:

hid an integer

Method `get_hypothesis_name()`: get name of a hypothesis given its index.

Usage:

```
GraphicalTesting$get_hypothesis_name(hid)
```

Arguments:

hid an integer

Method `get_weight()`: return weight between two nodes

Usage:

```
GraphicalTesting$get_weight(hid1, hid2)
```

Arguments:

hid1 an integer

hid2 an integer

Method `set_weight()`: update weight between two nodes

Usage:

GraphicalTesting\$set_weight(hid1, hid2, value)

Arguments:

hid1 an integer

hid2 an integer

value numeric value to be set as a weight two nodes

Method get_alpha(): return alpha allocated to a hypothesis when calling this function. Note that a function can be called several time with the graph is updated dynamically. Thus, returned alpha can be different even for the same hid.

Usage:

GraphicalTesting\$get_alpha(hid)

Arguments:

hid an integer

Method set_alpha(): update alpha of a hypothesis

Usage:

GraphicalTesting\$set_alpha(hid, value)

Arguments:

hid integer. Index of a hypothesis

value numeric value to be allocated

Method get_hypotheses_ids(): return all valid hid

Usage:

GraphicalTesting\$get_hypotheses_ids()

Method get_number_hypotheses(): return number of hypotheses, including those been rejected.

Usage:

GraphicalTesting\$get_number_hypotheses()

Method get_hids_not_in_graph(): return index of hypotheses that are rejected.

Usage:

GraphicalTesting\$get_hids_not_in_graph()

Method get_testable_hypotheses(): return index of hypotheses with non-zero alphas, thus can be tested at the current stage.

Usage:

GraphicalTesting\$get_testable_hypotheses()

Method has_testable_hypotheses(): determine whether at least one hypothesis is testable. If return FALSE, the testing procedure is completed.

Usage:

GraphicalTesting\$has_testable_hypotheses()

Method `is_in_graph()`: determine whether a hypothesis is not yet rejected (in graph).

Usage:

`GraphicalTesting$is_in_graph(hid)`

Arguments:

`hid` integer. Index of a hypothesis

Method `is_testable()`: determine whether a hypothesis has a non-zero alpha allocated.

Usage:

`GraphicalTesting$is_testable(hid)`

Arguments:

`hid` integer. Index of a hypothesis

Method `get_hid()`: convert hypothesis's name into (unique) index.

Usage:

`GraphicalTesting$get_hid(hypothesis)`

Arguments:

`hypothesis` character. Name of a hypothesis. It is different from `hid`, which is an index.

Method `reject_a_hypothesis()`: remove a node from graph when a hypothesis is rejected

Usage:

`GraphicalTesting$reject_a_hypothesis(hypothesis)`

Arguments:

`hypothesis` name of a hypothesis. It is different from `hid`, which is an index.

Method `set_trajectory()`: save new testing results at current stage

Usage:

`GraphicalTesting$set_trajectory(result)`

Arguments:

`result` a data frame of specific columns.

Method `get_trajectory()`: return testing results by the time this function is called. Note that graphical test is carried out in a sequential manner. Users may want to review the results anytime. Value returned by this function can possibly vary over time.

Usage:

`GraphicalTesting$get_trajectory()`

Method `test_hypotheses()`: test hypotheses using p-values (and other information in `stats`) base on the current graph. All rows should have the same order number.

Usage:

`GraphicalTesting$test_hypotheses(stats)`

Arguments:

`stats` a data frame. It must contain the following columns:

`order` integer. P-values (among others) of hypotheses that can be tested at the same time (e.g., an interim, or final analysis) should be labeled with the same order number. If a hypothesis is not tested at a stage, simply don't put it in stats with that order number.

`hypotheses` character. Name of hypotheses to be tested. They should be identical to those when calling `GraphicalTesting$new`.

`p` nominal p-values.

`info` observed number of events or samples at test. These will be used to compute information fractions in group sequential design.

`max_info` integers. Maximum information at test. At interim, `max_info` should be equal to `planned_max_info` when calling `GraphicalTesting$new`. At the final stage of a hypothesis, one can update it with observed numbers.

Method `test()`: test hypotheses using p-values (and other information in stats) base on the current graph. Users can call this function multiple times. P-values of the same order should be passed through stats together. P-values of multiple orders can be passed together as well. For example, if users only have p-values at current stage, they can call this function and update the graph accordingly. In this case, column order in stats is a constant. They can call this function again when p-values of next stage is available, where order is another integer. In simulation, if p-values of all stages are on hand, users can call this function to test them all in a single pass. In this case, column order in stats can have different values.

Usage:

```
GraphicalTesting$test(stats)
```

Arguments:

`stats` a data frame. It must contain the following columns:

`order` integer. P-values (among others) of hypotheses that can be tested at the same time (e.g., an interim, or final analysis) should be labeled with the same order number. If a hypothesis is not tested at a stage, simply don't put it in stats with that order number. If all p-values in stats are tested at the same stage, order can be absent.

`hypotheses` character. Name of hypotheses to be tested. They should be identical to those when calling `GraphicalTesting$new`.

`p` nominal p-values.

`info` observed number of events or samples at test. These will be used to compute information fractions in group sequential design.

`max_info` integers. Maximum information at test. At interim, `max_info` should be equal to `planned_max_info` when calling `GraphicalTesting$new`. At the final stage of a hypothesis, one can update it with observed numbers.

`alpha_spent` accumulative proportion of allocated alpha to be spent if `alpha_spending = "asUser"`. Set it to `NA_real_` otherwise. If no hypothesis uses "asUser" in stats, this column could be ignored.

Returns: a data frame returned by `get_current_testing_results`. It contains details of each of the testing steps.

Method `get_current_testing_results()`: return testing results with details by the time this function is called. This function can be called by users by multiple times, thus the returned value varies over time. This function is called by `GraphicalTesting::test`, and returns a data frame consisting of columns

hypothesis name of hypotheses.
 obs_p_value observed p-values.
 max_allocated_alpha maximum allocated alpha for the hypothesis.
 decision 'reject' or 'accept' the hypotheses.
 stages stage of a hypothesis.
 order order number that this hypothesis is tested for the last time. It is different from stages.
 typeOfDesign name of alpha spending functions.

Usage:

GraphicalTesting\$get_current_testing_results()

Method get_current_decision(): get current decisions for all hypotheses. Returned value could changes over time because it depends on the stages being tested already.

Usage:

GraphicalTesting\$get_current_decision()

Returns: a named vector of values "accept" or "reject". Note that if a hypothesis is not yet tested when calling this function, the decision for that hypothesis would be "accept".

Method print(): generic function for print

Usage:

GraphicalTesting\$print(graph = TRUE, trajectory = TRUE, ...)

Arguments:

graph logic. TRUE if visualizing the current graph, which can vary over time.
 trajectory logic. TRUE if print the current data frame of trajectory, which can vary over time.
 ... other arguments supported in gMCPLite::hGraph, e.g., trhw and trhh to control the size of transition box, and trdigits to control the digits displayed for transition weights.

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

Usage:

GraphicalTesting\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
## Example 1
## dry-run to study the behavior of a graph
## without group sequential design
if(interactive()){
  eps <- .01
  alpha <- c(.01, .04, 0, 0, 0)
  transition <- matrix(c(
    0, 0, 0, 0, 1,
    0, 0, .75, 0, .25,
    0, 1/2-eps/2, 0, eps, 1/2-eps/2,
    0, 0, 0, 0, 0,
    0, 1/2, 1/2, 0, 0
```

```

), nrow = 5, byrow = TRUE)

## dummy can be anything, we don't actually use it
asf <- rep('asOF', 5)
## dummy can be anything, we don't actually use it
max_info <- c(300, 1100, 1100, 1100, 500)

hs <- c('H1: UPCR IgA', 'H2: eGFR GN', 'H3: eGFR GN 10wk', 'H5: 2nd Endpoints', 'H4: eGFR IgA')

## initialize an object
gt <- GraphicalTesting$new(alpha, transition, asf, max_info, hs)
print(gt)

## reject hypotheses based on customized order
## to understand the behavior of a testing strategy
## Any other rejection order is possible
gt$reject_a_hypothesis('H1: UPCR IgA')
print(gt)

gt$reject_a_hypothesis('H2: eGFR GN')
print(gt)

gt$reject_a_hypothesis('H4: eGFR IgA')
print(gt)

gt$reject_a_hypothesis('H3: eGFR GN 10wk')
print(gt)

gt$reset()
}

## Example 2
## Example modified from vignettes in gMCP Lite:
## Graphical testing for group sequential design
if(interactive()){
  ## initial alpha split to each of the hypotheses
  alpha <- c(.01, .01, .004, .0, .0005, .0005)

  ## transition matrix of the initial graph
  transition <- matrix(c(
    0, 1, 0, 0, 0, 0,
    0, 0, .5, .5, 0, 0,
    0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, .5, .5,
    0, 0, 0, 0, 0, 1,
    .5, .5, 0, 0, 0, 0
  ), nrow = 6, byrow = TRUE)

  ## alpha spending functions per hypothesis
  asf <- c('asUser', 'asOF', 'asUser', 'asOF', 'asOF', 'asOF')

  ## planned maximum number of events per hypothesis
  max_info <- c(295, 800, 310, 750, 500, 1100)

```

```

## name of hypotheses
hs <- c('H1: OS sub',
        'H2: OS all',
        'H3: PFS sub',
        'H4: PFS all',
        'H5: ORR sub',
        'H6: ORR all')

gt <- GraphicalTesting$new(alpha, transition, asf, max_info, hs)

## print initial graph
gt

## nominal p-values at each stage
## Note: p-values with same order are calculated with the same locked data
## Note: alpha_spent is only specified for hypotheses using custom alpha
##       spending function "asUser"
stats <-
  data.frame(
    order = c(1:3, 1:3, 1:2, 1:2, 1, 1),
    hypotheses = c(rep('H1: OS sub', 3), rep('H2: OS all', 3),
                   rep('H3: PFS sub', 2), rep('H4: PFS all', 2),
                   'H5: ORR sub', 'H6: ORR all'),
    p = c(.03, .0001, .000001, .2, .15, .1, .2, .001, .3, .2, .00001, .1),
    info = c(185, 245, 295, 529, 700, 800, 265, 310, 675, 750, 490, 990),
    is_final = c(F, F, T, F, F, T, F, T, F, T, T, T),
    max_info = c(rep(295, 3), rep(800, 3), rep(310, 2), rep(750, 2), 490, 990),
    alpha_spent = c(c(.1, .4, 1), rep(NA, 3), c(.3, 1), rep(NA, 2), NA, NA)
  )

## test the p-values from the first analysis, plot the updated graph
gt$test(stats %>% dplyr::filter(order==1))

## test the p-values from the second analysis, plot the updated graph
gt$test(stats %>% dplyr::filter(order==2))

## test the p-values from the third analysis, plot the updated graph
## because no further test would be done, displayed results are final
gt$test(stats %>% dplyr::filter(order==3))

## plot the final status of the graph
print(gt, trajectory = FALSE)

## you can get final testing results as follow
gt$get_current_testing_results()

## if you want to see step-by-step details
print(gt$get_trajectory())

## equivalently, you can call gt$test(stats) for only once to get same results.
gt$reset()

```

```

gt$test(stats)

## if you only want to get the final testing results
gt$get_current_decision()
}

```

GroupSequentialTest *Class of GroupSequentialTest*

Description

Perform group sequential test for a single endpoint based on sequential one-sided p-values at each stages. Selected alpha spending functions, including user-defined functions, are supported. Boundaries are calculated with 'rpact'. At the final analysis, adjustment can be applied for over-running or under-running trial where observed final information is greater or lower than the planned maximum information. See Wassmer & Brannath, 2016, p78f. The test is based on p-values not z statistics because it is easier to not handling direction of alternative hypothesis in current implementation. In addition, only one-sided test is supported which should be sufficient for common use in clinical design.

Value

an R6Class generator object; create a test with `GroupSequentialTest$new()`.

Methods

Public methods:

- `GroupSequentialTest$new()`
- `GroupSequentialTest$get_name()`
- `GroupSequentialTest$get_alpha()`
- `GroupSequentialTest$set_alpha_spending()`
- `GroupSequentialTest$get_alpha_spending()`
- `GroupSequentialTest$get_max_info()`
- `GroupSequentialTest$set_max_info()`
- `GroupSequentialTest$get_stage()`
- `GroupSequentialTest$reset()`
- `GroupSequentialTest$set_trajectory()`
- `GroupSequentialTest$get_trajectory()`
- `GroupSequentialTest$get_stage_level()`
- `GroupSequentialTest$test_one()`
- `GroupSequentialTest$test()`
- `GroupSequentialTest$print()`
- `GroupSequentialTest$clone()`

Method `new()`: initialize a group sequential test. Now only support one-sided test based on p-values.

Usage:

```
GroupSequentialTest$new(
  alpha = 0.025,
  alpha_spending = c("asP", "asOF", "asUser"),
  planned_max_info,
  name = "H0",
  silent = TRUE
)
```

Arguments:

`alpha` familywise error rate

`alpha_spending` alpha spending function. Use "asUser" if custom alpha spending schedule is used.

`planned_max_info` integer. Planned maximum number of patients for non-tte endpoints or number of events for tte endpoints

`name` character. Name of the hypothesis, e.g. endpoint, subgroup, etc. Optional.

`silent` TRUE if muting all messages.

Method `get_name()`: get name of hypothesis

Usage:

```
GroupSequentialTest$get_name()
```

Method `get_alpha()`: get overall alpha

Usage:

```
GroupSequentialTest$get_alpha()
```

Method `set_alpha_spending()`: set alpha spending function. This is useful when set 'asUser' at the final stage to adjust for an under- or over-running trial.

Usage:

```
GroupSequentialTest$set_alpha_spending(asf)
```

Arguments:

`asf` character of alpha spending function.

Method `get_alpha_spending()`: return character of alpha spending function

Usage:

```
GroupSequentialTest$get_alpha_spending()
```

Method `get_max_info()`: return planned maximum information

Usage:

```
GroupSequentialTest$get_max_info()
```

Method `set_max_info()`: set planned maximum information. This is used at the final stage to adjust for an under- or over-running trial.

Usage:

```
GroupSequentialTest$set_max_info(obs_max_info)
```

Arguments:

`obs_max_info` integer. Maximum information, which could be observed number of patients or events at the final stage.

Method `get_stage()`: get current stage.

Usage:

```
GroupSequentialTest$get_stage()
```

Method `reset()`: an object of class `GroupSequentialTest` is designed to be used sequentially by calling `GroupSequentialTest$test`. When all planned tests are performed, no further analysis could be done. In that case keep calling `GroupSequentialTest$test` will trigger an error. To reuse the object for a new set of staged p-values, call this function to reset the status to stage 1. See examples. This implementation can prevent the error that more than the planned number of stages are tested.

Usage:

```
GroupSequentialTest$reset()
```

Method `set_trajectory()`: save testing result at current stage

Usage:

```
GroupSequentialTest$set_trajectory(result, is_final = FALSE)
```

Arguments:

`result` a data frame storing testing result at a stage.

`is_final` logical. TRUE if final test for the hypothesis, FALSE otherwise.

Method `get_trajectory()`: return testing trajectory until current stage. This function can be called at any stage. See examples.

Usage:

```
GroupSequentialTest$get_trajectory()
```

Method `get_stage_level()`: compute boundaries given current (potentially updated) settings. It returns different values if settings are changed over time.

Usage:

```
GroupSequentialTest$get_stage_level()
```

Method `test_one()`: test a hypothesis with the given p-value at current stage

Usage:

```
GroupSequentialTest$test_one(
  p_value,
  is_final,
  observed_info,
  alpha_spent = NA_real_
)
```

Arguments:

`p_value` numeric. A p-value.

`is_final` logical. TRUE if this test is carried out for the final analysis.

`observed_info` integer. Observed information at current stage. It can be the number of samples (non-tte) or number of events (tte) at test. If the current stage is final, `observed_info` will be used to update `planned_max_info`, the alpha spending function (`typeOfDesign` in `rpact`) will be updated to 'asUser', and the argument `userAlphaSpending` will be used when calling `rpact::getDesignGroupSequential`.

`alpha_spent` numeric if `alpha_spending = "asUser"`. It must be between 0 and alpha, the overall alpha of the test. NA_real_ for other alpha spending functions "asOF" and "asP".

Method `test()`: Carry out test based on group sequential design. If `p_values` is NULL, dummy values will be use and boundaries are calculated for users to review.

Usage:

```
GroupSequentialTest$test(
  observed_info,
  is_final,
  p_values = NULL,
  alpha_spent = NULL
)
```

Arguments:

`observed_info` a vector of integers, observed information at stages.

`is_final` logical vector. TRUE if the test is for the final analysis.

`p_values` a vector of p-values. If specified, its length should equal to the length of `observed_info`.

`alpha_spent` accumulative alpha spent at observed information. It is a numeric vector of values between 0 and 1, and of length that equals `length(observed_info)` if alpha-spending function is "asUser". Otherwise NULL.

Method `print()`: generic function for print

Usage:

```
GroupSequentialTest$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Note: examples showed here replicate the results from
## https://www.rpact.org/vignettes/planning/rpact_boundary_update_example/

## Example 1. Generate boundaries for a pre-fix group sequential design
gst <- GroupSequentialTest$new(
  alpha = .025, alpha_spending = 'asOF',
  planned_max_info = 387)

## without giving p-values, boundaries are returned without actual testing
```

```

gst$test(observed_info = c(205, 285, 393), is_final = c(FALSE, FALSE, TRUE))
gst

## Example 2. Calculate boundaries with observed information at stages
## No p-values are provided

## get an error without resetting an used object
try( gst$test(observed_info = 500, is_final = FALSE) )

## reset the object for re-use
gst$reset()
gst$test(observed_info = c(205, 285, 393), is_final = c(FALSE, FALSE, TRUE))
gst

## Example 3. Test stagewise p-values sequentially
gst$reset()

gst$test(observed_info = 205, is_final = FALSE, p_values = .09)
gst$test(285, FALSE, .006)

## print testing trajectory by now
gst

gst$test(393, TRUE, .002)

## print all testing trajectory
gst

## you can also test all stages at once
## the result is the same as calling test() for each of the stages
gst$reset()
gst$test(c(205, 285, 393), c(FALSE, FALSE, TRUE), c(.09, .006, .002))
gst

## Example 4. use user-define alpha spending
gst <- GroupSequentialTest$new(
  alpha = .025, alpha_spending = 'asUser',
  planned_max_info = 387)

gst$test(
  observed_info = c(205, 285, 393),
  is_final = c(FALSE, FALSE, TRUE),
  alpha_spent = c(.005, .0125, .025))
gst

```

Description

Define a listener. This is a user-friendly wrapper for the class constructor `Listener$new()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Listener is an important concept of `TrialSimulator`. Used with a trial object in a controller, a listener can monitor a running trial to execute user-defined actions when it determine condition of triggering a milestone is met. This mechanism allows the package users to focus on the development of action functions in a simulation.

Usage

```
listener(silent = FALSE)
```

Arguments

`silent` logical. TRUE to mute messages.

Value

an object of R6 class `Listeners`, monitoring and triggering milestones of a trial.

Examples

```
listener <- listener()
```

`Listeners`

Class of Listener

Description

Create a class of listener. A listener monitors the trial while checking condition of pre-defined milestones. Actions are triggered and executed automatically.

Public methods in this R6 class are used in developing this package. Thus, we have to export the whole R6 class which exposures all public methods. However, only the public methods in the list below are useful to end users.

- `$add_milestones()` register milestone(s) with the listener.
- `$get_milestone_names()` return names of registered milestones.

Internal machinery. The remaining public methods (`$monitor()`, `$mute()` and `$reset()`) are public only because they are invoked on a listener object by other components of the package (controllers), which R6 cannot grant through private members. Users should not call them directly.

Value

an R6Class generator object; use `listener()` to create a listener.

Methods

Public methods:

- `Listeners$new()`
- `Listeners$add_milestones()`
- `Listeners$get_milestone_names()`
- `Listeners$monitor()`
- `Listeners$mute()`
- `Listeners$reset()`
- `Listeners$clone()`

Method `new()`: initialize a listener

Usage:

```
Listeners$new(silent = FALSE)
```

Arguments:

`silent` logical. TRUE to mute messages.

Method `add_milestones()`: register milestones with listener. Order in ... matter as they are scanned and triggered in that order. It is users' responsibility to use reasonable order when calling this function, otherwise, the result of `Listeners$monitor()` can be problematic.

Usage:

```
Listeners$add_milestones(...)
```

Arguments:

... one or more objects returned from `milestone()`.

Examples:

```
listener <- listener()
interim <- milestone(name = 'interim',
                     when = eventNumber('endpoint', n = 100)
                     )
final <- milestone(name = 'final',
                  when = calendarTime(time = 24)
                  )
listener$add_milestones(interim, final)
```

Method `get_milestone_names()`: return names of registered milestones

Usage:

```
Listeners$get_milestone_names()
```

Method `monitor()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

scan, check, and trigger registered milestones. Milestones are triggered in the order when calling `Listener$add_milestones`.

Usage:

```
Listeners$monitor(trial)
```

Arguments:

`trial` a `Trial` object.

Method `mute()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

mute all messages (not including warnings)

Usage:

```
Listeners$mute(silent)
```

Arguments:

`silent` logical.

Method `reset()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

reset all milestones registered to the listener. Usually, this is called before a controller can run additional replicates of simulation.

Usage:

```
Listeners$reset()
```

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

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
##
## -----
## Method `Listeners$add_milestones`
## -----

listener <- listener()
interim <- milestone(name = 'interim',
                    when = eventNumber('endpoint', n = 100)
                    )
final <- milestone(name = 'final',
                  when = calendarTime(time = 24)
                  )
listener$add_milestones(interim, final)
```

milestone

Define a Milestone

Description

Define a milestone of a trial. This is a user-friendly wrapper for the class constructor `Milestones$new()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

A milestone means the time point to take an action, e.g., carrying out (futility, interim, final) analysis for adding/removing arms, or stopping a trial early. It can also be any more general time point where trial data is used in decision making or adaptation. For example, one can define a milestone for changing randomization scheme, sample size re-assessment, trial duration extension etc.

Refer to the [vignette](#) to learn how to define milestones when performing simulation using `TrialSimulator`.

Usage

```
milestone(name, when, action = doNothing, ...)
```

Arguments

name	character. Name of milestone.
when	condition to check if this milestone should be triggered. It takes value returned from functions <code>calendarTime()</code> , <code>enrollment()</code> , <code>eventNumber()</code> or their logic combinations.
action	function to execute when the milestone triggers. If no action to be executed but simply need to record triggering time and number of events/non-missing observations of endpoints at a milestone, action can be its default value, a built-in function <code>doNothing</code> .
...	(optional) arguments of action.

Value

an object of R6 class `Milestones`, representing a milestone of a trial.

Examples

```
## See vignette('conditionSystem')
```

Milestones

*Class of Milestones***Description**

Create a class of milestone. A milestone means the time point to take an action, e.g., carrying out (futility, interim, final) analysis for adding/removing arms, or stopping a trial early. It can also be any more general time point where trial data is used in decision making or adaptation. For example, one can define a milestone for changing randomization scheme, sample size re-assessment, trial duration extension etc.

Public methods in this R6 class are used in developing this package. Thus, we have to export the whole R6 class which exposures all public methods. However, **none of the public methods on this page is useful to end users**. Instead, refer to the [vignette](#) to learn how to define milestones when performing simulation using TrialSimulator.

Value

an R6Class generator object; use milestone() to create a milestone.

Methods**Public methods:**

- Milestones\$new()
- Milestones\$get_name()
- Milestones\$set_trigger_condition()
- Milestones\$set_action_function()
- Milestones\$get_trigger_status()
- Milestones\$reset()
- Milestones\$trigger_milestone()
- Milestones\$mute()
- Milestones\$clone()

Method new(): initialize milestone

Usage:

```
Milestones$new(name, type = name, trigger_condition, action = doNothing, ...)
```

Arguments:

name character. Name of milestone.

type character vector. Milestone type(s) (futility, interim, final), a milestone can be of multiple types. This is for information purpose so can be any string.

trigger_condition function to check if this milestone should trigger. See vignette Condition System for Triggering Milestones in a Trial.

action function to execute when the milestone triggers.

... (optional) arguments of action.

Method `get_name()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return name of milestone

Usage:

```
Milestones$get_name()
```

Method `set_trigger_condition()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

replace the trigger condition of a not-yet-triggered milestone. This is invoked by the listener when it applies an update requested through `Trials$update_milestone()` within an action function. The as-designed condition, frozen at construction, is restored by `reset()` between simulation replicates.

Usage:

```
Milestones$set_trigger_condition(trigger_condition)
```

Arguments:

`trigger_condition` an object of class `Condition`. See `trigger_condition` of `milestone()`.

Method `set_action_function()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

replace the action function (and its fixed arguments) of a not-yet-triggered milestone. This is invoked by the listener when it applies an update requested through `Trials$update_milestone()` within an action function. The action in effect at construction is restored by `reset()` between simulation replicates.

Usage:

```
Milestones$set_action_function(action, action_args = list())
```

Arguments:

`action` function to execute when the milestone triggers. See `action` of `milestone()`.

`action_args` named list. Fixed arguments of action, corresponding to ... of `milestone()`.

Method `get_trigger_status()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return trigger status

Usage:

```
Milestones$get_trigger_status()
```

Method `reset()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

reset an milestone so that it can be triggered again. Usually, this is called before the controller of a trial can run additional replicates of simulation. If the trigger condition or the action was replaced in-run through `Trials$update_milestone()`, the as-designed version is restored so that every replicate starts from the original design.

Usage:

```
Milestones$reset()
```

Method trigger_milestone(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

trigger an milestone (always TRUE) and execute action accordingly. It calls Trial\$get_data_lock_time() to lock data based on conditions implemented in Milestones\$trigger_condition. If time that meets the condition cannot be found, Trial\$get_data_lock_time() will throw an error and stop the program. This means that user needs to adjust their trigger_condition (e.g., target number of events (target_n_events) is impossible to reach).

Usage:

```
Milestones$trigger_milestone(trial)
```

Arguments:

trial a Trial object.

Method mute(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

mute all messages (not including warnings)

Usage:

```
Milestones$mute(silent)
```

Arguments:

silent logical.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

PiecewiseConstantExponentialRNG

Generate Time-to-Event Endpoint from Piecewise Constant Exponential Distribution

Description

This function can be used as generator to define endpoint. Implementation is based on [this algorithm](#). This distribution can be used to simulate delayed treatment effect.

Usage

```
PiecewiseConstantExponentialRNG(n, risk, endpoint_name)
```

Arguments

n	integer. Number of random numbers
risk	a data frame of columns
end_time	End time for a constant risk in a time window. The start time of the first time window is 0.
piecewise_risk	A constant risk in a time window, which is absolute risk * relative risk, or (h0 * g) in the link.
hazard_ratio	An optional column for simulating an active arm. If absent, a column of 1s will be added. Equivalently, user can multiply piecewise_risk by hazard_ratio manually and ignore this column.
endpoint_name	character. Name of endpoint. This should be the same as the name argument when calling function endpoint().

Value

a data frame of n rows and two columns

<endpoint_name> name of endpoint specified by users in endpoint_name.

<endpoint_name>_event event indicator with 0/1 as censoring and event, respectively. Note that due to the nature of the algorithm to generate data from this distribution, it is possible to have the endpoint being censoring at the last end_time unless it is set to Inf.

Examples

```
# example code
# In this example, absolute risk in [0, 1) and [26, 52] are 0.0181 and
# 0.0027, respectively.
risk <- data.frame(
  end_time = c(1, 4.33, 26.0, 52.0),
  piecewise_risk = c(1, 1.01, 0.381, 0.150) * exp(-4.01)
)
PiecewiseConstantExponentialRNG(10, risk, 'PFS')
```

plot.milestone_time_summary

Plot Triggering Time of Milestones in Simulated Trials

Description

Plot Triggering Time of Milestones in Simulated Trials

Usage

```
## S3 method for class 'milestone_time_summary'
plot(x, ...)
```

Arguments

`x` an object returned by `summarizeMilestoneTime()`.
`...` currently not supported.

Value

no return value, called to display a boxplot of milestone triggering times.

`plot.three_state_model`

Plot result of three-state illness-death model

Description

Plot result of three-state illness-death model

Usage

```
## S3 method for class 'three_state_model'
plot(x, ...)
```

Arguments

`x` an object returned by `solveThreeStateModel()`.
`...` currently not supported.

Value

no return value, called to print the calibration table and display the corresponding plot.

`qPiecewiseExponential` *Quantile Function of Piecewise Exponential Distribution*

Description

To generate endpoint that follows a piecewise exponential distribution and is correlated with other endpoints, the copula method is commonly used. The quantile function needs to be specified (e.g., in the `simdata` package). If a piecewise exponential distributed endpoint is independent to other endpoints, one can simply use `TrialSimulator::PiecewiseConstantExponentialRNG()` to specify the generator argument in `endpoint()`.

There are many R packages implementing the quantile function of the piecewise exponential distributed random variable. Why do I implement it again? The reason is that this function is extremely important for simulating time-to-event endpoint in clinical trial simulation, thus the speed matters. `qPiecewiseExponential()` is implemented purely in R for code transparency, and is much faster than other packages.

Usage

```
qPiecewiseExponential(p, times, piecewise_risk)
```

Arguments

p numeric. A vector of probabilities.

times numeric. A vector of time points where risk (event rates) change. 0 shouldn't be in times.

piecewise_risk numeric. A vector of constant risk (event rates) in a time window. `length(piecewise_risk) = length(times) + 1`. To fully specify a piecewise exponential distribution, the number of risk parameters is one greater than the number of changepoints in times.

Value

A vector of quantiles.

Examples

```
## This code snippet can take > 10s to execute
if(interactive()){
  library(TrialSimulator)

  run <- TRUE
  if(!requireNamespace("rpact", quietly = TRUE)){
    run <- FALSE
    message("Please install 'rpact' to run this example.")
  }

  if(!requireNamespace("PWEXP", quietly = TRUE)){
    run <- FALSE
    message("Please install 'PWEXP' to run this example.")
  }

  if(!requireNamespace("PWEALL", quietly = TRUE)){
    run <- FALSE
    message("Please install 'PWEALL' to run this example.")
  }

  if(run){

    x <- solvePiecewiseConstantExponentialDistribution(
      surv_prob = c(.9, .75, .64, .42, .28),
      times = c(.4, 1.2, 4, 5.5, 9)
    )

    p <- stats::runif(1e5)

    ## fast, and only five lines of R codes
    message('TrialSimulator::qPiecewiseExponential(): ')
    system.time(
```

```

    a <- qPiecewiseExponential(
      p, times = x$end_time, piecewise_risk = c(x$piecewise_risk, .1)
    )
  ) |> print()

## > 10s
message('rpact::getPiecewiseExponentialQuantile(): ')
system.time(
  b <- rpact::getPiecewiseExponentialQuantile(
    p, piecewiseSurvivalTime = c(0, x$end_time),
    piecewiseLambda=c(x$piecewise_risk, .1)
  )
) |> print()

## equally fast, but implemented in Fortran
message('PWEALL::qpwe(): ')
system.time(
  c <- PWEALL::qpwe(p, rate = c(x$piecewise_risk, .1), tchange = c(0, x$end_time))$q
) |> print()

## equally fast, long codes in R (maybe more versatile?)
message('PWEXP::qpwexp(): ')
system.time(
  d <- PWEXP::qpwexp(p, rate = c(x$piecewise_risk, .1), breakpoint = x$end_time)
) |> print()

message('a == b: ')
all.equal(a, b) |> print()
message('a == c: ')
all.equal(a, c) |> print()
message('a == d: ')
all.equal(a, d) |> print()

}
}

```

rconst

Generate Constant Variable

Description

A random number generator returning only a constant. This can be used to set dropout time. Currently it is the default value of dropout time, with value = Inf.

This function can also be used as a generator of endpoint() if a constant endpoint is needed.

Usage

```
rconst(n, value)
```

Arguments

- n integer. Number of observations.
- value scalar. Value of constant observations.

Value

a numeric vector of length n with all entries equal to value.

regimen	<i>Define a Regimen</i>
---------	-------------------------

Description

Define a regimen of a trial. This is a user-friendly wrapper for the class constructor `Regimens$new()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

A regimen defines the rules to select patients who switch treatments, to determine the time of switching, and to update patients' endpoint data.

Usage

`regimen(what, when, how, ...)`

Arguments

- what a function determining whether patients' data would be updated due to switching treatment. It takes `patient_data`, a data frame as argument, and returns a data frame of two columns `patient_id` and `new_treatment`, with one row per switching patient. The number of rows in the returned data frame may be smaller than the number of patients in the input data frame; patients that are left out are simply not switched. Values of `new_treatment` must not contain '@' or ';', which are reserved for encoding `regimen_trajectory`. Note that the returned object will be passed into function `'how()'`, which is also provide by users. This argument can also be a list of functions that will be executed sequentially. No default value.
- when a function determining the time at which a patient switches to another treatment regimen, measured from the time of enrollment. It takes `patient_data`, a data frame as argument, and returns a data frame of two columns `patient_id` and `switch_time` (from `enroll_time`). The number of rows in the returned data frame must equal the number of rows in `patient_data`, i.e., a switching time must be specified for every patient (missing values are not allowed). Note that the returned object will be passed into function `'how()'`, which is also provided by users. This argument can also be a list of functions that will be executed sequentially. No default value.

how	a function updating patients' data after treatment switching. Only modified columns and patient_id are returned. For a cell that should not change, return its original value. Only <i>post-switch</i> outcomes may be changed: returning a value that differs from the original for an endpoint whose readout/event is at or before switch_time (a pre-switch or already-observed outcome) raises an error, so leave such cells at their original value (e.g. ifelse(os > switch_time, new_os, os)). This argument can also be a list of functions that will be executed sequentially. No default value.
...	(optional) named arguments to be passed to one or more of what, when, and how. Each argument is routed to every function whose formal parameter list contains that name. All arguments must be named, and every name must match at least one parameter of at least one function in what, when, or how.

Value

an object of R6 class Regimens, representing a treatment regimen.

Regimens	<i>Class of Regimens</i>
----------	--------------------------

Description

Create a class of regimen. A regimen defines the rules to select treatments for patients switch, to determine the time of switching, and to update patients' endpoint data.

Public methods in this R6 class are used in developing this package. Thus, we have to export the whole R6 class which exposures all public methods. However, **none of the public methods on this page is useful to end users**. Instead, use regimen() to create a regimen and register it with a trial through trial\$add_regimen() or trial\$crossover().

Value

an R6Class generator object; use regimen() to create a regimen.

Methods**Public methods:**

- Regimens\$new()
- Regimens\$get_number_treatment_allocator()
- Regimens\$get_treatment_allocator()
- Regimens\$get_time_selector()
- Regimens\$get_data_modifier()
- Regimens\$get_treatment_allocator_args()
- Regimens\$get_time_selector_args()
- Regimens\$get_data_modifier_args()
- Regimens\$get_earliest_crossover_calendar_time()

- `Regimens$append_triplet()`
- `Regimens$clone()`

Method `new()`: initialize regimen

Usage:

```
Regimens$new(what, when, how, ..., earliest_crossover_calendar_time = 0)
```

Arguments:

`what` a function determining whether patients' data would be updated due to switching treatment. It takes `patient_data`, a data frame as argument, and returns a data frame of two columns `patient_id` and `new_treatment`, with one row per switching patient. The number of rows in the returned data frame may be smaller than the number of patients in the input data frame; patients that are left out are simply not switched. Values of `new_treatment` must not contain '@' or ';', which are reserved for encoding regimen_trajectory. Note that the returned object will be passed into function 'how()', which is also provide by users. This argument can also be a list of functions that will be executed sequentially. No default value.

`when` a function determining the time at which a patient switches to another treatment regimen, measured from the time of enrollment. It takes `patient_data`, a data frame as argument, and returns a data frame of two columns `patient_id` and `switch_time` (from `enroll_time`). The number of rows in the returned data frame must equal the number of rows in `patient_data`, i.e., a switching time must be specified for every patient (missing values are not allowed). Note that the returned object will be passed into function 'how()', which is also provided by users. This argument can also be a list of functions that will be executed sequentially. No default value.

`how` a function updating patients' data after treatment switching. Only modified columns and `patient_id` are returned. For a cell that should not change, return its original value. Only *post-switch* outcomes may be changed: returning a value that differs from the original for an endpoint whose readout/event is at or before `switch_time` raises an error. This argument can also be a list of functions that will be executed sequentially. No default value.

... (optional) named arguments routed to one or more of `what`, `when`, and `how`.

`earliest_crossover_calendar_time` numeric. The earliest calendar time at which the triplet(s) may take effect. 0 (default) is the classic enrollment-time regimen, applied from the first enrollment. A positive value marks the triplet(s) as a milestone-triggered crossover (eligibility filtering, switch-time validation and the post-switch data mask). This is set internally by `trial$crossover()`; it is not a user argument of `regimen()`.

Method `get_number_treatment_allocator()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return number of treatment allocators for regimen

Usage:

```
Regimens$get_number_treatment_allocator()
```

Method `get_treatment_allocator()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return user-defined new treatment for a patient

Usage:

```
Regimens$get_treatment_allocator(index = NULL)
```

Arguments:

index integer. Index of allocator. Return all allocators if NULL.

Method get_time_selector(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return user-defined time selector

Usage:

```
Regimens$get_time_selector(index = NULL)
```

Arguments:

index integer. Index of selector. Return all selectors if NULL.

Method get_data_modifier(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return user-defined endpoint data modifier

Usage:

```
Regimens$get_data_modifier(index = NULL)
```

Arguments:

index integer. Index of selector. Return all modifiers if NULL.

Method get_treatment_allocator_args(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return pre-bound arguments for the i-th treatment allocator

Usage:

```
Regimens$get_treatment_allocator_args(index)
```

Arguments:

index integer.

Method get_time_selector_args(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return pre-bound arguments for the i-th time selector

Usage:

```
Regimens$get_time_selector_args(index)
```

Arguments:

index integer.

Method get_data_modifier_args(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return pre-bound arguments for the i-th data modifier

Usage:

```
Regimens$get_data_modifier_args(index)
```

Arguments:

index integer.

Method `get_earliest_crossover_calendar_time()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return the earliest crossover calendar time of triplet(s)

Usage:

```
Regimens$get_earliest_crossover_calendar_time(index = NULL)
```

Arguments:

`index` integer. Index of triplet. Return all if NULL.

Method `append_triplet()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

append one more triplet to the regimen. Used by milestone-triggered crossover to stack a new what/when/how (with its own `earliest_crossover_calendar_time`) onto an existing regimen without overwriting earlier triplets. Triplets are executed in append order.

Usage:

```
Regimens$append_triplet(
  what,
  when,
  how,
  ...,
  earliest_crossover_calendar_time = 0
)
```

Arguments:

`what`, `when`, `how` see `regimen()`.

... (optional) named arguments routed to `what`, `when`, and/or `how`.

`earliest_crossover_calendar_time` numeric. Earliest calendar time for the appended triplet.

A positive value marks it as a crossover.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

remove_arms

Removing One or More Arms From a Trial

Description

remove arms from a trial. The application of this function includes, but is not limited to, dose selection, enrichment analysis (select sub-population).

Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and `TrialSimulator` has no way to track it. In addition, data of the removed arms are censored or truncated by the time of arm removal.

This is a user-friendly wrapper of the member function of trial, i.e., `Trials$remove_arms()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
remove_arms(trial, arms_name)
```

Arguments

trial	a trial object returned by <code>trial()</code> .
arms_name	character vector. Name of arms to be removed.

Value

no return value, called for its side effect of updating trial.

resize	<i>Resizing a Trial</i>
--------	-------------------------

Description

resize a trial with a greater sample size. This function is used to update the maximum sample size adaptively after sample size reassessment. Note that this function should be called within action functions. It is users' responsibility to ensure it and `TrialSimulator` has no way to track this.

This is a user-friendly wrapper of the member function of trial, i.e., `Trials$resize()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
resize(trial, n_patients)
```

Arguments

trial	a trial object returned by <code>trial()</code> .
n_patients	integer. Number of maximum sample size of a trial.

Value

no return value, called for its side effect of updating trial.

set_duration	<i>Extending Duration of a Trial</i>
--------------	--------------------------------------

Description

set trial duration in an adaptive designed trial. New duration must be longer than the old one. All patients enrolled before resetting the duration are truncated (non-TTE endpoints) or censored (TTE endpoints) at the original duration. This helps maintain proper type I error or family-wise error rate and control multiplicity when conducting testing statistics. For more details of why this is necessary, please refer to [Jorgens et al. 2019](#).

Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and TrialSimulator has no way to track it.

This is a user-friendly wrapper of the member function of trial, i.e., `Trials$set_duration()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
set_duration(trial, duration)
```

Arguments

trial	a trial object returned by <code>trial()</code> .
duration	new duration of a trial. It must be greater than the current duration.

Value

no return value, called for its side effect of updating trial.

solveMixtureExponentialDistribution	<i>Solve Parameters in a Mixture Exponential Distribution</i>
-------------------------------------	---

Description

This is a helper function to explore parameters for endpoint generator, likely in an enrichment design.

Assume that the overall population in an arm is a mixture of two exponential distributions with medians median1 (m_1) and median2 (m_2). Given the proportion of the first component (p_1) and the overall median m , we have

$$p_1(1 - e^{-\log(2)m/m_1}) + (1 - p_1)(1 - e^{-\log(2)m/m_2}) = 1/2$$

This function computes m_2 or m given p_1 and m_1 . These parameters can be used in custom random number generator to define exponential distributed endpoints.

Note that the math formula above may not be displayed correctly on a html page. You can read it with better format by running `?solveMixtureExponentialDistribution`.

Usage

```
solveMixtureExponentialDistribution(
  weight1,
  median1,
  median2 = NULL,
  overall_median = NULL
)
```

Arguments

<code>weight1</code>	numeric. The proportion of the first component.
<code>median1</code>	numeric. Median of the first component.
<code>median2</code>	numeric. Median of the second component. If <code>NULL</code> , then <code>overall_median</code> must be specified, and this function will calculate and return <code>median2</code> .
<code>overall_median</code>	numeric. Median of the overall population. If <code>NULL</code> , then <code>median2</code> must be specified, and this function will calculate and return <code>overall_median</code> .

Value

a named vector of `median2` or `overall_median`.

Examples

```
library(dplyr)

median2 <-
  solveMixtureExponentialDistribution(
    weight1 = .3,
    median1 = 10,
    overall_median = 8)

median2

n <- 1e6
ifelse(
  runif(n) < .3,
  rexp(n, rate=log(2)/10),
  rexp(n, rate=log(2)/median2)) %>%
  median() ## should be close to 8

overall_median <-
  solveMixtureExponentialDistribution(
    weight1 = .4,
    median1 = 12,
```

```

        median2 = 4)

overall_median

ifelse(
  runif(n) < .4,
  rexp(n, rate=log(2)/12),
  rexp(n, rate=log(2)/4)) %>%
  median() ## should be close to overall_median

```

solvePiecewiseConstantExponentialDistribution

Compute Constant Rates of Piecewise Exponential Distribution

Description

This function computes the rate parameters λ in each of the time windows. λ are natural parameters of piecewise exponential distribution, but in practice, users may define the distribution by specifying the survival probabilities at time points where event rates change.

This function returns a data frame, which can be used as input of the argument `risk` of the data generator `PiecewiseConstantExponentialRNG`.

Usage

```
solvePiecewiseConstantExponentialDistribution(surv_prob, times)
```

Arguments

<code>surv_prob</code>	numeric. A vector of survival probabilities at times.
<code>times</code>	numeric. A vector of time points where event rates change. <code>times</code> and <code>surv_prob</code> must be of equal length.

Value

a data frame of two columns

end_time End time for a constant event rate. The start time of the first time window is 0.

piecewise_risk A constant event rate in the time window ending with `end_time` on the same row.

Examples

```

solvePiecewiseConstantExponentialDistribution(
  surv_prob = c(.9, .75, .64, .42, .28),
  times = c(.4, 1.2, 4, 5.5, 9)
)

```

solveThreeStateModel *Solve Parameters in a Three-State Illness-death Model*

Description

The illness-death model consists of three states, initial, progression, and death. It can be used to model the progression-free survival (PFS) and overall survival (OS) in clinical trial simulation. It models the correlation between PFS and OS without assumptions on latent status and copula. Also, it does not assume PFS and OS satisfy the proportional hazard assumption simultaneously. The three-state illness-death model ensures a nice property that $PFS \leq OS$ with probability one. However, it requires three hazard parameters under the homogeneous Markov assumption. In practice, hazard parameters are hard to specify intuitively especially when no trial data is available at the planning stage.

This function reparametrizes the illness-death model in term of three parameters, i.e. median of PFS, median of OS, and correlation between PFS and OS. The output of this function, which consists of the three hazard parameters, can be used to generate PFS and OS with desired property. It can be used with the built-in data generator `CorrelatedPfsAndOs3()` when defining endpoints in `TrialSimulator`.

For more information, refer to [this vignette](#).

Usage

```
solveThreeStateModel(
  median_pfs,
  median_os,
  corr,
  h12 = seq(0.05, 0.2, length.out = 50)
)
```

Arguments

<code>median_pfs</code>	numeric. Median of PFS.
<code>median_os</code>	numeric. Median of OS.
<code>corr</code>	numeric vector. Pearson correlation coefficients between PFS and OS.
<code>h12</code>	numeric vector. A set of hazard from progression to death that may induce the target correlation <code>corr</code> given <code>median_pfs</code> and <code>median_os</code> . <code>solveThreeStateModel()</code> will do a grid search to find the best hazard parameters that matches to the medians of PFS and OS, and their correlations.

Value

a data frame with columns:

`corr` target Pearson's correlation coefficients.

`h01` hazard from stable to progression.

`h02` hazard from stable to death.

h12 hazard from progression to death.
error absolute error between target correlation and correlation derived from h01, h02, and h12.

Examples

```
dat <- CorrelatedPfsAndOs3(1e6, h01 = .1, h02 = .05, h12 = .12)

cor(dat$pfs, dat$os) ## 0.65

median(dat$pfs) ## 4.62

median(dat$os) ## 9.61

## find h01, h02, h12 that can match to median_pfs, median_os and corr
## should be close to h01 = 0.10, h02 = 0.05, h12 = 0.12 when corr = 0.65
ret <- solveThreeStateModel(median_pfs = 4.6, median_os = 9.6,
                           corr = seq(.5, .7, length.out=5))

ret
```

StaggeredRecruiter	<i>Generate Enrollment Time from Piecewise Constant Uniform Distribution</i>
--------------------	--

Description

It assumes a uniform enrollment with constant rate in each of the time windows. This function can be used as the enroller when calling trial() to define a trial.

Usage

```
StaggeredRecruiter(n, accrual_rate)
```

Arguments

- n integer. Number of enrollment times to generate.
- accrual_rate a data frame of columns
 - end_time End time for a constant rate in a time window. The start time of the first time window is 0. Values must be positive and strictly increasing; the last one must be Inf.
 - piecewise_rate A constant rate in a time window. So the number of patients being recruited in that window is window length x piecewise_rate. A rate of 0 pauses enrollment for that window. Rates must be non-negative and finite; the last must be positive.

Details

StaggeredRecruiter is the only enroller accepted by `trial()`: a piecewise constant accrual rate is flexible enough to approximate realistic recruitment in practice, e.g., site ramp-up, steady accrual, and temporary pauses.

The returned enrollment times are deterministic, not random. Within a window of positive rate, patients enroll one by one with spacing $1/\text{piecewise_rate}$; under a constant rate r , the k -th patient enrolls exactly at k / r . In particular, the first patient enrolls at $1/\text{piecewise_rate}$ rather than at time 0, and a milestone triggered by `enrollment(n = n)` occurs exactly at the time the planned cumulative accrual reaches n .

A window with `piecewise_rate = 0` models a recruitment pause (a hold for safety review, a site not yet activated, a seasonal gap, etc.): no patient is enrolled in that window, and enrollment resumes after its `end_time`. Pauses may occur in the first window or span several consecutive windows; a leading pause defers the first enrollment accordingly.

A valid `accrual_rate` must satisfy all of the following:

- it is a data frame with columns `end_time` and `piecewise_rate`;
- `end_time` is positive and strictly increasing, and the last entry is `Inf` with a positive rate, so that the schedule can supply any number of patients (TrialSimulator may internally request more than the planned sample size, e.g., for adaptive resizing via `resize()`);
- rates are non-negative and finite;
- a finite window with a positive rate must expect at least one patient, i.e., window length $\times$ `piecewise_rate` ≥ 1 . A tiny positive rate meant as a pause is rejected with an error; use `piecewise_rate = 0` for a true pause.

Value

a sorted numeric vector of n enrollment times.

Examples

```
## constant accrual of 25 patients/month: patient k enrolls at k / 25
accrual_rate <- data.frame(end_time = Inf, piecewise_rate = 25)

StaggeredRecruiter(30, accrual_rate)

## recruitment pause: 30/mo through month 12, paused during months 12-18,
## then 30/mo again. Monthly counts show months 13-17 are empty and
## enrollment resumes at the end of the pause (month 18).
accrual_rate <- data.frame(
  end_time = c(12, 18, Inf),
  piecewise_rate = c(30, 0, 30)
)

enroll_time <- StaggeredRecruiter(400, accrual_rate)
table(ceiling(enroll_time))

## leading pause (first rate is 0): enrollment opens 3 months after study
## start, e.g., the first site is activated with a delay, then 30/mo
```

```

accrual_rate <- data.frame(
  end_time = c(3, Inf),
  piecewise_rate = c(0, 30)
)

StaggeredRecruiter(30, accrual_rate)

## approximate a linear ramp-up by monthly steps: accrual grows by 5/mo
## each month, from 5/mo up to 30/mo, then stays steady at 30/mo
accrual_rate <- data.frame(
  end_time = c(1:6, Inf),
  piecewise_rate = c(seq(5, 30, by = 5), 30)
)

enroll_time <- StaggeredRecruiter(200, accrual_rate)

## monthly enrolled counts show the ramp (5, 10, ..., 30) and the plateau (30)
table(ceiling(enroll_time))

```

stop_followup

Stopping Follow-Up of Selected Patients in a Trial

Description

stop follow-up of a subset of enrolled patients at a specified time at or after the current milestone. Data of affected patients are censored (time-to-event endpoints) or set to missing (non-time-to-event endpoints with readout after the stopping time), as if those patients were no longer followed since then. This function can be used in adaptive designs; its application includes, but is not limited to, treatment discontinuation, early termination of follow-up for a sub-population, or enrichment design where follow-up of a de-selected sub-population is stopped after an interim analysis. It can also be called at a pre-specified milestone that splits a trial into cohorts, e.g., a milestone marking the last patient of the first cohort and the first patient of the second cohort. Such a milestone is usually event driven, so its time is unknown until the trial is simulated. Stopping follow-up of the earlier cohort at that milestone, or after a pre-specified, fixed additional_followup beyond it, makes statistics computed from the two cohorts independent, which facilitates tests requiring independence, e.g., combination tests.

Only patients who are enrolled by the time this function is called and satisfy the subset conditions in ... (if any) are affected. Patients enrolled afterwards are followed as usual.

Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and TrialSimulator has no way to track it. Calling it before any milestone has been triggered is an error.

This is a user-friendly wrapper of the member function of trial, i.e., Trials\$stop_followup(), which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
stop_followup(trial, ..., additional_followup = 0)
```

Arguments

`trial` a trial object returned by `trial()`.

`...` subset conditions compatible with `dplyr::filter`. Follow-up is stopped for selected patients only. If no condition is provided, follow-up is stopped for all patients enrolled by the time this function is called.

`additional_followup` numeric. Extra follow-up time granted to the selected patients after the current milestone. If 0 (default), follow-up stops at the milestone itself.

Value

no return value, called for its side effect of updating `trial`.

<code>summarizeDataFrame</code>	<i>Summarize A Data Frame</i>
---------------------------------	-------------------------------

Description

A minimum alternative to `summarytools::dfSummary` to avoid package dependency. This function is used to generate summary reports of endpoints and arms. No meant to be used by end users. However, users may find it helpful in their own applications if the interface is okay with them.

Usage

```
summarizeDataFrame(
  data,
  exclude_vars = NULL,
  tte_vars = NULL,
  event_vars = NULL,
  categorical_vars = NULL,
  title = "Summary",
  sub_title = ""
)
```

Arguments

`data` a data frame.

`exclude_vars` columns to be excluded from summary.

`tte_vars` character. Vector of time-to-event variables.

`event_vars` character. Vector of event indicators. Every time-to-event variable should be corresponding to an event indicator.

`categorical_vars` character. Vector of categorical variables. This can be used to specify variables with limited distinct values as categorical variables in summary.

`title` character. Title of the summary report.

`sub_title` character. Sub-title.

Value

a data frame of summary

Examples

```
set.seed(123)

n <- 1000
data <- data.frame(
  age = rnorm(n, 65, 10),
  gender = sample(c('M', 'F', NA), n, replace = TRUE, prob = c(.4, .4, .2)),
  time_to_death = rexp(n, .01),
  death = rbinom(n, 1, .6),
  type = sample(LETTERS[1:8], n, replace = TRUE)
)

summarizeDataFrame(data, tte_vars = 'time_to_death', event_vars = 'death')
```

summarizeMilestoneTime

Summary of Milestone Time from Simulated Trials

Description

Summary of Milestone Time from Simulated Trials

Usage

```
summarizeMilestoneTime(output)
```

Arguments

output a data frame. It assumes that triggering time of milestones are store in columns `milestone_time_<...>`. It can be a data frame returned by `controller$get_output()`, or row-binded from multiple data frames returned by `controller$get_output()` (e.g., users may run simulation under the `targets` framework).

Value

A data frame of class `milestone_time_summary`. It comes with a `plot` method for visualization.

Examples

```
# a minimum, meaningful, and executable example,
# where a randomized trial with two arms is simulated and analyzed.

control <- arm(name = 'control arm')
active <- arm(name = 'active arm')
```

```

pfs_in_control <- endpoint(name = 'PFS', type = 'tte', generator = rexp, rate = log(2) / 5)
control$add_endpoints(pfs_in_control)

pfs_in_active <- endpoint(name = 'PFS', type = 'tte', generator = rexp, rate = log(2) / 6)
active$add_endpoints(pfs_in_active)

accrual_rate <- data.frame(end_time = c(10, Inf), piecewise_rate = c(30, 50))
trial <- trial(name = 'trial',
              n_patients = 1000,
              duration = 40,
              enroller = StaggeredRecruiter,
              accrual_rate = accrual_rate,
              dropout = rweibull, shape = 2, scale = 38,
              silent = TRUE)

trial$add_arms(sample_ratio = c(1, 1), control, active)

action_at_final <- function(trial){
  locked_data <- trial$get_locked_data('final analysis')
  fitLogrank(Surv(PFS, PFS_event) ~ arm, placebo = 'control arm',
             data = locked_data, alternative = 'less')
  invisible(NULL)
}

final <- milestone(name = 'final analysis',
                  action = action_at_final,
                  when = eventNumber(endpoint = 'PFS', n = 300))

listener <- listener(silent = TRUE)
listener$add_milestones(final)

controller <- controller(trial, listener)
controller$run(n = 10, plot_event = FALSE, silent = TRUE)

output <- controller$get_output()
time <- summarizeMilestoneTime(output)
time

plot(time)

```

trial

Define a Trial

Description

Define a trial. This is a user-friendly wrapper for the class constructor `Trial$new()`. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Trial's name, planned size/duration, enrollment plan, dropout mechanism and seeding are specified in this function. Note that many of these parameters can be altered adaptively during a trial.

Note that it is users' responsibility to assure that the units of dropout time, trial duration, and readout of non-tte endpoints are consistent.

Usage

```
trial(
  name,
  n_patients,
  duration,
  description = name,
  seed = NULL,
  enroller = StaggeredRecruiter,
  dropout = NULL,
  stratification_factors = NULL,
  silent = FALSE,
  ...
)
```

Arguments

name	character. Name of trial. Usually, hmm..., useless.
n_patients	integer. Maximum (and initial) number of patients could be enrolled when planning the trial. It can be altered adaptively during a trial.
duration	Numeric. Trial duration. It can be altered adaptively during a trial.
description	character. Optional for description of the trial. By default it is set to be trial's name. Usually useless.
seed	random seed. If NULL, seed is set for each simulated trial automatically and saved in output. It can be retrieved in the seed column in <code>\$get_output()</code> . Setting it to be NULL is recommended. For debugging, set it to a specific integer.
enroller	enrollment-time generator. Must be <code>StaggeredRecruiter</code> (the default); any other value is rejected. Supply its <code>accrual_rate</code> via <code>...</code> . See <code>?TrialSimulator::StaggeredRecruiter</code> . The argument is kept (rather than dropped) for backward compatibility: existing code that passes <code>enroller = StaggeredRecruiter</code> explicitly keeps working unchanged, and a named argument gives a clearer error than an "unused argument" failure if some other enroller is supplied.
dropout	a function returning a vector of dropout time for patients. It can be any random number generator with first argument <code>n</code> , the number of enrolled patients. Usually <code>rexp</code> if dropout rate is set at a single time point, or <code>rweibull</code> if dropout rates are set at two time points. See <code>?TrialSimulator::weibullDropout</code> .
stratification_factors	character. Names of baseline characteristics to define stratum in stratified permuted block randomization. Stratification factors must be defined in <code>endpoint()</code> with <code>readout = 0</code> . As a natural assumption for randomized trial, <code>TrialSimulator</code> assumes that the baseline characteristics share the same distribution across arms,

but endpoints can have same or different distributions given baseline characteristics. NULL by default, i.e., unstratified permuted block randomization is executed.

`silent` logical. TRUE to mute messages. However, warning message is still displayed. Usually set it to TRUE in formal simulation. Default: FALSE.

`...` (optional) arguments of `enroller` and `dropout`.

Value

an object of R6 class `Trials`, representing a trial.

Examples

```
risk1 <- data.frame(
  end_time = c(1, 10, 26.0, 52.0),
  piecewise_risk = c(1, 1.01, 0.381, 0.150) * exp(-3.01)
)

pfs1 <- endpoint(name = 'pfs', type='tte',
  generator = PiecewiseConstantExponentialRNG,
  risk = risk1, endpoint_name = 'pfs')

orr1 <- endpoint(
  name = 'orr', type = 'non-tte',
  readout = c(orr=1), generator = rbinom,
  size = 1, prob = .4)

placebo <- arm(name = 'pbo')

placebo$add_endpoints(pfs1, orr1)

risk2 <- risk1
risk2$hazard_ratio <- .8

pfs2 <- endpoint(name = 'pfs', type='tte',
  generator = PiecewiseConstantExponentialRNG,
  risk = risk2, endpoint_name = 'pfs')

orr2 <- endpoint(
  name = 'orr', type = 'non-tte',
  generator = rbinom, readout = c(orr=3),
  size = 1, prob = .6)

active <- arm(name = 'ac')

active$add_endpoints(pfs2, orr2)

## Plan a trial, Trial-3415, of up to 100 patients.
## Enrollment accrues at a constant 10 patients per time unit.
trial <- trial(
  name = 'Trial-3415', n_patients = 100,
  seed = 31415926, duration = 100,
```

```

enroller = StaggeredRecruiter,
accrual_rate = data.frame(end_time = Inf, piecewise_rate = 10))

trial

trial$add_arms(sample_ratio = c(1, 2), placebo, active)

## updated information after arms are registered
trial

```

Trials

Class of Trial

Description

Create a class of trial.

Public methods in this R6 class are used in developing this package. Thus, we have to export the whole R6 class which exposures all public methods. However, only the public methods in the sections below are useful to end users, and users are encouraged to restrict themselves to them. The remaining public methods are internal machinery of the package and should not be called directly (see the last section).

Adaptation methods. The following methods adapt an ongoing trial and should be called within action functions of milestones. Each of them has a user-friendly wrapper of the same name, e.g., `set_duration(trial, ...)` for `trial$set_duration(...)`.

- `$set_duration()` set duration of a trial. This function can be used to extend duration under adaptive designs.
- `$resize()` set maximum sample size of a trial. This function can be used to increase sample size under adaptive designs (e.g., sample size reassessment).
- `$remove_arms()` drop arms from a trial. This function can be used in adaptive designs, e.g., dose selection, enrichment design, etc.
- `$update_sample_ratio()` change sample ratio of arm. This function can be used under adaptive designs, e.g., response-adaptive design, etc.
- `$update_generator()` change endpoint generator of arm. This function can be used in enrichment design.
- `$add_arms()` add arms to a trial. This function is used to add arms to a newly defined trial, or add arms under adaptive design, e.g., dose-ranging, etc.
- `$crossover()` apply a milestone-triggered crossover to eligible patients in the trial. Called inside a milestone action; only alters patients' post-switch endpoint values and leaves already-observed data intact.
- `$stop_followup()` stop follow-up of a subset of enrolled patients at or after a milestone. Their data are censored (time-to-event endpoints) or set to missing (non-time-to-event endpoints) accordingly. This function can be used in adaptive designs, e.g., to simulate treatment discontinuation, early termination of follow-up for a sub-population, or enrichment design

where follow-up of a de-selected sub-population is stopped after an interim analysis. It can also stop follow-up of an earlier cohort at a pre-specified, event-driven milestone (e.g., the last patient of the first cohort), optionally with a fixed additional follow-up beyond it, making statistics of the cohorts independent to facilitate, e.g., combination tests.

- `$update_accrual_rate()` update the accrual rate of the recruitment curve at a milestone, e.g., to revise or pause recruitment after dose selection or enrichment. `end_time` of the new accrual rate is measured from the milestone; patients not yet enrolled are re-planned and re-randomized under the new schedule.
- `$update_milestone()` update the trigger condition and/or the action of a not-yet-triggered milestone. The update takes effect right after the current action function returns.

Methods callable within action functions. In addition to the adaptation methods above, users can call the following methods within action functions to access and manipulate data, to query the current status of a trial, and to carry out statistical testing.

Data access and manipulation:

- `$get_locked_data()` request for data snapshot at a milestone. Calling this function is recommended as the first action in any action function as long as trial data is needed in statistical analysis or decision making.
- `$save()` save intermediate result for simulation summary. Results across multiple replicates of simulation are saved, which can be retrieved by calling `get_output()` anytime.
- `$bind()` row bind and save intermediate results across milestones if those results are data frames of similar formats. The life cycle of the save results is within a single replicate of simulation and is reset to NULL in next simulated trial. Saved results can be retrieved by calling `get()` anytime.
- `$save_custom_data()` save intermediate results of any format. The life cycle of the saved result is within a single replicate of simulation and is reset to NULL in next simulated trial. Saved results can be retrieved by calling `get()` anytime.
- `$get()` retrieve intermediate results saved by calling functions `save_custom_data()` or `bind()`.
- `$get_output()` retrieve intermediate results saved by calling function `save()`.

Trial status queries:

- `$get_current_time()` return the triggering time of the milestone that the calling action function is attached to.
- `$get_milestone_time()` return milestone time when triggering a given milestone.
- `$get_sample_ratio()` return current sample ratios of arms.
- `$get_arms_name()` return names of the arms in the trial at the time of calling, i.e., arms that have been added and not yet removed by `$remove_arms()`. Note that this can differ from the arms present in locked data, where data of removed arms remain available (censored at the time of removal).

Statistical testing:

- `$dunnettTest()` perform Dunnett's test.
- `$closedTest()` perform combination test based on Dunnett's test.

- `$conditionalPower()` compute conditional power at a triggered interim milestone for each treatment-vs-placebo comparison of a time-to-event endpoint, under a design with one interim and one final analysis and a constant allocation ratio.
- `$eventNumberReestimationFromConditionalPower()` re-estimate the number of events at the final analysis: the smallest whole number of events at which conditional power reaches a target, given the observations at a triggered interim milestone, optionally bounded by a practical cap.

Trial setup.

- `$add_regimen()` register a regimen object to a trial. Must be called before `$add_arms()`. Applied at enrollment. Unlike the adaptation methods above, it belongs to the setup stage of a trial and must not be called within action functions.

Internal machinery. The remaining public methods (`$lock_data()`, `$get_data_lock_time_by_calendar_time()`, `$get_data_lock_time_by_event_number()`, `$get_data_lock_time_by_enrollment()`, `$has_arm()`, `$event_plot()`, `$mute()`, `$tidy_output()`, `$reset()`, `$make_arms_snapshot()` and `$pop_milestone_updates()`) are public only because they are invoked on a trial object by other components of the package (milestones, listeners, controllers and triggering conditions), which R6 cannot grant through private members. Users should not call them directly. Note that `$save()` and `$get_output()` are invoked by those components too, but they are part of the user-facing API above at the same time.

Value

an R6Class generator object; use `trial()` to create a trial.

Methods

Public methods:

- `Trials$new()`
- `Trials$set_duration()`
- `Trials$resize()`
- `Trials$remove_arms()`
- `Trials$update_sample_ratio()`
- `Trials$update_generator()`
- `Trials$add_arms()`
- `Trials$crossover()`
- `Trials$stop_followup()`
- `Trials$update_accrual_rate()`
- `Trials$update_milestone()`
- `Trials$get_locked_data()`
- `Trials$save()`
- `Trials$get_output()`
- `Trials$bind()`
- `Trials$save_custom_data()`
- `Trials$get_custom_data()`
- `Trials$get()`

- `Trials$get_current_time()`
- `Trials$get_milestone_time()`
- `Trials$get_sample_ratio()`
- `Trials$get_arms_name()`
- `Trials$dunnettTest()`
- `Trials$closedTest()`
- `Trials$conditionalPower()`
- `Trials$eventNumberReestimationFromConditionalPower()`
- `Trials$add_regimen()`
- `Trials$lock_data()`
- `Trials$get_data_lock_time_by_calendar_time()`
- `Trials$get_data_lock_time_by_event_number()`
- `Trials$get_data_lock_time_by_enrollment()`
- `Trials$has_arm()`
- `Trials$event_plot()`
- `Trials$mute()`
- `Trials$tidy_output()`
- `Trials$reset()`
- `Trials$make_arms_snapshot()`
- `Trials$pop_milestone_updates()`
- `Trials$print()`
- `Trials$clone()`

Method `new()`: initialize a trial

Usage:

```
Trials$new(
  name,
  n_patients,
  duration,
  description = name,
  seed = NULL,
  enroller = StaggeredRecruiter,
  dropout = NULL,
  stratification_factors = NULL,
  silent = FALSE,
  ...
)
```

Arguments:

`name` character. Name of trial. Usually, hmm..., useless.

`n_patients` integer. Maximum (and initial) number of patients could be enrolled when planning the trial. It can be altered adaptively during a trial.

`duration` Numeric. Trial duration. It can be altered adaptively during a trial.

`description` character. Optional for description of the trial. By default it is set to be trial's name. Usually useless.

seed random seed. If NULL, seed is set for each simulated trial automatically and saved in output. It can be retrieved in the seed column in `$get_output()`. Setting it to be NULL is recommended. For debugging, set it to a specific integer.

enroller enrollment-time generator. Must be `StaggeredRecruiter` (the default); any other value is rejected. Supply its `accrual_rate` via `...`. See `?TrialSimulator::StaggeredRecruiter`. Kept (rather than dropped) for backward compatibility, so existing code that passes `enroller = StaggeredRecruiter` explicitly keeps working unchanged.

dropout a function returning a vector of dropout time for patients. It can be any random number generator with first argument `n`, the number of enrolled patients. Usually `rexp` if dropout rate is set at a single time point, or `rweibull` if dropout rates are set at two time points. See `?TrialSimulator::weibullDropout`.

stratification_factors character. Names of baseline characteristics to define strata in stratified permuted block randomization. Stratification factors must be defined in `endpoint()` with `readout = 0`. As a natural assumption for randomized trial, `TrialSimulator` assumes that the baseline characteristics share the same distribution across arms, but endpoints can have same or different distributions given baseline characteristics. NULL by default, i.e., unstratified permuted block randomization is executed.

silent logical. TRUE to mute messages. However, warning message is still displayed.

`...` (optional) arguments of `enroller` and `dropout`.

Method `set_duration()`: set trial duration in an adaptive designed trial. All patients enrolled before resetting the duration are truncated (non-tte endpoints) or censored (tte endpoints) at the original duration. Remaining patients are re-randomized. New duration must be longer than the old one.

Usage:

```
Trials$set_duration(duration)
```

Arguments:

`duration` new duration of a trial. It must be greater than the current duration.

Method `resize()`: resize a trial with a greater sample size. This function is used to update the maximum sample size adaptively after sample size reassessment. Note that this function should be called within action functions. It is users' responsibility to ensure it and `TrialSimulator` has no way to track this.

Usage:

```
Trials$resize(n_patients)
```

Arguments:

`n_patients` integer. Number of maximum sample size of a trial.

Method `remove_arms()`: remove arms from a trial. `enroll_patients()` will be called at the end of this function to enroll all remaining patients after `Trials$get_current_time()`, i.e. no more unenrolled patients could be randomized to removed arms. This function may be used with futility analysis, dose selection, enrichment analysis (sub-population) or interim analysis (early stop for efficacy).

Note that this function should only be called within action functions. It is users' responsibility to ensure it and `TrialSimulator` has no way to track this. In addition, data of the removed arms are censored or truncated by the time of arm removal.

Usage:

```
Trials$remove_arms(arms_name)
```

Arguments:

`arms_name` character vector. Name of arms to be removed.

Method `update_sample_ratio()`: update sample ratios of arms. This could happen after an arm is added or removed. Note that we may update sample ratios of unaffected arms as well. Once sample ratio is updated, trial data should be rolled back with updated randomization queue. Data of unenrolled patients are re-sampled as well.

Usage:

```
Trials$update_sample_ratio(arm_names, sample_ratios)
```

Arguments:

`arm_names` character vector. Name of arms.

`sample_ratios` numeric vector. New sample ratios of arms. If sample ratio is a whole number, the permuted block randomization is adopted; otherwise, `sample()` will be used instead, which can cause imbalance between arms by chance. However, this is fine for simulation.

Method `update_generator()`: update endpoint generator in an arm

Usage:

```
Trials$update_generator(arm_name, endpoint_name, generator, ...)
```

Arguments:

`arm_name` character. Name of an arm.

`endpoint_name` character. A vector of endpoint names whose generator is updated. It must cover all names registered together in the corresponding `endpoint()` call, but their order does not matter.

`generator` a random number generation (RNG) function. See `generator` of `endpoint()`.

`...` optional arguments for generator.

Method `add_arms()`: add one or more arms to the trial. `enroll_patients()` will be called at the end to enroll all remaining patients in `private$randomization_queue`. This function can be used in two scenarios: (1) add arms right after a trial is created (i.e., `Trials$new(...)`). `sample_ratio` and arms added through `...` should be of same length; (2) add arms to a trial already with `arm(s)`.

Note that this function should only be called within action functions. It is users' responsibility to ensure it and `TrialSimulator` has no way to track this.

The trial captures an independent deep copy of every arm it registers. Subsequent changes to the original arm object or its endpoint objects do not affect the trial, and adaptations within the trial (e.g., `update_generator()`) do not modify the original arm. Complete the configuration of an arm before registering it; after registration, change it only through the trial's adaptation methods. The copy covers the arm and endpoint objects themselves; a mutable environment or R6 object captured by a generator function is shared by design of R closures and is not isolated, so generators should not carry external mutable state.

Usage:

```
Trials$add_arms(sample_ratio, ...)
```

Arguments:

`sample_ratio` integer vector. Sample ratio for permuted block randomization. It will be appended to existing sample ratio in the trial.

... one or more objects returned from `arm()`. Randomization is carried out with updated sample ratio of newly added arm. It rolls back all patients after `Trials$get_current_time()`, i.e. redo randomization for those patients. This can be useful to add arms one by one when creating a trial. Note that we can run `Trials$add_arm(sample_ratio1, arm1)` followed by `Trials$add_arm(sample_ratio2, arm2)`. We would expect similar result with `Trials$add_arms(c(sample_ratio1, sample_ratio2), arm1, arm2)`. Note that these two methods won't return exactly the same trial because `randomization_queue` were generated twice in the first approach but only once in the second approach. But statistically, they are equivalent and of the same distribution.

Method `crossover()`: Apply a milestone-triggered crossover to eligible patients in the trial.

Unlike a regimen registered via `add_regimen()` (applied at enrollment), `crossover()` is meant to be called inside a milestone's action function. At the earliest crossover (calendar) time $T = \text{get_current_time()} + \text{delay}$, eligible patients may switch to a new treatment, and only their *post-switch* endpoint values are altered. The triplet is stacked onto the trial's regimen (so it is also re-applied to patients enrolled later), and applied immediately, in place, to all currently-eligible patients.

Eligibility (the pool passed to `what()`) = patients with at least one endpoint still "open" (unobserved, dropout-/duration-aware) at T ; fully-observed patients are excluded. `when()` must return a switch time with $\text{enroll_time} + \text{switch_time} \geq T$ (a crossover cannot predate its opening), otherwise an error is raised. `how()` may only change post-switch outcomes; returning a changed value for a pre-switch/locked cell raises an error.

Two helper columns are injected into `patient_data` for the triplet functions: `earliest_crossover_calendar_time` ($= T$) and `earliest_crossover_time_from_enrollment` ($= \max(T - \text{enroll_time}, 0)$).

Usage:

```
Trials$crossover(what, how, when = NULL, delay = 0, ...)
```

Arguments:

`what` a function selecting which eligible patients crossover and to what new_treatment (NA = no crossover). See `regimen()`. Values of `new_treatment` must not contain '@' or ';', which are reserved for encoding regimen_trajectory.

`how` a function returning the modified post-switch endpoint values.

`when` (optional) a function returning `switch_time` from enrollment. If NULL (default), patients switch at T ($\text{switch_time} = \text{earliest_crossover_time_from_enrollment}$).

`delay` numeric. Time after the milestone before crossover opens; $T = \text{get_current_time()} + \text{delay}$. Default 0.

... (optional) named arguments routed to `what`, `when`, and/or `how`.

Method `stop_followup()`: stop follow-up of a subset of patients at a specified time at or after the current milestone. Data of affected patients are censored (time-to-event endpoints) or set to missing (non-time-to-event endpoints with readout after the stopping time), as if those patients were no longer followed since then. This function can be used in adaptive designs, e.g., to simulate treatment discontinuation, early termination of follow-up for a sub-population, or enrichment design where follow-up of a de-selected sub-population is stopped after an interim analysis. It can

also be called at a pre-specified milestone that splits a trial into cohorts, e.g., a milestone marking the last patient of the first cohort and the first patient of the second cohort. Such a milestone is usually event driven, so its time is unknown until the trial is simulated. Stopping follow-up of the earlier cohort at that milestone, or after a pre-specified, fixed `additional_followup` beyond it, makes statistics computed from the two cohorts independent, which facilitates tests requiring independence, e.g., combination tests.

Only patients who are enrolled by the time this function is called and satisfy the subset conditions in `...` (if any) are affected. Patients enrolled afterwards are followed as usual.

Note that this function should only be called within action functions. It is users' responsibility to ensure it and `TrialSimulator` has no way to track this. Calling it before any milestone has been triggered is an error.

Usage:

```
Trials$stop_followup(..., additional_followup = 0)
```

Arguments:

`...` subset conditions compatible with `dplyr::filter`. Follow-up is stopped for selected patients only. If no condition is provided, follow-up is stopped for all patients enrolled by the time this function is called.

`additional_followup` numeric. Extra follow-up time granted to the selected patients after the current milestone. If 0 (default), follow-up stops at the milestone itself.

Method `update_accrual_rate()`: update the accrual rate of the recruitment curve at a milestone. The enroller of a trial is always `StaggeredRecruiter`; this function replaces its `accrual_rate` for patients not yet enrolled, while enrolled patients are left unchanged. It can be used in adaptive designs, e.g., to revise recruitment after dose selection or enrichment, or to pause recruitment for a period after an interim decision.

`end_time` in `accrual_rate` is measured from the time this function is called (i.e., the current milestone), not from the start of the trial. A milestone is usually event driven, so its calendar time is unknown until the trial is simulated, and a schedule on the calendar time scale could not be pre-specified. Measuring `end_time` from the milestone also lets users state the new plan simply as "from now on": e.g., `data.frame(end_time = c(3, Inf), piecewise_rate = c(20, 35))` means 20 patients per month for the 3 months following the milestone and 35 per month thereafter, whenever the milestone occurs. Following the convention of `StaggeredRecruiter`, the first re-planned patient is enrolled $1 / \text{piecewise_rate}$ after the milestone; a leading window with `piecewise_rate = 0` defers enrollment further. As with other adaptations, patients not yet enrolled are re-randomized and their data are regenerated under the new schedule.

Note that this function should only be called within action functions. Calling it before any milestone has been triggered is an error.

Usage:

```
Trials$update_accrual_rate(accrual_rate)
```

Arguments:

`accrual_rate` a data frame of columns `end_time` and `piecewise_rate` as in `StaggeredRecruiter`, with `end_time` measured from the current milestone. The last `end_time` must be `Inf` with a positive rate.

Method `update_milestone()`: update the trigger condition and/or the action of a not-yet-triggered milestone at a milestone. The milestone to be updated is identified by its name, which

cannot be changed. This function can be used in adaptive designs, e.g., when conditional power at an interim analysis is lower than expected, the final analysis can be postponed by increasing the target number of events in its triggering condition, or its triggering condition can be switched from a calendar time to an event count entirely.

The update is not applied immediately: it is queued and takes effect right after the current action function returns, before the next milestone is evaluated. The new trigger condition and action replace the old ones as a whole. Between simulation replicates the milestone is restored to its as-designed trigger condition and action, so every replicate starts from the original design. A milestone that has already been triggered cannot be updated.

Note that this function should only be called within action functions. Calling it before any milestone has been triggered is an error. Also note that milestones must trigger in their registration order: an updated triggering condition that makes a later-registered milestone fire before an earlier one stops the simulation with an error.

Usage:

```
Trials$update_milestone(name, when = NULL, action = NULL, ...)
```

Arguments:

name character. Name of the milestone to be updated. It must be registered with the listener and not yet triggered.

when (optional) new triggering condition, an object returned by `calendarTime()`, `enrollment()`, `eventNumber()` or their combinations using `&` and `|`. If `NULL`, the triggering condition is left unchanged.

action (optional) new action function. See `action` of `milestone()`. If `NULL`, the action is left unchanged.

... (optional) named arguments of the new action. Only allowed when `action` is provided. The new action is executed with exactly the arguments supplied here: fixed arguments of the previous action are never carried over.

Method `get_locked_data()`: return locked data, i.e. snapshot at a milestone. TTE data is censored and non-TTE data is truncated accounting for readout time and dropout time simultaneously by the triggering time of milestone.

Usage:

```
Trials$get_locked_data(milestone_name)
```

Arguments:

milestone_name character. Milestone name of which the locked data to be extracted.

Method `save()`: save a single value or a one-row data frame to trial's output for further analysis/summary later. Results saved by calling this function have a life cycle of the whole simulation. This means that all results are accumulated across multiple simulated trial and can be used for summary later.

Usage:

```
Trials$save(value, name = "", overwrite = FALSE)
```

Arguments:

value value to be saved. It can be a scalar (vector of length 1) or a data frame (of one row).

name character to name the saved object. It will be used to name a column in trial's output if value is a scalar. If value is a data frame, name will be the prefix pasted with the column name of value in trial's output. If user want to use value's column name as is in trial's output, set name to be '' as default. Otherwise, column name would be, e.g., "{<name>}_<{colnames(value)}>".

overwrite logic. TRUE if overwriting existing entries with warning, otherwise, throwing an error and stop.

Method `get_output()`: return a data frame of all current outputs saved by calling `Trials$save()`. Usually this function is call at the end of simulation for summary.

Usage:

```
Trials$get_output(cols = NULL, simplify = TRUE, tidy = FALSE)
```

Arguments:

cols columns to be returned from `Trial$output`. If NULL, all columns are returned.

simplify logical. Return value rather than a data frame of one column when `length(col) == 1` and `simplify == TRUE`.

tidy logical. `TrialSimulator` automatically records a set of standard outputs at milestones, even when `doNothing` is used as action functions. These includes time of triggering milestones, number of observed events for time-to-event endpoints, and number of non-missing readouts for non-TTE endpoints (see vignette('actionFunctions')). This usually mean a large number of columns in outputs. If users have no intent to summarize a trial on these columns, setting `tidy = TRUE` can eliminate these columns from `get_output()`. Note that currently we use regex `"^n_events_<.*?>_<.*?>$"` and `"^milestone_time_<.*?>$"` to match columns to be eliminated. If users plan to use `tidy = TRUE`, caution is needed when naming custom outputs in `save()`. Default FALSE.

Method `bind()`: row bind a data frame to existing data frame. If a data frame name is not existing in a trial, then it is equivalent to calling `Trials$save_custom_data()`. Extra columns in value are ignored. Columns in `Trials$custom_data[[name]]` but not in value are filled with NA.

This function can be used to save results across multiple milestones. For example, p-values and effect estimates of endpoints may be computed at multiple milestones. Users may want to bind them into a data frame for combination test or graphical test. In this case, this function can be called repeatedly in milestones. Once the data frame is fully conducted, statistical test can be performed on its final version retrieved by calling `Trials$get()`.

Note that data saved by calling this function has a short life cycle within a single simulated trial. It will be reset to NULL before simulated another trial. Thus, it cannot be used to save results that are used for summarizing the simulation.

Usage:

```
Trials$bind(value, name)
```

Arguments:

value a data frame to be saved. It can consist of one or multiple rows.

name character. Name of object to be saved.

Method `save_custom_data()`: save arbitrary (number of) objects into a trial so that users can use those to control the workflow. Most common use case is to store simulation parameters to be used in action functions.

Usage:

```
Trials$save_custom_data(value, name, overwrite = FALSE)
```

Arguments:

value value to be saved. Any type.

name character. Name of the value to be accessed later.

overwrite logic. TRUE if overwriting existing entries with warning, otherwise, throwing an error and stop.

Method `get_custom_data()`: return custom data saved by calling `Trials$save_custom_data()` or `Trials$bind()` with its name.

Usage:

```
Trials$get_custom_data(name)
```

Arguments:

name character. Name of custom data to be accessed.

Method `get()`: alias of function `get_custom_data` to make it short and cool.

Usage:

```
Trials$get(name)
```

Arguments:

name character. Name of custom data to be accessed.

Method `get_current_time()`: return current time of a trial

Usage:

```
Trials$get_current_time()
```

Method `get_milestone_time()`: return milestone time when triggering a given milestone

Usage:

```
Trials$get_milestone_time(milestone_name = NULL)
```

Arguments:

milestone_name character. Name of milestone. If NULL, time of all triggered milestones are returned.

Method `get_sample_ratio()`: return current sample ratio of the trial. The ratio can probably change during the trial (e.g., arm is removed or added)

Usage:

```
Trials$get_sample_ratio(arm_names = NULL)
```

Arguments:

arm_names character vector of arms.

Method `get_arms_name()`: return arms' name of trial

Usage:

```
Trials$get_arms_name()
```

Method `dunnettTest()`: carry out closed test based on Dunnett method under group sequential design.

Usage:

```
Trials$dunnettTest(
  formula,
  placebo,
  treatments,
  milestones,
  alternative,
  planned_info,
  ...
)
```

Arguments:

formula An object of class `formula` that can be used with `survival::coxph`. Must consist arm and endpoint in data. No covariate is allowed. Stratification variables are supported and can be added using `strata(...)`.

placebo character. Name of placebo arm.

treatments character vector. Name of treatment arms to be used in comparison.

milestones character vector. Names of triggered milestones at which either adaptation is applied or statistical testing for endpoint is performed. Milestones in `milestones` does not need to be sorted by their triggering time.

alternative a character string specifying the alternative hypothesis, must be one of "greater" or "less". No default value. "greater" means superiority of treatment over placebo is established by an hazard ratio greater than 1 when a log-rank test is used.

planned_info a data frame of planned number of events of time-to-event endpoint in each stage and each arm. Milestone names, i.e., `milestones` are row names of `planned_info`, and arm names, i.e., `c(placebo, treatments)` are column names. Note that it is not the accumulative but stage-wise event numbers. It is usually not easy to determine these numbers in practice, simulation may be used to get estimates. Note: `planned_info` can also be a character "default" so that `planned_info` are set to be number of newly randomized patients in the control arm in each of the stages. This assumes that event rate do not change over time and/or sample ratio between placebo and a treatment arm does not change as well, which may not be true. It is for the purpose of debugging or rapid implementation only. Using simulation to pick `planned_info` is recommended in formal simulation study. Another issue with `planned_info` set to be "default" is that it is possible patient recruitment is done before a specific stage, as a result, `planned_info` is zero which can crash the program.

... subset condition that is compatible with `dplyr::filter`. `survdif` will be fitted on this subset only to compute one-sided logrank statistics. It could be useful when comparison is made on a subset of treatment arms. By default it is not specified, all data (placebo plus one treatment arm at a time) in the locked data are used to fit the model.

Details: This function computes stage-wise p-values for each of the intersection hypotheses based on Dunnett test. If only one treatment arm is present, it is equivalent to compute the stage-wise p-values of elemental hypotheses. This function also computes inverse normal combination test statistics at each of the stages. The choice of `planned_info` can affect the calculation of stage-wise p-values. Specifically, it is used to compute the columns `observed_info` and

`p_inverse_normal` in returned data frame, which will be used in `Trial$closedTest()`. The choice of `planned_info` can affect the result of `Trial$closedTest()` so user should chose it with caution.

Note that in `Trial$closedTest()`, `observed_info`, which is derived from `planned_info`, will lead to the same closed testing results up to a constant. This is because the closed test uses information fraction $\text{observed_info}/\text{sum}(\text{observed_info})$. As a result, setting `planned_info` to, e.g., $10 * \text{planned_info}$ should give same closed test results.

Based on numerical study, setting `planned_info = "default"` leads to a much higher power (roughly 10%) than setting `planned_info` to median of event numbers at stages, which can be determined by simulation. I am not sure if regulator would support such practice. For example, if a milestone (e.g., interim analysis) is triggered at a pre-specified calendar time, the number of randomized patients is random and is unknown when planning the trial. If I understand it correctly, regulator may want the information fraction in closed test (combined with Dunnett test) to be pre-fixed. In addition, this choice for `planned_info` assumes that the event rates does not change over time which is obviously not true. It is recommended to always use pre-fixed `planned_info` for restrict control of family-wise error rate. It should be pointed out that the choice of pre-fixed `planned_info` can affect statistical power significantly so fine-tuning may be required.

Returns: a list with element names like `arm_name`, `arm1_name|arm2_name`, `arm1_name|arm2_name|arm3_name`, etc., i.e., all possible combination of treatment arms in comparison. Each element is a data frame, with its column names self-explained. Specifically, the columns `p_inverse_normal`, `observed_info`, `is_final` can be used with `GroupSequentialTest` to perform significance test.

Examples:

```
\dontrun{
trial$dunnettTest(Surv(pfs, pfs_event) ~ arm, 'pbo', c('high dose', 'low dose'),
                  listener$get_milestone_names(), 'default')
}
```

Method `closedTest()`: perform closed test based on Dunnett test

Usage:

```
Trials$closedTest(
  dunnett_test,
  treatments,
  milestones,
  alpha,
  alpha_spending = c("asP", "asOF")
)
```

Arguments:

`dunnett_test` object returned by `Trial$dunnettTest()`.

`treatments` character vector. Name of treatment arms to be used in comparison.

`milestones` character vector. Names of triggered milestones at which significance testing for endpoint is performed in closed test. Milestones in `milestones` does not need to be sorted by their triggering time.

`alpha` numeric. Allocated alpha.

alpha_spending alpha spending function. It can be "asP" or "asOF". Note that theoretically it can be "asUser", but it is not tested. It may be supported in the future.

Returns: a data frame of columns arm, decision (final decision on a hypothesis at the end of trial, "accept" or "reject"), milestone_at_reject, and reject_time. If a hypothesis is accepted at then end of a trial, milestone_at_reject is NA, and reject_time is Inf.

Note that if a hypothesis is tested at multiple milestones, the final decision will be "accept" if it is accepted at at least one milestone. The decision is "reject" only if the hypothesis is rejected at all milestones.

Examples:

```
\dontrun{
dt <- trial$dunnettTest(
  Surv(pfs, pfs_event) ~ arm,
  placebo = 'pbo',
  treatments = c('high dose', 'low dose'),
  milestones = c('dose selection', 'interim', 'final'),
  data.frame(pbo = c(100, 160, 80),
             low = c(100, 160, 80),
             high = c(100, 160, 80),
             row.names = c('dose selection', 'interim', 'final'))

trial$closedTest(dt, treatments = c('high dose', 'low dose'),
               milestones = c('interim', 'final'),
               alpha = 0.025, alpha_spending = 'asOF')
}
```

Method conditionalPower(): compute conditional power at a triggered interim milestone for each treatment-vs-placebo comparison of a time-to-event endpoint, under a group sequential design with one interim and one final analysis. Locked data of the milestone is pulled automatically; fitLogrank() is called internally to obtain, for every treatment arm vs placebo, the observed z statistic and the observed number of events d on the two arms of that comparison (after applying subset conditions in . . . , if any). Conditional power is then

$$CP = \Phi \left(\frac{\Phi^{-1}(\alpha) - \sqrt{d/D} z - \theta \sqrt{\omega D} (1 - d/D)}{\sqrt{1 - d/D}} \right)$$

under alternative = 'less' (mirrored for 'greater'), where θ is the log hazard ratio at which conditional power is evaluated and $\omega = r/(1+r)^2$ with r the allocation ratio of the pair recorded when the milestone's data was locked. Like z and d , r is an interim quantity: the result depends on the requested milestone only, not on adaptations applied after it.

The calculation assumes the trial continues as designed: the allocation ratio of the compared arms is constant from the start of enrollment through the final analysis, and the final analysis tests the planned statistic at the planned boundary. A data-dependent design change (e.g., updating the sample ratio based on interim results) alters both the final test statistic and its boundary; such adaptations require a combination-test analysis instead (see \$dunnettTest() and \$closedTest()). It is users' responsibility to call this function only when the calculation is legitimate – in particular, the allocation ratio of the compared arms has not been updated before the milestone, the compared arms are enrolled concurrently with placebo, and subset conditions in . . . are independent of randomization – TrialSimulator has no way to track this.

Conditional power can be requested for an arm that has been removed from the trial: its z and d are well-defined historical quantities, although no further event will accrue on it. With a numeric effect, however, an error is raised for an arm removed before the milestone, as no allocation ratio of the pair is recorded at the milestone.

Usage:

```
Trials$conditionalPower(
  milestone,
  formula,
  placebo,
  alternative,
  alpha,
  D,
  effect,
  ...
)
```

Arguments:

milestone character. Name of a triggered milestone at which the interim results are observed.
formula an object of class formula as in `fitLogrank()`, e.g., `Surv(pfs, pfs_event) ~ arm`.

Stratification via `strata(...)` is supported; no covariate is allowed.

placebo character. Name of the placebo arm.

alternative a character string specifying the alternative hypothesis, must be one of "greater" or "less". No default value. "greater" means superiority of treatment over placebo is established by a hazard ratio greater than 1. See `fitLogrank()`.

alpha numeric. The one-sided nominal significance level(s) corresponding to the planned final critical boundary, in (0, 1): under `alternative = 'less'` the final z statistic is compared with `qnorm(alpha)`. Under a group sequential design it is implied by the alpha spending function, e.g., `1 - pnorm(c)` for a final critical value c on the upper scale; in general it differs from both the total design alpha and the alpha spent, cumulatively or incrementally, at the final look. If a single treatment arm is compared with placebo, an unnamed scalar is accepted; otherwise alpha must be a named vector using treatment arm names, matching the names of **D**. Entries are matched to **D** by name, so the order of components does not matter.

D numeric. Planned number of events at the final analysis for each comparison, counted on the two arms of that comparison (placebo plus one treatment arm). If a single treatment arm is compared with placebo, an unnamed scalar is accepted; otherwise **D** must be a named vector using treatment arm names. A subset of the treatment arms can be specified, in which case conditional power is computed for that subset of comparisons only. **D** and **alpha** must be of the same length and, when named, use the identical set of arm names. An error is raised if the observed number of events d of a comparison already reaches **D**.

effect the treatment effect at which conditional power is evaluated. No default value. 'trend' extrapolates the effect observed at the interim; 'null' assumes no effect for the remaining events (conditional type I error); a single positive numeric value is interpreted as a hazard ratio (e.g., `effect = 0.75`), which is converted internally using the allocation ratio of each pair recorded at the milestone.

... subset conditions compatible with `dplyr::filter`, passed to `fitLogrank()`.

Returns: a data frame with one row per treatment-vs-placebo comparison, with columns `arm`, `placebo`, `z`, `d`, `D`, `info_fraction`, `alpha`, `effect` and `cp`.

Examples:

```
\donttest{
## a two-arm trial with a calendar-time interim
pbo <- arm(name = 'pbo')
pbo$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 10))

trt <- arm(name = 'trt')
trt$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 14))

accrual <- data.frame(end_time = Inf, piecewise_rate = 30)
tr <- trial(name = 'ex', n_patients = 400, duration = 40,
            seed = 31416, enroller = StaggeredRecruiter,
            accrual_rate = accrual, silent = TRUE)
add_arms(tr, sample_ratio = c(1, 1), pbo, trt)

lstn <- listener(silent = TRUE)
lstn$add_milestones(
  milestone(name = 'interim', when = calendarTime(time = 15)),
  milestone(name = 'final', when = calendarTime(time = 40))
)
controller(tr, lstn)$run(n = 1, silent = TRUE, plot_event = FALSE)

## conditional power at the interim trend, with 300 events planned
## at the final analysis and a final boundary at nominal level 0.022
tr$conditionalPower('interim', Surv(pfs, pfs_event) ~ arm,
                    placebo = 'pbo', alternative = 'less',
                    alpha = 0.022, D = 300, effect = 'trend')

## under an assumed hazard ratio instead of the interim trend
tr$conditionalPower('interim', Surv(pfs, pfs_event) ~ arm,
                    placebo = 'pbo', alternative = 'less',
                    alpha = 0.022, D = 300, effect = 0.75)
}
```

Method `eventNumberReestimationFromConditionalPower()`: re-estimate the number of events at the final analysis for each treatment-vs-placebo comparison of a time-to-event endpoint: the smallest whole number of events D , strictly greater than the event number observed at the interim, at which the conditional power computed by `$conditionalPower()` reaches a target, under a group sequential design with one interim and one final analysis. Locked data of the milestone is pulled automatically; `fitLogrank()` is called internally to obtain, for every treatment arm vs placebo, the observed z statistic and the observed number of events d on the two arms of that comparison (after applying subset conditions in ..., if any).

Usage:

```
Trials$eventNumberReestimationFromConditionalPower(
  milestone,
  formula,
  placebo,
```

```

    alternative,
    alpha,
    target_cp,
    effect,
    ...,
    D_cap = NULL
)

```

Arguments:

milestone character. Name of a triggered milestone at which the interim results are observed.
formula an object of class formula as in `fitLogrank()`, e.g., `Surv(pfs, pfs_event) ~ arm`.

Stratification via `strata(...)` is supported; no covariate is allowed.

placebo character. Name of the placebo arm.

alternative a character string specifying the alternative hypothesis, must be one of "greater" or "less". No default value. "greater" means superiority of treatment over placebo is established by a hazard ratio greater than 1. See `fitLogrank()`.

alpha numeric. The one-sided nominal significance level(s) corresponding to the planned final critical boundary, in (0, 1). See `$conditionalPower()`. If a single treatment arm is compared with placebo, an unnamed scalar is accepted; otherwise alpha must be a named vector using treatment arm names, matching the names of `target_cp`. A subset of the treatment arms can be specified, in which case the event number is re-estimated for that subset of comparisons only.

target_cp numeric. Target conditional power(s) in (0, 1). `alpha` and `target_cp` must be of the same length and, when named, use the identical set of arm names. Entries are matched to `alpha` by name, so the order of components does not matter.

effect the treatment effect at which conditional power is evaluated. No default value. 'trend' extrapolates the effect observed at the interim; a single positive numeric value is interpreted as a hazard ratio (e.g., `effect = 0.75`), which is converted internally using the allocation ratio of each pair recorded at the milestone. 'null' is not supported (see Details).

... subset conditions compatible with `dplyr::filter`, passed to `fitLogrank()`.

D_cap NULL or numeric. Practical upper bound(s) of the re-estimated event number, counted on the two arms of each comparison. Whole number(s) greater than the observed `d`, or Inf for an unbounded search. The default NULL is converted internally to scalar Inf for one comparison or a named vector of Inf for multiple comparisons. An explicit scalar Inf applies to all comparisons; a finite unnamed scalar is accepted when a single treatment arm is compared with placebo; otherwise `D_cap` must be a named vector over the same treatment arms as `alpha` and `target_cp`, in which individual entries may be Inf. Placed after `...`, so it must always be passed by name.

Details: The method returns the smallest whole number $D > d$ satisfying

$$CP(D) \geq \gamma,$$

where γ is `target_cp` and $CP(D)$ is the conditional power of `$conditionalPower()` at the same milestone, `alpha` and `effect`. The strict inequality requires a genuine future final analysis. It also applies when the interim z statistic already reaches the final boundary, because future observations can dilute the interim result and reduce conditional power below `target_cp`.

Conditional power need not be monotone in D , particularly when a numeric effect differs from the interim trend. The method finds the stationary points of conditional power by solving a cubic

equation, partitions the integer search range into monotone intervals, and searches them from left to right. Integer bisection within the first interval that reaches `target_cp` therefore returns the earliest crossing required by the rule above, even if conditional power later falls below the target and recovers.

`D_cap` bounds the search by the largest event number considered practical. The complete integer range through the cap is searched:

- if a finite solution exists in the requested range, the smallest such `D` is returned with its conditional power and `target_reached = TRUE`;
- if no finite solution exists in the requested range, `D` and `achieved_cp` are `NA`, while `target_reached = FALSE`. `D_cap` continues to report the requested search cap.

`effect = 'null'` is not supported; use `$conditionalPower()` to compute the conditional type I error at a given `D` instead.

Like `$conditionalPower()`, the calculation assumes the trial continues as designed: constant allocation ratio of the compared arms, planned final statistic, and the planned final boundary held fixed at nominal level `alpha` while `D` varies. Increasing the event number based on a promising interim while keeping the boundary unchanged is the practice studied in the sample size re-estimation literature; whether it is legitimate for the design at hand is users' responsibility, as `TrialSimulator` has no way to track this.

Returns: a data frame with one row per treatment-vs-placebo comparison, with columns `arm`, `placebo`, `z`, `d`, `D`, `D_cap`, `alpha`, `effect`, `target_cp`, `achieved_cp` and `target_reached`. `D` is the smallest solution found in the requested range and `achieved_cp` is the conditional power at that `D`. Both are `NA` when no solution is found. `target_cp` always records the requested target, `D_cap` always records the requested search cap (with the default `NULL` represented as `Inf`), and `target_reached` indicates whether a solution was found.

Examples:

```
\donttest{
## a two-arm trial with a calendar-time interim
pbo <- arm(name = 'pbo')
pbo$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 10))

trt <- arm(name = 'trt')
trt$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 14))

accrual <- data.frame(end_time = Inf, piecewise_rate = 30)
tr <- trial(name = 'ex', n_patients = 400, duration = 40,
            seed = 31416, enroller = StaggeredRecruiter,
            accrual_rate = accrual, silent = TRUE)
add_arms(tr, sample_ratio = c(1, 1), pbo, trt)

lstn <- listener(silent = TRUE)
lstn$add_milestones(
  milestone(name = 'interim', when = calendarTime(time = 15)),
  milestone(name = 'final', when = calendarTime(time = 40))
)
controller(tr, lstn)$run(n = 1, silent = TRUE, plot_event = FALSE)
```

```
## smallest number of final events reaching conditional power 0.9
## under an assumed hazard ratio
tr$eventNumberReestimationFromConditionalPower(
  'interim', Surv(pfs, pfs_event) ~ arm,
  placebo = 'pbo', alternative = 'less',
  alpha = 0.022, target_cp = 0.9, effect = 0.75)

## with a practical cap: when no event number through the cap reaches
## the target, D and achieved_cp are NA and target_reached is FALSE
tr$eventNumberReestimationFromConditionalPower(
  'interim', Surv(pfs, pfs_event) ~ arm,
  placebo = 'pbo', alternative = 'less',
  alpha = 0.022, target_cp = 0.9, effect = 'trend', D_cap = 500)
}
```

Method `add_regimen()`: register regimen to a trial. The regimen consists of three functions to determine the patients who may switch to other treatment during a trial, to determine the switching time and how to update patients' endpoint data accordingly.

The trial captures an independent deep copy of the regimen: triplets appended in-run by `crossover()` do not modify the caller's regimen object. The copy covers the regimen object itself; a mutable environment or R6 object captured by a what/when/how function is shared by design of R closures and is not isolated.

Usage:

```
Trials$add_regimen(regimen)
```

Arguments:

`regimen` an object created by `regimen()`.

Method `lock_data()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

lock data at specific calendar time. For time-to-event endpoints, their event indicator `*_event` should be updated accordingly. Locked data should be stored separately. **DO NOT OVERWRITE/UPDATE** `private$trial_data`! which can lose actual time-to-event information. For example, a patient may be censored at the first data lock. However, he may have event being observed in a later data lock.

Usage:

```
Trials$lock_data(at_calendar_time, milestone_name)
```

Arguments:

`at_calendar_time` time point to lock trial data

`milestone_name` assign milestone name as the name of locked data for future reference.

Method `get_data_lock_time_by_calendar_time()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

given the calendar time to lock the data, return it with event counts of each of the endpoints.

Usage:

```
Trials$get_data_lock_time_by_calendar_time(calendar_time)
```

Arguments:

calendar_time numeric. Calendar time to lock the data

Returns: data lock time

Method get_data_lock_time_by_event_number(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

given a set of endpoints and target number of events, determine the data lock time for a milestone (futility, interim, final, etc.). This function does not change trial object (e.g. rolling back not yet randomized patients after the found data lock time).

Usage:

```
Trials$get_data_lock_time_by_event_number(
  endpoints,
  arms,
  target_n_events,
  type = c("all", "any"),
  ...
)
```

Arguments:

endpoints character vector. Data lock time is determined by a set of endpoints.

arms a vector of arms' name on which number of events will be counted.

target_n_events target number of events for each of the endpoints.

type all if all target number of events are reached. any if the any target number of events is reached.

... subset conditions compatible with dplyr::filter. Number Time of milestone is based on event counts on the subset of trial data.

Returns: data lock time

Method get_data_lock_time_by_enrollment(): **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

given a target number of enrolled patients, determine the data lock time for a milestone (futility, interim, final, etc.). This function does not change trial object (e.g. rolling back not yet randomized patients after the found data lock time). It is similar to get_data_lock_time_by_event_number but only focus on patient_id.

Usage:

```
Trials$get_data_lock_time_by_enrollment(
  arms,
  target_n_patients,
  min_treatment_duration,
  ...
)
```

Arguments:

arms a vector of arms' name on which number of events will be counted.

target_n_patients target number of enrolled patients.

min_treatment_duration numeric. Zero or positive value. minimum treatment duration of enrolled patients. If 0, it looks for triggering time based on number of enrolled patients in population specified by ... and arms. If positive, it means that milestone is

triggered when a specific number of enrolled patients have received treatment for at least `min_treatment_duration` duration. It is users' responsibility to assure that the unit of `min_treatment_duration` are consistent with readout of non-tte endpoints, dropout time, and trial duration.

... subset conditions compatible with `dplyr::filter`. Number Time of milestone is based on event counts on the subset of trial data.

Returns: data lock time

Method `has_arm()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

check if the trial has any arm. Return TRUE or FALSE.

Usage:

```
Trials$has_arm()
```

Method `event_plot()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

plot of cumulative number of events/samples over calendar time.

Usage:

```
Trials$event_plot()
```

Method `mute()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

mute all messages (not including warnings)

Usage:

```
Trials$mute(silent)
```

Arguments:

`silent` logical.

Method `tidy_output()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

control whether the per-arm event count table is saved in trial output at every milestone. It is set by `controller$run()` through its argument `tidy`.

Usage:

```
Trials$tidy_output(tidy)
```

Arguments:

`tidy` logical. If TRUE, the per-arm event count table (output column `n_events_<milestone>_<arms>`) is not saved in trial output; the per-endpoint totals and milestone times are still saved. The table remains available in the attributes of locked data, so `event_plot()` is unaffected.

Method `reset()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

reset a trial to its snapshot taken before it was executed. Seed will be reassigned with a new one. Enrollment time are re-generated. If the trial already have arms when this function is called, they are added back to recruit patients again.

Usage:

```
Trials$reset()
```

Method `make_arms_snapshot()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

make a snapshot of arms

Usage:

```
Trials$make_arms_snapshot()
```

Method `pop_milestone_updates()`: **INTERNAL MACHINERY: DO NOT CALL THIS METHOD DIRECTLY.**

return and clear the queue of milestone update requests scheduled by `update_milestone()` within the current action function. It is called by the listener right after each action function returns, to apply the requested updates to its registered milestones.

Usage:

```
Trials$pop_milestone_updates()
```

Method `print()`: print a trial

Usage:

```
Trials$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Instead of using Trials$new, please use trial(), a user-friendly
# wrapper. See examples in ?trial.

## -----
## Method `Trials$dunnettTest`
## -----

## Not run:
trial$dunnettTest(Surv(pfs, pfs_event) ~ arm, 'pbo', c('high dose', 'low dose'),
                  listener$get_milestone_names(), 'default')

## End(Not run)

## -----
## Method `Trials$closedTest`
## -----

## Not run:
dt <- trial$dunnettTest(
  Surv(pfs, pfs_event) ~ arm,
```

```

placebo = 'pbo',
treatments = c('high dose', 'low dose'),
milestones = c('dose selection', 'interim', 'final'),
data.frame(pbo = c(100, 160, 80),
            low = c(100, 160, 80),
            high = c(100, 160, 80),
            row.names = c('dose selection', 'interim', 'final'))

trial$closedTest(dt, treatments = c('high dose', 'low dose'),
                 milestones = c('interim', 'final'),
                 alpha = 0.025, alpha_spending = 'asOF')

## End(Not run)

## -----
## Method `Trials$conditionalPower`
## -----

## a two-arm trial with a calendar-time interim
pbo <- arm(name = 'pbo')
pbo$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 10))
trt <- arm(name = 'trt')
trt$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 14))

accrual <- data.frame(end_time = Inf, piecewise_rate = 30)
tr <- trial(name = 'ex', n_patients = 400, duration = 40,
            seed = 31416, enroller = StaggeredRecruiter,
            accrual_rate = accrual, silent = TRUE)
add_arms(tr, sample_ratio = c(1, 1), pbo, trt)

lstn <- listener(silent = TRUE)
lstn$add_milestones(
  milestone(name = 'interim', when = calendarTime(time = 15)),
  milestone(name = 'final', when = calendarTime(time = 40))
)
controller(tr, lstn)$run(n = 1, silent = TRUE, plot_event = FALSE)

## conditional power at the interim trend, with 300 events planned
## at the final analysis and a final boundary at nominal level 0.022
tr$conditionalPower('interim', Surv(pfs, pfs_event) ~ arm,
                   placebo = 'pbo', alternative = 'less',
                   alpha = 0.022, D = 300, effect = 'trend')

## under an assumed hazard ratio instead of the interim trend
tr$conditionalPower('interim', Surv(pfs, pfs_event) ~ arm,
                   placebo = 'pbo', alternative = 'less',
                   alpha = 0.022, D = 300, effect = 0.75)

```

```
## -----
## Method `Trials$eventNumberReestimationFromConditionalPower`
## -----

## a two-arm trial with a calendar-time interim
pbo <- arm(name = 'pbo')
pbo$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 10))
trt <- arm(name = 'trt')
trt$add_endpoints(endpoint(name = 'pfs', type = 'tte',
                           generator = rexp, rate = log(2) / 14))

accrual <- data.frame(end_time = Inf, piecewise_rate = 30)
tr <- trial(name = 'ex', n_patients = 400, duration = 40,
            seed = 31416, enroller = StaggeredRecruiter,
            accrual_rate = accrual, silent = TRUE)
add_arms(tr, sample_ratio = c(1, 1), pbo, trt)

lstn <- listener(silent = TRUE)
lstn$add_milestones(
  milestone(name = 'interim', when = calendarTime(time = 15)),
  milestone(name = 'final', when = calendarTime(time = 40))
)
controller(tr, lstn)$run(n = 1, silent = TRUE, plot_event = FALSE)

## smallest number of final events reaching conditional power 0.9
## under an assumed hazard ratio
tr$eventNumberReestimationFromConditionalPower(
  'interim', Surv(pfs, pfs_event) ~ arm,
  placebo = 'pbo', alternative = 'less',
  alpha = 0.022, target_cp = 0.9, effect = 0.75)

## with a practical cap: when no event number through the cap reaches
## the target, D and achieved_cp are NA and target_reached is FALSE
tr$eventNumberReestimationFromConditionalPower(
  'interim', Surv(pfs, pfs_event) ~ arm,
  placebo = 'pbo', alternative = 'less',
  alpha = 0.022, target_cp = 0.9, effect = 'trend', D_cap = 500)
```

update_accrual_rate *Updating the Accrual Rate of a Trial at a Milestone*

Description

update the accrual rate of the recruitment curve at a milestone. The enroller of a trial is always `StaggeredRecruiter`; this function replaces its `accrual_rate` for patients not yet enrolled, while enrolled patients are left unchanged. This function can be used in adaptive designs; its application

includes, but is not limited to, revising recruitment after dose selection or enrichment, or pausing recruitment for a period after an interim decision.

end_time in accrual_rate is measured from the time this function is called (i.e., the current milestone), not from the start of the trial. A milestone is usually event driven, so its calendar time is unknown until the trial is simulated, and a schedule on the calendar time scale could not be pre-specified. Measuring end_time from the milestone also lets users state the new plan simply as "from now on": e.g., data.frame(end_time = c(3, Inf), piecewise_rate = c(20, 35)) means 20 patients per month for the 3 months following the milestone and 35 per month thereafter, whenever the milestone occurs. Following the convention of StaggeredRecruiter, the first re-planned patient is enrolled 1/piecewise_rate after the milestone; a leading window with piecewise_rate = 0 defers enrollment further. As with other adaptations, patients not yet enrolled are re-randomized and their data are regenerated under the new schedule.

Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and TrialSimulator has no way to track it. Calling it before any milestone has been triggered is an error.

This is a user-friendly wrapper of the member function of trial, i.e., Trials\$update_accrual_rate(), which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

update_accrual_rate(trial, accrual_rate)

Arguments

- trial a trial object returned by trial().
- accrual_rate a data frame of columns end_time and piecewise_rate as in StaggeredRecruiter, with end_time measured from the current milestone. The last end_time must be Inf with a positive rate.

Value

no return value, called for its side effect of updating trial.

update_generator	<i>Updating Data Generators of One or More Endpoints in an Arm</i>
------------------	--

Description

update endpoint generator in an arm. This function can be useful in, e.g., enrichment design where generator is updated so that patients are enrolled from sub-population afterwards. This function can also be used when data model changes over time, i.e., generating data in a new way after a milestone. This function can be called multiple times to update generators of endpoints one by one. Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and TrialSimulator has no way to track it.

This is a user-friendly wrapper of the member function of trial, i.e., `Trials$update_generator()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
update_generator(trial, arm_name, endpoint_name, generator, ...)
```

Arguments

<code>trial</code>	a trial object returned by <code>trial()</code> .
<code>arm_name</code>	character. Name of an arm.
<code>endpoint_name</code>	character. A vector of endpoint names whose generator is updated.
<code>generator</code>	a random number generation (RNG) function. See <code>generator of endpoint()</code> .
<code>...</code>	optional arguments for generator.

Value

no return value, called for its side effect of updating `trial`.

<code>update_milestone</code>	<i>Updating a Not-Yet-Triggered Milestone of a Trial</i>
-------------------------------	--

Description

update the trigger condition and/or the action of a not-yet-triggered milestone. The milestone to be updated is identified by its name, which cannot be changed. This function can be used in adaptive designs, e.g., when conditional power at an interim analysis is lower than expected, the final analysis can be postponed by increasing the target number of events in its triggering condition, or its triggering condition can be switched from a calendar time to an event count entirely.

The update is not applied immediately: it is queued and takes effect right after the current action function returns, before the next milestone is evaluated. The new trigger condition and action replace the old ones as a whole. Between simulation replicates the milestone is restored to its as-designed trigger condition and action, so every replicate starts from the original design. A milestone that has already been triggered cannot be updated.

Note that this function should only be called within action functions of milestones. It is users' responsibility to ensure that and `TrialSimulator` has no way to track it. Calling it before any milestone has been triggered is an error. Also note that milestones must trigger in their registration order: an updated triggering condition that makes a later-registered milestone fire before an earlier one stops the simulation with an error.

This is a user-friendly wrapper of the member function of trial, i.e., `Trials$update_milestone()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
update_milestone(trial, name, when = NULL, action = NULL, ...)
```

Arguments

<code>trial</code>	a trial object returned by <code>trial()</code> .
<code>name</code>	character. Name of the milestone to be updated. It must be registered with the listener and not yet triggered.
<code>when</code>	(optional) new triggering condition, an object returned by <code>calendarTime()</code> , <code>enrollment()</code> , <code>eventNumber()</code> or their combinations using <code>&</code> and <code> </code> . If <code>NULL</code> , the triggering condition is left unchanged.
<code>action</code>	(optional) new action function. See <code>action</code> of <code>milestone()</code> . If <code>NULL</code> , the action is left unchanged.
<code>...</code>	(optional) named arguments of the new action. Only allowed when <code>action</code> is provided. The new action is executed with exactly the arguments supplied here: fixed arguments of the previous action are never carried over.

Value

no return value, called for its side effect of updating `trial`.

<code>update_sample_ratio</code>	<i>Updating Sampling Ratios of Existing Arms in a Trial</i>
----------------------------------	---

Description

update sample ratios of arms. This could be called after an arm is added or removed. Sample ratios can be updated for any existing arms.

This is a user-friendly wrapper of the member function of `trial`, i.e., `Trials$update_sample_ratio()`, which is used in vignettes. Users who are not familiar with the concept of classes may consider using this wrapper directly.

Usage

```
update_sample_ratio(trial, arm_names, sample_ratios)
```

Arguments

<code>trial</code>	a trial object returned by <code>trial()</code> .
<code>arm_names</code>	character vector. Name of arms.
<code>sample_ratios</code>	numeric vector. New sample ratios of arms. If sample ratio is a whole number, the permuted block randomization is adopted; otherwise, <code>sample()</code> will be used instead, which can cause imbalance between arms by chance. However, this is usually fine for simulation.

Value

no return value, called for its side effect of updating trial.

weibullDropout	<i>Calculate Parameters of Weibull Distribution as a Dropout Method</i>
----------------	---

Description

Fit scale and shape parameters of the Weibull distribution to match dropout rates at two specified time points. Weibull distribution can be used as a dropout distribution because it has two parameters. Note that It is users' responsibility to assure that the units of dropout time, readout of non-tte endpoints, and trial duration are consistent.

Usage

```
weibullDropout(time, dropout_rate)
```

Arguments

time a numeric vector of two time points at which dropout rates are specified.
dropout_rate a numeric vector of dropout rates at time.

Value

a named vector for scale and shape parameters.

Examples

```
## dropout rates are 8% and 18% at time 12 and 18.  
weibullDropout(time = c(12, 18), dropout_rate = c(.08, .18))
```

Index

`add_arms`, [3](#)
`arm`, [4](#)
`Arms`, [5](#)

`calendarTime`, [8](#), [26](#), [27](#)
`controller`, [9](#)
`Controllers`, [10](#)
`CorrelatedPfsAndOs2`, [12](#)
`CorrelatedPfsAndOs3`, [14](#)
`CorrelatedPfsAndOs4`, [15](#)
`crossover`, [16](#)

`doNothing`, [17](#)
`DynamicRNGFunction`, [18](#)

`endpoint`, [19](#)
`Endpoints`, [22](#)
`enrollment`, [8](#), [26](#), [27](#)
`eventNumber`, [8](#), [26](#), [27](#)
`expandRegimen`, [28](#)

`fitCoxph`, [29](#)
`fitFarringtonManning`, [30](#)
`fitLinear`, [31](#)
`fitLogistic`, [32](#)
`fitLogrank`, [33](#)

`getAdaptiveDesignOutput`, [34](#)
`getConditionalPowerOutput`, [35](#)
`getDoseRangingOutput`, [35](#)
`getFixedDesignOutput`, [35](#)
`GraphicalTesting`, [36](#)
`GroupSequentialTest`, [44](#)

`listener`, [48](#)
`Listeners`, [49](#)

`milestone`, [52](#)
`Milestones`, [53](#)

`PiecewiseConstantExponentialRNG`, [55](#)

`plot.milestone_time_summary`, [56](#)
`plot.three_state_model`, [57](#)

`qPiecewiseExponential`, [57](#)

`rconst`, [59](#)
`regimen`, [60](#)
`Regimens`, [61](#)
`remove_arms`, [64](#)
`resize`, [65](#)

`set_duration`, [66](#)
`solveMixtureExponentialDistribution`,
[66](#)
`solvePiecewiseConstantExponentialDistribution`,
[68](#)
`solveThreeStateModel`, [69](#)
`StaggeredRecruiter`, [70](#)
`stop_followup`, [72](#)
`summarizeDataFrame`, [73](#)
`summarizeMilestoneTime`, [74](#)

`trial`, [75](#)
`Trials`, [78](#)

`update_accrual_rate`, [101](#)
`update_generator`, [102](#)
`update_milestone`, [103](#)
`update_sample_ratio`, [104](#)

`weibullDropout`, [105](#)