Package 'simtte'

November 7, 2026

Title Simulate Time-to-Event Data from PK/PD and Bespoke ODE Models

Version 2.0.0

Language en-US

Description Simulates time-to-event (survival) datasets for clinical trial design and analysis, including datasets driven by a joint pharmacokinetic/pharmacodynamic (PK/PD) model solved via the 'mrgsolve' backend. Event times can be located directly during ODE integration ('in-solver' event detection) using a library of built-in PK/PD hazard models or a user-supplied 'mrgsolve' model converted for this purpose, or generated by inverse transform sampling from a closed-form Weibull or flexible M-spline baseline hazard, or from any externally simulated survival trajectory. See Bender et al. (2005) <doi:10.1002/sim.2059> for the inverse transform sampling methodology and Royston and Parmar (2002) <doi:10.1002/sim.1203> for flexible parametric survival models.

License GPL (>= 2)

Encoding UTF-8

LazyData true

Depends R (>= 4.0.0)

Imports dplyr, magrittr, methods, mrgsolve, tools

Suggests callr, testthat (>= 3.0.0), knitr, pkgload, rmarkdown, splines2, survival

VignetteBuilder knitr

URL https://github.com/csetraynor/simtte

BugReports https://github.com/csetraynor/simtte/issues

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

RoxygenNote 8.0.0

NeedsCompilation no

Author Carlos Traynor [aut, cre]

Maintainer Carlos Traynor <carlos.traynor.qcp@gmail.com>

Repository CRAN

Date/Publication 2026-11-07 17:00:02 UTC

Contents

add_censoring . . . 2
add_interval_censoring . . . 4
censoring_rate_for . . . 6
draw_censoring_times . . . 7
explore_pi_tq_surv . . . 8
ms_data . . . 10
print.simtte_model . . . 10
print.simtte_ode_sim . . . 11
print.simtte_prepared_model . . . 11
simtte_example_model . . . 12
simtte_example_models . . . 13
simtte_model_cache . . . 14
simtte_model_cache_clear . . . 15
simtte_prepare_model . . . 16
sim_tte . . . 18
sim_tte_df . . . 22
sim_tte_ode . . . 25
sim_tte_ode_models . . . 35
thin_visits . . . 36
tte_model . . . 37
visit_schedule . . . 39
visit_schedule_informative . . . 41

Index **44**

`add_censoring` *Apply independent right censoring to a simulated events data frame*

Description

Adds a second, independent censoring process to an already-simulated time-to-event data frame: a censoring time C_i is drawn per subject from `censoring`, and the observed outcome becomes $\min(T_i, C_i, end)$, with `sim_status` = 1 only if T_i (the original `sim_time`) is the minimum. Works on the output of `sim_tte_ode`, `sim_tte`, or `sim_tte_df` alike – all three already share the same `ID`/`sim_time`/`sim_status` shape.

Usage

```
add_censoring(
  events,
  censoring,
  end,
  id_var = "ID",
  time_var = "sim_time",
  status_var = "sim_status",
  seed = NULL
)
```

Arguments

events	Data frame of simulated outcomes: typically the `$events` element returned by `sim_tte_ode`, or the output of `sim_tte`/`sim_tte_df`. Needs `ID`, `sim_time`, `sim_status` columns by default – column names are configurable via `id_var`/`time_var`/`status_var` for a data frame using different names. Any other column is left untouched.
censoring	A list describing the censoring-time distribution, with a `dist` element: `"exponential"` `rate` (positive scalar). `C ~ Exp(rate)`. `"weibull"` `shape`, `scale` (positive scalars). `C ~ Weibull(shape, scale)` (`rweibull` parameterization). `"uniform"` `min`, `max` (`0 <= min < max`). `C ~ Uniform(min, max)` – the common case for a uniform-accrual administrative censoring time. `"lognormal"` `meanlog` (finite scalar), `sdlog` (positive scalar). `C ~ Lognormal(meanlog, sdlog)` (`rlnorm` parameterization). `"gamma"` `shape`, `rate` (positive scalars). `C ~ Gamma(shape, rate)` (`rgamma` parameterization) – `rate` only, matching `"exponential"`'s own rate-only convention (a `scale` element is an error, not silently accepted). `"user"` `fun`, a function `n -> numeric(n)` returning `n` non-negative, finite censoring times. The escape hatch for any distribution not covered above.
end	Numeric scalar. Administrative censoring horizon – the third term of `min(T, C, end)`. Always wins: no observed time in the result exceeds `end`. Required (part of the censoring formula itself, not inferred from `events`).
id_var, time_var, status_var	Character scalars naming the subject-ID, event/censoring-time, and status columns of `events`. Default `"ID"`/`"sim_time"`/`"sim_status"` (the column names every **simtte** simulation function already uses).
seed	Optional integer. If supplied, `set.seed(seed)` is called before C is drawn. Leave `NULL` when calling this from inside a larger already-seeded workflow (this is how `sim_tte_ode`'s own `censoring =` argument uses it, so that one `seed` reproduces `U`, any between-subject draw, and the censoring draw together, in that fixed order – see `?sim_tte_ode` "Reproducibility").

Value

`events` with `time_var`/`status_var` updated in place, plus one new column, `sim_reason`: `"event"`, `"censored"` (the independent draw, or a pre-existing censoring time from `events` itself, was earlier

than both the event and end), or "administrative" (the observed time equals end and the subject is not an event). Every other column of events is returned unchanged.

See Also

sim_tte_ode, whose $events this usually post-processes (or pass its censoring argument to do this in one call); add_interval_censoring, the next step if you also need assessment intervals. censoring_rate_for, to choose rate/ scale for a target censoring fraction; draw_censoring_times, to draw from the same censoring-shaped spec directly, with no events data frame needed; the "Right censoring" section of vignette("pkpd-time-to-event", package = "simtte") for a worked example, including the scale-helper workflow.

Examples

```
sim <- sim_tte_ode(model = "exponential", param = list(H0 = 0.05),
  n = 100, end = 30, delta = 2, seed = 1)
out <- add_censoring(sim$events, censoring = list(dist = "exponential",
  rate = 0.02), end = 30, seed = 2)
table(out$sim_reason)

# Also works directly on the legacy sim_tte() output.
lp <- matrix(rnorm(50, 0, 0.5), nrow = 50)
legacy <- sim_tte(pi = lp, mu = -1, coefs = 1.1,
  time = seq(0, 30, by = 1), type = "weibull", end_time = 30)
out2 <- add_censoring(legacy, censoring = list(dist = "uniform",
  min = 5, max = 30), end = 30, seed = 3)
table(out2$sim_reason)
```

---

add_interval_censoring

Map simulated event/censoring times onto an assessment-time interval

---

Description

Adds interval-censoring bounds to an already-simulated (and, if applicable, already right-censored – see "Ordering" below) events data frame: given a visit schedule, an event known to have occurred exactly at sim_time is only *detectable* at the first visit at or after it, and a censored subject is only known event-free through their last visit at or before their own censoring time. Works on the output of sim_tte_ode, sim_tte, or sim_tte_df alike, same as add_censoring.

Usage

```
add_interval_censoring(
  events,
  visits,
  id_var = "ID",
  time_var = "sim_time",
  status_var = "sim_status"
)
```

Arguments

`events` Data frame of simulated outcomes: typically the `$events` element returned by `sim_tte_ode`, or the output of `sim_tte`/`sim_tte_df`, optionally after `add_censoring`. Needs `ID`, `sim_time`, `sim_status` columns by default (rename via `id_var`/`time_var`/`status_var`). Every other column is left untouched, and so are these three: this function only adds `sim_time_left`/`sim_time_right` (see "Return" below).

`visits` A visit schedule, either:

a numeric vector applied to every subject (e.g. `c(0, 4, 8, 12, 24)`). Must start at `0` and be strictly increasing (sorted, no duplicate visit times).

a data frame with `ID`/`time` columns, for a per-subject schedule. Every ID in `events` must have a schedule (starting at `0`, strictly increasing), and every ID in `visits` must be one that appears in `events`.

`visit_schedule` builds the data-frame form from a fixed spacing plus optional per-subject jitter, rather than this function accepting a jitter spec itself – see its own documentation.

`id_var`, `time_var`, `status_var`
Character scalars naming the subject-ID, event/censoring-time, and status columns of `events`. Default `"ID"`/`"sim_time"`/`"sim_status"`.

Value

`events` with two new columns:

sim_time_left Left (open) bound of the interval, `L`.

sim_time_right Right (closed) bound, `R`, or `Inf` for an interval with no known upper bound (a censored subject, or an event that occurred after the subject's last visit and so was never confirmed by the schedule – `sim_status` still reads `1` in that case: the exact columns are the simulation's ground truth, unaffected by this mapping. The two columns answer different questions: `sim_status` says what happened, the bounds say what the schedule could observe).

An event with `sim_time_left == 0` (an event before the first post-baseline visit) **is** a left-censored observation – the event is known to have occurred before `sim_time_right`, exact time unknown – with no separate mechanism needed; see the vignette's "Left censoring" subsection for a worked example. `sim_time`/`sim_status`/`sim_reason` (if present) are returned unchanged – this function only adds columns, so every existing consumer of those three keeps working. For `survival::Surv(time = sim_time_left, time2 = sim_time_right, type = "interval2")`, which expects `NA` rather than `Inf` for an open upper bound, convert with `ifelse(is.infinite(sim_time_right), NA, sim_time_right)` – see the vignette's "Interval censoring" section for a full worked example.

Ordering with right censoring

Call this **after** `add_censoring` (or after the exact simulation, if no independent right censoring was requested), not before: the interval mapping reads `sim_time`/`sim_status` as given, so a dropout applied afterward would leave the already-added interval columns stale and inconsistent with the final exact columns. `sim_tte_ode`'s own `censoring`/ `visits` convenience arguments apply them in this order internally.

See Also

visit_schedule, to build a per-subject schedule with fixed spacing and optional jitter; thin_visits, to drop visits at random (missed visits, dropout); visit_schedule_informative, for visits that react to the simulated outcome. add_censoring, which should run first. sim_tte_ode, which can apply this step itself through its visits argument.

Examples

```
sim <- sim_tte_ode(model = "exponential", param = list(H0 = 0.05),
  n = 50, end = 24, delta = 2, seed = 1)
out <- add_interval_censoring(sim$events, visits = c(0, 4, 8, 12, 16, 20, 24))
head(out)
```

---

censoring_rate_for *Solve for a censoring-distribution parameter giving a target censoring fraction*

Description

Users think in terms of "about 20 end", not in the rate/scale units add_censoring needs, and the right units depend on the model's own time scale. This solves for one parameter of a fixed-family censoring distribution – the exponential/gamma rate, the Weibull scale at a fixed shape, or the lognormal meanlog at a fixed sdlog – that would give a target independent-censoring fraction, treating times – the event times from an already-run, **uncensored** simulation – as an empirical sample of the latent event time T: it solves mean(F_C(times)) == target via uniroot, where F_C is the censoring distribution's own CDF (pexp/ pweibull/plnorm/ pgamma). This is the Monte Carlo estimate of $P(C < T)$ under independence, using the supplied sample in place of T's true (generally not closed-form once between-subject variability or covariates are in the model) distribution – **approximate, and dependent on the run it is given**: a different (even same-seed-different-n) uncensored run will give a slightly different answer. Re-check the realized fraction on a large cohort after drawing with the solved parameter; see vignette("pkpd-time-to-event", package = "simtte") "Right censoring" for a worked before/after check.

Usage

```
censoring_rate_for(
  times,
  target,
  dist = c("exponential", "weibull", "lognormal", "gamma"),
  shape = NULL,
  sdlog = NULL
)
```

Arguments

times	Numeric vector of event times (sim_status == 1 only) from an uncensored simulation. Must be non-negative and finite.
target	Numeric scalar in $(0, 1)$. The desired $P(C < T)$.
dist	"exponential" (default), "weibull", "lognormal", or "gamma".
shape	Required (positive numeric scalar) for dist = "weibull" (solves for scale) or dist = "gamma" (solves for rate); the shape is fixed by the caller in either case. Unused otherwise.
sdlog	Required (positive numeric scalar) for dist = "lognormal" (solves for meanlog, with sdlog fixed by the caller). Unused otherwise.

Details

Only genuine event times are informative here: an administratively-censored subject's true T_i is unknown beyond end, so it carries no information about where independent censoring would actually bite. Pass times = events$sim_time[events$sim_status == 1] from an uncensored run, not the full events$sim_time.

Value

Numeric scalar: rate for dist = "exponential" or "gamma", scale for dist = "weibull", meanlog for dist = "lognormal".

See Also

add_censoring, which takes the solved parameter in its censoring list; draw_censoring_times, to check the draws directly; sim_tte_ode, whose uncensored $events supply times.

Examples

```
sim <- sim_tte_ode(model = "exponential", param = list(H0 = 0.05),
  n = 500, end = 30, delta = 2, seed = 1)
event_times <- sim$events$sim_time[sim$events$sim_status == 1]
rate <- censoring_rate_for(event_times, target = 0.2)
out <- add_censoring(sim$events, censoring = list(dist = "exponential",
  rate = rate), end = 30, seed = 2)
mean(out$sim_reason == "censored") # close to 0.2, on the events subset
```

---

draw_censoring_times *Draw independent times from a censoring/delay distribution spec*

Description

A thin, exported wrapper over the same distribution dispatch add_censoring uses internally for its own censoring argument, and visit_schedule_informative uses for extra_visit_after_event. Exported so a caller that only wants n raw draws from a distribution spec – to plan a scenario's expected censoring/dropout burden, or to pre-validate a spec before running many replicates of it – does not need to build an events-shaped data frame just to reach this logic. add_censoring/visit_schedule_informative are unaffected and still call the same, unexported dispatcher directly; this function does not replace either.

Usage

```
draw_censoring_times(spec, n, seed = NULL)
```

Arguments

spec	A list describing the distribution, in exactly the shape documented for add_censoring's own censoring argument (dist one of "exponential", "weibull", "uniform", "lognormal", "gamma", "user", plus that distribution's own parameters).
n	Integer. Number of draws.
seed	Optional integer. If supplied, set.seed(seed) is called before the draw (same convention as add_censoring's own seed).

Value

Numeric vector of length n, all finite and non-negative.

See Also

add_censoring and visit_schedule_informative, the two functions that use this same spec shape internally; censoring_rate_for, to choose a spec's rate for a target censoring fraction.

Examples

```
draw_censoring_times(list(dist = "exponential", rate = 0.05), n = 10,
  seed = 1)
```

---

explore_pi_tq_surv *Explore Prognostic Index and Quantile Survival Times*

Description

Calculates the difference in survival probability at a given quantile time across a range of prognostic index values. Useful for understanding how different log hazard ratios affect survival outcomes.

Usage

```
explore_pi_tq_surv(q = 0.5, pi, mu, shape, type, times, basehaz, end_time, ...)
```

Arguments

q	Numeric scalar. Survival quantile of interest (default 0.5, i.e. median survival).
pi	Numeric vector. Prognostic index values to evaluate. Must be finite.
mu	Numeric scalar. Intercept parameter. Must be finite.
shape	Numeric scalar or vector. Shape parameter(s) for the Weibull model. Must be finite and strictly positive.
type	Character string. Model type: "weibull" or "ms".
times	Numeric vector. Output time grid; see the "Time grid and event-time resolution" and "M-spline hazard carry convention" sections of sim_tte for the exact contract. For M-spline models this must also be the time points corresponding to the rows of basehaz. If missing, defaults to seq(0, end_time, by = 1).
basehaz	Numeric matrix. Baseline hazard (M-spline models), one row per element of times. Must be finite and non-negative; see sim_tte's "M-spline hazard carry convention" for how it is interpreted between supplied times.
end_time	Numeric scalar. Administrative censoring time. See the end_time contract documented in sim_tte (including the M-spline min(times) <= end_time <= max(times) requirement).
...	Additional arguments passed to mrgsim. tgrid, obsonly, nocb, carry_out/carry.out, and data cannot be overridden; see sim_tte.

Value

A data frame including columns lp, p11, survdiff_tq, and model parameters.

See Also

sim_tte, the closed-form engine this function diagnoses, and the function to call once you have picked pi, mu and shape; sim_tte_df, for the underlying trajectory-to-event mechanism.

Examples

```
# Small fast example
data_sim <- explore_pi_tq_surv(
  pi = seq(-1, 1, by = 0.5),
  mu = -1,
  shape = 1.1,
  end_time = 10,
  type = "weibull"
)
head(data_sim)

# Larger range example
data_sim2 <- explore_pi_tq_surv(
  pi = seq(-3, 3, by = 0.1),
  mu = -1,
  shape = 1.1,
  end_time = 200,
  type = "weibull"
```

```
)
plot(survdiff_tq ~ lp, data = data_sim2)
```

---

ms_data *Simulated Data for M-Splines Model Demonstration*

Description

A list containing parameters for an M-spline survival model, suitable for use with `sim_tte`.

Usage

```
ms_data
```

Format

A list with 4 elements:

mu Numeric scalar. Intercept parameter.

basis Numeric matrix. M-spline basis matrix with rows corresponding to time points and columns to basis functions.

coefs Numeric vector. Coefficients for each basis function.

time Numeric vector. Time points corresponding to the rows of the basis matrix.

Source

Simulated example data.

---

print.simtte_model *Print a converted PK/PD model*

Description

Print a converted PK/PD model

Usage

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

Arguments

`x`	A `"simtte_model"` object.
`...`	Ignored.

print.simtte_ode_sim *Print an in-solver simulation result*

Description

Print an in-solver simulation result

Usage

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

Arguments

x	A "simtte_ode_sim" object.
...	Ignored.

print.simtte_prepared_model *Print a prepared model*

Description

Print a prepared model

Usage

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

Arguments

x	A "simtte_prepared_model" object.
...	Ignored.

`simtte_example_model` *Load a bundled simtte example mrgsolve model*

Description

Compiles (or loads from cache) one of simtte's shipped example mechanistic mrgsolve models; see `simtte_example_models` for the list of available names and what each model represents. This is a thin wrapper around `mread_cache` pointed at the package's example-model directory; it adds nothing beyond discoverability and an informative error for an unrecognized name.

Usage

```
simtte_example_model(name, ...)
```

Arguments

`name`	Character scalar naming one bundled model; must be an element of `simtte_example_models()`.
`...`	Additional arguments passed to `mread_cache` (for example `quiet = TRUE`).

Details

The returned model is a standard **mrgsolve** model object: use `mrgsim` to simulate it, exactly as with any other **mrgsolve** model. The underlying `.cpp` source file can be found under `system.file("models", "examples", package = "simtte")` and copied out and modified freely with any **mrgsolve** tooling; loading it through this function does not require continuing to use this function afterward.

Value

A compiled **mrgsolve** model object.

Intended workflow

These example models are for use with `sim_tte_df`, the package's entry point for custom/mechanistic models with an ODE-integrated survival compartment (see its "Trajectory contract" section); they are not used by, and cannot be passed to, `sim_tte`, which only ever simulates its own built-in Weibull and M-spline models.

```
mod  <- simtte_example_model("pkpd_linear_hazard")
data <- mrgsolve::ev(amt = 100, cmt = 1, time = 0)
out  <- as.data.frame(mrgsolve::mrgsim(mod, data = data, end = -1,
          add = seq(0, 60, by = 0.5), obsonly = TRUE))
sim_tte_df(out[, c("ID", "time", "p11")])
```

See Also

`simtte_example_models`, to list available names; `sim_tte_df`, which turns this model's simulated trajectory into event times. To simulate a model of your own with `sim_tte_ode`, convert it with `tte_model`.

Examples

```
mod <- simtte_example_model("pkpd_linear_hazard")
```

`simtte_example_models` *List simtte's bundled example mrgsolve models*

Description

Returns the names of the mechanistic mrgsolve models shipped as user-facing examples/templates in `inst/models/examples/`. These are plain, editable `.cpp` model source files, not part of the package's internal Weibull/M-spline simulation engine (see `sim_tte`) and not used by it. They demonstrate the `dxdt_p11 = -p11 * HAZ` survival co-integration pattern that a user's own custom mrgsolve model is expected to follow when working with `sim_tte_df` directly; see "Shipped example models" below.

Usage

```
simtte_example_models()
```

Details

Discovery is a directory listing of `*.cpp` files in `inst/models/examples/`, not a maintained registry or catalog: adding a new example model in a future package version requires only adding a new `.cpp` file to that directory, with no change to this function.

Value

Character vector of example model names, suitable for passing to `simtte_example_model`.

Shipped example models

`pkpd_idr_hazard` A one-compartment oral-absorption PK model coupled to an indirect-response (turnover) mediator model, in which drug exposure inhibits mediator production and the event hazard is proportional to the current mediator level. Includes between-subject variability (`$OMEGA`) on clearance, volume, and potency. This is the model described in the package's companion manuscript ("ODE-coupled PK/PD hazard"), shipped here as a standalone, loadable file.

`pkpd_linear_hazard` A minimal one-compartment IV-bolus PK model with a hazard that decreases linearly in drug concentration from a constant baseline. No between-subject variability or turnover dynamics. Both the PK and hazard are linear, so this model has an independent closed-form survival function, making it a short, readable starting point for a custom model as well as this package's validation anchor for mechanistic hazard models (see `inst/validation/` in the package's source repository – a developer-only directory, not installed with the package).

See Also

simtte_example_model, to load one; sim_tte_df, the function these models are for. sim_tte_ode_models lists the built-in models for sim_tte_ode, which are a separate set.

Examples

```
simtte_example_models()
```

`simtte_model_cache` *Where simtte builds and caches compiled models*

Description

Resolves and returns the directory sim_tte_ode and simtte_prepare_model pass as **mrgsolve**'s own `soloc` argument (where a model's compiled shared object is built and stored) – simtte itself never sets `options()` on a user's behalf, for `mrgsolve`'s own options or this package's `simtte.cache_dir`; this function only ever *reads* `getOption("simtte.cache_dir")` and appends a version-qualified subdirectory before returning it.

Usage

```
simtte_model_cache()
```

Details

Default (`simtte.cache_dir` unset): a directory under `tempdir()`, i.e. session-scoped and automatically cleaned up – CRAN-safe (nothing is ever written outside a session temp directory unless you opt in below) and the right choice for any single script or session, including a local **parallel** cluster (every worker of a *local* cluster shares the parent's filesystem, so a model built once under the parent's own cache directory is directly reachable by every worker's `loadso` call – see `?sim_tte_ode` "Parallel simulation").

Persistent cache: set `options(simtte.cache_dir = tools::R_user_dir("simtte", "cache"))` (the standard, CRAN-compliant location for a package's own cache data – simtte_model_cache never chooses this for you) to keep a compiled model's shared object across R restarts, avoiding a repeat compile in a later session. This matters most for a slow-to-compile tte_model-converted model reused across many scripts, or a genuinely distributed cluster whose worker nodes do *not* share the parent's filesystem (where every node needs to be pointed, itself, at the same shared, network- visible directory).

Version-key caveat: whichever directory `simtte.cache_dir` resolves to, a subdirectory keyed by the current R version, this **simtte** version, and the installed **mrgsolve** version is always appended before it is returned or created. This is deliberate, not incidental: **mrgsolve**'s own build-cache key is content/mtime based, not version-aware (verified: its own `?mread` documentation states plainly that `soloc` is "a directory location where the model shared object is built and stored", with no mention of invalidating it across an R or package upgrade) – a *persistent* cache surviving an R version change or a **simtte/mrgsolve** upgrade risks reusing a compiled `.so` built against a different, ABI-incompatible R/package version. The version key makes that class of staleness unreachable by

construction rather than relying on **mrgsolve**'s own file-level staleness detection to catch it (which can be defeated by two builds landing in the same filesystem-timestamp second – see `tte_model`'s own `name` documentation). The **default**, `tempdir()`-based location needs no such protection on its own (a session cannot upgrade R or a package mid-run), but the key is applied unconditionally anyway, so there is exactly one code path to reason about, not two.

Value

Character scalar: an existing, writable directory path.

See Also

`simtte_prepare_model`, which builds into this directory (as **mrgsolve**'s `soloc`). `simtte_model_cache_clear`, to remove what accumulates here. `sim_tte_ode` and `tte_model` also build into it; see "Parallel simulation" in `?sim_tte_ode`.

Examples

```
simtte_model_cache()
```

`simtte_model_cache_clear` *Clear simtte's compiled-model cache*

Description

Removes compiled model artifacts that `simtte_prepare_model`/`sim_tte_ode` built under `simtte_model_cache`. Only ever removes directories *this package itself* created (version-qualified subdirectories named exactly like `simtte_model_cache`'s own return value) – never the configured base directory itself, and never anything under **mrgsolve**'s own default `tempdir()` locations that this package did not create. Safe to call even if `simtte.cache_dir` happens to be pointed at a directory with other, unrelated content in it (e.g. a user's home directory): only the recognizably-named version subdirectories are ever candidates for removal, and each is double-checked to actually resolve under the configured base before being deleted.

Usage

```
simtte_model_cache_clear(all = FALSE)
```

Arguments

`all`	Logical, default `FALSE`: remove only the *current* version-qualified subdirectory – the one `simtte_model_cache` would return right now. If `TRUE`, remove every version-qualified subdirectory found under the configured base – useful after an R, simtte, or mrgsolve upgrade, when a persistent `simtte.cache_dir` has accumulated subdirectories for versions no longer in use.

Value

Character vector of the directory paths actually removed, invisibly (`character(0)` if there was nothing to remove).

See Also

`simtte_model_cache`, which returns the directory this clears. `simtte_prepare_model`, which rebuilds a model the next time you call it. `sim_tte_ode`, which does the same implicitly.

Examples

```
simtte_model_cache()
simtte_model_cache_clear()
```

`simtte_prepare_model` *Prepare a model for simulation on this process, once*

Description

Builds (or, if this process has already built the exact same model before – `mread_cache`'s/`mcode_cache`'s own in-session cache – reuses) and `loadso`s a model, returning it wrapped in a class `sim_tte_ode` recognizes and never rebuilds. `sim_tte_ode` calls this function internally for every `model` it is given (a character library name, a `tte_model`-converted `simtte_model`, or an object this function already returned) – there is exactly one model-loading path, whether or not a caller prepares anything ahead of time.

Usage

```
simtte_prepare_model(model, knots = NULL)
```

Arguments

`model`	A character library-model name (as `sim_tte_ode`'s own `model` argument), a `simtte_model` from `tte_model`, or an object this function already returned.
`knots`	Only used, and required, when `model = "mspline"`: its *length* selects which of the three shipped knot-count variants (`?sim_tte_ode` "M-spline knot-count variants") gets compiled – the knot *values* themselves, like `coefs`/`boundary_knots`, only ever affect `$PARAM` values, applied later by `sim_tte_ode` itself, never which compiled variant is loaded (checked directly: this function's only use of `knots` is `length(knots)`). `coefs`/ `boundary_knots` are therefore not parameters of this function at all – supplying either raises R's own "unused argument" error, naming it, rather than silently accepting a value that would not affect what gets built.

Details

The point of calling this yourself: a **parallel** PSOCK worker is a separate R process with no memory of what the parent process has already built or loaded. Calling `sim_tte_ode()` directly, unprepared, on each worker is not wrong – every path through it now calls `loadso` – but it does mean each worker independently pays its own full compile cost the first time (harmless for a fast-compiling built-in model; measured up to ~1.5s for a `tte_model()`-converted one, "Parallel simulation" section of `?sim_tte_ode`). Calling `simtte_prepare_model()` once per worker at cluster initialization avoids that redundant work *and* is the pattern verified race-free: a worker receiving an already-compiled model object (via ordinary **parallel** serialization – `parallel::clusterCall()`/`clusterExport()` both work) and only ever calling `loadso()` on it never races, unlike a worker independently re-triggering `mcode_cache()`/`mread_cache()` against a directory another process might be writing to at the same time.

Idempotent: calling this on its own return value (exactly what happens when the same `model =` argument is broadcast to every worker via `clusterCall()`, each worker then calling this function on it again) only re-`loadso()`s (cheap, safe to repeat) and returns the object unchanged – never a second build.

A genuinely separate process (not a **parallel** cluster worker that received the compiled object by serialization, but e.g. a chunked/array-job task reading a `tte_model` object back from a manifest file written by an earlier, already-exited process): a `simtte_model`'s compiled shared object lives wherever the *building* process's own cache put it, which a later process cannot see. Calling this function there rebuilds it from `$code`/`$name` automatically, into *this* process's own `simtte_model_cache` – deterministic (the same code and name always produce the same compiled model), and never a global `mrgsolve` option, so correctness never depends on the two processes sharing a filesystem. A built-in library model needs no such rebuild (it is always (re)built from the shipped `.cpp` source into this process's own cache directly, every time).

Value

An object of class `"simtte_prepared_model"`. Pass it directly as `sim_tte_ode`'s own `model` argument.

See Also

`sim_tte_ode`, which takes the returned object as `model` (see "Parallel simulation" there for the full cluster-init pattern). `tte_model`, which builds the `simtte_model` to pass in for your own model. `simtte_model_cache`, where the compiled model is stored, and `simtte_model_cache_clear`, to remove it.

Examples

```
prepared <- simtte_prepare_model("exponential")
sim <- sim_tte_ode(model = prepared, param = list(H0 = 0.1), n = 10,
  end = 20, delta = 2, seed = 1)
head(sim$events)
```

`sim_tte` *Simulate Time-to-Event Data*

Description

Main function of **simtte**. Simulates a time-to-event dataset (survival dataset) using either a Weibull parametric model or an M-spline flexible baseline hazard model. Event times are generated via inverse transform sampling from the cumulative hazard function computed by **mrgsolve**.

Usage

```
sim_tte(
  pi,
  log_pi = TRUE,
  mu = -3,
  coefs = 0,
  basis = NULL,
  time = seq(0, 100, by = 1),
  end_time,
  type = "weibull",
  event_time_method = c("grid", "log_survival"),
  lp_data = NULL,
  ...
)
```

Arguments

`pi`	Numeric matrix or vector. Prognostic index (linear predictor) for each individual. Given a covariate matrix X and coefficient vector b, `pi = X %*% b`. Must be non-empty and finite (no `NA`/`NaN`/`Inf`/`-Inf`).
`log_pi`	Logical; is the prognostic index already on the log scale? Default `TRUE`.
`mu`	Numeric scalar. Intercept parameter of the model. Must be finite. Default `-3`.
`coefs`	Numeric vector. For M-spline models, the coefficients of each spline basis function. For Weibull models, the shape parameter (scalar), which must be strictly positive. Must be finite in all cases.
`basis`	Numeric matrix. Basis matrix for M-spline models, with one row per element of `time` and one column per coefficient in `coefs`. Must be finite. Ignored for Weibull models. Default `NULL`.
`time`	Numeric vector. The output time grid: the exact set of times at which the survival trajectory is simulated and reported, and therefore the set of times at which an event can be detected (see "Time grid and event-time resolution" below). For M-spline models, `time` must additionally be sorted in non-decreasing order and have one element per row of `basis` (the baseline hazard is only defined at these points). Must be non-negative, finite, and contain no `NA`/`NaN`. Default `seq(0, 100, by = 1)`.

`end_time` Numeric scalar. Administrative censoring horizon. Defaults to `max(time)`. Any requested `time` points beyond `end_time` are dropped, and `end_time` itself is always added to the output grid so that censoring lands exactly on it. Must be non-negative; `end_time = 0` is accepted (a degenerate zero-duration follow-up: the output grid is just time 0, and since $S(0) = 1$ for every supported model, every subject is censored at 0). For Weibull models `end_time` may otherwise be any non-negative value, including below `min(time)` or above `max(time)` (the closed-form hazard is defined for all `t >= 0`). For M-spline models `end_time` must satisfy `min(time) <= end_time <= max(time)`: the baseline hazard is only defined on the supplied `time` grid and this package does not extrapolate it in either direction.

`type` Character string. Model type: `"weibull"` or `"ms"` (M-splines). Default `"weibull"`.

`event_time_method` Character string. `"grid"` (default) or `"log_survival"`. See "Event-time interpolation" below. Forwarded explicitly to `sim_tte_df`; it is unrelated to the **mrgsolve** simulation step and is never passed via `...`.

`lp_data` `NULL` (default) or a data frame with columns `time`, `lp`, and optionally `ID`, giving a time-varying linear predictor `lp(t)` for the built-in Weibull/M-spline models. See "Time-varying `lp(t)`" below. `NULL` preserves every prior behavior of `sim_tte()` byte-for-byte.

`...` Additional arguments passed to `mrgsim`. `tgrid`, `obsonly`, `nocb`, `carry_out`/`carry.out`, and `data` are controlled internally to guarantee the contracts documented here and cannot be overridden; supplying them raises an error naming the argument.

Value

A data frame with columns:

sim_time Simulated event or censoring time.

sim_status Event indicator (1 = event, 0 = censored).

ID Subject identifier.

lp Log prognostic index (linear predictor).

Time grid and event-time resolution

The output grid used for simulation is `sort(unique(time))`, restricted to values `<= end_time`, with `end_time` appended if it is not already present. **mrgsolve** is instructed to report the survival trajectory *only* at these times (via its `tgrid` mechanism), so the grid is deterministic and is not silently replaced by **mrgsolve**'s own default output schedule.

Event times are then resolved by `sim_tte_df` as the first grid point at which the survival probability falls to or below a uniform random draw. For `event_time_method = "grid"` (the default), no interpolation between grid points is performed: the resolution of the simulated event time is exactly the spacing of `time`. For `event_time_method = "log_survival"`, a crossing that falls strictly between two grid points is refined by interpolation; see "Event-time interpolation" below and the corresponding section of `sim_tte_df` for the full mathematical contract (this is the function `sim_tte()` delegates to internally, forwarding `event_time_method` explicitly). A subject whose survival never crosses the threshold on the grid is censored at `end_time` under either method.

Event-time interpolation

See the "Event-time interpolation" section of `sim_tte_df` for the full mathematical contract and edge-case rules; `sim_tte()` simply forwards `event_time_method` to it. In summary: `"log_survival"` assumes the hazard is constant between consecutive reported trajectory points and linearly interpolates cumulative hazard accordingly. For the M-spline model this recovers the event time implied by the discretized (piecewise-constant, via the "M-spline hazard carry convention" below) hazard exactly. For the Weibull model with `shape != 1`, whose true hazard is not constant within an interval, it is an approximation whose accuracy improves as `time` is refined (exact when `shape == 1`, the one case where the Weibull hazard genuinely is constant).

M-spline hazard carry convention

For `type = "ms"`, `basis %*% coefs` gives a baseline hazard value at each element of `time`; this is supplied to **mrgsolve** as a time-varying input, not evaluated as a continuous spline function at arbitrary times. The package fixes the piecewise interpretation explicitly (**mrgsolve**'s `nocb = FALSE`, i.e. last-observation-carried-forward): the hazard value reported at `time[i]` applies from `time[i]` (inclusive) until `time[i + 1]` (exclusive) – a right-continuous step function in the hazard. Concretely, for knots/hazards `time = c(0, 1, 2)`, `basehaz = c(1, 2, 4)`: $S(0.5) = \exp(-0.5)$, $S(1) = \exp(-1)$, $S(1.5) = \exp(-2)$, $S(2) = \exp(-3)$ (verified directly against **mrgsolve** output; this is **not** the **mrgsolve** default, which is next-observation-carried-backward and would instead give $S(0.5) = \exp(-1)$).

A consequence of this convention: the hazard value associated with the *last* element of `time` has no interval after it to apply to, so it never affects the survival trajectory (it would only matter if `end_time` extended past `max(time)`, which is not permitted; see `end_time` above). `basis`/`coefs` therefore define `length(time) - 1` active piecewise-constant hazard segments, not `length(time)`.

This is a grid-based piecewise-constant hazard, not continuous M-spline evaluation: the package receives precomputed basis values at specific times and never re-evaluates the spline basis at arbitrary times.

Time-varying `lp(t)`

Supplying `lp_data` allows the linear predictor itself to vary over time, for both built-in models. `lp_data` must contain `time` (finite, non-negative) and `lp` (finite; unlike a hazard, negative values are valid) columns, and, optionally, an `ID` column:

- no `ID`: a single population-level $lp(t)$ trajectory, applied identically to every subject;
- `ID` present: subject-specific trajectories. `sort(unique(lp_data$ID))` must equal `1:length(pi)` exactly (one full trajectory per subject; no partial coverage).

In both cases, `pi` keeps its existing role as each subject's baseline linear-predictor offset: $lp_{\text{effective}}(t) = pi_i + lp(t)$. `lp_data` rows are not sorted or deduplicated automatically (same philosophy as `sim_tte_df`'s trajectory contract): unsorted times are an error; duplicate times are accepted only with an identical `lp` value, otherwise an error. `lp(t)` is piecewise constant under last-observation-carried-forward (the same convention as `basehaz`, above): the value at `lp_data$time[i]` applies from that time until the next `lp_data` knot for that subject. `lp_data`'s own time grid is independent of, and need not coincide with, the main `time` argument.

Coverage requirement: every subject's `lp_data` must include an observation at `time = 0` (there is no implicit $lp(0)$). For M-spline models, which cannot extrapolate, it must additionally reach at

least `end_time`; for Weibull models this is not required, since the last known `lp` value is carried forward for any remaining follow-up.

For `type = "weibull"`, a separate, distinct model (`weibull_tv`) is used internally whenever `lp_data` is supplied; the constant-`lp` closed-form Weibull model (`inst/models/weibull.cpp`) used whenever `lp_data` is `NULL` is not modified or replaced. `weibull_tv` computes $S(t)$ by an **exact** closed-form cumulative-hazard summation over the piecewise-constant $lp(t)$ segments – it does not use ODE integration or internal-solver event detection (the latter was investigated and rejected; see `PHASE_E_THRESHOLD_TRACKING_REPORT.md`) – so it retains the same solver-free numerical robustness established for the baseline model. For `type = "ms"`, the existing `basehaz(t)` mechanism and `lp(t)` combine as $h(t) = basehaz(t)\exp[\mu + lp(t)]$, both under the same last-observation-carried-forward convention.

References

Bender R, Augustin T, Blettner M (2005). Generating survival times to simulate Cox proportional hazards models. *Statistics in Medicine*, 24(11), 1713–1723. doi:10.1002/sim.2059

Royston P, Parmar MKB (2002). Flexible parametric proportional-hazards and proportional-odds models for censored survival data, with application to prognostic modelling and estimation of treatment effects. *Statistics in Medicine*, 21(15), 2175–2197. doi:10.1002/sim.1203

See Also

`sim_tte_df`, which turns any `(ID, time, p11)` trajectory, including one from your own **mrgsolve** model, into event times. `sim_tte_ode`, for joint PK/PD and time-to-event simulation with in-solver event detection (also the route for Cox-model data; see its "Simulating from a Cox model" section). `add_censoring`, to add independent right censoring to this function's output (call it directly on the returned data frame; there is no argument here for it), then `add_interval_censoring`, to map the result onto an assessment-time interval. `explore_pi_tq_surv`, to see how the prognostic index moves survival before simulating.

Examples

```
# Fast Weibull example with a small dataset
set.seed(1)
lp <- matrix(rnorm(5, 0, 0.5), nrow = 5)
result <- sim_tte(
  pi = lp, mu = -1, coefs = 1.1,
  time = seq(0.1, 10, by = 0.5),
  type = "weibull", end_time = 10
)
head(result)

# Larger examples using bundled ms_data
data("ms_data")
mu <- ms_data$mu
basis <- ms_data$basis
coefs <- ms_data$coefs
time <- ms_data$time
lp <- matrix(runif(nrow(basis)), nrow = nrow(basis))
wei_sim <- sim_tte(pi = lp, mu = -1, coefs = 1.1,
```

```
  time = time, type = "weibull", end_time = 100)
ms_sim <- sim_tte(pi = lp, mu = mu, basis = basis,
  coefs = coefs, time = time, type = "ms")
```

`sim_tte_df` *Simulate Time-to-Event Data from a Custom mrgsolve Output*

Description

Applies inverse transform sampling to a data frame produced by an **mrgsolve** simulation that contains a survival probability column. This function is useful when you have a custom time-to-event model implemented in **mrgsolve** and want to generate event times.

Usage

```
sim_tte_df(
  dat,
  surv_var = "p11",
  id_var = "ID",
  time_var = "time",
  xdata = NULL,
  event_time_method = c("grid", "log_survival")
)
```

Arguments

`dat`	Data frame. Output from `mrgsim` containing at minimum columns for time, subject ID, and survival probability. Columns are resolved by name (not position).
`surv_var`	Character string. Name of the column containing the survival probability (probability of remaining event-free). Default `"p11"`.
`id_var`	Character string. Name of the subject ID column. Default `"ID"`.
`time_var`	Character string. Name of the time column. Default `"time"`.
`xdata`	Optional data frame of additional covariates to merge into the output. Must contain a column matching `id_var`.
`event_time_method`	Character string. `"grid"` (default) or `"log_survival"`. See "Event-time interpolation" below. Partial matching is allowed (resolved via `match.arg`).

Value

A data frame with columns:

sim_time Simulated event or censoring time.

sim_status Event indicator (1 = event, 0 = censored).

ID Subject identifier.

Plus any additional columns from `xdata`.

Trajectory contract

`dat` must contain, for each subject, one ordered survival trajectory satisfying:

- at least one row;
- `time` values that are non-negative, finite, and strictly increasing within the subject, with no duplicates. Rows are **not** sorted automatically: unsorted or duplicated times raise an error rather than being silently re-ordered or resolved by row order;
- survival probabilities in $[0, 1]$ (a small numerical tolerance of 10^{-8} is allowed for floating-point/ODE solver noise: values outside $[0, 1]$ but within this tolerance are accepted and then deterministically **clamped** to exactly 0 or 1 before monotonicity checking and event-time selection; values further outside $[0, 1]$ are a hard error, never silently clamped or dropped). This clamping cannot change whether a subject is classified as an event or censored, because `stats::runif()` draws from $[0, 1)$: a raw value already below 0 and its clamped value of exactly 0 both satisfy $value \leq U$ for every achievable U, and symmetrically a raw value already above 1 and its clamped value of exactly 1 both fail it;
- survival that is non-increasing within the subject (checked on the clamped values). A trajectory that increases by more than the same 10^{-8} tolerance at any step is rejected rather than silently repaired, because that pattern cannot arise from a valid survival function and usually indicates a data or model error; an increase within the tolerance is treated as numerical noise and accepted as-is (not further adjusted).

A trajectory is **not** required to start at survival = 1 or at time = 0: `sim_tte_df()` never invents an implicit $S(0) = 1$ state. Whatever survival value is reported at the first observed time is used exactly as given (see "Censoring and event-time resolution" below for what this implies when that first value is already at or below the sampled threshold). Subjects may have different time ranges; see below for what this means for censoring.

Censoring and event-time resolution

For each subject, a uniform random variate $U \sim \mathrm{Uniform}(0, 1)$ is drawn, and the **first reported time** at which survival is at or below U is located. If the first reported observation already satisfies $S(t_1) \leq U$, that first time is returned as the event time; no earlier crossing is inferred, and this is identical for both values of `event_time_method` (there is no earlier reported point to interpolate from). What happens next depends on `event_time_method`: see "Event-time interpolation" below. Exactly one `stats::runif(1)` is drawn per subject regardless of `event_time_method`.

If a subject's trajectory never crosses U, that subject is censored (`sim_status = 0`) at the **last reported time in their own trajectory**, i.e. `max(time)` for that subject, for both values of `event_time_method`. Because this censoring time is subject-specific, `sim_tte_df()` does **not** inherently represent a common administrative censoring cutoff unless every subject's trajectory happens to share the same final time — each subject's final observation time is effectively that subject's own censoring horizon. There is currently no separate `end_time` argument to `sim_tte_df()` itself; `sim_tte` provides one for its built-in Weibull/M-spline models by ensuring every subject's generated trajectory is extended to a common `end_time` (see `?sim_tte`).

Event-time interpolation

`event_time_method = "grid"` (the default) resolves event times purely by grid-based lookup, not interpolation: the returned time is always one of the times present in `dat` for that subject, never a value between two reported times.

`event_time_method = "log_survival"` additionally refines a crossing that occurs strictly after the first reported observation: it assumes the hazard is constant between the two consecutive reported trajectory points surrounding the crossing, and linearly interpolates cumulative hazard ($H = -\log(S)$) between them accordingly. For trajectories whose hazard is genuinely piecewise-constant on the reported grid (e.g. this package's M-spline model; see `sim_tte`'s "M-spline hazard carry convention"), this recovers the event time implied by that discretized hazard exactly. For trajectories generated from a continuously-varying hazard (e.g. the Weibull model with `shape != 1`, or a custom mechanistic model), it is an approximation whose accuracy improves as the reported time grid is refined. `event_time_method` never changes which subjects are classified as events vs. censored, and never changes the returned time for censoring or a first-observation crossing — it can only change `sim_time` for a crossing that falls strictly between two reported points, and the interpolated time is always within that interval (never before the earlier point, never after the subject's final reported time).

Reproducibility

`sim_tte_df()` draws exactly one `stats::runif(1)` per subject, in the order subjects are first encountered in `dat` (row order), not in ID-sorted order. With a fixed `set.seed()`, results are therefore reproducible **for a fixed row order** of `dat`, but reordering the rows of `dat` so that subjects are first encountered in a different sequence changes which random draw each subject receives, even with the same seed and the same set of per-subject trajectories. This is standard, unmodified R RNG behavior (a single shared random stream consumed sequentially), not a bug; `sim_tte_df()` does not currently use per-subject or per-ID RNG streams.

See Also

`sim_tte`, which builds the Weibull or M-spline trajectory this function consumes. `simtte_example_model` loads a bundled **mrgsolve** model whose simulated `p11` trajectory can be passed as `dat`. `sim_tte_ode`, for in-solver event detection with no separate trajectory step. `add_censoring` and `add_interval_censoring`, to post-process this function's output (call `add_censoring()` first if using both).

Examples

```
# Create a mock survival probability data frame (no mrgsolve required)
mock_dat <- data.frame(
  ID = rep(1:3, each = 50),
  time = rep(seq(0.1, 10, length.out = 50), 3),
  p11 = rep(exp(-0.3 * seq(0.1, 10, length.out = 50)), 3)
)
result <- sim_tte_df(mock_dat)
head(result)

# Custom column names
mock_dat2 <- data.frame(
  subj = rep(101:103, each = 50),
  tgrid = rep(seq(0.1, 10, length.out = 50), 3),
  surv = rep(exp(-0.3 * seq(0.1, 10, length.out = 50)), 3)
)
result2 <- sim_tte_df(mock_dat2, surv_var = "surv",
  id_var = "subj", time_var = "tgrid")
head(result2)
```

`sim_tte_ode` *Simulate Time-to-Event Data via In-Solver Event Detection*

Description

The event/censoring time is located *during* ODE integration (at the solver's own internal evaluations), by drawing a per-subject $U \sim \mathrm{Uniform}(0,1)$ and integrating the survival probability `p11` as a state, latching the first solver time at which `p11 <= U`. This is a different mechanism from `sim_tte`/`sim_tte_df`, which resolve event times *after* the fact from an already-materialized trajectory – the two are separate code paths, not two modes of one function.

Usage

```
sim_tte_ode(
  model,
  param = list(),
  omega = NULL,
  sigma = NULL,
  n = 1L,
  end = 100,
  delta = 1,
  add = NULL,
  idata = NULL,
  data = NULL,
  covariates = NULL,
  beta = NULL,
  formula = NULL,
  knots = NULL,
  boundary_knots = NULL,
  coefs = NULL,
  censoring = NULL,
  visits = NULL,
  seed = NULL,
  keep_trajectory = FALSE,
  ...
)
```

Arguments

`model`	What to simulate. One of: a character name from `sim_tte_ode_models()` (`"exponential"`, `"weibull"`, `"gompertz"`, `"mspline"`, `"pk_hazard"`, `"irm1_hazard"`–`"irm4_hazard"`, `"tmdd_hazard"`); a `"simtte_model"` returned by `tte_model` (your own converted **mrgsolve** model); or the `"simtte_prepared_model"` returned by `simtte_prepare_model`, for parallel or multi-process use. A bare compiled `mrgmod` is not accepted: convert it with `tte_model` first.

param — Named list or named numeric vector of population-level $PARAM overrides, forwarded to param. U and END are ordinarily supplied per subject via idata (see below), not here.

omega — Optional square matrix. For a model that already declares a matching $OMEGA block (any user-supplied model with its own ETA() wiring), forwarded to omat as before. For a built-in PK/PD hazard-library model with no declared block, applied instead as per-subject between-subject variability via idata columns – see "Between-subject variability" below.

sigma — Optional square matrix, forwarded to smat. User-model-only: no built-in library model's hazard depends on a residual-error-perturbed quantity, so there is no idata-based analogue for sigma.

n — Integer. Number of subjects. Ignored (and inferred from nrow(idata)) when idata is supplied. Also ignored (and inferred from nrow(covariates) instead) when idata is NULL and a baseline, no-ID formula covariate frame (see "Covariates and the linear predictor" below) has more than one row – supplying n explicitly in that case is only valid if it matches that row count exactly (an error names both numbers otherwise). When a dosing data with an ID column is also supplied, n (or idata's row count) must equal length(unique(data$ID)) exactly – inferred from it when n/idata are both left unresolved, checked otherwise (an error names both numbers; see data below).

end — Numeric scalar. Administrative censoring horizon. Written into the model's END parameter for every subject that does not already supply its own END column in idata, and used to guard in-solver event detection against latching a time past the requested follow-up (see "Boundary guard" below).

delta — Numeric scalar. Reported-grid spacing, used to build the requested output grid seq(0, end, by = delta) when add is not supplied. Ignored if add is supplied.

add — Numeric vector of explicit reported times. If NULL (default), seq(0, end, by = delta) is used. Either way, end itself is always included in the resolved grid (via .resolve_output_grid()), which the censoring rule and refinement step below both depend on.

idata — NULL (default) or a data frame with an ID column and, optionally, U and/or END columns. Missing U is drawn as stats::runif(nrow(idata)) (one independent draw per row/subject; see "Reproducibility" below). Missing END is filled with end. Any other column is passed through to **mrgsolve** as an ordinary per-subject covariate/parameter.

data — Optional dosing event data, forwarded to mrgsim's own data argument. If covariates is also supplied, the covariate-update rows built from it are appended to data (see "Covariates and the linear predictor" below); either or both may be NULL. When supplied together with more than one resolved subject, data must have an ID column whose unique values equal the resolved subject set (from n/idata) exactly – every subject dosed via data must have exactly one idata/ n row, and vice versa (an error, checked before simulating, names any mismatch and the fix). Before this check existed, n exceeding data's own ID count silently dropped the extra subjects from the result with no warning, and n short of it failed with **mrgsolve**'s own opaque "ID found in the data set, but not in idata" – both now error immediately from here instead.

`covariates` NULL (default) or a covariate data frame, either baseline (one row per subject, no `time` column) or time-varying (a `time` column, an optional `ID` column, and one numeric column per covariate) – see "Covariates and the linear predictor" below. Required together with `beta`; an error if only one of the two is supplied.

`beta` NULL (default) or a named numeric vector of log-linear coefficients. Without `formula`, one name per raw column of `covariates` (`lp(t) = sum(beta * covariates(t))`, the original interface, unchanged). With `formula`, one name per column of the design matrix `formula` implies (`lp(t) = X(t) %*% beta`) – see "Covariates and the linear predictor" below.

`formula` NULL (default, preserving the original `covariates`/`beta` behaviour exactly) or a one-sided formula, e.g. `~ age + arm`, built into a design matrix via `model.matrix` against `covariates`. See "Covariates and the linear predictor" below.

`knots`, `boundary_knots`, `coefs`
Only used (and, together with `coefs`, required) when `model = "mspline"`; an error if any is supplied for another `model`. `knots`: numeric vector of interior knots, strictly increasing, strictly inside `boundary_knots` – its length selects which shipped knot-count variant is used (see "M-spline knot-count variants" below). `boundary_knots`: numeric `c(lower, upper)`, default `c(0, end)`. `coefs`: numeric vector of non-negative M-spline coefficients, length `length(knots) + 3` (2 = spline degree, fixed).

`censoring` NULL (default) or a censoring spec list (see `add_censoring`). When supplied, an independent censoring time is drawn per subject and applied to `$events` via `add_censoring`, inside this call's own seeded block – see "Right censoring" below.

`visits` NULL (default), a visit schedule (a numeric vector or per-subject data frame, see `add_interval_censoring`), or a jitter spec `list(every = <spacing>, jitter = <optional>, jitter_dist = <optional, "uniform"/"normal">, jitter_trunc = <optional>)`. When supplied, `add_interval_censoring` is applied to `$events` **after** any `censoring` above – see "Interval censoring" below. The jitter-spec form calls `visit_schedule` internally, inside this call's own seeded block (`add_interval_censoring` itself does not accept this third form).

`seed` Optional integer. If supplied, `set.seed(seed)` is called before `U` is drawn.

`keep_trajectory`
Logical. If `TRUE`, the full `mrgsim()` output is retained in the returned object's `$trajectory`. Default `FALSE` (the coarse-output/in-solver-detection design is partly motivated by *not* requiring a dense materialized trajectory for large cohorts).

`...` Forwarded to `mrgsim`. `tgrid`, `obsonly`, `nocb`, and `carry_out`/`carry.out` are controlled internally to guarantee the contracts documented here and cannot be overridden; supplying them raises an error naming the argument (the same `.check_reserved_dots()` mechanism `sim_tte` already uses). This is where solver-tolerance arguments relevant to in-solver event-time accuracy (`rtol`, `atol`, `hmax`, ...) are set.

Details

Typical order of use: pick a model (a built-in name from `sim_tte_ode_models`, or convert your own **mrgsolve** model with `tte_model` first), call this function, then optionally post-process the `$events` it returns with `add_censoring` and `add_interval_censoring` (or pass `censoring`/`visits` to do both in this call).

The sections below follow one reading order: the mechanism itself ("Boundary guard", "Event-time refinement"), then what varies by `model` ("Weibull shape support", "M-spline knot-count variants", "PK/PD-linked hazard models"), then the cross-cutting inputs any model can take ("Covariates and the linear predictor", "Between-subject variability", "Right censoring", "Interval censoring"), and finally "Reproducibility", which ties every draw above together.

`model %in% c("exponential", "weibull", "gompertz")` are the three closed-hazard-form library models; `model = "mspline"` is the flexible baseline hazard (see "M-spline knot-count variants" below); `"pk_hazard"`, `"irm1_hazard"`–`"irm4_hazard"`, and `"tmdd_hazard"` are the six PK/PD-linked models (see "PK/PD-linked hazard models" below). `model` may also be a `simtte_model` produced by `tte_model`, which converts a user's own **mrgsolve** PK/PD model into one usable here – see `tte_model`'s own documentation, or `sim_tte_ode_models` to list the built-in names programmatically. A bare compiled `mrgmod` is not accepted directly; convert it first.

Value

An object of class `"simtte_ode_sim"`, a list with:

events Data frame with columns `ID`, `sim_time`, `sim_status` (1 = event, 0 = censored), `sim_reason` (`"event"`, `"censored"`, or `"administrative"` – see "Right censoring" below); plus `sim_time_left`/`sim_time_right` when `visits` is supplied (see "Interval censoring" below).

trajectory The full `mrgsim()` output data frame, or `NULL` unless `keep_trajectory = TRUE`.

model The compiled `mrgmod` object used.

seed The `seed` argument, as supplied (possibly `NULL`).

call The matched call, for reproducibility bookkeeping.

Boundary guard

the administrative-censoring contract: The shipped library models latch `TEVT` (the in-solver event time) only while `SOLVERTIME <= END`, so an internal solver evaluation past the requested follow-up horizon can never be recorded as an event time. `sim_tte_ode()` additionally re-applies the boundary rule on the R side, independent of the in-model guard: a subject is censored (`sim_status = 0`, `sim_time = end`) if no event was latched at all, **or** if the latched time is `>= end`. This is deliberately belt-and-braces: the in-model guard is an internal robustness measure; the R-side rule above is the documented contract.

Event-time refinement

The raw in-solver `TEVT` is quantized to the ODE solver's own internal step grid, which is generally finer than, but not independent of, the requested output grid and solver tolerance. For every subject classified as an event, `sim_tte_ode()` therefore refines the final `sim_time` using the already-validated cumulative-hazard interpolation `.interpolate_log_survival()` (the same method `sim_tte_df` offers as `event_time_method = "log_survival"`), applied **between the solver's**

own last pre-crossing evaluation and the crossing evaluation itself – an interval one internal solver step wide, captured directly from the model, not between the two *reported* trajectory rows bracketing `U`. `TEVT` itself therefore only decides **whether** a subject had an event before `end`; the reported `sim_time` for an event subject comes from this bracket interpolation, not from the raw `TEVT` value.

This makes refinement accuracy independent of `delta`**/** `add`: because the bracket is a solver step, not a reporting interval, its width (and hence the constant-hazard-within-bracket approximation error) does not grow as the requested output grid gets coarser – confirmed directly across `delta` of 4, 1, and 0.25 for Weibull `shape` = `0.5, 1, 2, 5` and Gompertz `gamma` = `+/-`, where an earlier, grid-based refinement approach had instead gotten *worse* than raw `TEVT` at coarse `delta` for shapes far from 1. `delta`/`add` therefore now control only the resolution of the returned `$trajectory` (when `keep_trajectory` = `TRUE`) and of the reported-grid *fallback* described next – not the accuracy of a refined event time.

The reported-grid method is kept only as a fallback, used for a subject when the in-solver bracket is degenerate (its pre-crossing evaluation coincides with, or is not earlier than, the crossing itself; `p11(0)` = `1` > `U` for every valid `U` < `1` rules this out at `t` = `0` specifically, but it is not ruled out later in integration – see the next paragraph) or unavailable (a user-supplied trajectory missing the `T_PRE`/`P_PRE`/ `P_POST` capture columns). **One** `message()` **is emitted per** `sim_tte_ode()` **call whenever any subject falls back**, naming how many did; a finer `delta`/`add` improves those subjects' event times specifically (the fallback is exactly the earlier grid-based method, so its accuracy still depends on the grid). This message never changes the returned result and is suppressible with `suppressMessages()`, like every guardrail message this function emits (see also "Weibull shape support" below).

Two fallback-triggering mechanisms have been characterized so far, with different remedies, and the message differentiates them (classified purely from the already-captured bracket columns, no new model-side logic): a **Weibull-type** signature (`P_POST` outside `[0, 1]`, a genuine solver overshoot across one large internal step, remedied by a finer `delta`/`add`), and an **M-spline-type** signature (`T_PRE` == `TEVT` exactly, with both `P_PRE` and `P_POST` legitimate probabilities – a same-timestamp, multiple-corrector-iteration degeneracy near a locally steep hazard, remedied by tighter `rtol`/`atol`, **not** `delta`). A subject matching neither signature (including every subject when the bracket columns are unavailable at all) is reported as unclassified, with generic advice. Only the non-zero buckets are named in the message.

A very steep hazard at a coarse `delta` **can trigger this fallback for an entire cohort at once**: mrgsolve/lsoda does not take an internal solver step past the next requested output time, so a coarse `delta` can force one large internal step across a region where the hazard changes by orders of magnitude, and a trial/corrector evaluation partway through that step can be momentarily unphysical – detected and safely handled by the fallback described above, not a silent wrong answer. Measured directly for Weibull `shape` = `5` and `shape` = `10`: fallback engagement drops from 100% of subjects to 0% between `delta` = `1` and `delta` = `0.5` at `shape` = `10`. The remedy is exactly what the fallback message already states: request a finer `delta`/`add`. This is a named, disclosed, non-blocking regime, not a separate guardrail.

Weibull shape support

The closed-form `sim_tte`'s Weibull model has no restriction on `shape` (an exact analytical expression is evaluated at every reported time). This function's `"weibull"` library model genuinely integrates the Weibull hazard as an ODE state, which diverges as $t \to 0^+$ for `shape` < `1` – a real

mathematical property of the hazard, not a floating-point artifact (integrating this exact hazard as an ODE fails outright for small shapes without a floor). `weibull_ode.cpp` floors the time value used *inside the hazard expression only* (never `SOLVERTIME` itself) at `1e-100`, which was verified directly to eliminate solver failure/NaN/non-monotone `p11` entirely across `shape` from 0.01 to 10, with accuracy at or below mrgsolve's own default solver tolerance for `shape >= 0.05`, degrading gracefully (larger but still finite, still monotone bias) below that. **This model always runs (no error) for any** `shape > 0`; for `shape < 0.05` specifically, prefer `sim_tte(type = "weibull")` if exactness near `t = 0` matters more than the in-solver mechanism's other properties. `sim_tte_ode()` additionally emits one `message()` per call when the effective `shape` (after `param` overrides) is below `0.05`, pointing back to this section; the simulation itself is unaffected (suppressible with `suppressMessages()`, like the refinement-fallback message described above).

M-spline knot-count variants

`model = "mspline"` evaluates a degree-2 (quadratic) normalized M-spline basis (Ramsay 1988 scaling, matching `splines2::mSpline(..., degree = 2, intercept = TRUE)` exactly – verified to floating-point precision) **inside the compiled model, in C++**, not as a precomputed covariate the way `sim_tte(type = "ms")` does: `HAZ = eta * sum_m coefs[m] * M_m(t)`, a genuinely continuous, smooth hazard, integrated as a real `$ODE` state like every other `sim_tte_ode()` model (chosen specifically so the in-solver bracket refinement above, which assumes a smooth hazard, applies to the M-spline model too – a piecewise-constant covariate does not give that).

Because the basis is compiled in, only a small, fixed set of interior-knot counts is shipped (**3, 5, or 7** interior knots, a moderate, fixed-set compromise) – `length(knots)` selects which shipped variant is used; any other count errors, naming the supported counts and pointing to `sim_tte(type = "ms")`, which accepts an arbitrary knot count (at the cost of being a piecewise-constant hazard on its own reporting grid, not a continuous one). `knots` (interior), `boundary_knots` (`c(lower, upper)`, default `c(0, end)`), and `coefs` (length `length(knots) + 3`, **non-negative** – the basis itself is non-negative, so a negative coefficient would make the hazard negative) together define the same hazard `sim_tte(type = "ms")` would produce from a `splines2::mSpline()` basis built with the same knots/degree and `coefs` as its own `coefs` argument – knots and coefficients are directly portable between the two functions.

No time floor is needed (unlike `"weibull"`'s divergence at `t = 0`): the M-spline basis is bounded and finite everywhere on `[boundary_knots[1], boundary_knots[2]]`, including exactly at the boundary – verified directly, not assumed. The hazard **is** identically zero outside that interval, though (a consequence of the clamped-knot spline construction): `end` (and every `idata$END`) must not exceed `boundary_knots[2]`, enforced as an error, not a silent extrapolation – the same restriction `sim_tte(type = "ms")`'s own `end_time > max(time)` check already applies, for the same reason.

Simulating from a Cox model

A Cox model specifies $h(t \mid X) = h_0(t)\exp(X\beta)$ but leaves the baseline $h_0(t)$ unspecified, so simulating from one requires choosing a baseline. `model = "mspline"` supplies a flexible one: its hazard is `exp(mu + lp) * sum_m coefs[m] * M_m(t)`, i.e. $h_0(t) = e^{\mu}\sum_m c_m M_m(t)$ with covariates entering multiplicatively through `lp = X %*% beta` (see "Covariates and the linear predictor" below). The simulated data therefore satisfy proportional hazards exactly, and `survival::coxph()` fitted to them estimates `beta`. See `vignette("pkpd-time-to-event", package = "simtte")` for a worked example recovering `beta` this way.

PK/PD-linked hazard models

Six library models link the hazard to a PK/PD quantity computed by a genuine **mrgsolve** PK/PD backbone, each reusing an unedited `mrgsolve::modlib()` model verbatim and adding only the survival scaffold and the hazard link itself (the per-model mechanical-edit list is documented at the top of each '`.cpp`' file, and is exactly what `tte_model` automates for a model of your own):

- `"pk_hazard"` (backbone `"pk2cmt"`): concentration- driven, log-linear: `HAZ = H0 * exp(beta_cp * CP)`.
- `"irm1_hazard"`–`"irm4_hazard"` (backbones `"irm1"`–`"irm4"`, the four indirect-response types): response-driven, relative to the subject's own pre-dose baseline: `HAZ = H0 * exp(beta_r * (RESP / RESP0 - 1))`, where `RESP0 = KIN / KOUT`. The sign of `beta_r` gives the direction (positive: a response above baseline raises the hazard; negative: it lowers it).
- `"tmdd_hazard"` (backbone `"tmdd"`): linked to `RC`, the drug-target complex, not to free drug concentration – chosen because `RC` is the quantity unique to target-mediated disposition (a `CP`-based link would not differ in kind from `"pk_hazard"`); `HAZ = H0 * exp(beta_rc * RC)`.

Every one of the six reduces exactly to the exponential model with hazard `H0` when its `beta_*` parameter is 0. Dosing and other event data are supplied via this function's existing `data` argument (**mrgsolve** event data, e.g. `evid`/`amt`/ `cmt`/`time` rows) exactly as for any **mrgsolve** model; `covariates`/`beta` (see "Covariates and the linear predictor" above) compose with these models unchanged – `lp` is folded into `eta` the same way the closed-form models do, wherever a given backbone itself multiplies a hazard-scale quantity by `eta` (verified directly, not assumed, for the models where `lp` is wired into the hazard link this way). **TMDD** is the intentionally stiff member of this set (fast binding kinetics next to slow target turnover) and is this library's designated steepness stress case. An outright ODE solver failure (not the reported-grid fallback described above, an **mrgsolve**/`lsoda` error that stops the call) is possible for `"tmdd_hazard"` at a sufficiently extreme parameter/dose combination – far past any shipped default – and is not wrapped with a more informative message: it means the parameters are too extreme for the requested tolerance; try tighter `rtol`/`atol` or a smaller dose.

Covariates and the linear predictor

`covariates`/`beta` generalize `sim_tte()`'s single-covariate `lp_data` mechanism to an arbitrary named set: $lp(t) = \sum_k \beta_k X_k(t)$, computed in R and supplied to the model as an ordinary time-varying `lp` covariate (last-observation-carried-forward, the same convention `sim_tte()` already uses) – no library model file needs to change to support this, since every one of them already recomputes `eta = exp(mu + lp)` inside `$ODE`. Coverage follows the same rule as `sim_tte()`'s Weibull `lp_data` path: every subject's `covariates` must include an observation at `time = 0`; the last known value is carried forward for any remaining follow-up (no M-spline-style "must reach end" requirement, since these are genuine ODE models, not a piecewise-constant-on-a-fixed-grid closed form). **Limitation**: when `covariates` has an `ID` column, it must equal `1:nrow(idata)` exactly – subject-specific covariates for a non-default (non-`1:n`) subject numbering are not yet supported. **Combining with** `data`: when a dosing `data` is also supplied, every dosing record's `lp` is filled with the covariate value already in force for that subject at that time (the same last-observation-carried-forward rule as above), so a dosing regimen and time-varying covariates compose freely, with no **mrgsolve** warning. `data` already carrying its own `lp` column is an error (which value should apply at a dosing record would otherwise be ambiguous).

Legacy form (formula not supplied): every name in beta must be a raw numeric column of covariates, used as-is – no factor coding, interactions, or transforms.

Formula form: supplying formula builds a design matrix X via model.matrix(formula,covariates), and lp(t) = X(t) %*% beta; beta's names must then match colnames(X) exactly (an error lists the expected names otherwise). covariates may be **baseline** (no time column – equivalent to a single time = 0 row per subject, carried forward for the whole follow-up) or **time-varying** (as in the legacy form). A baseline frame with no ID column is either a **population-level constant**, recycled to every subject, if it has exactly one row, or **one row per subject** if it has more than one row – in the latter case, n (see above) is inferred from its row count when not supplied explicitly. An implied intercept column is dropped automatically, with a message() naming why (the model's own baseline hazard parameter, e.g. H0/mu, already plays that role, so an intercept in lp would be additively unidentifiable against it) – write formula = ~ 0 + ... to suppress that message and get one design-matrix column per factor level instead of the usual k - 1 treatment-contrast columns for the *first* factor term in formula (R's own model.matrix convention for keeping a no-intercept design full rank; every other factor term still gets k - 1 columns). A separate message() names which term this happened for, since it changes what beta's expected names are.

Because X only ever needs to be linear in beta, not in the covariates themselves, terms nonlinear in a *covariate* are in scope: formula = ~ 0 + age + I(age^2) + log(dose) + splines::ns(weight, 3) + arm, or a covariate column already computed as an Emax-shaped transform of a *known, fixed* EC50 (e.g. a column dose / (EC50 + dose) for a literal numeric EC50), all work directly. Terms nonlinear in a *parameter* the model itself estimates or a caller varies across scenarios – an Emax link whose EC50 is itself unknown or varied – are out of scope for lp/formula: that belongs in the model's HAZ expression instead (tte_model(hazard = "H0 * (CP / (EC50 + CP))", params = list(EC50 = 5))), where it composes with PK/PD states, BSV, and dosing directly. The two express the same kind of relationship, but only the second lets EC50 itself be estimated or varied; a fixed EC50 can be written either way, and gives the same event-time distribution either way (verified directly).

Between-subject variability

omega adds between-subject variability (BSV) to the **structural PK/PD parameters** of the six models above (not to H0/lp/beta_*, the hazard-scale parameters this package's own scaffold adds): a log-normal draw, $\theta_i = \theta_{pop} \cdot \exp(\eta_i)$, computed in R with mvgauss and supplied as ordinary idata columns – exactly the mechanism already used for U (no model file needs a declared $OMEGA block for this to work). The valid targets are every structural parameter of the model's own **mrgsolve** backbone (see each *_hazard.cpp file's own [PROB] block for the exact list, e.g. pk_hazard: CL, V2, Q, V3, KA, KA2, VMAX, KM).

omega may be **named** (matching dimnames, any subset of the model's targets, any order – e.g. BSV on CL only) or **unnamed** (dimension must equal the full target list, applied positionally in the order documented for that model). Population values come from the *resolved* parameters (param overrides applied first), so param = list(CL = 3) together with omega on "CL" centers the draw on 3. Supplying both an idata column and an omega target with the same name is an error (no silent override in either direction).

Coexistence with a user-supplied model (a compiled mrgmod with its own $OMEGA/ETA() wiring): omega is dispatched on whether the resolved model already declares a matching block. A model that does (any hand-written model using the standard TVCL/ETA() idiom) goes through omat exactly as before – unaffected by this mechanism. Only a model with **no** declared block falls through to

the idata route, and only if it is one of the named models above; omega on any other model (the closed-form-style library models, or a user model with neither) keeps the existing informative error naming the missing $OMEGA block.

sigma is unaffected by any of this and stays exactly what it was: forwarded to smat, usable only for a model (necessarily user-supplied, for now) that already declares a $SIGMA block – no built-in library model's hazard depends on a residual-error-perturbed quantity.

Right censoring

$events always carries sim_reason, distinguishing "event" from administrative censoring at end ("administrative"). Supplying censoring adds a second, independent censoring process on top – a per-subject censoring time drawn from the requested distribution, with the observed outcome becoming min(event time, censoring time, end) and sim_reason = "censored" for a subject pulled earlier by the draw. See add_censoring for the full spec (the same function this argument calls internally); this is applied in R after simulation rather than as a second ODE compartment (out of scope for this release: censoring that depends on a subject's own simulated PK/PD trajectory, e.g. dropout driven by toxicity).

Interval censoring

Supplying visits maps the exact (and, if censoring was also supplied, already right-censored) outcome onto an interval, via add_interval_censoring: an event is only knowable at the first visit at or after it happened, giving (L, R]; a censored subject is only known event-free through their last visit at or before their own censoring time, giving (L, Inf). sim_time/sim_status/sim_reason are unchanged – sim_time_left/sim_time_right are added on top. Always applied **after** any censoring (see add_interval_censoring "Ordering with right censoring" for why the order matters); a list(every = , jitter =) value builds the schedule via visit_schedule first, inside this call's own seeded block – a post-simulation R mapping, not an in-solver mechanism.

A realistic (missed-visit, dropout, or outcome-reactive) schedule is **not** a visits = form this function accepts directly – build it standalone, then pass the result as visits:

```
sched <- visit_schedule(n = 50, every = 4, end = 24, jitter = 1, seed = 1)
realized <- thin_visits(sched, p_miss = 0.1, seed = 2)
sim <- sim_tte_ode(model = "exponential", param = list(H0 = 0.05),
  n = 50, end = 24, delta = 2, visits = realized, seed = 1)
```

visit_schedule_informative (which needs $events, so it cannot run before simulation at all) composes the same way, applied to sim$events after the fact and fed back through add_interval_censoring directly rather than through this function's visits argument a second time.

Reproducibility

If idata does not supply U, exactly one stats::runif(1) is drawn per row of idata (one row per subject), in row order, *before* **mrgsolve** is called. With a fixed seed (or a set.seed() call by the caller), results are therefore reproducible for a fixed idata row order.

The idata-based omega draw ("Between-subject variability" above) is likewise reproducible for a fixed seed – verified directly (not assumed from **mrgsolve**'s own documentation) that mvgauss's draw, called without its own seed argument, is insensitive to any runif()/rnorm() draws made in

between (such as the U draw above), so one seed deterministically reproduces both regardless of draw order.

Parallel simulation

model is a character library name or a simtte_model every time; either way, sim_tte_ode() builds (or reuses an already-built) compiled **mrgsolve** model and loadsos it before simulating – always, whether or not you prepare anything yourself. On a single process this needs no attention. On a makePSOCKcluster worker – a separate R process, with no memory of what the parent process already built – each worker independently paying its own first-call compile cost is not wrong, just wasteful (harmless for a fast-compiling built-in model; can be a real cost, up to a couple of seconds, for a tte_model-converted one).

simtte_prepare_model builds a model once and returns an object sim_tte_ode() never rebuilds; call it once per worker at cluster initialization, then pass its return value as model in every later sim_tte_ode() call on that worker:

```
cl <- parallel::makePSOCKcluster(2)
parallel::clusterCall(cl, simtte::simtte_prepare_model,
  model = "irm1_hazard")
results <- parallel::parLapply(cl, seeds, function(s) {
  sim_tte_ode(model = "irm1_hazard", param = list(H0 = 0.01, beta_r = 1),
    n = 50, end = 30, delta = 2, seed = s)
})
parallel::stopCluster(cl)
```

For a tte_model-converted model, build it *once* in the parent and broadcast the already-compiled object instead of having every worker convert/compile it independently:

```
tm <- tte_model(my_model, hazard = "H0 * exp(lp + beta * CP)",
  params = list(H0 = 0.01, beta = 0))
prepared <- simtte_prepare_model(tm)
parallel::clusterCall(cl, simtte::simtte_prepare_model, model = prepared)
results <- parallel::parLapply(cl, seeds, function(s) {
  sim_tte_ode(model = prepared, n = 50, end = 30, delta = 2, seed = s)
})
```

This is also the only pattern verified race-free under repeated, fresh-cache stress testing: a worker that only ever calls loadso() on an already-compiled object cannot race (a pure read, never a write), unlike a worker independently re-triggering its own build against a directory another process might be writing to at the same time – see the investigation report cited below for the measurements.

For model = "mspline", simtte_prepare_model only takes knots (only its *length* matters – it selects which shipped variant is compiled); coefs/boundary_knots are supplied to sim_tte_ode() as usual, not to simtte_prepare_model() (see simtte_prepare_model's own documentation for why).

Fork clusters (mclapply, Unix-only): a forked worker inherits the parent process's memory directly, compiled model included, so none of the above is needed – simulate immediately, with no simtte_prepare_model()/clusterCall() step at all.

simtte_model_cache controls where a model is built (default: a directory under tempdir(), i.e. session-scoped and CRAN-safe); see its own documentation for persisting a compiled model across R sessions, and simtte_model_cache_clear to remove what accumulates there (e.g. after an R/simtte/mrgsolve upgrade, for a persistent cache). sim_tte_ode()/simtte_prepare_model() never call options() themselves – mrgsolve.project/mrgsolve.soloc are passed explicitly, never set globally on your behalf.

See Also

Choosing a model: sim_tte_ode_models lists the built-in names; tte_model converts your own **mrgsolve** model into something model accepts. Post-processing the returned $events: add_censoring for independent right censoring (what censoring calls internally; censoring_rate_for helps choose its rate) and add_interval_censoring for assessment-time intervals (what visits calls; build the schedule with visit_schedule). Parallel or multi-process runs: simtte_prepare_model builds a model once per process, simtte_model_cache says where it is stored, and simtte_model_cache_clear removes it (see "Parallel simulation" above). The older grid-based engine: sim_tte and sim_tte_df, which this function does not replace.

Examples

```
sim <- sim_tte_ode(model = "exponential", param = list(H0 = 0.1),
  n = 50, end = 30, delta = 5, seed = 1)
head(sim$events)
```

---

sim_tte_ode_models *List sim_tte_ode()'s bundled library models*

Description

The built-in, bundled model names accepted by sim_tte_ode's model argument – the character-string case, not a tte_model-converted user model. Exported so a caller building a scenario grid over model (e.g. a companion package comparing several hazard mechanisms) can enumerate and validate model names programmatically, and look up each model's valid omega targets (see ?sim_tte_ode "Between-subject variability"), instead of hardcoding a copy of names(.ODE_LIBRARY_FILES) (private, @noRd) or reaching in via :::.

Usage

```
sim_tte_ode_models()
```

Details

"mspline" is included with bsv_targets = NA: it takes no between-subject variability (its hazard is entirely determined by knots/boundary_knots/coefs), which is a different situation from the exponential/Weibull/Gompertz models' empty-but-valid target list (also NA here, for the same reason: omega is rejected outright for those three, exactly as for "mspline", since none of the four has a structural PK/PD parameter for BSV to attach to).

Value

A data frame, one row per model: `model` (character, passable to `sim_tte_ode(model = ...)`), `bsv_targets` (character, comma-separated valid `omega` target names, or `NA` when the model takes no `omega` at all).

See Also

`sim_tte_ode`, where these names go as `model`. `tte_model`, to bring in a model beyond this list. `simtte_example_models` lists a different set: editable templates for `sim_tte_df`, not `sim_tte_ode()` models.

Examples

```
sim_tte_ode_models()
```

---

thin_visits *Thin a visit schedule with missed visits and dropout (C1)*

Description

Layers schedule-only randomness on top of a fixed visit schedule – no simulated outcome is read, so this is the "safe", always- non-informative member of the visit-process generators (contrast with `visit_schedule_informative`, which reads the simulated outcome by design). Each non-baseline visit is independently missed with probability `p_miss`; independently of that, each subject may also start dropping out: with probability `p_dropout`, one of their non-baseline visits is chosen uniformly at random as the dropout onset, and it and every later visit is dropped. The baseline visit (`time = 0`) is never dropped by either mechanism.

Usage

```
thin_visits(visits, p_miss = 0, p_dropout = 0, ids = NULL, seed = NULL)
```

Arguments

`visits`	A per-subject schedule (a data frame with `ID`/`time` columns, e.g. from `visit_schedule`), or a common numeric vector – in which case `ids` is required, and the vector is expanded to one copy per ID first (via the same internal validation `add_interval_censoring` itself uses).
`p_miss`, `p_dropout`	Numeric scalars in `[0, 1]`, default `0` (no thinning; `visits` is returned as given). The dropout-onset visit is drawn from the subject's *original* non-baseline visits, independently of any `p_miss` draws for that subject.
`ids`	Required only when `visits` is a numeric vector; the subject IDs to expand it across (e.g. `events$ID`).
`seed`	Optional integer. If supplied, `set.seed(seed)` is called before any draw. Leave `NULL` inside a larger already-seeded workflow.

Value

A data frame with ID/time columns – a (possibly shorter, per subject) schedule directly usable as add_interval_censoring's visits argument. Must be applied before add_interval_censoring, and composes with add_censoring in either relative order (they touch disjoint objects: this function only ever reads/writes a visit schedule, never events).

See Also

visit_schedule, which builds the schedule to thin; visit_schedule_informative, the outcome-reactive counterpart; add_interval_censoring, which takes the thinned schedule as visits.

Examples

```
sched <- visit_schedule(n = 20, every = 4, end = 24, seed = 1)
thinned <- thin_visits(sched, p_miss = 0.1, p_dropout = 0.05, seed = 2)
nrow(thinned) <= nrow(sched)
```

---

tte_model *Convert a PK/PD mrgsolve model into a 'sim_tte_ode()'-ready model*

Description

Writes the survival scaffold (p11, U, END, the TEVT/event_found/T_PRE/P_PRE/P_POST in-solver event-detection latch – sim_tte_ode's own mechanism) onto a user's own **mrgsolve** PK/PD model, exactly the mechanical edit list every shipped *_hazard.cpp library model already documents by hand at the top of its [PROB] block – generalized here to any model with an $ODE/$DES block, not just the six shipped backbones. See vignette("bring-your-own-model", package = "simtte") for the full mechanics (block-locating/splicing, the name-collision policy) and worked examples.

Usage

```
tte_model(model, hazard, params = list(), name = NULL, bsv_targets = NULL)
```

Arguments

model	One of: a file path to a model file; a character vector of model code (either one string with embedded newlines, or already one line per element – both accepted); or a compiled mrgmod (its own source is read back from mod@code). Must declare an $ODE or $DES block – a $PKMODEL/$PRED-only model is rejected, since the survival compartment has to be co-integrated in the same ODE system; use the ODE-form equivalent from modlib instead (e.g. "pk2cmt", not "pk2"). A simtte_model (this function's own return class) is rejected – convert the original model again with a different hazard/params instead of converting an already-converted one.

`hazard` Character scalar: one C++ expression for the instantaneous hazard `HAZ`, written in terms of any name already in scope inside the model's own `$ODE`/`$DES` (states, `$GLOBAL` `#defines`, `$MAIN` locals, `$PARAM` values) plus whatever names `params` adds – e.g. `"H0 * exp(lp + beta_cp * CP)"`. Include `lp` in the expression if `sim_tte_ode`'s `covariates`/ `beta` mechanism should affect this model; `lp` is always declared (default 0) whether or not the expression uses it.

`params` Named list, the hazard's own parameters and their population-default values (e.g. `list(H0 = 0.01, beta_cp = 0)`). Must not name `lp`, `U`, or `END` – those three are always added automatically (defaults `0`, `0`, `1e9`, matching every shipped `*_hazard.cpp` model) and are rejected if present here.

`name` Character scalar or `NULL` (default): passed to `mcode_cache` as the model name (caching and `print()` only). Left at `NULL`, a name is derived from an MD5 hash of the converted code, so two different conversions never share a build directory/model name even if called back to back – `mcode_cache()` is normally content-hash aware under a repeated name (verified, verified directly), but two builds under the same name landing in the same filesystem-timestamp second can defeat its underlying `make`-based rebuild check (verified directly – exactly the pattern an automated test loop hits, converting many small variants back to back); hashing the default name sidesteps this rather than relying on `mcode_cache()`'s own detection. A caller-supplied `name` does not get this treatment (it is used verbatim, for predictable caching across sessions) – avoid reusing one `name` for genuinely different `hazard`/ `params` content called in quick succession.

`bsv_targets` Character vector or `NULL` (default): the parameter names `sim_tte_ode`'s `omega` argument may target via its per-subject-`idata` route. Ignored (with a `message()`) if `model` already declares its own `$OMEGA` block – that model already has a working BSV route (`omat`), and the two never combine for the same model.

The compiled model is built into `simtte_model_cache` (never a global `mrgsolve` option). **Using the result in a separate process** (a cluster worker, a chunked/array job task – see `?sim_tte_ode` "Parallel simulation"): pass this object to `simtte_prepare_model` there; it rebuilds from `$code` automatically if the compiled object this function returned does not resolve in that process (its shared object lives wherever *this* process's own cache put it), so correctness never depends on the two processes sharing a filesystem.

Details

Call this first, then pass the result to `sim_tte_ode` as its `model` argument. In a parallel or multi-process run, run the result through `simtte_prepare_model` on each worker.

Nothing in the input model is edited in place except its `$GLOBAL`/`$MAIN` blocks (both singleton in **mrgsolve**; every other addition – a new `$PARAM`, `$INIT`, an `$ODE` block carrying only the hazard link, a `$CAPTURE` block – is appended as a brand-new block at the end of the file, never spliced into the model's own `$ODE`/`$DES` content.

Value

An object of class `"simtte_model"`, a list with `code` (the converted model source, printable directly with `cat(x$code, sep = "\n")`), `mod` (the compiled `mrgmod`), `hazard`, `hazard_params` (the fully

resolved parameter list, including the always-added `lp/ U/END`), `bsv_targets`, `name`, and `source` (how `model` was supplied: `"code"`, `"file"`, or `"mrgmod"`). Pass the whole object as `sim_tte_ode`'s `model` argument.

See Also

`sim_tte_ode`, which takes this function's return value as its `model` argument and runs the simulation. `sim_tte_ode_models`, if a built-in model already fits and no conversion is needed. `simtte_prepare_model`, to make the converted model usable in another R process; `simtte_model_cache` shows where it is compiled.

Examples

```
# mrgsolve::modlib() itself needs mrgsolve *attached* (library(mrgsolve)),
# not just installed; mrgsolve::mread() with an explicit 'project' reaches
# the same internal model library without that requirement.
irm1 <- mrgsolve::mread(model = "irm1",
  project = system.file("models", package = "mrgsolve"), compile = FALSE)
tm <- tte_model(irm1,
  hazard = "H0 * exp(lp + beta_r * (RESP / (KIN / KOUT) - 1.0))",
  params = list(H0 = 0.01, beta_r = 0),
  bsv_targets = c("CL", "V2", "KIN", "KOUT"))
tm
sim <- sim_tte_ode(model = tm, n = 20, end = 30, delta = 5, seed = 1)
head(sim$events)
```

---

visit_schedule *Generate a per-subject visit schedule with fixed spacing and jitter*

Description

Builds the per-subject data-frame form of `visits` that `add_interval_censoring` accepts: assessments every `every` time units from `0` to `end`, optionally perturbed per subject by up to `jitter` (a common pattern in trial simulation – scheduled visits rarely land exactly on the nominal day). The baseline visit (`time = 0`) is never jittered, so the result always satisfies `add_interval_censoring()`'s "must start at 0" requirement.

Usage

```
visit_schedule(
  n,
  every,
  end,
  jitter = 0,
  jitter_dist = c("uniform", "normal"),
  jitter_trunc = 2,
  seed = NULL
)
```

Arguments

n	Integer. Number of subjects; schedules are generated for ID = 1:n (remap the ID column afterward if your events uses different subject identifiers).
every	Positive numeric scalar. Nominal spacing between visits.
end	Non-negative numeric scalar. Last nominal visit time (the schedule is seq(0, end, by = every) before jitter).
jitter	Non-negative numeric scalar (default 0, no jitter). Each non-baseline visit is perturbed by an independent draw from jitter_dist: jitter is the draw's half-width for "uniform" (stats::runif(1, -jitter, jitter)) or its standard deviation for "normal", which is truncated at +/- jitter_trunc * jitter (see jitter_trunc) and drawn via inverse-CDF (stats::qnorm() on a stats::runif() restricted to the truncated range) – one uniform draw per visit either way, so both distributions consume the seeded stream the same shape of way.
jitter_dist	"uniform" (default; unchanged behavior and seeds from before this argument existed) or "normal" (truncated, see jitter_trunc; its seeds changed when the truncation/bound was added – see NEWS.md).
jitter_trunc	Positive numeric scalar, default 2. Only used for jitter_dist = "normal": the truncation half-width, in standard deviations, of the truncated normal jitter draw. Ignored for "uniform".
seed	Optional integer. If supplied, set.seed(seed) is called before any jitter is drawn. Leave NULL when calling this from inside a larger already-seeded workflow (this is how sim_tte_ode's own visits = argument uses it when given a jitter spec, so one seed reproduces U, any between-subject draw, censoring, and the visit jitter together).

Details

jitter is validated against every *before* any draw is made: the maximum possible deviation (jitter for "uniform", jitter_trunc * jitter for "normal") must be less than every / 2, which guarantees – by construction, not by a post-hoc check – that jittered visits can never cross order or collide (adjacent nominal visits are every apart; two deviations each smaller than half of that cannot close the gap) or go negative (the first non-baseline visit is at least every / 2 above 0). A jitter violating this bound is an error, naming the bound, rather than a silently reordered or clamped schedule.

Value

A data frame with ID/time columns, one row per subject per visit – directly usable as add_interval_censoring's visits argument.

See Also

add_interval_censoring, thin_visits and visit_schedule_informative, to layer missed visits/dropout on top of the schedule this function returns.

Examples

```
visit_schedule(n = 5, every = 4, end = 20, jitter = 1, seed = 1)
visit_schedule(n = 5, every = 4, end = 20, jitter = 0.4,
  jitter_dist = "normal", seed = 1)
```

`visit_schedule_informative` *Generate an outcome-informative visit schedule (C2)*

Description

Layers visit-process randomness that **reacts to the simulated event time** on top of a fixed visit schedule: a scheduled visit shortly before a subject's event may be missed at an elevated rate (`miss_near_event`, e.g. "too unwell to attend"), and/or an unscheduled extra visit may be added shortly after it (`extra_visit_after_event`, e.g. "brought in for follow-up after a reported symptom"). Both are optional and independent of each other; supplying neither is equivalent to `thin_visits` with `p_miss = p_miss_base`.

Usage

```
visit_schedule_informative(
  visits,
  events,
  end,
  miss_near_event = NULL,
  extra_visit_after_event = NULL,
  p_miss_base = 0,
  id_var = "ID",
  time_var = "sim_time",
  status_var = "sim_status",
  seed = NULL
)
```

Arguments

`visits`	A per-subject schedule (data frame with `ID`/`time`), or a common numeric vector (expanded per subject using `events[[id_var]]` – no separate `ids` argument needed here, unlike `thin_visits`, since `events` already supplies the ID list).
`events`	Data frame of simulated outcomes (typically the `$events` element of `sim_tte_ode`) with `ID`, `sim_time`, `sim_status` columns by default, **after** any `add_censoring` call (see "Pipeline position" below) – column names configurable via `id_var`/`time_var`/`status_var`.
`end`	Numeric scalar. An unscheduled extra visit (`extra_visit_after_event`) landing after `end` is dropped, not added – the same administrative horizon `add_censoring` uses.

`miss_near_event`
: `NULL` (default, no effect), or `list(window = <positive scalar>, p = <scalar in [0, 1]>)`. For an event subject (`sim_status == 1`), a scheduled visit `v` with `t - window <= v < t` (`t` the event time) is missed with probability `1 - (1 - p_miss_base) * (1 - p)` – `p_miss_base`'s background miss chance and `p`'s near-event miss chance are independent, competing risks, so they add rather than one overriding the other (this collapses exactly to `p` when `p_miss_base = 0`, the common case). Has no effect on a censored subject, or on a visit outside the window – both use `p_miss_base` alone.

`extra_visit_after_event`
: `NULL` (default, no effect), or a distribution spec in `add_censoring`'s `censoring` shape (`list(dist = "exponential", rate = ...)`, `"weibull"`, `"uniform"`, `"lognormal"`, `"gamma"`, or `"user"`). For an event subject only, one delay is drawn and an unscheduled visit is added at `sim_time + delay`, unless that would exceed `end`.

`p_miss_base`
: Numeric scalar in `[0, 1]`, default `0`. Background miss probability applied to every non-baseline visit not otherwise covered by `miss_near_event` (i.e. every visit for a censored subject, and every out-of-window visit for an event subject) – the same mechanism as `thin_visits`'s `p_miss`, folded in here so a caller does not need to chain both functions just to add a uniform background miss rate on top of the near-event effect.

`id_var`, `time_var`, `status_var`
: Character scalars naming the subject-ID, event/censoring-time, and status columns of `events`. Default `"ID"`/`"sim_time"`/`"sim_status"`.

`seed`
: Optional integer. If supplied, `set.seed(seed)` is called before any draw.

Value

A data frame with `ID`/`time` columns – directly usable as `add_interval_censoring`'s `visits` argument.

Informative assessment

This function's output is not a neutral assessment schedule. By construction, whether/when a subject is assessed now depends on whether/when they had the simulated event – exactly the situation most interval-censored analyses (Turnbull NPMLE, standard parametric interval-censored likelihoods) assume does *not* hold when they treat assessment times as ignorable. Using this function's schedule is appropriate for *studying* that bias (e.g. comparing a fit against `thin_visits`'s non-informative schedule at the same nominal miss rate, as the vignette's "Interval censoring" section does), not as a drop-in, general-purpose visit generator. A `message()` naming this is emitted on every call.

Pipeline position

`events` should already reflect any independent right censoring: exact simulation -> `add_censoring` -> `visit_schedule_informative()` -> `add_interval_censoring`. Generating the schedule from pre-censoring event times would make the near-event/extra-visit mechanics react to a `sim_time` that the final `events` no longer has – the same ordering reasoning as `add_interval_censoring`'s own "Ordering with right censoring" section.

See Also

visit_schedule, which builds the base schedule passed as visits; thin_visits, the non-informative counterpart; add_interval_censoring, which takes the returned schedule as its visits; add_censoring, which should run before this.

Examples

```
sim <- sim_tte_ode(model = "exponential", param = list(H0 = 0.05),
  n = 50, end = 24, delta = 2, seed = 1)
sched <- visit_schedule(n = 50, every = 4, end = 24, seed = 1)
realized <- visit_schedule_informative(sched, sim$events, end = 24,
  miss_near_event = list(window = 3, p = 0.5), seed = 2)
out <- add_interval_censoring(sim$events, visits = realized)
head(out)
```

Index

∗ **datasets**
ms_data, 10

add_censoring, 2, *4–8*, *21*, *24*, *27, 28*, *33*, *35*, *37*, *41–43*
add_interval_censoring, *4*, 4, *21*, *24*, *27, 28*, *33*, *35–37*, *39, 40*, *42, 43*

censoring_rate_for, *4*, 6, *8*, *35*

draw_censoring_times, *4*, *7*, 7

explore_pi_tq_surv, 8, *21*

loadso, *14*, *16, 17*, *34*

makePSOCKcluster, *34*
match.arg, *22*
mclapply, *34*
mcode_cache, *16*, *38*
model.matrix, *27*, *32*
modlib, *37*
mread_cache, *12*, *16*
mrgsim, *9*, *12*, *19*, *22*, *26, 27*
ms_data, 10
mvgauss, *32, 33*

omat, *26*, *32*, *38*

param, *26*
pexp, *6*
pgamma, *6*
plnorm, *6*
print.simtte_model, 10
print.simtte_ode_sim, 11
print.simtte_prepared_model, 11
pweibull, *6*

rgamma, *3*
rlnorm, *3*
rweibull, *3*

sim_tte, *2–5*, *9, 10*, *12, 13*, 18, *23–25*, *27*, *29, 30*, *35*
sim_tte_df, *2–5*, *9*, *12–14*, *19–21*, 22, *25*, *28*, *35, 36*
sim_tte_ode, *2–7*, *12*, *14–17*, *21*, *24*, 25, *35–41*
sim_tte_ode_models, *14*, *25*, *28*, *35*, 35, *39*
simtte_example_model, 12, *13, 14*, *24*
simtte_example_models, *12*, 13, *36*
simtte_model_cache, *14*, 14, *15–17*, *35*, *38, 39*
simtte_model_cache_clear, *15*, 15, *17*, *35*
simtte_prepare_model, *14–16*, 16, *25*, *34, 35*, *38, 39*
smat, *26*, *33*

tempdir, *14*, *35*
thin_visits, *6*, 36, *40–43*
tte_model, *12*, *14–17*, *25*, *28*, *31, 32*, *34–36*, 37

uniroot, *6*

visit_schedule, *5, 6*, *27*, *33*, *35–37*, 39, *43*
visit_schedule_informative, *6*, *8*, *33*, *36, 37*, *40*, 41