Package 'TrialSimulator'

December 19, 2025

Type Package

Title Clinical Trial Simulator

Version 1.7.0

Description Simulate phase II and/or phase III clinical trials. It supports various types of endpoints 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, dplyr, emmeans, ggplot2, gMCPLite, htmltools, mvtnorm, R6, rlang, rpact, rstudioapi, survival, utils

RoxygenNote 7.3.2

Suggests DoseFinding, graphicalMCP, kableExtra, knitr, 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 no

Author Han Zhang [cre, aut]

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

Repository CRAN

Date/Publication 2025-12-19 17:50:02 UTC

Contents

`arm` *Define an Arm*

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 `Trials$get_locked_data()` in action functions.
...	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 `....`

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.

- `$add_endpoints()`
- `$print()`

Methods

Public methods:

- `Arms$new()`
- `Arms$add_endpoints()`
- `Arms$get_name()`
- `Arms$get_number_endpoints()`
- `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

`...` 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()`: return name of arm.

Usage:

```
Arms$get_name()
```

Method `get_number_endpoints()`: return number of endpoints in the arm.

Usage:

```
Arms$get_number_endpoints()
```

Method `has_endpoint()`: 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()`: return name of endpoints registered to the arm.

Usage:

```
Arms$get_endpoints_name()
```

Method `update_endpoint_generator()`: 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

trial	an object returned from `trial()`.
listener	an object returned from `listener()`.

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.

- `$run()`
- `$get_output()`
- `$reset()`

Methods

Public methods:

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 `get_listener()`**:** return listener in a controller.

Usage:

```
Controllers$get_listener()
```

Method `get_trial()`**:** return trial in a controller.

Usage:

```
Controllers$get_trial()
```

Method `mute()`**:** mute all messages (not including warnings).

Usage:

```
Controllers$mute()
```

Arguments:

`silent` logical.

Method `reset()`**:** reset the trial and listener registered to the controller before running additional replicate of simulation. This is usually done between two calls of `controller$run()`.

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.

Usage:

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

Arguments:

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

`plot_event` logical. Create event plot if `FALSE`. Users should set it to be `FALSE` if `n > 1`.

`silent` logical. `TRUE` if muting all messages during a trial. Note that warning messages are still displayed.

`dry_run` logical. We are considering retire this argument. `TRUE` if action function provided by users is ignored and an internal default action `.default_action` is called instead. This default function only locks data when the milestone is triggered. Milestone time and number of endpoints' events or sample sizes are saved. It is suggested to set `dry_run = TRUE` to estimate distributions of triggering time and number of events before formally using custom action functions if a fixed design is in use. This helps determining planned maximum information for group sequential design and reasonable time of milestone of interest when planning a trial. Set it to `FALSE` for formal simulations. However, for an adaptive design where arm(s) could possibly be added or removed, setting `dry_run` to `TRUE` is usually not helpful because adaption should be executed before estimating the milestone time.

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

```
##
```

CorrelatedPfsAndOs3 *Generate Correlated PFS and OS*

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

`n`	integer. Number of observations.
`h01`	constant transition hazard from state "initial" to state "progression".
`h02`	constant transition hazard from state "initial" to state "death".
`h12`	constant transition hazard from state "progression" to state "death".
`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

```
## 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

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

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
```

---

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 *A wrapper of random number generator.*

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

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

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 *Define Endpoints*

Description

Define one or multiple endpoints. This is a user-friendly wrapper for the class constructor `Endpoint$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 = c("tte", "non-tte"), 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. `TrialSimulator` will do some verification if an endpoint is of type `"tte"`. However, no special manipulation is done for non-tte endpoints.
`readout`	numeric vector named by non-tte endpoint(s). `readout` should 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 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 `c(baseline = 0, ep1 = 2, ep2 = 4)`. 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 `NULL`.
`generator`	a RNG function. Its first argument must be `n`, number of patients. It must return a data frame of `n` rows. It supports all univariate random number generators, like those in `stats`, e.g., `stats::rnorm`, `stats::rexp`, etc. that with `n` as the first argument for number of observations. `generator` could be any custom functions as long as (1) its first argument is `n`; 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.

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('simulatePfsAndOs')
```

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()`

Methods

Public methods:

- `Endpoints$new()`
- `Endpoints$test_generator()`
- `Endpoints$get_generator()`
- `Endpoints$update_generator()`
- `Endpoints$get_readout()`
- `Endpoints$get_uid()`
- `Endpoints$get_name()`
- `Endpoints$get_type()`
- `Endpoints$print()`
- `Endpoints$clone()`

Method `new()`**:** initialize an endpoint.

Usage:

```
Endpoints$new(name, type = c("tte", "non-tte"), 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., continous, 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.

`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 been 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()`**:** 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_readout()`**:** return readout function

Usage:

```
Endpoints$get_readout()
```

Method `get_uid()`**:** return uid

Usage:

```
Endpoints$get_uid()
```

Method `get_name()`**:** return endpoints' name

Usage:

```
Endpoints$get_name()
```

Method `get_type()`**:** return endpoints' type

Usage:

```
Endpoints$get_type()
```

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 dplyr::filter. 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 *Triggering Condition by Number of Events or Non-missing Observations of an Endpoint*

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 `endpoint()`.
`n`	integer. Targeted number of events or non-missing obervations, depending on the type of endpoint.
`...`	subset conditions compatible with `dplyr::filter`. 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 `remove_arms()`) by the time of calculation.

Value

an object of class 'Condition'

`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 survival::coxph. 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 data$arm.
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 dplyr::filter. coxph 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., fitCoxph(formula, 'pbo', data, 'less', 'log hazard ratio', arm %in% c('pbo', 'low dose'), x > 0.5), which is equivalent to: fitCoxph(formula, 'pbo', data, 'less', 'log hazard ratio', 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 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 three 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

endpoint	Character. Name of the endpoint in data.
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 rate difference greater than 'delta'.
...	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., fitFarringtonManning('remission', 'pbo', data, delta, arm %in% c('pbo', 'low dose'), cfb > 0.5), which is equivalent to: fitFarringtonManning('remission', 'pbo', data, delta, 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.
delta	the rate difference between a treatment arm and placebo under the null. 0 by default.

Value

a data frame with three 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` *Fit linear regression model*

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 `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., `fitLinear(cfb ~ arm, 'pbo', data, 'greater', arm %in% c('pbo', 'low dose'), cfb > 0.5)`, which is equivalent to: `fitLinear(cfb ~ 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` *Carry out log rank test*

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 `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. String of placebo in `data$arm`.
`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`. `survival::coxph` with `ties = "exact"` will be fitted 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 three 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 statistics of log hazard ratios.

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.

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 *Class of GraphicalTesting*

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 gMCPLite:
## 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 futher 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.

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
```

`listener` *Define a Listener*

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.

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()`

Methods

Public methods:

- `Listeners$new()`
- `Listeners$add_milestones()`
- `Listeners$get_milestones()`
- `Listeners$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_milestones()`**:** return registered milestones

Usage:

```
Listeners$get_milestones(milestone_name = NULL)
```

Arguments:

`milestone_name` return `Milestone` object with given name(s). If `NULL`, all registered milestones are returned.

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

Usage:

```
Listeners$get_milestone_names()
```

Method `monitor()`**:** scan, check, and trigger registered milestones. Milestones are triggered in the order when calling `Listener$add_milestones`.

Usage:

```
Listeners$monitor(trial, dry_run)
```

Arguments:

`trial` a `Trial` object.

`dry_run` logical. See `Controller::run` for more information.

Method `mute()`**:** mute all messages (not including warnings)

Usage:

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

Arguments:

`silent` logical.

Method `reset()`**:** 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 taks value returned from functions `calendarTime()`, `enrollment()`, `eventNumber()` 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 `doNothing`.
`...`	(optional) arguments of `action`.

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`.

Methods

Public methods:

- `Milestones$new()`
- `Milestones$get_name()`
- `Milestones$get_type()`
- `Milestones$get_trigger_condition()`
- `Milestones$get_action()`
- `Milestones$set_dry_run()`
- `Milestones$execute_action()`
- `Milestones$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()`: return name of milestone

Usage:

```
Milestones$get_name()
```

Method `get_type()`: return type(s) of milestone

Usage:

```
Milestones$get_type()
```

Method `get_trigger_condition()`: return trigger_condition function

Usage:

```
Milestones$get_trigger_condition()
```

Method `get_action()`: return action function

Usage:

```
Milestones$get_action()
```

Method `set_dry_run()`: set if dry run should be carried out for the milestone. For more details, refer to `Controller::run`.

Usage:

```
Milestones$set_dry_run(dry_run)
```

Arguments:

`dry_run` logical.

Method `execute_action()`: execute action function

Usage:

```
Milestones$execute_action(trial)
```

Arguments:

`trial` a `Trial` object.

Method `get_trigger_status()`: return trigger status

Usage:

```
Milestones$get_trigger_status()
```

Method `reset()`: 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.

Usage:

```
Milestones$reset()
```

Method `trigger_milestone()`: 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()`: 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.

plot.three_state_model *Plot result of three-state ill-death model*

Description

Plot result of three-state ill-death model

Usage

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

Arguments

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

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.

`solveMixtureExponentialDistribution` *Solve Parameters in a Mixture Exponential Distribution*

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

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

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
```

---

solveThreeStateModel *Solve Parameters in a Three-State Ill-Death Model*

Description

The ill-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 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 ill-death model ensures a nice property that PFS <= 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 ill-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

`median_pfs`	numeric. Median of PFS.
`median_os`	numeric. Median of OS.
`corr`	numeric vector. Pearson correlation coefficients between PFS and OS.
`h12`	numeric vector. A set of hazard from progression to death that may induce the target correlation `corr` given `median_pfs` and `median_os`. `solveThreeStateModel()` 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 Peason'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 *Generate Enrollment Time from Piecewise Constant Uniform Distribution*

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 random numbers.
`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.
	`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`.

Examples

```
accrual_rate <- data.frame(
  end_time = c(12, 13:17, Inf),
  piecewise_rate = c(15, 15 + 6 * (1:5), 45)
)

StaggeredRecruiter(30, accrual_rate)

accrual_rate <- data.frame(
  end_time = c(3, 4, 5, 8, Inf),
  piecewise_rate = c(1, 2, 2, 3, 4)
)

StaggeredRecruiter(30, accrual_rate)
```

summarizeDataFrame *Summarize A Data Frame*

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,
  dropout = 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	a function returning a vector enrollment time for patients. Its first argument n is the number of enrolled patients. Set it to StaggeredRecruiter can handle most of the use cases. See ?TrialSimulator::StaggeredRecruiter for more information.
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.
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.

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 time follows an exponential distribution, with median 5
trial <- trial(
  name = 'Trial-3415', n_patients = 100,
  seed = 31415926, duration = 100,
  enroller = rexp, rate = log(2) / 5)

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 list below are useful to end users.

- `$set_duration()` set duration of a trial. This function can be used to extend duration under adaptive designs.
- `$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.
- `$add_arms()` add arms to a trial. This function is used to add arms to a newly defined trial, or add arms under adaptive design, e.g., dose-ranging, etc.
- `$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()`.
- `$get_output()` retrieve intermediate results saved by calling function `save()`.
- `$dunnettTest()` perform Dunnett's test.
- `$closedTest()` perform combination test based on Dunnett's test.

Methods

Public methods:

- `Trials$new()`
- `Trials$get_trial_data()`
- `Trials$get_duration()`
- `Trials$set_duration()`
- `Trials$set_enroller()`
- `Trials$get_enroller()`
- `Trials$set_dropout()`
- `Trials$get_dropout()`
- `Trials$roll_back()`
- `Trials$remove_arms()`
- `Trials$update_sample_ratio()`
- `Trials$update_generator()`
- `Trials$add_arms()`
- `Trials$get_name()`
- `Trials$get_description()`
- `Trials$get_arms()`
- `Trials$get_arms_name()`
- `Trials$get_number_arms()`
- `Trials$has_arm()`
- `Trials$get_an_arm()`
- `Trials$get_sample_ratio()`
- `Trials$get_number_patients()`
- `Trials$get_number_enrolled_patients()`
- `Trials$get_number_unenrolled_patients()`
- `Trials$get_randomization_queue()`
- `Trials$get_enroll_time()`

- Trials$enroll_patients()
- Trials$set_current_time()
- Trials$get_current_time()
- Trials$get_event_tables()
- Trials$get_data_lock_time_by_event_number()
- Trials$get_data_lock_time_by_calendar_time()
- Trials$get_locked_data()
- Trials$get_locked_data_name()
- Trials$get_event_number()
- Trials$save_milestone_time()
- Trials$get_milestone_time()
- Trials$lock_data()
- Trials$event_plot()
- Trials$censor_trial_data()
- Trials$save()
- Trials$bind()
- Trials$save_custom_data()
- Trials$get_custom_data()
- Trials$get()
- Trials$get_output()
- Trials$mute()
- Trials$independentIncrement()
- Trials$dunnettTest()
- Trials$closedTest()
- Trials$get_seed()
- Trials$print()
- Trials$get_snapshot_copy()
- Trials$make_snapshot()
- Trials$make_arms_snapshot()
- Trials$reset()
- Trials$set_arm_added_time()
- Trials$get_arm_added_time()
- Trials$set_arm_removal_time()
- Trials$get_arm_removal_time()
- Trials$clone()

Method `new()`**:** initialize a trial

Usage:

```
Trials$new(
  name,
  n_patients,
  duration,
  description = name,
```

```
  seed = NULL,
  enroller,
  dropout = 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` a function returning a vector enrollment time for patients. Its first argument `n` is the number of enrolled patients. Set it to `StaggeredRecruiter` can handle most of the use cases. See `?TrialSimulator::StaggeredRecruiter` for more information.

`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`.

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

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

Method `get_trial_data()`**:** return trial data of enrolled patients at the time of this function is called

Usage:

```
Trials$get_trial_data()
```

Method `get_duration()`**:** return maximum duration of a trial

Usage:

```
Trials$get_duration()
```

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 `set_enroller()`**:** set recruitment curve when initialize a trial.

Usage:

```
Trials$set_enroller(func, ...)
```

Arguments:

`func` function to generate enrollment time. It can be built-in function like 'rexp' or customized functions like 'StaggeredRecruiter'.

`...` (optional) arguments for `func`.

Method `get_enroller()`**:** get function of recruitment curve

Usage:

```
Trials$get_enroller()
```

Method `set_dropout()`**:** set distribution of drop out time. This can be done when initialize a trial, or when updating a trial in adaptive design.

Usage:

```
Trials$set_dropout(func, ...)
```

Arguments:

`func` function to generate dropout time. It can be built-in function like 'rexp' or customized functions.

`...` (optional) arguments for `func`.

Method `get_dropout()`**:** get generator of dropout time

Usage:

```
Trials$get_dropout()
```

Method `roll_back()`**:** roll back data to current time of trial. By doing so, `Trial$trial_data` will be cut at current time, and data after then are deleted. However, `Trial$enroll_time` after current time are kept unchanged because that is planned enrollment curve.

Usage:

```
Trials$roll_back()
```

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.

`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.

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$add_arm(sample_ratio2, arm2)`. We would expected similar result with `Trials$add_arms(c(sample_ratio1, sample_ratio2), arm1, arm2)`. Note that these two method 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 `get_name()`**:** return name of trial

Usage:

```
Trials$get_name()
```

Method `get_description()`**:** return description of trial

Usage:

```
Trials$get_description()
```

Method `get_arms()`: return a list of arms in the trial

Usage:

```
Trials$get_arms()
```

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

Usage:

```
Trials$get_arms_name()
```

Method `get_number_arms()`: get number of arms in the trial

Usage:

```
Trials$get_number_arms()
```

Method `has_arm()`: check if the trial has any arm. Return `TRUE` or `FALSE`.

Usage:

```
Trials$has_arm()
```

Method `get_an_arm()`: return an arm

Usage:

```
Trials$get_an_arm(arm_name)
```

Arguments:

`arm_name` character, name of arm to be extracted

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_number_patients()`: return number of patients when planning the trial

Usage:

```
Trials$get_number_patients()
```

Method `get_number_enrolled_patients()`: return number of enrolled (randomized) patients

Usage:

```
Trials$get_number_enrolled_patients()
```

Method `get_number_unenrolled_patients()`: return number of unenrolled patients

Usage:

```
Trials$get_number_unenrolled_patients()
```

Method `get_randomization_queue()`: return randomization queue of planned but not yet enrolled patients. This function does not update randomization_queue, just return its value for debugging purpose.

Usage:

```
Trials$get_randomization_queue(index = NULL)
```

Arguments:

`index` index to be extracted. Return all queue if `NULL`.

Method `get_enroll_time()`**:** return enrollment time of planned but not yet enrolled patients. This function does not update enroll_time, just return its value for debugging purpose.

Usage:

```
Trials$get_enroll_time(index = NULL)
```

Arguments:

`index` index to extract. Return all enroll time if `NULL`.

Method `enroll_patients()`**:** assign new patients to pre-planned randomization queue at pre-specified enrollment time.

Usage:

```
Trials$enroll_patients(n_patients = NULL)
```

Arguments:

`n_patients` number of new patients to be enrolled. If `NULL`, all remaining patients in plan are enrolled. Error may be triggered if n_patients is greater than remaining patients as planned.

Method `set_current_time()`**:** set current time of a trial. Any data collected before could not be changed. private$now should be set after a milestone is triggered (through Milestones class, futility, interim, etc), an arm is added or removed at a milestone

Usage:

```
Trials$set_current_time(time)
```

Arguments:

`time` current calendar time of a trial.

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

Usage:

```
Trials$get_current_time()
```

Method `get_event_tables()`**:** count accumulative number of events (for TTE) or non-missing samples (otherwise) over calendar time (enroll time + tte for TTE, or enroll time + readout otherwise)

Usage:

```
Trials$get_event_tables(arms = NULL, ...)
```

Arguments:

`arms` a vector of arms' name on which the event tables are created. if `NULL`, all arms in the trial will be used.

`...` subset conditions compatible with `dplyr::filter`. Event tables will be counted on subset of trial data only.

Method `get_data_lock_time_by_event_number()`: 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_calendar_time()`: 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, arms)
```

Arguments:

`calendar_time` numeric. Calendar time to lock the data

`arms` a vector of arms' name on which number of events will be counted.

Returns: data lock time

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 `get_locked_data_name()`: return names of locked data

Usage:

```
Trials$get_locked_data_name()
```

Method `get_event_number()`: return number of events at lock time of milestones

Usage:

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

Arguments:

`milestone_name` names of triggered milestones. Use all triggered milestones if `NULL`.

Method `save_milestone_time()`**:** save time of a new milestone.

Usage:

```
Trials$save_milestone_time(milestone_time, milestone_name)
```

Arguments:

`milestone_time` numeric. Time of new milestone.

`milestone_name` character. Name of new milestone.

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 `lock_data()`**:** 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 `event_plot()`**:** plot of cumulative number of events/samples over calendar time.

Usage:

```
Trials$event_plot()
```

Method `censor_trial_data()`**:** censor trial data at calendar time

Usage:

```
Trials$censor_trial_data(
  censor_at = NULL,
  selected_arms = NULL,
  enrolled_before = Inf
)
```

Arguments:

`censor_at` time of censoring. It is set to trial duration if `NULL`.

`selected_arms` censoring is applied to selected arms (e.g., removed arms) only. If `NULL`, it will be set to all available arms in trial data. Otherwise, censoring is applied to user-specified arms only. This is necessary because number of events/sample size in removed arms should be fixed unchanged since corresponding milestone is triggered. In that case, one can update trial data by something like `censor_trial_data(censor_at = milestone_time, selected_arms = removed_arms)`.

`enrolled_before` censoring is applied to patients enrolled before specific time. This argument would be used when trial duration is updated by `set_duration`. Adaptation happens when `set_duration` is called so we fix duration for patients enrolled before adaptation to maintain independent increment. This should work when trial duration is updated for multiple times.

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 `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_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 `mute()`: mute all messages (not including warnings)

Usage:

```
Trials$mute(silent)
```

Arguments:

`silent` logical.

Method `independentIncrement()`**:** calculate independent increments from a given set of milestones

Usage:

```
Trials$independentIncrement(
  formula,
  placebo,
  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. String of placebo in trial's locked data.

`milestones` a character vector of milestone names in the trial, e.g., `listener$get_milestone_names()`.

`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 vector of planned accumulative number of event of time-to-event endpoint. It is named by milestone names. Note: `planned_info` can also be a character `"oracle"` so that planned number of events are set to be observed number of events, in that case inverse normal z statistics equal to one-sided logrank statistics. This is for the purpose of debugging only. In formal simulation, `"oracle"` should not be used if adaptation is present. Pre-fixed `planned_info` should be used to create weights in combination test that controls the family-wise error rate in the strong sense.

`...` subset condition that is compatible with `dplyr::filter`. `survdiff` will be fitted on this subset only to compute one-sided logrank statistics. It could be useful when a trial consists of more than two arms. By default it is not specified, all data will be used to fit the model.

Returns: This function returns a data frame with columns:

`p_inverse_normal` one-sided p-value for inverse normal test based on logrank test (alternative hypothesis: risk is higher in placebo arm). Accumulative data is used.

`z_inverse_normal` z statistics of `p_inverse_normal`. Accumulative data is used.

`p_lr` one-sided p-value for logrank test (alternative hypothesis: risk is higher in placebo arm). Accumulative data is used.

`z_lr` z statistics of `p_lr`. Accumulative data is used.

`info` observed accumulative event number.

`planned_info` planned accumulative event number.

`info_pbo` observed accumulative event number in placebo.

`info_trt` observed accumulative event number in treatment arm.

wt weights in z_inverse_normal.

Examples:

```
\dontrun{
trial$independentIncrement(Surv(pfs, pfs_event) ~ arm, 'pbo',
                           listener$get_milestone_names(),
                           'less', 'oracle')
}
```

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`. `survdiff` 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 `observed_info/sum(observed_info)`. As a result, setting `planned_info` to, e.g., `10 * 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 `get_seed()`**:** return random seed

Usage:

```
Trials$get_seed()
```

Method `print()`**:** print a trial

Usage:

```
Trials$print()
```

Method `get_snapshot_copy()`**:** return a snapshot of a trial before it is executed.

Usage:

```
Trials$get_snapshot_copy()
```

Method `make_snapshot()`**:** make a snapshot before running a trial. This can be useful when resetting a trial. This is only called when initializing a 'Trial' object, when arms have not been added yet.

Usage:

```
Trials$make_snapshot()
```

Method `make_arms_snapshot()`**:** make a snapshot of arms

Usage:

```
Trials$make_arms_snapshot()
```

Method `reset()`**:** 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 `set_arm_added_time()`**:** save time when an arm is added to the trial

Usage:

```
Trials$set_arm_added_time(arm, time)
```

Arguments:

`arm` name of added arm.

`time` time when an arm is added.

Method `get_arm_added_time()`**:** get time when an arm is added to the trial

Usage:

```
Trials$get_arm_added_time(arm)
```

Arguments:

`arm` arm name.

Method `set_arm_removal_time()`**:** save time when an arm is removed to the trial

Usage:

```
Trials$set_arm_removal_time(arm, time)
```

Arguments:

`arm` name of removed arm.

`time` time when an arm is removed.

Method `get_arm_removal_time()`**:** get time when an arm is removed from the trial

Usage:

```
Trials$get_arm_removal_time(arm)
```

Arguments:

`arm` arm name.

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$independentIncrement`
## ------------------------------------------------


## Not run:
trial$independentIncrement(Surv(pfs, pfs_event) ~ arm, 'pbo',
                           listener$get_milestone_names(),
                           'less', 'oracle')

## End(Not run)

## ------------------------------------------------
## 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)
```

`weibullDropout` *Calculate Parameters of Weibull Distribution as a Dropout Method*

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