Package ‘dRiftDM’

December 1, 2025

Type Package

Title Estimating (Time-Dependent) Drift Diffusion Models

Version 0.3.1

License MIT + file LICENSE

Description Fit and explore Drift Diffusion Models (DDMs), a common tool in psychology for describing decision processes in simple tasks. It can handle both time-independent and time-dependent DDMs. You either choose prebuilt models or create your own, and the package takes care of model predictions and parameter estimation. Model predictions are derived via the numerical solutions provided by Richter, Ulrich, and Janczyk (2023, <doi:10.1016/j.jmp.2023.102756>).

Suggests testthat (>= 3.0.0), cowsay, knitr, rmarkdown, DMCfun, truncnorm, vdiffr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.3.3

Imports withr, parallel, pbapply, mirai, purrr, DEoptim, dfoptim, Rcpp, Rdpack, progress, stats, lifecycle, coda

LinkingTo Rcpp

Depends R (>= 4.1.0)

VignetteBuilder knitr

RdMacros Rdpack

URL `https://github.com/bucky2177/dRiftDM`, `https://bucky2177.github.io/dRiftDM/`

BugReports `https://github.com/bucky2177/dRiftDM/issues`

NeedsCompilation yes

Author Valentin Koob [cre, aut, cph],
Thomas Richter [aut, cph],
Markus Janczyk [aut]

Maintainer Valentin Koob <v.koob@web.de>

Repository CRAN

Date/Publication 2025-12-01 13:20:12 UTC

Contents

`b_coding<-` *The Coding of the Boundaries*

Description

Functions to get or set the "boundary coding" of an object.

Usage

```
b_coding(object, ...) <- value

## S3 replacement method for class 'drift_dm'
b_coding(object, ...) <- value

b_coding(object, ...)

## S3 method for class 'drift_dm'
b_coding(object, ...)

## S3 method for class 'fits_ids_dm'
b_coding(object, ...)

## S3 method for class 'fits_agg_dm'
b_coding(object, ...)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments.
`value`	a named list, specifying how boundaries are coded (see Details).

Details

`b_coding()` is a generic accessor function, and `b_coding<-()` a generic replacement function. The default methods get and set the "boundary coding", which is an attribute of drift_dm model.

The boundary coding summarizes which response time belongs to which boundary and how the boundaries shall be "labeled". The list specifies three entries:

- `column`, contains a single character string, indicating which column in an observed data set codes the boundaries.
- `u_name_value`, contains a numeric or character vector of length 1. The name of this vector gives a label for the upper boundary, and the entry gives the value stored in `obs_data[[column]]` coding the upper boundary.
- `l_name_value`, contains a numeric or character vector of length 1. The name of this vector gives a label for the lower boundary, and the entry gives the value stored in `obs_data[[column]]` coding the lower boundary.

The package `dRiftDM` has a default boundary coding:

- `column` = "Error"
- `u_name_value` = c("corr" = 0)
- `l_name_value` = c("err" = 1)

Thus, per default, dRiftDM assumes that any observed data set has a column "Error", providing the values 0 and 1 for the upper and lower boundary, respectively. The upper and lower boundaries are labeled "corr" and "err", respectively. These labels are used, for example, when calculating statistics (see calc_stats).

When calling `b_coding<-()` with `value` = `NULL`, the default "accuracy" coding is evoked

Value

For `b_coding()` a list containing the boundary coding For `b_coding<-()` the updated `drift_dm` or `fits_ids_dm` object

See Also

`drift_dm()`

Examples

```
# show the default accuracy coding of dRiftDM
my_model <- ratcliff_dm() # get a pre-built model
b_coding(my_model)

# can be modified/replaced
b_coding(my_model)[["column"]] <- "Response"

# accessor method also available for fits_ids_dm objects
# get an exemplary fits_ids_dm object (see estimate_model_ids)
fits <- get_example_fits("fits_ids_dm")
names(b_coding(fits))
```

`calc_stats` *Calculate Statistics*

Description

`calc_stats` provides an interface for calculating statistics/metrics on model predictions and/or observed data. Supported statistics include basic statistics on mean and standard deviation, Conditional Accuracy Functions (CAFs), Quantiles, Delta Functions, and fit statistics. Results can be aggregated across individuals.

Usage

```
calc_stats(object, type, ...)

## S3 method for class 'data.frame'
calc_stats(
  object,
  type,
  ...,
  conds = NULL,
  resample = FALSE,
  progress = 1,
  level = "individual",
  b_coding = NULL
)

## S3 method for class 'drift_dm'
calc_stats(object, type, ..., conds = NULL, resample = FALSE)

## S3 method for class 'fits_ids_dm'
calc_stats(
  object,
  type,
  ...,
  conds = NULL,
  resample = FALSE,
  progress = 1,
  level = "individual"
)

## S3 method for class 'fits_agg_dm'
calc_stats(
  object,
  type,
  ...,
  conds = NULL,
  resample = FALSE,
```

```
  progress = 1,
  level = "group",
  messaging = TRUE
)

## S3 method for class 'stats_dm'
print(
  x,
  ...,
  round_digits = NULL,
  print_rows = NULL,
  some = NULL,
  show_header = NULL,
  show_note = NULL
)

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

Arguments

`object`	an object for which statistics are calculated. This can be a data.frame of observed data, a drift_dm object, a `fits_ids_dm` object, or a `fits_agg_dm` object (see `estimate_dm()`).
`type`	a character vector, specifying the statistics to calculate. Supported values include `"basic_stats"`, `"cafs"`, `"quantiles"`, `"delta_funs"`, `"densities"`, and `"fit_stats"`.
`...`	additional arguments passed to the respective method and the underlying calculation functions (see Details for mandatory arguments).
`conds`	optional character vector specifying conditions to include. Conditions must match those found in the `object`.
`resample`	logical. If `TRUE`, then data is (re-)sampled to create an uncertainty estimate for the requested summary statistic. See Details for more information. Default is `FALSE`. Note that resampling does not work with `type = "fit_stats"`.
`progress`	integer, indicating if information about the progress should be displayed. 0 -> no information, 1 -> a progress bar. Default is 1.
`level`	a single character string, indicating at which "level" the statistic should be calculated. Options are `"group"` or `"individual"`. If `"individual"`, the returned `stats_dm` object contains an `"ID"` column.
`b_coding`	a list for boundary coding (see b_coding). Only relevant when `object` is a data.frame. For other `object` types, the `b_coding` of the `object` is used.
`messaging`	logical, if `FALSE`, no message is provided.
`x`	an object of type `stats_dm` or `stats_dm_list`, as returned by the function `calc_stats()`.
`round_digits`	integer, controls the number of digits shown. Default is 3.
`print_rows`	integer, controls the number of rows shown.

`some`	logical. If `TRUE`, a subset of randomly sampled rows is shown.
`show_header`	logical. If `TRUE`, a header specifying the type of statistic will be displayed.
`show_note`	logical. If `TRUE`, a footnote is displayed indicating that the underlying data.frame can be accessed as usual.

Details

`calc_stats` is a generic function to handle the calculation of different statistics/metrics for the supported object types. Per default, it returns the requested statistics/metrics.

List of Supported Statistics:

Basic Statistics

With "basic statistics", we refer to a summary of the mean and standard deviation of response times, including a proportion of response choices.

Conditional Accuracy Function (CAFs)

CAFs are a way to quantify response accuracy against speed. To calculate CAFs, RTs (whether correct or incorrect) are first binned and then the percent correct responses per bin is calculated.

When calculating model-based CAFs, a joint CDF combining both the pdf of correct and incorrect responses is calculated. Afterwards, this CDF is separated into even-spaced segments and the contribution of the pdf associated with a correct response relative to the joint CDF is calculated.

The number of bins can be controlled by passing the argument `n_bins`. The default is 5.

Quantiles

For observed response times, the function stats::quantile is used with default settings.

Which quantiles are calcuated can be controlled by providing the probabilites, `probs`, with values in $[0, 1]$. Default is `seq(0.1, 0.9, 0.1)`.

Delta Functions

Delta functions calculate the difference between quantiles of two conditions against their mean:

- $Delta_i = Q_{i,j} - Q_{i,k}$
- $Avg_i = 0.5 \cdot Q_{i,j} + 0.5 \cdot Q_{i,k}$

With i indicating a quantile, and j and k two conditions.

To calculate delta functions, users have to specify:

- `minuends`: character vector, specifying condition(s) j. Must be in `conds(drift_dm_obj)`.
- `subtrahends`: character vector, specifying condition(s) k. Must be in `conds(drift_dm_obj)`
- `dvs`: character, indicating which quantile columns to use. Default is "Quant_<u_label>". If multiple dvs are provided, then minuends and subtrahends must have the same length, and matching occurs pairwise. In this case, if only one minuend/subtrahend is specified, minuend and subtrahend are recycled to the necessary length.
- specifying `probs` is possible (see Quantiles)

Densities

With "densities", we refer to a summary of the distribution of observed or predicted data. For observed data, histogram values and kernel density estimates are provided. For predicted data, the model's predicted PDFs are provided.

Optional arguments are:

- `discr`: numeric, the band-width when calculating the histogram or the kernel density estimates. Defaults to `0.015` seconds

- `t_max`: numeric, the maximum time window when calculating the distribution summaries of observe data. Defaults to the longest RT (for observed data) or the maximum of the time domain of a model (which is the preferred choice, if possible). If necessary, `t_max` is slightly adjusted to match with `discr`.
- `scale_mass`: logical, only relevant if observed data is available. If `TRUE`, density masses are scaled proportional to the number of trials per condition.

Fit Statistics

Calculates the Log-Likelihood, Akaike and Bayesian Information Criteria, and root-mean squared-error statistic.

Optional arguments are:

- `k`: numeric, for penalizing the AIC statistic (see also stats::AIC and AIC.fits_ids_dm).
- `n_bins`, `probs`: numeric vectors, see the section on CAFs and Quantiles above
- `weight_err`: numeric scalar, determines how CAFs and quantiles are weighted. Default is `1.5`.

Resampling:

When `resampling = TRUE`, an uncertainty interval is provided via simulation. The default number of iterations is `R = 100`, which can be changed by passing the optional argument `R`.

If resampling is requested, the returned `stats_dm` object contains the column `"Estimate"`, coding the interval. The interval width is controlled via the optional argument `interval_level`, a single numeric value between 0 and 1 (default: `0.95`). The interpretation of this interval depends on the specific situation (see below).

Resampling at the Individual Level

If `object` is a `drift_dm` object (i.e., a single model created by `drift_dm()`), synthetic data are simulated under the model, and for each synthetic data set the requested statistic is calculated. The interval then reflects the range of these simulated statistics. To determine the number of trials for each synthetic data set, dRiftDM either uses the observed data attached to the model (if available) or the optional argument `n_sim` (passed to `simulate_data()`). Note that `n_sim` must be provided if no observed data are available, and that `n_sim` always has priority.

If `object` is a `drift_dm` object with attached observed data, resampling is also performed for the observed data. In this case, trials are bootstrapped, and for each bootstrap sample the requested statistic is calculated.

If `object` is a `data.frame`, `fits_agg_dm`, or `fits_ids_dm` object, resampling is performed for each individual if `level = "individual"`. For both models and observed data, synthetic or bootstrapped data sets are generated as described above.

Resampling at the Group Level

Group-level resampling is possible only if `object` is a `data.frame` (with an `"ID"` column), `fits_agg_dm`, or `fits_ids_dm` object. To request this, set `level = "group"`. Participants are then bootstrapped, and for each bootstrapped sample the aggregated statistic is calculated.

Interpretation of Intervals

For `level = "group"`, intervals represent bootstrapped confidence intervals For `level = "individual"`, intervals represent the variability in the statistic when data for a single participant are resampled or simulated under the model.

Note

For objects of type `fits_agg_dm`, which contain a mixture of group- and individual-level information, the `level` argument only affects resampling for the observed data. For the model itself, resampling is always performed under the fitted model, in the same way as for a `drift_dm` object.

Value

If `type` is a single character string, then a subclass of data.frame is returned, containing the respective statistic. Objects of type `sum_dist` will have an additional attribute storing the boundary encoding (see also b_coding). The reason for returning subclasses of data.frame is to provide custom `plot()` methods (e.g., plot.cafs). To get rid of the subclass label and additional attributes (i.e., to get just the plain underlying data.frame, users can use `unpack_obj()`).

If `type` contains multiple character strings (i.e., is a character vector) a subclass of list with the calculated statistics is returned. The list will be of type `stats_dm_list` (to easily create multiple panels using the respective `plot.stats_dm_list()` method).

The print methods `print.stats_dm()` and `print.stats_dm_list()` each invisibly return the supplied object `x`.

Note

When a model's predicted density function integrates to a value of less than `drift_dm_skip_if_contr_low()`, means and quantiles return the values `NA`. Users can alter this by explicitly passing the argument `skip_if_contr_low` when calling `calc_stats()` (e.g., `calc_stats(..., skip_if_contr_low = -Inf)`)

Examples

```
# Example 1: Calculate CAFs and Quantiles from a model ---------------------
# get a model for demonstration purpose
a_model <- ssp_dm()
# and then calculate cafs and quantiles
some_stats <- calc_stats(a_model, type = c("cafs", "quantiles"))
print(some_stats)

# Example 2: Calculate a Delta Function from a data.frame ------------------
# get a data set for demonstration purpose
some_data <- ulrich_simon_data
conds(some_data) # relevant for minuends and subtrahends
some_stats <- calc_stats(
  a_model,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)
print(some_stats, print_rows = 5)


# Example 3: Calculate Quantiles from a fits_ids_dm object -----------------
# get an auxiliary fits_ids_dm object
all_fits <- get_example_fits("fits_ids_dm")
some_stats <- calc_stats(all_fits, type = "quantiles")
print(some_stats, print_rows = 5) # note the ID column

# one can also request that the statistics are averaged across individuals
print(
  calc_stats(all_fits, type = "quantiles", average = TRUE)
```

```
)
```

`check_discretization` *Check time/space discretization via reference comparison*

Description

`check_discretization()` helps you choose or check time (`dt`) and space (`dx`) discretization settings. It computes a high-precision *reference* solution of the model's PDFs with `dt_ref`/`dx_ref`, and then compares the reference PDFs to the discretization settings of the supplied object, using the Hellinger distance per condition. Smaller distances indicate closer agreement with the reference — i.e., a sufficiently fine grid.

There are not yet overall and officially published recommendations on how large the Hellinger distance can be without affecting model precision, and this very likely will depend on the model itself. Based on some preliminary simulations using `dmc_dm()`, we would recommend trying to keep the Hellinger Distance at best below 10 percent. However, we also observed for extreme parameter values that the Hellinger distance can be even larger without sacrificing the qualitative model behavior, and vice versa! It is thus best to iterate between plotting model predictions and calculating the Hellinger Distance, to ensure that you can best interpret this quantity for your model at hand. Furthermore, we recommend to run parameter recoveries using `simulate_data()` and `estimate_dm()`, to check if you can recover data generated under your model with fine discretization using that same model with coarse discretization.

Usage

```
check_discretization(object, ...)

## S3 method for class 'drift_dm'
check_discretization(
  object,
  ...,
  dt_ref = 0.001,
  dx_ref = 0.001,
  round_digits = 5
)

## S3 method for class 'fits_ids_dm'
check_discretization(object, ...)

## S3 method for class 'fits_agg_dm'
check_discretization(object, ...)
```

Arguments

`object`	a drift_dm, `fits_agg_dm`, or `fits_ids_dm` object. (the latter two are returned by `estimate_dm()`)

`...`	further arguments passed forward to the respective method.
`dt_ref`, `dx_ref`	numeric scalars, providing a fine time or space step size for the reference solution. Defaults to `0.001`.
`round_digits`	number of decimal places to which the final Hellinger distances are rounded (default: `5`).

Details

Under the hood, for each condition, we concatenate the lower- and upper- boundary PDFs (`pdf_l`, `pdf_u`), interpolate the model PDFs to a time space matching with the reference PDFs, and then compute the Hellinger distance: $H(p,q) = \sqrt{1 - \int \sqrt{p(t)\,q(t)}\,dt}$

There are not yet overall, officially published recommendations on how large the Hellinger distance can be without affecting model precision, and this may even depend on the specific model. Based on preliminary simulations, we recommend trying to keep the average Hellinger distance below 5\

The reference discretizations (`dt_ref`/`dx_ref`) must be at least as fine as the object's current discretization settings (`dt_model`/`dx_model`). If `dt_model` < `dt_ref` or `dx_model` < `dx_ref`, an error is raised because the "reference" would not be the finest solution.

Value

a named numeric vector of Hellinger distances (one per condition) if `object` is of type drift_dm or `fits_agg_dm`. A data.frame of Hellinger distances across IDs and conditions if `object` is of type `fits_ids_dm`. Hellinger distances are in `[0, 1]`, where `0` means identical to the reference.

See Also

`estimate_dm()`, `trapz()`

Examples

```
# Example:
my_model <- ratcliff_dm()

# Assess current (dt=0.0075, dx=0.02) against a fine reference:
check_discretization(my_model)

# If distances are near zero across conditions, the current grid is adequate.
```

`coef<-` *Access Coefficients of a Model*

Description

Extract or set the coefficients/parameters objects supported by dRiftDM.

Usage

```
coef(object, ...) <- value

## S3 replacement method for class 'drift_dm'
coef(object, ..., eval_model = FALSE) <- value

## S3 method for class 'drift_dm'
coef(object, ..., select_unique = TRUE, select_custom_prms = TRUE)

## S3 method for class 'fits_agg_dm'
coef(object, ...)

## S3 method for class 'fits_ids_dm'
coef(object, ...)

## S3 method for class 'mcmc_dm'
coef(object, ..., .f = mean, id = NULL)

## S3 method for class 'coefs_dm'
print(
  x,
  ...,
  round_digits = drift_dm_default_rounding(),
  print_rows = 10,
  some = FALSE,
  show_header = TRUE,
  show_note = TRUE
)
```

Arguments

`object`	an object of type drift_dm, `fits_agg_dm`, `fits_ids_dm` (see also `estimate_dm()`), or `mcmc_dm`.
`...`	additional arguments passed forward (to `coef.drift_dm()` for objects of type `fits_agg_dm`; to `.f` for objects of type `mcmc_dm`.
`value`	numerical, a vector with valid values to update the model's parameters. Must match with the number of (unique and free) parameters.
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the parameters (see re_evaluate_model). Default is `FALSE`.
`select_unique`	logical, indicating if only those parameters shall be returned that are considered unique (e.g., when a parameter is set to be identical across three conditions, then the parameter is only returned once). Default is `TRUE`. This will also return only those parameters that are estimated. The argument is currently not supported for objects of type `mcmc_dm`.
`select_custom_prms`	logical, indicating if custom parameters shall be returned as well. Only has an effect if `select_unique = FALSE`. The argument is currently not supported for

	objects of type `mcmc_dm`.
`.f`	the function to be applied to each parameter of a chain. Must either return a single value or a vector (with always the same length). Default is `mean` (i.e., the mean function).
`id`	an optional numeric or character vector specifying the IDs of participants from whom to summarize samples. Only applicable when the model was estimated hierarchically. Use `id = NA` as a shorthand to summarize samples for all individuals in the chain object.
`x`	an object of type `coefs_dm`, as returned by the function `coef()` when supplied with a `fits_ids_dm` object.
`round_digits`	integer, controls the number of digits shown. Default is 3.
`print_rows`	integer, controls the number of rows shown.
`some`	logical. If `TRUE`, a subset of randomly sampled rows is shown.
`show_header`	logical. If `TRUE`, a header specifying the type of statistic will be displayed.
`show_note`	logical. If `TRUE`, a footnote is displayed indicating that the underlying data.frame can be accessed as usual.

Details

`coef.*()` are methods for the generic `stats::coef()` function; `coefs<-()` is a generic replacement function, currently supporting objects of type drift_dm.

The argument `value` supplied to the `coefs<-()` function must match with the vector returned from `coef(<object>)`. It is possible to update just part of the (unique) parameters.

Whenever the argument `select_unique` is `TRUE`, `dRiftDM` tries to provide unique parameter labels.

Value

For objects of type drift_dm, `coefs()` returns either a named numeric vector if `select_unique = TRUE`, or a matrix if `select_unique = FALSE`. If `select_custom_prms = TRUE`, custom parameters are added to the matrix.

For objects of type `fits_ids_dm`, `coefs()` returns a data.frame. If `select_unique = TRUE`, the columns will be the (unique, free) parameters, together with a column coding `IDs`. If `select_unique = FALSE`, the columns will be the parameters as listed in the columns of `prms_matrix` (see drift_dm), together with columns coding the conditions and `IDs`. If `select_custom_prms = TRUE`, the data.frame will also contain columns for the custom parameters. The returned data.frame has the class label `coefs_dm` to easily plot histograms for each parameter (see hist.coefs_dm).

For objects of type `fits_agg_dm`, returns the same as `coef.drift_dm()` (i.e., as if calling `coef()` with an object of type `drift_dm`)

For objects of type `mcmc_dm`, the return type depends on the model structure and the `.f` output:

- If the model is non-hierarchical or `id` is a single value (not `NA`), the function returns either a `vector` or a `matrix`, depending on whether `.f` returns a single value or a vector.
- In the hierarchical case, when `id` is a vector or `NA`, the function returns a `data.frame`. If `.f` returns a single value, the `data.frame` will contain one row per participant (with an `ID` column and one column per parameter). If `.f` returns a vector, the `data.frame` will include an additional column `.f_out`, coding the output of `.f` in long format.

See Also

`drift_dm()`

Examples

```
# get a pre-built model and a data set for demonstration purpose
# (when creating the model, set the discretization to reasonable values)
a_model <- dmc_dm()
coef(a_model) # gives the free and unique parameters
coef(a_model, select_unique = FALSE) # gives the entire parameter matrix
```

`component_shelf` *Diffusion Model Components*

Description

This function is meant as a convenient way to access pre-built model component functions.

Usage

```
component_shelf()
```

Details

The function provides the following functions:

- `mu_constant`, provides the component function for a constant drift rate with parameter `muc`.
- `mu_dmc`, provides the drift rate of the superimposed diffusion process of DMC. Necessary parameters are `muc` (drift rate of the controlled process), `a` (shape..), `A` (amplitude...), `tau` (scale of the automatic process).
- `mu_ssp`, provides the drift rate for SSP. Necessary parameters are `p` (perceptual input of flankers and target), `sd_0` (initial spotlight width), `r` (shrinking rate of the spotlight) and 'sign' (an auxiliary parameter for controlling the contribution of the flanker stimuli). Note that no `mu_int_ssp` exists.
- `mu_int_constant`, provides the complementary integral to `mu_constant`.
- `mu_int_dmc`, provides the complementary integral to `mu_dmc`.
- `x_dirac_0`, provides a dirac delta for a starting point centered between the boundaries (no parameter required).
- `x_uniform`, provides a uniform distribution for a start point centered between the boundaries. Requires a parameter `range_start` (between 0 and 2).
- `x_beta`, provides the function component for a symmetric beta-shaped starting point distribution with parameter `alpha`.
- `b_constant`, provides a constant boundary with parameter `b`.

- `b_hyperbol`, provides a collapsing boundary in terms of a hyperbolic ratio function with parameters `b0` as the initial value of the (upper) boundary, `kappa` the size of the collapse, and `t05` the point in time where the boundary has collapsed by half.
- `b_weibull`, provides a collapsing boundary in terms of a Weibull distribution with parameters `b0` as the initial value of the (upper) boundary, `lambda` controlling the time of the collapse, `k` the shape of the collapse, and `kappa` the size of the collapse.
- `dt_b_constant`, the first derivative of `b_constant`.
- `dt_b_hyperbol`, the first derivative of `b_hyperbol`.
- `nt_constant`, provides a constant non-decision time with parameter `non_dec`.
- `nt_uniform`, provides a uniform distribution for the non-decision time. Requires the parameters `non_dec` and `range_non_dec`.
- `nt_truncated_normal`, provides the component function for a normally distributed non-decision time with parameters `non_dec`, `sd_non_dec`. The Distribution is truncated to $[0, t_{max}]$.
- `dummy_t` a function that accepts all required arguments for `mu_fun` or `mu_int_fun` but which throws an error. Might come in handy when a user doesn't require the integral of the drift rate.

See `vignette("customize_ddms", "dRiftDM")` for more information on how to set/modify/customize the components of a diffusion model.

Value

A list of the respective functions; each entry/function can be accessed by "name" (see the Example and Details).

Examples

```
pre_built_functions <- component_shelf()
names(pre_built_functions)
```

`comp_funs<-` *The Component Functions of A Model*

Description

Functions to get or set the "component functions" of an object. The component functions are a list of functions providing the drift rate, boundary, starting point distribution, and non-decision time distribution They are at the heart of the package and shape the model's behavior.

Usage

```
comp_funs(object, ...) <- value

## S3 replacement method for class 'drift_dm'
comp_funs(object, ..., eval_model = FALSE) <- value
```

```
comp_funs(object, ...)

## S3 method for class 'drift_dm'
comp_funs(object, ...)

## S3 method for class 'fits_ids_dm'
comp_funs(object, ...)

## S3 method for class 'fits_agg_dm'
comp_funs(object, ...)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments passed down to the specific method.
`value`	a named list which provides the component functions to set (see Details)
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the component funtions (see re_evaluate_model). Default is `FALSE`.

Details

`comp_funs()` is a generic accessor function, and `comp_funs<-()` is a generic replacement function. The default methods get and set the "component functions". The component functions are a list of functions, with the following names (see also `vignette("customize_ddms", "dRiftDM")` for examples):

- `mu_fun` and `mu_int_fun`, provide the drift rate and its integral, respectively, across the time space.
- `x_fun` provides a distribution of the starting point across the evidence space.
- `b_fun` and `dt_b_fun` provide the values of the upper decision boundary and its derivative, respectively, across the time space. It is assumed that boundaries are symmetric.
- `nt_fun` provides a distribution of the non-decision component across the time space.

All of the listed functions are stored in the list `comp_funs` of the respective model (see also `drift_dm()`).

Each component function must take the model's parameters (i.e., one row of `prms_matrix`), the parameters for deriving the PDFs, the time or evidence space, a condition, and a list of optional values as arguments. These arguments are provided with values when `dRiftDM` internally calls them.

In order to work with `dRiftDM`, `mu_fun`, `mu_int_fun`, `b_fun`, `dt_b_fun`, and `nt_fun` must have the following declaration: `my_fun = function(prms_model, prms_solve, t_vec, one_cond, ddm_opts)`. Here, `prms_model` is one row of `prms_matrix`, prms_solve the parameters relevant for deriving the PDFs, `t_vec` the time space, going from 0 to `t_max` with length `nt + 1` (see drift_dm), and `one_cond` a single character string, indicating the current condition. Finally `dmm_opts` may contain additional values. Each function must return a numeric vector of the same length as `t_vec`. For `mu_fun`, `mu_int_fun`, `b_fun`, `dt_b_fun` the returned values provide the respective boundary/drift rate (and their derivative/integral) at every time step t. For `nt_fun` the returned values provide the density of the non-decision time across the time space (which get convoluted with the pdfs when solving the model)

In order to work with `dRiftDM`, `x_fun` must have the following declaration: `my_fun = function(prms_model, prms_solve,` Here, `x_vec` is the evidence space, going from -1 to 1 with length `nx + 1` (see drift_dm). Each function must return a numeric vector of the same length as `x_vec`, providing the density values of the starting points across the evidence space.

Drift rate and its integral::

The drift rate is the first derivative of the expected time-course of the diffusion process. For instance, if we assume that the diffusion process X is linear with a slope of v...

$$E(X) = v \cdot t$$

...then the drift rate at every time step t is the constant v, obtained by taking the derivative of the expected time-course with respect to t:

$$\mu(t) = v$$

Conversely, the integral of the drift rate is identical to the expected time-course:

$$\mu_{int}(t) = v \cdot t$$

For the drift rate `mu_fun`, the default function when calling `drift_dm()` is a numeric vector containing the number 3. Its integral counterpart `mu_int_fun` will return a numeric vector containing the values `t_vec*3`.

Starting Point Distribution::

The starting point of a diffusion model refers to the initial value taken by the evidence accumulation process at time $t = 0$. This is a PDF over the evidence space.

The default function when calling `drift_dm()` will be a function returning a dirac delta on zero, meaning that every potential diffusion process starts at 0.

Boundary::

The Boundary refers to the values of the absorbing boundaries at every time step t in a diffusion model. In most cases, this will be a constant. For instance:

$$b(t) = b$$

In this case, its derivative with respect to t is 0.

The default function when calling `drift_dm()` will be function for `b_fun` returning a numeric vector of length `length(t_vec)` containing the number 0.5. Its counterpart `dt_b` will return a numeric vector of the same length containing its derivative, namely, `0`.

Non-Decision Time::

The non-decision time refers to an additional time-requirement. Its distribution across the time space will be convoluted with the PDFs derived from the diffusion process.

In psychology, the non-decision time captures time-requirements outside the central decision process, such as stimulus perception and motor execution.

The default function when calling `drift_dm()` returns a dirac delta on $t = 0.3$.

Value

For `comp_funs()` the list of component functions.

For `comp_funs<-()` the updated drift_dm object.

Note

There is only a replacement function for drift_dm objects. This is because replacing the component functions after the model has been fitted (i.e., for a `fits_ids_dm` object) doesn't make sense.

See Also

`drift_dm()`

Examples

```
# get a pre-built model for demonstration
my_model <- ratcliff_dm()
names(comp_funs(my_model))

# direct replacement (see customize_ddms for a more information on
# how to write custom component functions)
# 1. Choose a uniform non-decision time from the pre-built component_shelf()
nt_uniform <- component_shelf()$nt_uniform
# swap it in
comp_funs(my_model)[["nt_fun"]] <- nt_uniform

# now update the flex_prms object to ensure that this model has the required
# parameters
prms <- c(muc = 3, b = 0.6, non_dec = 0.3, range_non_dec = 0.05)
conds <- "null"
new_flex_prms <- flex_prms(prms, conds = conds)
flex_prms(my_model) <- new_flex_prms

# accessor method also available for fits_ids_dm objects
# (see estimate_model_ids)
# get an exemplary fits_ids_dm object
fits <- get_example_fits("fits_ids_dm")
names(comp_funs(fits))
```

`conds<-` *The Conditions of an Object*

Description

Extract the conditions from a (supported) object.

Usage

```
conds(object, ...) <- value

## S3 replacement method for class 'drift_dm'
conds(object, ..., eval_model = FALSE, messaging = TRUE) <- value

conds(object, ...)

## S3 method for class 'drift_dm'
conds(object, ...)

## S3 method for class 'fits_ids_dm'
conds(object, ...)

## S3 method for class 'fits_agg_dm'
conds(object, ...)

## S3 method for class 'data.frame'
conds(object, ...)

## S3 method for class 'traces_dm_list'
conds(object, ...)
```

Arguments

`object`	an `R` object, see details
`...`	additional arguments passed forward.
`value`	a character vector, providing labels for the model's new conditions.
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the conditions (see re_evaluate_model). Default is `FALSE`.
`messaging`	logical, indicating if messages shall be displayed or not.

Details

`conds()` is a generic accessor function and `conds<-()` is a generic replacement function. The replacement method currently only supports drift_dm objects. The default methods get and set the conditions of an object.

When replacing the conditions of a drift_dm object, a new flex_prms object is created and then set to the model, resetting all parameter specifications and setting all parameter values to those of the previously first condition. In addition, if data was attached to the model, the data is removed. This is because there is no meaningful way for dRiftDM to know how the model should behave for the newly introduced condition(s), and how these new conditions relate to the old ones. Messages reminding the user of this behavior are displayed per default.

Value

For `conds()` `NULL` or a character vector with the conditions. `NULL` is given if the object has no conditions (e.g., when a data.frame has no `Cond` column).

For `conds<-()` the updated drift_dm object.

See Also

`drift_dm()`

Examples

```
# get a pre-built model to demonstrate the conds() function
my_model <- dmc_dm()
conds(my_model)

# accessor functions also work with other object types provided by dRiftDM
# (simulated traces; see the documentation of the respective function)
some_traces <- simulate_traces(my_model, k = 1)
conds(some_traces)

# get an exemplary fits_ids_dm object (see estimate_model_ids)
fits <- get_example_fits("fits_ids_dm")
conds(fits)

# also works with data.frames that have a "Cond" column
conds(dmc_synth_data)
```

`cost_function<-` *Access/Replace the Cost Function Label and Access the Cost Function Value*

Description

Functions to access/replace the cost function label of a `dRiftDM object` and to access the current cost function value. The cost function label codes which cost function is used during estimation (e.g., the negative log-likelihood). The cost function value indicates the current value of the cost function given the current set of parameters and the data.

Usage

```
cost_function(object, ...) <- value

## S3 replacement method for class 'drift_dm'
cost_function(object, ..., eval_model = FALSE) <- value

cost_function(object, ...)

## S3 method for class 'drift_dm'
cost_function(object, ...)

## S3 method for class 'fits_ids_dm'
```

```
cost_function(object, ...)

## S3 method for class 'fits_agg_dm'
cost_function(object, ...)

cost_value(object, ...)

## S3 method for class 'drift_dm'
cost_value(object, ...)

## S3 method for class 'fits_ids_dm'
cost_value(object, ...)

## S3 method for class 'fits_agg_dm'
cost_value(object, ...)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments passed down to `update_stats_agg()` when setting the cost function label.
`value`	a character string, providing the cost function label (options are `"neg_log_like"` or `"rmse"`)
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the conditions (see re_evaluate_model). Default is `FALSE`.

Value

- `cost_function()` returns a single character string, specifying the used cost function
- `cost_function<-()` returns the model object with the updated cost function.
- `cost_value()` returns a single numeric if `object` is of type `drift_dm` or `fits_agg_dm`. If there is no data attached to an object of type `drift_dm`, the function returns `NULL`. If `object` is of type `fits_ids_dm`, the function returns a data.frame with all cost values across participants.

See Also

`drift_dm()`, `re_evaluate_model()`

Examples

```
# get a pre-built model for demonstration purpose
a_model <- ratcliff_dm(obs_data = ratcliff_synth_data)
cost_function(a_model)
cost_value(a_model)

# switch the default cost function to rmse
cost_function(a_model) <- "rmse"
out <- estimate_dm(a_model, verbose = 0, messaging = FALSE)
# -> the model was estimated using the RMSE statistic
```

`ddm_opts<-` *Optional Arguments for the Component Functions*

Description

Functions to get or set the optional, user-defined R objects attached to a model object.

Usage

```
ddm_opts(object, ...) <- value

## S3 replacement method for class 'drift_dm'
ddm_opts(object, ..., eval_model = FALSE) <- value

ddm_opts(object, ...)

## S3 method for class 'drift_dm'
ddm_opts(object, ...)

## S3 method for class 'fits_agg_dm'
ddm_opts(object, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments passed down to the specific method.
`value`	an arbitrary R object.
`eval_model`	logical, indicating if the model should be re-evaluated or not after attaching the arbitrary R object to the model (see re_evaluate_model). Default is `FALSE`.

Details

When deriving model predictions, the model's component functions (see `comp_funs()`) are evaluated and the returned values are passed forward to dedicated numerical methods implemented in dRiftDM. To allow users to access arbitrary R objects within their custom component functions, models may contain a `ddm_opts` entry (see also `drift_dm()` and the end of `vignette("customize_ddms", "dRiftDM")` for an example).

`ddm_opts()` is a generic accessor function, and `ddm_opts<-()` is a generic replacement function. The default methods get and set the optional R object.

Value

For `ddm_opts()` the optional R object that was once supplied by the user, or `NULL`.

For `ddm_opts<-()` the updated drift_dm object.

See Also

`drift_dm()`, `comp_funs()`

Examples

```
# get a pre-built model for demonstration
a_model <- ratcliff_dm()
ddm_opts(a_model) <- "Hello World"
ddm_opts(a_model)
```

`dmc_dm` *Create the Diffusion Model for Conflict Tasks*

Description

This function creates a drift_dm object that corresponds to the Diffusion Model for Conflict Tasks by Ulrich et al. (2015).

Usage

```
dmc_dm(
  var_non_dec = TRUE,
  var_start = TRUE,
  instr = NULL,
  obs_data = NULL,
  sigma = 1,
  t_max = 3,
  dt = 0.0075,
  dx = 0.02,
  b_coding = NULL
)
```

Arguments

`var_non_dec`, `var_start`	logical, indicating whether the model should have a normally-distributed non-decision time or beta-shaped starting point distribution, respectively. (see `nt_truncated_normal` and `x_beta` in component_shelf). Defaults are `TRUE`. If `FALSE`, a constant non-decision time and starting point is set (see `nt_constant` and `x_dirac_0` in component_shelf).
`instr`	optional string with "instructions", see `modify_flex_prms()`.
`obs_data`	data.frame, an optional data.frame with the observed data. See obs_data.
`sigma`, `t_max`, `dt`, `dx`	numeric, providing the settings for the diffusion constant and discretization (see drift_dm)
`b_coding`	list, an optional list with the boundary encoding (see b_coding)

Details

The Diffusion Model for Conflict Tasks is a model for describing conflict tasks like the Stroop, Simon, or flanker task.

It has the following properties (see component_shelf):

- a constant boundary (parameter `b`)
- an evidence accumulation process that results from the sum of two subprocesses:
  - a controlled process with drift rate `muc`
  - a gamma-shaped process with a scale parameter `tau`, a shape parameter `a`, and an amplitude `A`.

If `var_non_dec = TRUE`, a (truncated) normally distributed non-decision with mean `non_dec` and standard deviation `sd_non_dec` is assumed. If `var_start = TRUE`, a beta-shaped starting point distribution is assumed with shape and scale parameter `alpha`.

If `var_non_dec = TRUE`, a constant non-decision time at `non_dec` is set. If `var_start = FALSE`, a starting point centered between the boundaries is assumed (i.e., a dirac delta over 0).

Per default the shape parameter `a` is set to 2 and not allowed to vary. This is because the derivative of the scaled gamma-distribution function does not exist at `t = 0` for `a < 2`. Currently, we recommend keeping `a` fixed to 2. If users decide to set `a != 2`, then a small value of `tol = 0.001` (default) is added to the time vector `t_vec` before calculating the derivative of the scaled gamma-distribution as originally introduced by Ulrich et al. (2015). Users can control this value by passing a value via `ddm_opts()` (see the example below). Note, however, that varying `a` can lead to large numerical inaccuracies if `a` gets smaller.

The model assumes the amplitude `A` to be negative for incompatible trials. Also, the model contains the custom parameter `peak_l`, containing the peak latency (`(a-2)*tau`).

Value

An object of type `drift_dm` (parent class) and `dmc_dm` (child class), created by the function `drift_dm()`.

Note

The scaling of the parameters in `dRiftDM` is different to Ulrich et al. (2015). This is because `dRiftDM` works in seconds and with a diffusion constant of 1, while the original DMC parameterization is in milliseconds and with a diffusion constant of 4. We describe how to convert the parameters on our website.

References

Ulrich R, Schröter H, Leuthold H, Birngruber T (2015). "Automatic and controlled stimulus processing in conflict tasks: Superimposed diffusion processes and delta functions." *Cognitive Psychology*, **78**, 148–174. doi:10.1016/j.cogpsych.2015.02.005.

Examples

```
# the model with default settings
my_model <- dmc_dm()
```

```
# the model with no variability in the starting point and a finer
# discretization
my_model <- dmc_dm(var_start = FALSE, dt = .005, dx = .01)

# we don't recommend this, but if you really want a != 2, just do...
# (see the Details for more warnings/information about this)
my_model <- dmc_dm(instr = "a ~!")
coef(my_model)["a"] <- 1.9
# -> if you want to control the small value that is added to t_vec when
# calculating the drift rate for a != 2, just use ...
ddm_opts(my_model) <- 0.0001 # ==> t_vec + 0.0001
ddm_opts(my_model) <- NULL # default ==> t_vec + 0.001
```

`dmc_synth_data` *A synthetic data set with two conditions*

Description

This dataset was simulated by using the Diffusion Model for Conflict tasks (see `dmc_dm()`) with parameter settings that are typical for a Simon task.

Usage

```
dmc_synth_data
```

Format

A data frame with 600 rows and 3 columns:

RT Response Times

Error Error Coding (Error Response = 1; Correct Response = 0)

Cond Condition ('comp' and 'incomp')

`drift_dm` *Create a drift_dm object*

Description

This function creates an object of type drift_dm, which serves as the parent class for all further created drift diffusion models (all of which have a child class label, e.g., `dmc_dm`). The objects created by `drift_dm()` are the backbone of the dRiftDM package. For a list of all pre-built models, see `vignette("dRiftDM", "dRiftDM")`.

Usage

```
drift_dm(
  prms_model,
  conds,
  subclass,
  instr = NULL,
  obs_data = NULL,
  sigma = 1,
  t_max = 3,
  dt = 0.001,
  dx = 0.001,
  solver = "kfe",
  cost_function = "neg_log_like",
  mu_fun = NULL,
  mu_int_fun = NULL,
  x_fun = NULL,
  b_fun = NULL,
  dt_b_fun = NULL,
  nt_fun = NULL,
  b_coding = NULL
)

## S3 method for class 'drift_dm'
print(x, ..., round_digits = drift_dm_default_rounding())
```

Arguments

`prms_model`	a named numeric vector of the model parameters. The names indicate the model's parameters, and the numeric entries provide the current parameter values.
`conds`	a character vector, giving the names of the model's conditions. values within `conds` will be used when addressing the data and when deriving the model's predictions.
`subclass`	a character string, with a name for the newly created diffusion model (e.g., `my_dmc_dm`). This will be the child class.
`instr`	an optional character string, providing "instructions" for the underlying flex_prms object.
`obs_data`	an optional data.frame, providing a data set (see `obs_data()` for more information).
`sigma`	the diffusion constant. Default is `1`.
`t_max`	the maximum of the time space. Default is set `3` (seconds).
`dt`, `dx`	the step size of the time and evidence space discretization, respectively. Default is set to `.001` (which refers to seconds for dt). Note that these values are set conservatively per default. In many cases, users can increase the discretization.
`solver`	a character string, specifying which approach to use for deriving the first passage time. Options are `kfe` or `im_zero`. Default is `kfe`, which provides access to the numerical discretization of the Kolmogorov Forward Equation.

`cost_function`	a character string, specifying the cost function used during estimation. Options are `neg_log_like` (negative log-likelihood), `rmse` (root-mean-squared error). Default is `neg_log_like`.
`mu_fun`, `mu_int_fun`, `x_fun`, `b_fun`, `dt_b_fun`, `nt_fun`	Optional custom functions defining the components of a diffusion model. See `comp_funs()`. If an argument is `NULL`, dRiftDM falls back to the respective default functions, which are documented in `comp_funs()`.
`b_coding`	an optional list, specifying how boundaries are coded. See `b_coding()`. Default refers to accuracy coding.
`x`	an object of type `drift_dm`
`...`	additional parameters
`round_digits`	integer, controls the number of digits shown for `print.drift_dm()`. Default is `3`.

Details

To modify the entries of a model users can use the replacement methods and the `modify_flex_prms()` method (see also `vignette("dRiftDM", "dRiftDM")` and `vignette("customize_ddms", "dRiftDM")`).

Value

For `drift_dm()`, a list with the parent class label `"drift_dm"` and the child class label `<subclass>`. The list contains the following entries:

- An instance of the class flex_prms for controlling the model parameters. Provides information about the number of parameters, conditions etc.
- Parameters used for deriving the model predictions, prms_solve, containing the diffusion constant (`sigma`), the maximum of the time space (`t_max`), the evidence and space discretization (`dt` and `dx`, respectively), and the resulting number of steps for the time and evidence space discretization (`nt` and `nx`, respectively).
- A character string solver, indicating the method for deriving the model predictions.
- A character string cost_function, indicating the cost function used for model estimation.
- A list of functions called comp_funs, providing the components of the diffusion model (i.e., `mu_fun`, `mu_int_fun`, `x_fun`, `b_fun`, `dt_b_fun`, `nt_fun`). These functions are called in the depths of the package and will determine the behavior of the model

If (optional) observed data were passed via `obs_data()`, the list will contain an entry `obs_data`. This is a (nested) list with stored response times for the upper and lower boundary and with respect to each condition. If the cost function is a summary statistic requiring quantiles, CAFs, etc., the model also contains the entries `stats_agg` and `stats_agg_info`. The former is a (nested) list with descriptive statistics. The latter contains information about the descriptive statistics (e.g., the quantile levels).

If the model has been evaluated (see `re_evaluate_model()`), the list will contain...

- ... the cost value; can be addressed via `cost_value()`.
- ... the PDFs of the first passage time; can be addressed via `pdfs()`.

If the model was estimated (which includes its evaluation), the list will contain `estimate_info`. This entry contains a convergence flag (`conv_flag`, logical) and the `optimizer` (a string).

Finally, if arbitrary R objects were passed via `ddm_opts()` (to access these objects when evaluating the component functions) the list will contain an entry `ddm_opts`.

Every model also has the attribute b_coding, which summarizes how the boundaries are labeled.

For `print.drift_dm()`, the supplied `drift_dm` object `x` (invisible return).

See Also

`conds()`, `flex_prms()`, `prms_solve()`, `solver()`, `obs_data()`, `comp_funs()`, `b_coding()`, `coef()`, `pdfs()`

Examples

```
# Plain call, with default component functions -----------------------------
# create parameter and condition vectors
prms <- c(muc = 4, b = 0.5)
conds <- c("one", "two")

# then call the backbone function (note that we don't provide any component
# functions, so dRiftDM uses the default functions as documented in
# comp_funs())
my_model <- drift_dm(prms_model = prms, conds = conds, subclass = "example")
print(my_model)
```

`estimate_dm` *Fit a DDM to Observed Data*

Description

`estimate_dm()` is the main function to fit a drift diffusion model (DDM) in `dRiftDM`. Several ways of fitting a model are supported: fitting a single participant, fitting multiple participants separately or aggregated, and fitting a (hierarchical) Bayesian model. The particular way is controlled via the `approach` argument.

Usage

```
estimate_dm(
  drift_dm_obj,
  obs_data = NULL,
  approach = NULL,
  optimizer = NULL,
  control = list(),
  n_cores = 1,
  parallelization_strategy = NULL,
  lower = NULL,
```

```
  upper = NULL,
  start_vals = NULL,
  means = NULL,
  sds = NULL,
  shapes = NULL,
  rates = NULL,
  n_chains = 40,
  burn_in = 500,
  samples = 1000,
  prob_migration = 0.1,
  prob_re_eval = 1,
  messaging = TRUE,
  seed = NULL,
  ...
)

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

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

## S3 method for class 'mcmc_dm'
print(x, ..., round_digits = drift_dm_default_rounding())
```

Arguments

`drift_dm_obj`	a drift_dm object containing the model to be fitted.
`obs_data`	an optional data.frame (see also obs_data). If no `ID` column is present, a single-individual setup is assumed. If an `ID` column is present, the model is fitted separately for each individual.
`approach`	an optional character string, specifying the approach to fitting the model. Options are `"sep_c"`, `"agg_c"`, `"sep_b"`, `"hier_b"` (see the Details).
`optimizer`	a character string. For classical optimization, one of `"nmkb"`, `"Nelder-Mead"`, `"BFGS"`, `"L-BFGS-B"`, `"DEoptim"`. For the Bayesian framework, only `"DE-MCMC"` is currently supported. If `NULL` and if a classical optimization approach is used, defaults to `"DEoptim"` or `"Nelder-Mead"`, depending on whether `lower/upper` are provided or not. If `NULL` and if a Bayesian framework is used, defaults to `"DE-MCMC`. Note that `"BFGS"` and `"L-BFGS-B"` are often unstable.
`control`	a list of control parameters passed to the optimizer (for Nelder-Mead, BFGS, and L-BFGS-B, see stats::optim; for nmkb, see dfoptim::nmkb; for DEoptim, see DEoptim::DEoptim). Per default, we set the `trace` control argument for DEoptim::DEoptim to `FALSE`. Also, we set the `parscale` control argument for "Nelder-Mead" via stats::optim to `pmax(x0, 1e-6)`.
`n_cores`	an integer > 0, indicating the number of CPU cores/threads to use (at the moment, this doesn't have an effect when fitting a single individual within the Bayesian framework).

`parallelization_strategy`	an integer, controlling how parallelization is performed when fitting multiple individuals with the classical approach. If `1`, parallelization is across individuals. If `2`, parallelization is within individuals (currently only supported for `"DEoptim"`). Defaults to `1`.
`lower`, `upper`	numeric vectors or lists, specifying the lower and upper bounds on each parameter to be optimized (see Details).
`start_vals`	optional starting values for classical single-subject fits and when using an optimizer that requires a starting value. Can be a numeric vector of model parameters when fitting a single individual, or a `data.frame` with columns for each model parameter. In the latter case, enables multi-start (one row per start). For `'approach = "separately"'`, a `data.frame` with an `ID` column is required.
`means`, `sds`, `shapes`, `rates`	optional numeric vectors for prior specification (when using the Bayesian framework, see Details).
`n_chains`	an integer, providing the number of MCMC chains (Bayesian framework).
`burn_in`	an integer, number of burn-in iterations (Bayesian framework).
`samples`	an integer, number of post-burn-in samples per chain (Bayesian framework).
`prob_migration`	a numeric in `[0,1]`, controlling the migration probability of the `DE-MCMC` algorithm (Bayesian framework).
`prob_re_eval`	a numeric in `[0,1]`, probability to re-evaluate the model at current group-level parameters during sampling (Bayesian framework; only relevant for the hierarchical case).
`messaging`	a logical, if `TRUE` progress/info messages are printed
`seed`	an optional integer to set the RNG seed for reproducibility.
`...`	additional arguments forwarded to lower-level routines. Options are: `progress/verbose` (integers, for controlling progress bars and verbosity of estimation infos), `round_digits` (for controlling the number of digits for rounding when printing individual model evaluations; if `verbose = 2`), `return_runs` (when fitting a single individual and starting the estimation routine with multiple starting points; if `TRUE`, then a list of all routines is returned), `probs/n_bins` (the quantile levels and the number of CAF bins when fitting aggregated data using the RMSE cost function), `use_ez/n_lhs` (logical and integer; the first controls if EZ-Diffusion Parameter Estimates shall be used for determining starting points; the latter controls the number of parameters to sample per dimension for the latin hypercube sampling when searching for starting values)
`x`	an object of type `fits_agg_dm`, `fits_ids_dm`, or `mcmc_dm`
`round_digits`	integer, specifying the number of decimal places for rounding in the printed summary. Default is 3.

Details

Fitting Approaches:

The function supports different "approaches" to fitting data.

- `"sep_c"`: This means that data is always considered separately for each participant (if there are multiple participants) and that a classical approach to parameter optimization is used. This means that a standard cost_function is minimized (e.g., the negative log-likelihood). If users provide only a single participant or a data set without an `ID` column, then the model is fitted just once to that data set.
- `"agg_c"`: This fits the model to aggregated data. For each individual in a data set, summary statistics (e.g., quantiles, accuracies) are calculated, and the model is fitted once to the average of these summary statistics.
- `"sep_b"`: Similar to `sep_b"`, although a Bayesian approach is used to sample from the posterior distribution.
- `"hier_b"`: A hierarchical approach to parameter estimation. In this case all participants are considered simultaneously and samples are drawn both at the individual-level and group-level.

The optimizers `"nmkb"`, `"L-BFGS-B"`, and `"DEoptim"` (for classical parameter optimization) require the specification of the `lower/upper` arguments.

Fitting to Aggregated Data:

For aggregated fits, aggregated statistics are set to the model and the cost function is switched to `"rmse"`. If incompatible settings are requested, the function switches to a compatible configuration and informs the user with messages (these messages can be suppressed via the `messaging` argument).

Specifying `lower/upper` for Classical optimization:

the function `estimate_model_dm()` provides a flexible way of specifying the optimization space; this is identical to specifying the parameter simulation space in `simulate_data.drift_dm()`.

Users have three options to specify the search space (see also the examples below):

- Plain numeric vectors (not very much recommended). In this case, `lower/upper` must be sorted in accordance with the parameters in the underlying flex_prms object of `drift_dm_obj` that vary for at least one condition (call `print(drift_dm_obj)` and have a look at the columns of the `Parameter Settings` output; for each column that has a number > 0, specify an entry in `lower/upper`).
- Named numeric vectors. In this case `lower/upper` have to provide labels in accordance with the parameters that are considered "free" at least once across conditions (call `coef(drift_dm_obj)` and provide one named entry for each parameter; dRiftDM will try to recycle parameter values across conditions).
- The most precise way is when `lower/upper` are lists. In this case, the list requires an entry called "default_values" which specifies the named or plain numeric vectors as above. If the list only contains this entry, then the behavior is as if `lower/upper` were already numeric vectors. However, the `lower/upper` lists can also provide entries labeled as specific conditions, which contain named (!) numeric vectors with parameter labels. This will modify the value for the upper/lower parameter space with respect to the specified parameters in the respective condition.

Specifying Priors for Bayesian Estimation:

(Default) Prior settings in the non-hierarchical case:

Let $\theta^{(j)}$ indicate parameter j of a model (e.g., the drift rate). The prior on $\theta^{(j)}$ is a truncated normal distribution:

$$\theta^{(j)} \sim NT(\mu^{(j)}, \sigma^{(j)}, l^{(j)}, u^{(j)})$$

With $\mu^{(j)}$ and $\sigma^{(j)}$ representing the mean and standard deviation of parameter j. $l^{(j)}$ and $u^{(j)}$ represent the lower and upper boundary. $\mu^{(j)}$ is taken from the `mean` argument or the currently set model parameters (i.e., from `coef(drift_dm_obj)`) when calling the function. $\sigma^{(j)}$ is, per default, equal to $\mu^{(j)}$. This can be changed by passing the `sd` argument. The lower and upper boundaries of the truncated normal are `-Inf` and `Inf` per default. This can be altered by passing the arguments `lower` and `upper` (see the examples below).

(Default) Prior settings in the hierarchical case:

Let $\theta_i^{(j)}$ indicate parameter j for participant i (e.g., the drift rate estimated for individual i). The prior on $\theta_i^{(j)}$ is a truncated normal distribution:

$$\theta_i^{(j)} \sim NT(\mu^{(j)}, \sigma^{(j)}, l^{(j)}, u^{(j)})$$

With $\mu^{(j)}$ and $\sigma^{(j)}$ representing the mean and standard deviation of parameter j at the group level. $l^{(j)}$ and $u^{(j)}$ represent the lower and upper boundary. The lower and upper boundaries of the truncated normal are `-Inf` and `Inf` per default. This can be altered by passing the arguments `lower` and `upper`.

For a group-level mean parameter, $\mu^{(j)}$, the prior is also a truncated normal distributions:

$$\mu^{(j)} \sim NT(M^{(j)}, SD^{(j)}, l^{(j)}, u^{(j)})$$

With $M^{(j)}$ specified by the `mean` argument or the currently set model parameters. $SD^{(j)}$ is, per default, equal to $M^{(j)}$. This can be changed by passing the `sd` argument.

For a group-level standard deviation parameter, $\sigma^{(j)}$, the prior is a gamma distribution:

$$\sigma^{(j)} \sim \Gamma(shape^{(j)}, rate^{(j)})$$

With $shape^{(j)}$ and $rate^{(j)}$ being `1` by default. This can be changed by passing the arguments `shape` and `rate`.

Specifying Prior Settings/Arguments

Argument specification for `mean`, `sd`, `lower`, `upper`, `shape` and `rate` is conceptually identical to specifying `lower/upper` for the classical optimization approach (see the subsection above and the examples below).

Value

- If fitting a single individual: either a `drift_dm` object with fitted parameters and additional fit information (for the classical optimization framework) or an object of type `mcmc_dm` (for the Bayesian framework)
- If fitting multiple individuals separately: a `fits_ids_dm` object or a list of `mcmc_dm` objects, containing all the individual model fits.
- If fitting aggregated data: a `fits_agg_dm` object containing the model itself and the raw data.
- If fitting multiple individuals hierarchically: an object of type `mcmc_dm`.

Note

`estimate_dm` dispatches to underlying estimation routines that are not exported:

- Classical optimization of one individual via `estimate_classical()`

- Classical optimization of multiple individuals via `estimate_classical_wrapper()`
- Bayesian estimation via `estimate_bayesian()`.
- Aggregated fitting is handled within `estimate_dm()` in combination with `estimate_classical()`

When fitting a model with `optimizer = "DEoptim"`, the corresponding minimization routine always runs for 200 iterations by default, irrespective of whether a minimum has already been reached (see DEoptim::DEoptim.control). Therefore, with default optimization settings, `estimate_dm()` returns the convergence flag `NA` for `optimizer = "DEoptim"`, because the termination of the routine does not necessarily indicate convergence. However, this is typically not an issue, as 200 iterations are generally sufficient for the algorithm to find the global minimum. If users explicitly define convergence criteria via the `control` argument of `estimate_dm()` (which is passed on to DEoptim::DEoptim.control), valid convergence messages and flags are returned.

See Also

`estimate_classical()`, `estimate_bayesian()`, `estimate_classical_wrapper()`, `get_parameters_smart()`

Examples

```
##########
# Note: The following examples were trimmed for speed to ensure they run
# within seconds. They do not always provide realistic settings.
##########

####
# Setup

# get a model for the examples (DMC with just two free parameters)
model <- dmc_dm(
  instr = "
   b <!>
   non_dec <!>
   sd_non_dec <!>
   tau <!>
   alpha <!>
   "
)

# get some data (the first two participants in the data set of Ulrich et al.)
data <- ulrich_flanker_data[ulrich_flanker_data$ID %in% 1:2, ]


####
# Fit a single individual (using unbounded Nelder-Mead)
fit <- estimate_dm(
  drift_dm_obj = model,
  obs_data = data[data$ID == 1, ],
  optimizer = "Nelder-Mead"
)
print(fit)
```

```
####
# Fit a single individual (using DEoptim)
l_u <- get_lower_upper(model)
set.seed(2)
fit <- estimate_dm(
  drift_dm_obj = model,
  obs_data = data[data$ID == 1, ],
  optimizer = "DEoptim",
  lower = l_u$lower, upper = l_u$upper,
  control = list(itermax = 5) # use default itermax in practice!
)
print(fit)


####
# Fit multiple individuals (separately; using bounded Nelder-Mead)
l_u <- get_lower_upper(model)
fit <- estimate_dm(
  drift_dm_obj = model,
  obs_data = data, # contains the data for two individuals
  optimizer = "nmkb",
  lower = l_u$lower, upper = l_u$upper,
)
print(fit)
coef(fit)


###
# Fit to aggregated data (using unbounded Nelder-Mead)
fit <- estimate_dm(
  drift_dm_obj = model,
  obs_data = data, # contains data for two individuals
  optimizer = "Nelder-Mead",
  approach = "agg_c"
)
print(fit)
coef(fit)


###
# EXPERIMENTAL
# Fit a single individual (using DE-MCMC; Bayesian; custom priors)
fit <- estimate_dm(
  drift_dm_obj = model,
  obs_data = data[data$ID == 1, ],
  approach = "sep_b",
  burn_in = 1, # higher in practice (e.g., 500)
  samples = 1, # higher in practice (e.g., 1000)
  n_chains = 5, # higher in practice (e.g., 40)
  mean = c(muc = 3, A = 0.9),
```

```
  sd = c(muc = 2, A = 0.8),
)
print(fit)
coef(fit)


###
# EXPERIMENTAL
# Fit multiple individuals (using DE-MCMC; hierarchical Bayesian)
fit <- estimate_dm(
  drift_dm_obj = model,
  approach = "hier_b",
  obs_data = data, # contains data for two individuals
  burn_in = 1, # higher in practice (e.g., 500)
  samples = 1, # higher in practice (e.g., 1000)
  n_chains = 5, # higher in practice (e.g., 40)
  n_cores = 1, # higher in practice (depending on your machine and data set)
)
print(fit)
coef(fit)
```

`estimate_model` *Estimate the Parameters of a drift_dm Model*

Description

[Deprecated] This function was deprecated in dRiftDM version v.0.3.0, please use the more general `estimate_dm()` function.

Old documentation: Find the 'best' parameter settings by fitting a drift_dm models' predicted probability density functions (PDFs) to the observed data stored within the respective object. The fitting procedure is done by minimizing the negative log-likelihood of the model.

Users have three options:

- Estimate the parameters via Differential Evolution (Default)
- Estimate the parameters via (bounded) Nelder-Mead
- Use Differential Evolution followed by Nelder-Mead.

See also `vignette("dRiftDM", "dRiftDM")`

Usage

```
estimate_model(
  drift_dm_obj,
  lower,
  upper,
  verbose = 0,
  use_de_optim = TRUE,
```

```
    use_nmkb = FALSE,
    seed = NULL,
    de_n_cores = 1,
    de_control = list(reltol = 1e-08, steptol = 50, itermax = 200, trace = FALSE),
    nmkb_control = list(tol = 1e-06)
  )
```

Arguments

`drift_dm_obj`	an object inheriting from drift_dm
`lower, upper`	numeric vectors or lists, specifying the lower and upper bounds on each parameter to be optimized (see Details).
`verbose`	numeric, indicating the amount of information displayed. If 0, no information is displayed (default). If 1, basic information about the start of Differential Evolution or Nelder-Mead and the final estimation result is given. If 2, each evaluation of the log-likelihood function is shown. Note that `verbose` is independent of the information displayed by DEoptim::DEoptim.
`use_de_optim`	logical, indicating whether Differential Evolution via DEoptim::DEoptim should be used. Default is `TRUE`
`use_nmkb`	logical, indicating whether Nelder-Mead via dfoptim::nmkb should be used. Default is `FALSE`.
`seed`	a single numeric, providing a seed for the Differential Evolution algorithm
`de_n_cores`	a single numeric, indicating the number of cores to use. Run `parallel::detectCores()` to see how many cores are available on your machine. Note that it is generally not recommended to use all of your cores as this will drastically slow down your machine for any additional task.
`de_control, nmkb_control`	lists of additional control parameters passed to DEoptim::DEoptim and dfoptim::nmkb.

Details

Specifying lower/upper:

the function `estimate_model` provides a flexible way of specifying the search space; identical to specifying the parameter simulation space in simulate_data.drift_dm.

Users have three options to specify the simulation space:

- Plain numeric vectors (not very much recommended). In this case, `lower/upper` must be sorted in accordance with the parameters in the `flex_prms_obj` object that vary for at least one condition (call `print(drift_dm_obj)` and have a look at the `Parameter Settings` output)
- Named numeric vectors. In this case `lower/upper` have to provide labels in accordance with the parameters that are considered "free" at least once across conditions.
- The most flexible way is when `lower/upper` are lists. In this case, the list requires an entry called "default_values" which specifies the named or plain numeric vectors as above. If the list only contains this entry, then the behavior is as if `lower/upper` were already numeric

vectors. However, the `lower/upper` lists can also provide entries labeled as specific conditions, which contain named (!) numeric vectors with parameter labels. This will modify the value for the upper/lower parameter space with respect to the specified parameters in the respective condition.

Details on Nelder-Mead and Differential Evolution:

If both `use_de_optim` and `use_nmkb` are `TRUE`, then Nelder-Mead follows Differential Evolution. Note that Nelder-Mead requires a set of starting parameters for which either the parameter values of `drift_dm_obj` or the estimated parameter values by Differential Evolution are used.

Default settings will lead DEoptim::DEoptim to stop if the algorithm is unable to reduce the negative log-likelihood by a factor of `reltol * (abs(val) + reltol)`after `steptol = 50` steps, with `reltol = 1e-8` (or if the default itermax of 200 steps is reached). Similarly, dfoptim::nmkb will stop if the absolute difference of the log-likelihood between successive iterations is below `tol = 1e-6`.See DEoptim::DEoptim.control and the details of dfoptim::nmkb for further information.

Value

the updated `drift_dm_obj` (with the estimated parameter values, log-likelihood, and probability density functions of the first passage time)

See Also

estimate_model_ids

`estimate_model_ids` *Fit Multiple Individuals and Save Results*

Description

[Deprecated] This function was deprecated in dRiftDM version 0.3.0. Please use the more general `estimate_dm()` instead. NOTE: dRiftDM now supports multiple ways of estimating a model. To ensure a more consistent function interface, individual fits are no longer saved to disk when fitting multiple participants. Instead, `estimate_dm()` directly returns an object of type `fits_ids_dm`, which users can save manually if desired.

Old documentation: Provides a wrapper around estimate_model to fit multiple individuals. Each individual will be stored in a folder. This folder will also contain a file `drift_dm_fit_info.rds`, containing the main arguments of the function call. One call to this function is considered a "fit procedure". Fit procedures can be loaded via load_fits_ids.

Usage

```
estimate_model_ids(
  drift_dm_obj,
  obs_data_ids,
  lower,
  upper,
  fit_procedure_name,
```

```
  fit_path,
  fit_dir = "drift_dm_fits",
  folder_name = fit_procedure_name,
  seed = NULL,
  force_refit = FALSE,
  progress = 2,
  start_vals = NULL,
  ...
)
```

Arguments

`drift_dm_obj`	an object inheriting from drift_dm that will be estimated for each individual in `obs_data_ids`.
`obs_data_ids`	data.frame, see obs_data. An additional column `ID` necessary, to identify a single individual.
`lower`, `upper`	numeric vectors or lists, providing the parameter space, see estimate_model.
`fit_procedure_name`	character, providing a name of the fitting procedure. This name will be stored in `drift_dm_fit_info.rds` to identify the fitting procedure, see also load_fits_ids.
`fit_path`	character, a path, pointing to the location where all fits shall be stored (i.e., `fit_dir` will be created in this location). From the user perspective, the path will likely be identical to the current working directory.
`fit_dir`	character, a directory where (multiple) fitting procedures can be stored. If the directory does not exist yet, it will be created via `base::create.dir(fit_dir, recursive = TRUE)` in the location provided by `fit_path`. Default is `"drift_dm_fits"`.
`folder_name`	character, a folder name for storing all the individual model fits. This variable should just state the name, and should not be a path. Per default `folder_name` is identical to `fit_procedure_name`.
`seed`	numeric, a seed to make the fitting procedure reproducable (only relevant for differential evolution, see estimate_model). Default is `NULL` which means no seed.
`force_refit`	logical, if `TRUE` each individual of a fitting routine will be fitted once more. Default is `FALSE`.
`progress`	numerical, indicating if and how progress shall be displayed. If 0, no progress is shown. If 1, the currently fitted individual is printed out. If 2, a progressbar is shown. Default is 2.
`start_vals`	optional data.frame, providing values to be set before calling estimate_model. Can be used to control the starting values for each individual when calling Nelder-Mead. Note that this will only have an effect if DEoptim is not used (i.e., when setting `use_de_optim = FALSE`; see estimate_model). The data.frame must provide a column `ID` whose entries match the `ID` column in `obs_data_ids`, as well as a column for each parameter of the model matching with `coef(drift_dm_obj, select_unique = TRUE)`.
`...`	additional arguments passed down to estimate_model.

Details

Examples and more information can also be found in `vignette("dRiftDM", "dRiftDM")`.

When developing the fitting routine we had three levels of files/folders in mind:

- In a directory/folder named `fit_dir` multiple fitting routines can be stored (default is "drift_dm_fits")
- Each fitting routine has its own folder with a name as given by `folder_name` (e.g., "ulrich_flanker", "ulrich_simon", ...)
- Within each folder, a file called `drift_dm_fit_info.rds` contains the main information about the function call. That is, the time when last modifying/calling a fitting routine, the `lower` and `upper` parameter boundaries, the `drift_dm_object` that was fitted to each individual, the original data set `obs_data_ids`, and the identifier `fit_procedure_name`. In the same folder each individual has its own `<individual>.rds` file containing the modified `drift_dm_object`.

Value

nothing (`NULL`; invisibly)

See Also

load_fits_ids

`flex_prms<-` *Flex_Prms*

Description

Functions for creating, accessing replacing, or printing a `flex_prms` object. Any object of type `flex_prms` provides a user-friendly way to specify dependencies, parameter values etc. for a model.

Usage

```
flex_prms(object, ...) <- value

## S3 replacement method for class 'drift_dm'
flex_prms(object, ..., eval_model = FALSE) <- value

flex_prms(object, ...)

## S3 method for class 'numeric'
flex_prms(object, ..., conds, instr = NULL, messaging = NULL)

## S3 method for class 'flex_prms'
flex_prms(object, ...)

## S3 method for class 'drift_dm'
```

```
flex_prms(object, ...)

## S3 method for class 'flex_prms'
print(
  x,
  ...,
  round_digits = drift_dm_default_rounding(),
  dependencies = TRUE,
  cust_parameters = TRUE
)
```

Arguments

`object`	an `R` object (see Details)
`...`	additional arguments passed on to the specific method.
`value`	an object of type `flex_prms`.
`eval_model`	logical, indicating if the model should be re-evaluated or not when replacing the `flex_prms` object (see re_evaluate_model).
`conds`	A character vector, giving the names of the model's conditions. values within `conds` will be used when addressing the data and when deriving the model's predictions.
`instr`	optional string with "instructions", see `modify_flex_prms()`.
`messaging`	optional logical, indicates if messages shall be displayed when processing `instr`.
`x`	an object of type `flex_prms`
`round_digits`	integer, controls the number of digits shown when printing out a `flex_prms` object. Default is `3`.
`dependencies`	logical, controlling if a summary of the special dependencies shall be printed.
`cust_parameters`	logical, controlling if a summary of the custom parameters shall be printed.

Details

Objects of type `flex_prms` can be modified using the generic `modify_flex_prms()` function and a corresponding set of "instructions" (see the respective function for more details).

`flex_prms()` is a generic function. If called with a named numeric vector, then this will create an object of type `flex_prms` (requires `conds` to be specified). If called with other data types, gives the respective `flex_prms` object
`flex_prms<-()` is a generic replacement function. Currently this only supports objects of type drift_dm. It will replace/update the model with a new instance of type `flex_prms`.

Value

The specific value returned depends on which method is called

Creating an object of type `flex_prms`:

Can be achieved by calling `flex_prms()` with a named numeric vector, thus when calling the underlying method `flex_prms.numeric` (see the example below). In this case a list with the class label `"flex_prms"` is returned. It containts three entries:

- A nested list `internal_list`. This list specifies the dependencies and restrains enforced upon the parameters across conditions. Integers >= 1 indicate that this parameter will be estimated for a specific condition, and conditions with the same number refer to a single parameter. Integers == 0 indicate thtat this parameter will not be esitmated for a specific condition (i.e., it is considered "fixed"). Expressions will be evaluated at run time and specify special dependencies among parameters.
- A nested list `linear_internal_list`. This list essentially contains the same information as `internal_list`, but the parameters are sorted so that they can be mapped to an integer vector (relevant only in the depths of the package for the minimization routines).
- A numeric matrix `prms_matrix` which contains the currently set values for each parameter across all conditions. Per default, the values of each parameter are set equal across all conditions. Additionally, each parameter is assumed to be restrained as equal across all conditions. The values for all parameters given a condition will be passed to the component functions (see comp_funs).
- (optional) A list of additional parameters `cust_prms` that are derived from the parameters in `prms_matrix`.

Accessing an object of type `flex_prms`:

Users can access/get the `flex_prms` object when calling `flex_prms()` with an object of type drift_dm, `fits_ids_dm` (see `estimate_model_ids()`), or `flex_prms`. In this case, the stored `flex_prms` object is returned.

Replacing an object of type `flex_prms`:

The `flex_prms` object stored within an object of type drift_dm can be replaced by calling the generic `flex_prms<-` replacement function. In this case, the modified drift_dm object is returned.

Printing an object of type `flex_prms`:

The `print.flex_prms()` method invisibly returns the supplied `flex_prms` object.

Note

There is only a replacement function for drift_dm objects. This is because replacing the solver settings after the model has been fitted (i.e., for a `fits_ids_dm` object) doesn't make sense.

See Also

`estimate_model_ids()`, `drift_dm()`, `summary.flex_prms()`, `modify_flex_prms()`

Examples

```
# Create a flex_prms object -----------------------------------------------
conds <- c("one", "two")
prms <- c(muc = 3, b = 0.5)
one_instr <- "muc ~ one + two"
flex_prms_obj <- flex_prms(
```

```
  prms,
  conds = conds,
  instr = one_instr
)
print(flex_prms_obj)


# Access a flex_prms object of a model -----------------------------------
my_model <- ratcliff_dm() # the Ratcliff DDM comes with dRiftDM
print(flex_prms(my_model))


# Replace the flex_prms object of a model ---------------------------------
# create a new flex_prms object
conds <- c("one", "two")
prms <- c(muc = 3, b = 0.6, non_dec = 0.3)
new_flex_prms_obj <- flex_prms(
  prms,
  conds = conds
)

flex_prms(my_model) <- new_flex_prms_obj

# acess the new flex_prms object
print(flex_prms(my_model))


# Control the print method ------------------------------------------------
dmc_model <- dmc_dm() # another, more complex, model; comes with dRiftDM
print(flex_prms(dmc_model), round_digits = 1, cust_parameters = FALSE)
```

`get_example_fits` *Auxiliary Function to load a* `fits_ids_dm`*,* `fits_agg_dm`*, or* `mcmc_dm` *object*

Description

The function is merely helper functions to create an object of type `fits_ids_dm`, `fits_agg_dm`, or `mcmc_dm`. It is used for example code.

Usage

```
get_example_fits(class, hierarchical = FALSE)
```

Arguments

`class`	a string of either `"fits_ids_dm"`, `"fits_agg_dm"`, or `"mcmc_dm"` (can be abbreviated)
`hierarchical`	a logical, relevant when `class = "mcmc_dm"`. If `TRUE`, an object from a hierarchical fit is returned. If `FALSE`, an object from an individual fit is returned.

Details

For `"fits_ids_dm"`, the returned object comprises DMC (see `dmc_dm()`) fitted to three participants of the `ulrich_flanker_data`.

For `"fits_agg_dm"`, the returned object comprises the Ratcliff model (see `ratcliff_dm()`) fitted to synthetic data of three participants.

For `"mcmc_dm"` and `hierarchical = FALSE`, the returned object comprises the Ratcliff model (see `ratcliff_dm()`) fitted to synthetic data of one participant.

For `"mcmc_dm"` and `hierarchical = TRUE`, the returned object comprises the Ratcliff model (see `ratcliff_dm()`) fitted to synthetic data of ten participants.

Value

An object of type `fits_ids_dm`, `fits_agg_dm`, or `mcmc_dm`, mimicking a result from calling `estimate_dm()`.

Examples

```
get_example_fits(class = "fits_agg")
```

`get_lower_upper` *Get Default Parameter Ranges for a Model*

Description

`get_lower_upper()` returns suggested default values for parameter bounds of a `drift_dm` model. The function inspects the model's component functions (e.g., drift, boundary, non-decision time, start) and provides heuristic defaults for some of the pre-built components. Only parameters that are currently considered *free* in the model are returned.

Usage

```
get_lower_upper(object, ...)

## S3 method for class 'drift_dm'
get_lower_upper(object, ..., warn = TRUE)
```

Arguments

`object`	a drift_dm model.
`...`	additional arguments passed forward to the respective method.
`warn`	a single logical, if `TRUE` issue a warning listing components and parameters where no defaults could be provided.

Details

Supported components include: `mu_constant`, `mu_dmc`, `mu_ssp`, `b_constant`, `x_uniform`, `x_beta`, `nt_constant`, `nt_uniform`, `nt_truncated_normal`. For some defaults we use the model's discretization (`dt`, `dx`) to ensure sensible minima.

If a component is not recognized (or refers to currently unsupported components), no defaults are provided for that component. When `warn = TRUE`, a single warning lists components without defaults and any free parameters that remain unmatched. In this case, the user has to add the missing parameter ranges before attempting to fit the model.

The default ranges are **heuristics** intended to provide a reasonable starting point for new users. They are not guaranteed to be appropriate for every model or data set. Always review and, if needed, adjust the returned values as needed.

Value

a list with two named numeric vectors:

- `lower` — suggested lower bounds for free parameters
- `upper` — suggested upper bounds for free parameters

Examples

```
# get a model for the example
model <- dmc_dm(obs_data = dmc_synth_data)

# get the parameter ranges
lu <- get_lower_upper(model)
lu$lower
lu$upper

# then continue to estimate
# estimate_dm(model, lower = lu$lower, upper = lu$upper, optimizer = "nmkb")
```

`hist.coefs_dm` *Plot Parameter Distribution(s)*

Description

This function creates a histogram for each parameter in a `coefs_dm` object, resulting from a call to coef.fits_ids_dm.

Usage

```
## S3 method for class 'coefs_dm'
hist(
  x,
  ...,
```

```
  conds = NULL,
  col = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "value",
  ylab = NULL,
  bundle_plots = TRUE
)
```

Arguments

`x`	an object of class `coefs_dm` (see coef.fits_ids_dm)
`...`	additional graphical arguments passed to `graphics::hist()`. Not supported are the `plot` and `probability` arguments (the latter can be controlled via the supported `freq` argument). For further plotting arguments, see also `set_default_arguments()`.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.
`col`	character vector, specifying colors for each condition, if conditions are present.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`ylim`	a numeric vector of length 2, specifying the y-axis limits.
`xlab`, `ylab`	character strings for the x- and y-axis labels.
`bundle_plots`	logical, indicating whether to display separate panels in a single plot layout (`FALSE`), or to plot them separately (`TRUE`).

Details

The `hist.coefs_dm` function is designed for visualizing parameter distributions.

If multiple conditions are present, it overlays histograms for each condition with adjustable transparency.

When `bundle_plots` is set to `TRUE`, histograms for each parameter are displayed in a grid layout within a single graphics device.

This function has some customization options, but they are limited. If you want to have a highly customized histogram, it is best to create it on your own using R's `graphics::hist()` function (see the examples below).

Value

Nothing (`NULL`; invisibly)

Examples

```
# get an auxiliary fit procedure result (see the function load_fits_ids)
all_fits <- get_example_fits("fits_ids")
coefs <- coef(all_fits)
print(coefs)
hist(coefs, bundle_plots = FALSE) # calls hist.coefs_dm method of dRiftDM
```

```
# how to fall back to R's hist() function for heavy customization
coefs <- unpack_obj(coefs) # provides the plain data.frame
hist(coefs$muc, main = expression(mu[c])) # calls graphics::hist()
```

`load_fits_ids` *Load Estimates of a Fit Procedure*

Description

[Deprecated] This function was deprecated in dRiftDM version 0.3.0, because dRiftDM no longer saves model fits to disk when fitting multiple participants. When estimating multiple individuals with the new function `estimate_dm()`, an object of type `fits_ids_dm` is returned directly.

Usage

```
load_fits_ids(
  path = "drift_dm_fits",
  fit_procedure_name = "",
  detailed_info = FALSE,
  check_data = TRUE,
  progress = 2
)
```

Arguments

`path`	character, a path pointing to a folder or directory containing the individual model fits.
`fit_procedure_name`	character, an optional name that identifies the fit procedure that should be loaded
`detailed_info`	logical, controls the amount of information displayed in case multiple fit procedures were found and the user is prompted to explicitly choose one
`check_data`	logical, should the data be checked before passing them back? This checks the observed data and the properties of the model. Default is `TRUE`
`progress`	numerical, indicating if and how progress shall be depicted. If 0, no progress is shown. If 1, basic infos about the checking progress is shown. If 2, multiple progressbars are shown. Default is 2.

Details

Old documentation: This function loads the results of a fit procedure where a model was fitted to multiple individuals (see estimate_model_ids). It is also the function that creates an object of type `fits_ids_dm`.

with respect to the logic outlined in the details of estimate_model_ids on the organization of fit procedures, `path` could either point to a directory with (potentially) multiple fit routines or to a specific folder with the individual fits. In either case the intended location is recursively searched for files named `drift_dm_fit_info.rds`.

If the fit procedure was uniquely located, either because only one fit routine was found in the intended location or because only one `drift_dm_fit_info.rds` contains the optional identifier specified in `fit_procedure_name`, then all individual model fits including the information `fit_procedure_name` are loaded and returned.

In case multiple fit procedures are identified, the user is prompted with a utils::menu, listing information about the possible candidates. The intended fit procedure can then interactively be chosen by the user. The amount of displayed information is controlled via `detailed_info`.

The `print()` method for objects of type `fits_ids_dm` prints out basic information about the fit procedure name, the fitted model, time of (last) call, and the number of individual data sets.

Value

For `load_fits_ids()`, an object of type `fits_ids_dm`, which essentially is a list with two entries:

- `drift_dm_fit_info`, containing a list of the main arguments when estimate_model_ids was originally called, including a time-stamp.
- `all_fits`, containing a list of all the modified/fitted `drift_dm` objects. The list's entry are named according to the individuals' identifier (i.e., `ID`).

For `print.fits_ids_dm()`, the supplied `fit_ids_dm` object `x` (invisible return).

See Also

`estimate_model_ids()`

`logLik.drift_dm` *Extract Log-Likelihood for a drift_dm Object*

Description

This method extracts the log-likelihood for a `drift_dm` object if possible.

Usage

```
## S3 method for class 'drift_dm'
logLik(object, ...)
```

Arguments

`object`	a drift_dm object containing observed data
`...`	additional arguments

Value

A `logLik` object containing the log-likelihood value for the drift_dm object. This value has attributes for the number of observations (`nobs`) and the number of model parameters (`df`).

Returns `NULL` if the log-likelihood is not available (e.g., when the model has no observed data attached).

Examples

```
# get a pre-built model and a data set for demonstration purpose
# (when creating the model, set the discretization to reasonable values)
a_model <- dmc_dm()
obs_data(a_model) <- dmc_synth_data
logLik(a_model)
```

`logLik.fits_ids_dm` *Extract Model Statistics for fits_ids_dm Object*

Description

These methods are wrappers to extract specific model fit statistics (log-likelihood, AIC, BIC) for each model in a `fits_ids_dm` object.

Usage

```
## S3 method for class 'fits_ids_dm'
logLik(object, ...)

## S3 method for class 'fits_ids_dm'
AIC(object, ..., k = 2)

## S3 method for class 'fits_ids_dm'
BIC(object, ...)
```

Arguments

`object`	a `fits_ids_dm` object (see estimate_model_ids)
`...`	additional arguments (currently not used)
`k`	numeric; penalty parameter for the AIC calculation. Defaults to 2 (standard AIC).

Details

Each function retrieves the relevant statistics by calling calc_stats with `type = "fit_stats"` and selects the columns for `ID` and the required statistic.

Value

An object of type `fit_stats` containing the respective statistic in one column (named `Log_Like`, `AIC`, or `BIC`) and a corresponding `ID` column. If any of the statistics can't be calculated, the function returns `NULL`.

See Also

`stats::AIC()`, `stats::BIC()`, logLik.drift_dm

Examples

```
# get an auxiliary fits_ids object for demonstration purpose;
# such an object results from calling load_fits_ids
all_fits <- get_example_fits("fits_ids_dm")

# AICs
AIC(all_fits)

# BICs
BIC(all_fits)

# Log-Likelihoods
logLik(all_fits)

# All unique and free parameters
coef(all_fits)

# Or all parameters across all conditions
coef(all_fits, select_unique = FALSE)
```

`modify_flex_prms` *Set Instructions to a flex_prms object*

Description

Functions to carry out the "instructions" on how to modify a flex_prms object, specified as a string.

Usage

```
modify_flex_prms(object, instr, ...)

## S3 method for class 'drift_dm'
modify_flex_prms(object, instr, ..., eval_model = FALSE)

## S3 method for class 'flex_prms'
modify_flex_prms(object, instr, ..., messaging = NULL)
```

Arguments

`object`	an object of type `drift_dm` or `flex_prms`.
`instr`	a character string, specifying a set of instructions (see Details).
`...`	further arguments passed forward to the respective method.
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating modifying the flex_prms object (see re_evaluate_model). Default is `FALSE`.
`messaging`	logical, indicating if messages shall be displayed or not. Can happen, for example, when setting a parameter value for a specific condition, although the parameter values are assumed to be the identical across conditions.

Details

`modify_flex_prms` is a generic function. The default methods pass forward a set of "instructions" to modify the (underlying) flex_prms object.

These instructions are inspired by the model syntax of the `lavaan` package. Note that specifying multiple instructions is possible, but each instruction has to be defined in its own line. Comments with '#' are possible, also line continuations are possible, if the last symbol is a "+","-", "*", "/", "(", or "[". The following instructions are implemented:

The **"vary"** instruction:

- Looks something like "a ~ foo + bar"
- This means that the parameter 'a' is allowed to vary independently for the conditions 'foo' and 'bar'
- Thus, when estimating the model, the user will have independent values for 'a' in conditions 'foo' and 'bar'

The **"restrain"** instruction:

- Looks something like "a ~! foo + bar "
- This means that the parameter 'a' is assumed to be identical for the conditions 'foo' and 'bar'
- Thus, when estimating the model, the user will have only a single value for 'a' in conditions 'foo' and 'bar'

The **"set"** instruction:

- Users may not always estimate a model directly but rather explore the model behavior. In this case setting the value of a parameter is necessary.
- The corresponding instruction looks something like "a ~ foo => 0.3"
- This will set the value for 'a' in condition 'foo' to the value of 0.3

The **"fix"** instruction:

- Oftentimes, certain parameters of a model are considered "fixed", so that they don't vary while the remaining parameters are estimated. An example would be the shape parameter 'a' of DMC (see dmc_dm).
- The corresponding instruction looks something like "a <!> foo + bar"
- Usually, users want to call the "set" instruction prior or after the "fix" instruction, to set the corresponding parameter to a certain value.

The **"special dependency"** instruction:

- Sometimes, users want to allow one parameter to depend on another. For instance, in DMC (see dmc_dm), the parameter A is positive in the congruent condition, but negative in the incongruent condition. Thus, parameters may have a 'special depencency' which can be expressed as an equation.
- To define a special dependency, users can use the operation "==". The parameter that should have the dependency is on the left-hand side, while the mathematical relationship to other parameters is defined on the right-hand side.

- This then looks something like "a ~ foo == -(a ~ bar)".
- This means that the parameter a in condition foo will always be -1 * the parameter a in condition bar. Thus, if a in condition bar has the value 5, then a in condition foo will be -5.
- The expression on the right-side can refer to any arbitrary mathematical relation.
- Important: Make sure that each 'parameter ~ condition' combination on the right-hand side of the equation are set in brackets.
- Another example: Parameter a in condition foo should be the mean of the parameter b in conditions bar and baz; this would be the instruction "a ~ foo == 0.5*(b ~ bar) + 0.5*(b ~ baz)"

The **"additional/custom parameter combination"** instruction:

- Sometimes, users may want to combine multiple parameters to summarize a certain property of the model. For example, in DMC (see dmc_dm), the shape and rate parameter jointly determine the peak latency.
- To avoid having to calculate this manually, users can define "custom" parameter combinations using the ":=" operation:
- An exemplary instruction might look like this: "peak_l := (a - 1) * tau"
- Expressions and values that provide calculations for those parameters are stored in a separate list `cust_prms`.

Value

For drift_dm objects, the updated drift_dm object.

For flex_prms, the updated flex_prms object.

See Also

`flex_prms()`

Examples

```
# Example 1: Modify a flex_prms object  directly ---------------------------
# create an auxiliary flex_prms object
a_flex_prms_obj <- flex_prms(
  c(muc = 3, b = 0.5, non_dec = 0.3),
  conds = c("foo", "bar")
)

# then carry out some "instructions". Here (arbitrary operations):
# 1.) Consider b as fixed
# 2.) Let muc vary independently for the conditions foo and bar
# 3.) Set non_dec in condition bar to be half as large as non_dec in
#     condition bar
instr <-
  "b <!>
 muc ~
 non_dec ~ bar == (non_dec ~ foo) / 2
"
```

```
modify_flex_prms(object = a_flex_prms_obj, instr = instr)


# Example 2: Modify a flex_prms object stored inside a drift_dm object -----
a_model <- ratcliff_dm() # get a model for demonstration purpose
modify_flex_prms(object = a_model, instr = "muc ~ => 4")
```

`nobs.drift_dm` *Get the Number of Observations for a drift_dm Object*

Description

This method retrieves the total number of observations in the `obs_data` list of a `drift_dm` object.

Usage

```
## S3 method for class 'drift_dm'
nobs(object, ...)
```

Arguments

`object`	a drift_dm object, which potentially contains the observed data in `object$obs_data`.
`...`	additional arguments

Details

The function iterates over each element in `object$obs_data`, counts the entries in each nested component, and returns the cumulative sum as the total observation count.

It was written to provide an `nobs` method for calculating the log-likelihood (logLik), AIC (stats::AIC), and BIC (stats::BIC) statistics for objects of type drift_dm.

Value

An integer representing the total number of observations across all conditions in `object$obs_data`. If `obs_data` doesn't exist, the function returns 0

Examples

```
# get a pre-built model and data set for demonstration purpose
a_model <- dmc_dm()
obs_data(a_model) <- dmc_synth_data

# then get the number of observations by accessing the model
nobs(a_model)

# same number of observations as in the original data set
nrow(dmc_synth_data)
```

`obs_data<-` *The Observed Data*

Description

Functions to get or set the "observed data" of an object.

Usage

```
obs_data(object, ...) <- value

## S3 replacement method for class 'drift_dm'
obs_data(object, ..., eval_model = FALSE) <- value

obs_data(object, ...)

## S3 method for class 'drift_dm'
obs_data(object, ..., messaging = TRUE)

## S3 method for class 'fits_ids_dm'
obs_data(object, ...)

## S3 method for class 'fits_agg_dm'
obs_data(object, ...)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments passed down to the specific method.
`value`	a data.frame which provides three columns: (1) `RT` for the response times, (2) a column for boundary coding according to the model's `b_coding()`, (3) `Cond` for specifying the conditions.
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the solver settings (see re_evaluate_model). Default is `FALSE`.
`messaging`	logical, indicating if messages shall be displayed or not.

Details

`obs_data()` is a generic accessor function, and `obs_data<-()` is a generic replacement function. The default methods get and set the "observed data". Their behavior, however, may be a bit unexpected.

In drift_dm objects, the observed data are not stored as a data.frame. Instead, any supplied observed data set is disassembled into RTs for the upper and lower boundary and with respect to the different conditions (ensures more speed and easier programming in the depths of the package). Yet, `obs_data()` returns a `data.frame` for drift_dm objects. This implies that `obs_data()` does not merely access the observed data, but re-assembles it. Consequently, a returned data.frame for the observed data is likely sorted differently than the data.frame that was originally set to the model via `obs_data<-()`. Also, when the originally supplied data set provided more conditions than the model, the unused conditions will not be part of the returned data.frame.

For `fits_ids_dm` (see load_fits_ids), the observed data are stored as a data.frame in the general fit procedure info. This is the data.frame that `obs_data()` will return. Thus, the returned data.frame will match with the data.frame that was initially supplied to estimate_model_ids, although with unused conditions being dropped.

In theory, it is possible to update parts of the "observed data". However, because `obs_data()` returns a re-assembled data.frame for drift_dm objects, great care has to be taken with respect to the ordering of the argument `value`. A message is displayed to remind the user that the returned data.frame may be sorted differently than expected.

Value

For `obs_data()` a data.frame of the observed data. The method `obs_data.drift_dm()` per default displays a message to remind the user that the returned data.frame is likely sorted differently than expected.

For `obs_data<-()` the updated drift_dm object.

Note

There is only a replacement function for drift_dm objects. This is because replacing the observed data after the model has been fitted (i.e., for a `fits_ids_dm` object) doesn't make sense.

See Also

`drift_dm()`

Examples

```
# Set some data to a model --------------------------------------------------
my_model <- dmc_dm() # DMC is pre-built and directly available
# synthetic data suitable for DMC; comes with dRiftDM
some_data <- dmc_synth_data
obs_data(my_model) <- some_data

# Extract data from a model -------------------------------------------------
head(obs_data(my_model))

# Important: ----------------------------------------------------------------
# The returned data.frame may be sorted differently than the one initially
# supplied.
some_data <- some_data[sample(1:nrow(some_data)), ] #' # shuffle the data set
obs_data(my_model) <- some_data
all.equal(obs_data(my_model), some_data)
# so don't do obs_data(my_model)["Cond"] <- ...

# Addition: -----------------------------------------------------------------
# accessor method also available for fits_ids_dm objects
# (see estimate_model_ids)
```

```
# get an exemplary fits_ids_dm object
fits <- get_example_fits("fits_ids_dm")
head(obs_data(fits))
```

`pdfs` *Access the Probability Density Functions of a Model*

Description

Functions to obtain the probability density functions (PDFs) of a model. These PDFs represent the convolution of the first-passage-time (decision time) with the non-decision time.

Usage

```
pdfs(object, ...)

## S3 method for class 'drift_dm'
pdfs(object, ...)

## S3 method for class 'fits_agg_dm'
pdfs(object, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments passed down to the specific method.

Details

If the model has not been evaluated, `re_evaluate_model()` is called before returning the PDFs.

Value

A list with the entries:

- `pdfs`, contains another named list with entries corresponding to the conditions of the model (see `conds()`). Each of these elements is another named list, containing the entries `pdf_u` and `pdf_l`, which are numeric vectors for the PDFs of the upper and lower boundary, respectively.
- `t_vec`, containing a numeric vector of the time domain.

See Also

`drift_dm()`, `re_evaluate_model()`, `conds()`

Examples

```
# get a pre-built model for demonstration purpose
a_model <- dmc_dm()
str(pdfs(a_model))
```

`plot.cafs` *Plot Conditional Accuracy Functions (CAFs)*

Description

Visualizes conditional accuracy functions (CAFs) for observed and/or predicted data. This is useful for assessing model fit or exploring response patterns across conditions or participants.

Usage

```
## S3 method for class 'cafs'
plot(
  x,
  ...,
  id = NULL,
  conds = NULL,
  col = NULL,
  xlim = NULL,
  ylim = c(0, 1),
  xlab = "Bins",
  ylab = NULL,
  interval_obs = TRUE,
  interval_pred = TRUE
)
```

Arguments

`x`	an object of type = `"cafs"`, typically returned by `calc_stats()`.
`...`	additional graphical arguments passed to plotting functions. See `set_default_arguments()` for the full list of supported options.
`id`	a numeric or character, specifying the ID of a single participant to plot. If `length(id)` > `1`, `plot.cafs()` is called recursively for each entry. Each `id` must match an entry in the `ID` column of `x`.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.
`col`	a character vector specifying colors for each condition. If a single color is provided, it is repeated for all conditions.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`ylim`	a numeric vector of length 2, specifying the y-axis limits.

`xlab`, `ylab`	character strings for the x- and y-axis labels.
`interval_obs`, `interval_pred`	logicals; if `TRUE` and `x` contains a column named `Estimate`, error bars for observed data and shaded contours for predicted data are drawn, respectively.

Details

If `x` contains multiple `ID`s and no specific `id` is provided, the function aggregates across participants before plotting.

Observed CAFs are shown as points, and predicted CAFs as lines. When `interval` = `TRUE` and the input includes interval estimates (i.e., the column `Estimate` exists), the plot includes error bars for observed data and shaded contours for model predictions.

Colors, symbols, and line styles can be customized via `...`.

Value

Returns `NULL` invisibly. The function is called for its side effect of generating a plot.

Examples

```
# Example 1: Model predictions only ---------------------------------------
a_model <- dmc_dm()
cafs <- calc_stats(a_model, type = "cafs")
plot(cafs)
plot(cafs, col = c("green", "red"), ylim = c(0.5, 1))

# Example 2: Observed and predicted data ----------------------------------
obs_data(a_model) <- dmc_synth_data
cafs <- calc_stats(a_model, type = "cafs")
plot(cafs)

# Example 3: Observed data only -------------------------------------------
cafs <- calc_stats(dmc_synth_data, type = "cafs")
plot(cafs)

# Example 4: Observed data with interval ----------------------------------
cafs <- calc_stats(dmc_synth_data, type = "cafs", resample = TRUE)
plot(cafs)
```

`plot.delta_funs` *Plot Delta Functions*

Description

Visualizes delta functions for observed and/or predicted data. This is useful for assessing model fit or exploring the model behavior

Usage

```
## S3 method for class 'delta_funs'
plot(
  x,
  ...,
  id = NULL,
  conds = NULL,
  dv = NULL,
  col = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "RT [s]",
  ylab = expression(Delta),
  interval_obs = TRUE,
  interval_pred = TRUE
)
```

Arguments

`x`	an object of `type = "delta_funs"`, typically returned by `calc_stats()`.
`...`	additional graphical arguments passed to plotting functions. See `set_default_arguments()` for the full list of supported options.
`id`	a numeric or character, specifying the ID of a single participant to plot. If `length(id) > 1`, `plot.cafs()` is called recursively for each entry. Each `id` must match an entry in the `ID` column of `x`.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.
`dv`	a character vector indicating the delta function(s) to plot. Defaults to all columns in `x` that begin with `"Delta_"`.
`col`	a character vector specifying colors for each condition. If a single color is provided, it is repeated for all conditions.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`ylim`	a numeric vector of length 2, specifying the y-axis limits.
`xlab`, `ylab`	character strings for the x- and y-axis labels.
`interval_obs`, `interval_pred`	logicals; if `TRUE` and `x` contains a column named `Estimate`, error bars for observed data and shaded contours for predicted data are drawn, respectively.

Details

If `x` contains multiple `ID`s and no specific `id` is provided, the function aggregates across participants before plotting.

Observed delta functions are shown as points, and predicted delta functions as lines. When `interval_obs = TRUE` or `interval_pred = TRUE` and the input includes interval estimates (i.e., the column `Estimate` exists), the plot includes error bars for observed data and shaded contours for model predictions.

Colors, symbols, and line styles can be customized via `...`.

Value

Returns `NULL` invisibly. The function is called for its side effect of generating a plot.

Examples

```
# Example 1: Model predictions only ---------------------------------------
a_model <- dmc_dm()
deltas <- calc_stats(
  a_model,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)
plot(deltas)
plot(deltas, col = "black", lty = 2, xlim = c(0.2, 0.65))

# Example 2: Observed and predicted data ----------------------------------
obs_data(a_model) <- dmc_synth_data
deltas <- calc_stats(
  a_model,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)
plot(deltas)

# Example 3: Observed data only -------------------------------------------
deltas <- calc_stats(
  dmc_synth_data,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)
plot(deltas)

# Example 4: Observed data with intervals ---------------------------------
deltas <- calc_stats(
  dmc_synth_data,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp",
  resample = TRUE
)
plot(deltas)
```

`plot.densities` *Plot Distributions of Predicted and Observed Data*

Description

Visualizes observed and/or predicted response time distributions. Useful for assessing model fit or exploring model behavior.

Usage

```
## S3 method for class 'densities'
plot(
  x,
  ...,
  id = NULL,
  conds = NULL,
  col = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "RT [s]",
  ylab = "Density",
  obs_stats = "hist",
  interval_obs = FALSE,
  interval_pred = TRUE
)
```

Arguments

`x`	an object of type = `"densities"`, typically returned by `calc_stats()`.
`...`	additional graphical arguments passed to plotting functions. See `set_default_arguments()` for the full list of supported options.
`id`	a numeric or character, specifying the ID of a single participant to plot. If `length(id)` > 1, `plot.cafs()` is called recursively for each entry. Each `id` must match an entry in the `ID` column of `x`.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.
`col`	a character vector specifying colors for each condition. If a single color is provided, it is repeated for all conditions.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`ylim`	a numeric vector of length 2, specifying the y-axis limits.
`xlab`, `ylab`	character strings for the x- and y-axis labels.
`obs_stats`	a character vector specifying which observed statistics to plot. Options include `"hist"` for histograms and `"kde"` for kernel density estimates. Defaults to `"hist"`.
`interval_obs`, `interval_pred`	logicals; if `TRUE` and `x` contains a column named `Estimate`, error bars for observed data and shaded contours for predicted data are drawn, respectively.

Details

If `x` contains multiple `IDs` and no specific `id` is provided, the function aggregates across participants before plotting. You can provide a vector of `ids` to produce separate plots for each participant.

Observed densities are shown as histograms (default: gray shaded areas), or KDE lines (default: black, dotted). Predicted densities are shown as lines (default: colorized). Distributions associated with the upper boundary are shown with values > 0 (i.e., the upper part of the plot), distributions associated with the lower boundary are shown with values < 0 (i.e., the lower part of the plot).

Axis limits, colors, and styling options can be customized via `...`. If interval information is provided (i.e., the column `Estimate` exists in `x`), error bars or shading will be added, depending on the type of statistic.

A legend is only displayed if there is predicted data.

Value

Returns `NULL` invisibly. The function is called for its side effect of generating a plot.

Examples

```
# Example 1: Predicted densities only -------------------------------------
a_model <- dmc_dm()
dens <- calc_stats(a_model, type = "densities")
plot(dens, xlim = c(0, 1))
plot(dens, xlim = c(0, 1), conds = "comp")


# Example 2: Observed and predicted densities -----------------------------
obs_data(a_model) <- dmc_synth_data
dens <- calc_stats(a_model, type = "densities")
plot(dens, xlim = c(0, 1), conds = "comp", col = "blue")

# Example 3: Observed densities only --------------------------------------
dens <- calc_stats(dmc_synth_data, type = "densities")
plot(dens, conds = "comp", obs.hist.col = "green", alpha = 1)

# Example 4: With interval estimates --------------------------------------
dens <- calc_stats(dmc_synth_data, type = "densities", resample = TRUE)
plot(dens, interval_obs = TRUE, conds = "comp")
```

`plot.drift_dm` *Plot Components of a Drift Diffusion Model*

Description

This function generates plots for all components of a drift diffusion model (DDM), such as drift rate, boundary, and starting condition. Each component is plotted against the time or evidence space, allowing for visual inspection of the model's behavior across different conditions.

Usage

```
## S3 method for class 'drift_dm'
plot(x, ..., conds = NULL, col = NULL, xlim = NULL, bundle_plots = TRUE)
```

Arguments

`x`	an object of class drift_dm
`...`	additional graphical arguments passed to plotting functions. See `set_default_arguments()` for the full list of supported options.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.
`col`	a character vector specifying colors for each condition. If a single color is provided, it is repeated for all conditions.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`bundle_plots`	logical, indicating whether to display separate panels in a single plot layout (`FALSE`), or to plot them separately (`TRUE`).

Details

The `plot.drift_dm` function provides an overview of key DDM components, which include:

- `mu_fun`: Drift rate over time.
- `mu_int_fun`: Integrated drift rate over time (if required by the specified `solver` of the model).
- `x_fun`: Starting condition as a density across evidence values.
- `b_fun`: Boundary values over time.
- `dt_b_fun`: Derivative of the boundary function over time.
- `nt_fun`: Non-decision time as a density over time.

Value

Nothing (`NULL`; invisibly)

Examples

```
# plot the component functions of the Ratcliff DDM
plot(ratcliff_dm())
plot(ratcliff_dm(var_non_dec = TRUE))
# Note: the variability in the drift rate for the Ratcliff DDM
# is not plotted! This is because it is not actually stored as a component
# function.

# plot the component functions of the DMC model
plot(dmc_dm(), col = c("green", "red"))
```

`plot.mcmc_dm` *Plot MCMC Results and Diagnostics for* `mcmc_dm` *Objects*

Description

Visualize MCMC results and diagnostics for `mcmc_dm` objects. The function `plot.mcmc()` is typically called when users supply an `mcmc_dm` object returned by `estimate_dm()` to the generic `base::plot()` function.

Usage

```
## S3 method for class 'mcmc_dm'
plot(x, ..., id = NULL, what = "trace", bundle_plots = TRUE)
```

Arguments

`x`	an object of class `mcmc_dm`, as returned by `estimate_dm()`.
`...`	optional arguments passed on to the underlying plotting functions `plot_mcmc_trace()`, `plot_mcmc_marginal()`, and `plot_mcmc_auto()`. See the respective documentations for a list of optional arguments and the examples below. Probably the most relevant optional argument is `which_prms` that allows users to select a specific subset of parameters.
`id`	optional character vector, specifying the id(s) of participants to plot. If `length(id)` > 1, `plot.mcmc_dm()` is called recursively, iterating over each entry in `id`. Each `id` must match with the relevant dimension names of the used chains array stored in `x`.
`what`	a character string indicating the type of plot to produce. Must be either `"trace"`, `"density"`, or `"auto"`. See the Details below. Default is `"trace"`.
`bundle_plots`	logical, indicating whether to display separate panels in a single plot layout (`FALSE`), or to plot them separately (`TRUE`).

Details

This function provides diagnostic and summary visualizations of MCMC samples. It handles results from both hierarchical and non-hierarchical MCMC runs:

- If `id` is provided, the plot refers to the requested participant, with MCMC results extracted at the individual level.
- If `id` is omitted, plots refer to group-level parameters (i.e., the hyperparameters)

The following plot types are supported:

- Trace plots (`what = "trace"`): These plots show sampled parameter values across MCMC iterations for each chain. They are primarily used to inspect convergence and mixing behavior. Ideally, all chains should appear well-mixed (i.e., they should overlap and sample in a similar range). Lack of convergence is indicated by chains that remain in separate regions or exhibit trends over time.

- Density plots (`what = "density"`): These plots display smoothed marginal posterior distributions for each parameter, collapsed over chains and iterations. They are useful for understanding the central tendency, variance, and shape of the posterior distributions.
- Autocorrelation plots (`what = "auto"`): These plots display the autocorrelation at different lags, averaged across chains. They are useful to judge how quickly the chains produced independent samples.

Value

Returns `NULL` invisibly.

See Also

`plot_mcmc_trace()`, `plot_mcmc_marginal()`, `plot_mcmc_auto()`

Examples

```
# get an examplary `mcmc_dm` object
chains_obj <- get_example_fits("mcmc")
plot(chains_obj)
plot(chains_obj, what = "density")
plot(chains_obj, what = "density", which_prm = "b", bundle_plots = FALSE)
```

`plot.quantiles` *Plot Response Time Quantiles*

Description

Visualizes response time quantiles for observed and/or predicted data across experimental conditions. This is useful for assessing model fit or exploring response patterns across conditions or participants.

Usage

```
## S3 method for class 'quantiles'
plot(
  x,
  ...,
  id = NULL,
  conds = NULL,
  dv = NULL,
  col = NULL,
  xlim = NULL,
  ylim = c(0, 1),
  xlab = "RT [s]",
  ylab = "F(RT)",
  interval_obs = TRUE,
```

```
  interval_pred = TRUE
)
```

Arguments

`x`	an object of `type = "quantiles"`, typically returned by `calc_stats()`.
`...`	additional graphical arguments passed to plotting functions. See `set_default_arguments()` for the full list of supported options.
`id`	a numeric or character, specifying the ID of a single participant to plot. If `length(id) > 1`, `plot.cafs()` is called recursively for each entry. Each `id` must match an entry in the `ID` column of `x`.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.
`dv`	a character string indicating the dependent variable to plot. Defaults to the quantiles for the upper boundary.
`col`	a character vector specifying colors for each condition. If a single color is provided, it is repeated for all conditions.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`ylim`	a numeric vector of length 2, specifying the y-axis limits.
`xlab`, `ylab`	character strings for the x- and y-axis labels.
`interval_obs`, `interval_pred`	logicals; if `TRUE` and `x` contains a column named `Estimate`, error bars for observed data and shaded contours for predicted data are drawn, respectively.

Details

If `x` contains multiple `ID`s and no specific `id` is provided, the function aggregates across participants before plotting.

Observed quantiles are shown as points, and predicted quantiles as lines. When `interval = TRUE` and the input includes interval estimates (i.e., the column `Estimate` exists), the plot includes error bars for observed data and shaded contours for model predictions.

Colors, symbols, and line styles can be customized via `...`.

Value

Returns `NULL` invisibly. The function is called for its side effect of generating a plot.

Examples

```
# Example 1: Model predictions only ---------------------------------------
a_model <- dmc_dm()
quantiles <- calc_stats(a_model, type = "quantiles")
plot(quantiles)
plot(quantiles, col = c("green", "red"), xlim = c(0.2, 0.6))

# Example 2: Observed and predicted data ----------------------------------
obs_data(a_model) <- dmc_synth_data
```

```
quantiles <- calc_stats(a_model, type = "quantiles")
plot(quantiles)

# Example 3: Observed data only -------------------------------------------
quantiles <- calc_stats(dmc_synth_data, type = "quantiles")
plot(quantiles)

# Example 4: Observed data with interval ----------------------------------
cafs <- calc_stats(dmc_synth_data, type = "quantiles", resample = TRUE)
plot(cafs)
```

`plot.stats_dm_list` *Plot Multiple Statistics*

Description

This function iterates over a list of statistics data, resulting from a call to `calc_stats()`, and subsequently plots each statistic. It allows for a simple arrangement of multiple plots on a single graphics device.

Usage

```
## S3 method for class 'stats_dm_list'
plot(x, ..., mfrow = NULL)
```

Arguments

`x`	an object of type `stats_dm_list`, which is essentially a list of multiple statistics, resulting from a call to `calc_stats()`.
`...`	additional arguments passed to the plot function for each individual `stats_dm` object in `x`.
`mfrow`	an optional numeric vector of length 2, specifying the number of rows and columns for arranging multiple panels in a single plot (e.g., `c(1, 3)`). Plots are provided sequentially if `NULL` (default), using the current graphics layout of a user.

Details

The `plot.stats_dm_list()` function "merely" iterates over each entry of `x` and calls the respective `plot()` method. If `dRiftDM` doesn't provide a `plot()` method for an object stored in `x`, the respective entry is skipped and a message is displayed.

When users want more control over each plot, it is best to call the `plot()` function separately for each statistic in the list (e.g., `plot(x$cafs)`; `plot(x$quantiles)`)

Value

Nothing (`NULL`; invisibly)

See Also

`plot.cafs()`, `plot.quantiles()`, `plot.delta_funs()`, `plot.densities()`

Examples

```
# get a list of statistics for demonstration purpose
all_fits <- get_example_fits("fits_ids_dm")
stats <- calc_stats(all_fits, type = c("cafs", "quantiles"))

# then call the plot function.
plot(stats, mfrow = c(1, 2))
```

`plot.traces_dm_list` *Plot Traces of a Drift Diffusion Model*

Description

Creates a plot of simulated traces (i.e., simulated evidence accumulation processes) from a drift diffusion model. Such plots are useful for exploring and testing model behavior.

Usage

```
## S3 method for class 'traces_dm_list'
plot(
  x,
  ...,
  conds = NULL,
  col = NULL,
  col_b = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "Time",
  ylab = "Evidence"
)

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

Arguments

`x`	an object of type `traces_dm_list` or `traces_dm`, containing the traces to be plotted, resulting from a call to `simulate_traces()`.
`...`	additional graphical arguments passed to plotting functions. See `set_default_arguments()` for the full list of supported options.
`conds`	a character vector specifying the conditions to plot. Defaults to all available conditions.

`col`	a character vector specifying colors for each condition. If a single color is provided, it is repeated for all conditions.
`col_b`	a character vector, specifying the color of the boundary for each condition. If a single color is provided, it is repeated for all conditions. Default is `"black"`.
`xlim`	a numeric vector of length 2, specifying the x-axis limits.
`ylim`	a numeric vector of length 2, specifying the y-axis limits.
`xlab`, `ylab`	character strings for the x- and y-axis labels.

Details

`plot.traces_dm_list()` iterates over all conditions and plots the traces. It includes a legend with condition labels.

`plot.traces_dm` plots a single set of traces. Because `simulate_traces()` returns an object of type `traces_dm_list` per default, users will likely call `plot.traces_dm_list()` in most cases; and not `plot.traces_dm`. `plot.traces_dm` is only relevant if users explicitly extract and provide an object of type `traces_dm`.

The function automatically generates the upper and lower boundaries based on the information stored within `x`.

Value

`NULL` invisibly

See Also

simulate_traces

Examples

```
# get a couple of traces for demonstration purpose
a_model <- dmc_dm()
some_traces <- simulate_traces(a_model, k = 3)

# Plots for traces_dm_list objects ----------------------------------------
# basic plot
plot(some_traces)

# a slightly more beautiful plot :)
plot(some_traces,
  col = c("green", "red"),
  xlim = c(0, 0.35),
  xlab = "Time [s]",
  ylab = bquote(Realizations ~ of ~ X[t]),
  legend_pos = "bottomright"
)

# Plots for traces_dm objects ---------------------------------------------
# we can also extract a single set of traces and plot them
one_set_traces <- some_traces$comp
```

```
plot(one_set_traces)

# modifications to the plot work in the same way
plot(one_set_traces,
  col = "green",
  xlim = c(0, 0.35),
  xlab = "Time [s]",
  ylab = bquote(Realizations ~ of ~ X[t]),
  legend = "just comp"
)
```

`print.summary.fits_agg_dm`

Summary and Printing for `fits_agg_dm` *Objects*

Description

Methods for summarizing and printing objects of the class `fits_agg_dm`, which contain model fits based on aggregated data across participants.

Usage

```
## S3 method for class 'summary.fits_agg_dm'
print(x, ..., just_header = FALSE, round_digits = drift_dm_default_rounding())

## S3 method for class 'fits_agg_dm'
summary(object, ..., select_unique = FALSE)
```

Arguments

`x`	an object of class `summary.fits_agg_dm`.
`...`	additional arguments (currently unused).
`just_header`	logical, if `TRUE` only print the header information without details. Default is `FALSE`.
`round_digits`	an integer, specifying the number of decimal places for rounding in the printed summary. Default is 3.
`object`	an object of class `fits_agg_dm`, typically generated by a call to estimate_dm.
`select_unique`	logical, passed to `coef.drift_dm()`.

Details

The `summary.fits_agg_dm` function creates a structured summary of a `fits_agg_dm` object, containing:

- **summary_drift_dm_obj**: A list with information about the underlying drift diffusion model (as returned by `summary.drift_dm()`).

- **prms**: Parameter estimates obtained from the model fit. This is equivalent to a call to `coef.drift_dm()` on the stored model object.
- **obs_data**: A list providing the number of individual participants and the average number of trials per condition across participants.

The `print.summary.fits_agg_dm` function formats and prints the above summary in a human-readable form.

Value

`summary.fits_agg_dm()` returns a list of class `summary.fits_agg_dm` (see Details for its structure).

`print.summary.fits_agg_dm()` returns the input object invisibly.

See Also

summary.drift_dm, coef.drift_dm

Examples

```
# Load example fit object
fits_agg <- get_example_fits("fits_agg")
sum_obj <- summary(fits_agg)
print(sum_obj, round_digits = 2)
```

---

print.summary.fits_ids_dm

Summary and Printing for fits_ids_dm Objects

Description

Methods for summarizing and printing objects of the class `fits_ids_dm`, which contain multiple fits across individuals.

Usage

```
## S3 method for class 'summary.fits_ids_dm'
print(x, ..., just_header = FALSE, round_digits = drift_dm_default_rounding())

## S3 method for class 'fits_ids_dm'
summary(object, ..., select_unique = FALSE)
```

Arguments

`x`	an object of class `summary.fits_ids_dm`.
`...`	additional arguments (currently unused).
`just_header`	logical, if `TRUE` only print the header information without details. Default is `FALSE`.
`round_digits`	an integer, specifying the number of decimal places for rounding in the printed summary. Default is 3.
`object`	an object of class `fits_ids_dm`, generated by a call to load_fits_ids.
`select_unique`	logical, passed to `coef.drift_dm()`.

Details

The `summary.fits_ids_dm` function creates a summary object. The contents of this summary object depends on whether the user supplies a `fits_ids_dm` object that was created with `estimate_dm()` or the deprecated function `load_fits_ids()`.

- In the first case, the object contains:
- **summary_drift_dm_obj**: A list with information about the underlying drift diffusion model (as returned by `summary.drift_dm()`).
- **prms**: All parameter values across all conditions (essentially a call to coef() with the argument select_unique = FALSE).
- **stats**: A named list of matrices for each condition, including mean and standard error for each parameter.
- **obs_data**: A list providing the number of individual participants and the average number of trials per condition across participants.
- **optimizer**: A string of the optimizer that was used
- **conv_info**: A list providing a summary of the convergance and messages for all IDs
- In the second case, the object contains:
- **lower** and **upper**: Lower and upper bounds of the search space.
- **model_type**: Description of the model type, based on class information.
- **prms**: All parameter values across all conditions (essentially a call to coef() with the argument select_unique = FALSE).
- **stats**: A named list of matrices for each condition, including mean and standard error for each parameter.
- **N**: The number of individuals.

The `print.summary.fits_ids_dm` function displays the summary object in a formatted manner.

Value

`summary.fits_ids_dm()` returns a list of class `summary.fits_ids_dm` (see the Details section summarizing each entry of this list).

`print.summary.fits_ids_dm()` returns invisibly the `summary.fits_ids_dm` object.

Examples

```
# get an auxiliary object of type fits_ids_dm for demonstration purpose
all_fits <- get_example_fits("fits_ids_dm")
sum_obj <- summary(all_fits)
print(sum_obj, round_digits = 2)
```

`prms_solve<-` *The Parameters for Deriving Model Predictions*

Description

Functions to get or set the "solver settings" of an object. This includes the diffusion constant and the discretization of the time and evidence space.

Usage

```
prms_solve(object, ...) <- value

## S3 replacement method for class 'drift_dm'
prms_solve(object, ..., eval_model = FALSE) <- value

prms_solve(object, ...)

## S3 method for class 'drift_dm'
prms_solve(object, ...)

## S3 method for class 'fits_ids_dm'
prms_solve(object, ...)

## S3 method for class 'fits_agg_dm'
prms_solve(object, ...)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments (i.e., `eval_model`).
`value`	a named numeric vector providing new values for the `prms_solve` vector (see `drift_dm()`).
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the solver settings (see re_evaluate_model). Default is `FALSE`.

Details

`prms_solve()` is a generic accessor function, and `prms_solve<-()` is a generic replacement function. The default methods get and set the "solver settings".

It is possible to update parts of the "solver setttings" (i.e., parts of the underlying `prms_solve` vector). However, modifying `"nx"` or `"nt"` is not allowed! Any attempts to modify the respective entries will silently fail (no explicit error/warning etc. is thrown).

Value

For `prms_solve()` the vector `prms_solve` (see `drift_dm()`).

For `prms_solve<-()` the updated drift_dm object.

Note

There is only a replacement function for drift_dm objects. This is because replacing the solver settings after the model has been fitted (e.g., for a `fits_ids_dm` object) doesn't make sense.

See Also

`drift_dm()`

Examples

```
# get some default model to demonstrate the prms_solve() functions
my_model <- ratcliff_dm()
# show the discretization and scaling of the model
prms_solve(my_model)
# partially modify these settings
prms_solve(my_model)[c("dx", "dt")] <- c(0.005)
prms_solve(my_model)

# accessor method also available for fits_ids_dm objects
# (see estimate_model_ids)
# get an exemplary fits_ids_dm object
fits <- get_example_fits("fits_ids_dm")
prms_solve(fits)
```

`ratcliff_dm` *Create a Basic Diffusion Model*

Description

This function creates a drift_dm model that corresponds to the basic Ratcliff Diffusion Model

Usage

```
ratcliff_dm(
  var_non_dec = FALSE,
  var_start = FALSE,
  var_drift = FALSE,
  instr = NULL,
  obs_data = NULL,
  sigma = 1,
  t_max = 3,
  dt = 0.0075,
  dx = 0.02,
  solver = "kfe",
  b_coding = NULL
)
```

Arguments

`var_non_dec`, `var_start`, `var_drift`	logical, indicating whether the model should have a variable non-decision time , starting point (uniform), or drift rate (normally-distributed). (see also `nt_uniform` and `x_uniform` in component_shelf)
`instr`	optional string with "instructions", see `modify_flex_prms()`.
`obs_data`	data.frame, an optional data.frame with the observed data. See obs_data.
`sigma`, `t_max`, `dt`, `dx`	numeric, providing the settings for the diffusion constant and discretization (see drift_dm)
`solver`	character, specifying the solver.
`b_coding`	list, an optional list with the boundary encoding (see b_coding)

Details

The classical Ratcliff Diffusion Model is a diffusion model with a constant drift rate `muc` and a constant boundary `b`. If `var_non_dec` = `FALSE`, a constant non-decision time `non_dec` is assumed, otherwise a uniform non-decision time with mean `non_dec` and range `range_non_dec`. If `var_start` = `FALSE`, a constant starting point centered between the boundaries is assumed (i.e., a dirac delta over 0), otherwise a uniform starting point with mean 0 and range `range_start`. If `var_drift` = `FALSE`, a constant drift rate is assumed, otherwise a normally distributed drift rate with mean `mu_c` and standard deviation `sd_muc` (can be computationally intensive). Important: Variable drift rate is only possible with dRiftDM's `mu_constant` function. No custom drift rate is yet possible in this case.

Value

An object of type `drift_dm` (parent class) and `ratcliff_dm` (child class), created by the function `drift_dm()`.

References

Ratcliff R (1978). "A theory of memory retrieval." *Psychological Review*, **85**(2), 59–108. doi:10.1037/0033295X.85.2.59.

See Also

`component_shelf()`, `drift_dm()`

Examples

```
# the model with default settings
my_model <- ratcliff_dm()

# the model with a variable non-decision time and with finer space
# discretization
my_model <- ratcliff_dm(var_non_dec = TRUE, dx = .01)
```

`ratcliff_synth_data` *A synthetic data set with one condition*

Description

This dataset was simulated by using the classical Ratcliff diffusion model (see `ratcliff_dm()`).

Usage

```
ratcliff_synth_data
```

Format

A data frame with 300 rows and 3 columns:

RT Response Times

Error Error Coding (Error Response = 1; Correct Response = 0)

Cond Condition ('null')

`re_evaluate_model` *Re-evaluate the model*

Description

Updates the PDFs of a model. If observed data is available (e.g., via the obs_data entry or the `stats_agg` entry; depending on the cost_function, see also `drift_dm()`), the cost_value is also updated.

Usage

```
re_evaluate_model(drift_dm_obj, eval_model = TRUE)
```

Arguments

`drift_dm_obj`	an object of type drift_dm
`eval_model`	logical, indicating if the model should be evaluated or not. If `FALSE`, PDFs and the value of the cost function are deleted from the model. Default is `True`.

Details

More in-depth information about the mathematical details for deriving the PDFs can be found in Richter et al. (2023)

Value

Returns the passed `drift_dm_obj` object, after (re-)calculating the PDFs and (if observed data is set) the cost_value.

- the PDFs an be addressed via `drift_dm_obj$pdfs`
- the cost_value can be addressed via `drift_dm_obj$cost_value`

Note that if `re_evaluate` model is called before observed data was set, the function silently updates the `pdfs`, but not `cost_value`.

See Also

`drift_dm()`

Examples

```
# choose a pre-built model (e.g., the Ratcliff model)
# and set the discretization as needed
my_model <- ratcliff_dm()

# then calculate the model's predicted PDF
my_model <- re_evaluate_model(my_model)
str(my_model$pdfs) # show the structure of the attached pdfs
```

```
# if you want the cost_function, make sure some data is attached to the
# model (see also the documentation of obs_data())
obs_data(my_model) <- ratcliff_synth_data # this data set comes with dRiftDM
my_model <- re_evaluate_model(my_model)
str(my_model$pdfs)
print(my_model$cost_value)
```

`simulate_data` *Simulate Synthetic Responses*

Description

This function simulates data based on the provided model. To this end, random samples from the predicted PDFs are drawn via approximate inverse CDF sampling.

Usage

```
simulate_data(object, ...)

## S3 method for class 'drift_dm'
simulate_data(
  object,
  ...,
  n,
  conds = NULL,
  k = 1,
  lower = NULL,
  upper = NULL,
  df_prms = NULL,
  seed = NULL,
  progress = 1
)
```

Arguments

`object`	an object inheriting from drift_dm.
`...`	further arguments passed on to other functions, i.e., `simulate_values()` and `simulate_one_data_set()`. This allows users to control the distribution from which original parameter values are drawn (if `k` > 0) and the number of decimal places that the simulated RTs should have. If users want to use a different distribution than uniform for `simulate_values()`, they must provide the additional arguments (e.g., `means` and `sds`) in a format like `lower`/`upper`.
`n`	numeric, the number of trials per condition to draw. If a single numeric, then each condition will have `n` trials. Can be a (named) numeric vector with the same length as there are conditions to allow a different number of trials per condition.

`conds`	character vector, specifying the conditions to sample from. Default `NULL` is equivalent to `conds(object)`.
`k`	numeric larger than 0, indicating how many data sets shall be simulated. If > 1, users must specify `lower/upper`.
`lower, upper`	vectors or a list, specifying the simulation space for each parameter of the model (see Details). Only relevant for `k > 1`
`df_prms`	an optional data.frame providing the parameters that should be used for simulating the data. `df_prms` must provide column names matching with (`coef(object, select_unique = TRUE)`), plus a column `ID` that will identify each simulated data set.
`seed`	a single numeric, an optional seed for reproducible sampling
`progress`	an integer, indicating if information about the progress should be displayed. 0 -> no information, 1 -> a progress bar. Default is 1. Only effective when `k > 1`.

Details

`simulate_data` is a generic function for simulating data based on approximate inverse CDF sampling. CDFs are derived from the model's PDFs and data is drawn by mapping samples from a uniform distribution (in $[0, 1]$) to the values of the CDF. Note that sampled response times will correspond to the values of the time space (i.e., they will correspond to `seq(0, t_max, dt)`, see drift_dm).

For `drift_dm` objects, the behavior of `simulate_data` depends on `k`. If `k = 1` and no `lower/upper` or `df_prms` arguments are supplied, then the parameters currently set to the model are used to generate the synthetic data. If `k > 1`, then `k` parameter combinations are either randomly drawn via simulate_values or gathered from the provided data.frame `df_prms`, and then data is simulated for each parameter combination.

When specifying `lower/upper`, parameter combinations are simulated via simulate_values. This comes in handy for simple parameter recovery exercises. If `df_prms` is specified, then the parameter combinations from this data.frame is used. Note that the column names in `df_prms` must match with the (unique) parameter combinations of the model (see `print(coef(object))`)

Details on how to specify `lower/upper`.:

When users want to simulate data with `k > 1` and `lower/upper`, then parameter values have to be drawn. One great aspect about the flex_prms object within each drift_dm model, is that users can easily allow certain parameters to vary freely across conditions. Consequently, the actual number of parameters varies with the settings of the flex_prms object. In many cases, however, the simulation space for a parameter is the same across conditions. For instance, in a model, the parameter "mu" may vary across the conditions "easy", "medium", or "hard", but the lower/upper limits are the same across conditions. To avoid that users always have to re-specify the simulation space via the `lower/upper` arguments, the `lower` and `upper` arguments refer to the parameter labels, and `dRiftDM` figures out how to map these to all parameters that vary across conditions.

Here is an example: Assume you have the model with parameters "A" and "B", and the conditions "foo" and "bar". Now assume that "A" is allowed to vary for "foo" and "bar". Thus, there are actually three parameters; "A~foo", "A~bar", and "B". `dRiftDM`, however, can help with this. If we provide `lower = c(A = 1, B = 2)`, `upper = c(A = 3, B = 4)`, `simulate_data` checks the model, and creates the vectors `temp_lower = c(1,1,2)` and `temp_upper = c(3,3,4)` as a basis to simulate the parameters.

Users have three options to specify the simulation space:

- Plain numeric vectors (not very much recommended). In this case, `lower/upper` must be sorted in accordance with the free parameters in the `flex_prms_obj` object (call `print(<model>)` and have a look at the `Parameter Settings` output)
- Named numeric vectors. In this case `lower/upper` have to provide labels in accordance with the parameters that are considered "free" at least once across conditions.
- The most flexible way is when `lower/upper` are lists. In this case, the list requires an entry called "default_values" which specifies the named or plain numeric vectors as above. If the list only contains this entry, then the behavior is as if `lower/upper` were already numeric vectors. However, the `lower/upper` lists can also provide entries labeled as specific conditions, which contain named (!) numeric vectors with parameter labels. This will modify the value for the upper/lower parameter space with respect to the specified parameters in the respective condition.

Value

The return value depends on whether a user specifies `lower/upper` or `df_prms`. If none of these are specified and if `k = 1`, then a data.frame containing the columns `RT`, `Error`, and `Cond` is returned.

If `lower/upper` or `df_prms` are provided, then a list with entries `synth_data` and `prms` is returned. The entry `synth_data` contains a data.frame, with the columns `RT`, `<b_column>`, `Cond`, and `ID` (the name of the second column, `<b_column>`, depends on the b_coding of the model object). The entry `prms` contains a data.frame with an `ID` column and the parameters used for simulating each synthetic data set.

Note

A function for `fits_ids_dm` will be provided in the future.

Examples

```
# Example 1 ----------------------------------------------------------------
# get a pre-built model for demonstration
a_model <- ratcliff_dm()

# define a lower and upper simulation space
lower <- c(1, 0.4, 0.1)
upper <- c(6, 0.9, 0.5)

# now simulate 5 data sets with each 100 trials
data_prms <- simulate_data(a_model,
  n = 100, k = 5, lower = lower,
  upper = upper, seed = 1, progress = 0
)
head(data_prms$synth_data)
head(data_prms$prms)

# Example 2 ----------------------------------------------------------------
# more flexibility when defining lists for lower and upper
# get a pre-built model, and allow muc to vary across conditions
a_model <- dmc_dm(instr = "muc ~ ")
```

```
# define a lower and upper simulation space
# let muc vary between 2 and 6, but in incomp conditions, let it vary
# between 1 and 4
lower <- list(
  default_values = c(
    muc = 2, b = 0.4, non_dec = 0.1,
    sd_non_dec = 0.01, tau = 0.02, A = 0.05,
    alpha = 3
  ),
  incomp = c(muc = 1)
)
upper <- list(
  default_values = c(
    muc = 6, b = 0.9, non_dec = 0.4,
    sd_non_dec = 0.15, tau = 0.15, A = 0.15,
    alpha = 7
  ),
  incomp = c(muc = 4)
)

data_prms <- simulate_data(a_model,
  n = 100, k = 5, lower = lower,
  upper = upper, seed = 1, progress = 0
)
range(data_prms$prms$muc.comp)
range(data_prms$prms$muc.incomp)
```

`simulate_traces` *Simulate Trajectories/Traces of a Model*

Description

Simulates single trajectories/traces of a model (i.e., evidence accumulation processes) using forward Euler.

Might come in handy when exploring the model's behavior or when creating figures (see also plot.traces_dm_list)

Usage

```
simulate_traces(object, k, ...)

## S3 method for class 'drift_dm'
simulate_traces(
  object,
  k,
  ...,
  conds = NULL,
```

```
  add_x = FALSE,
  sigma = NULL,
  seed = NULL,
  unpack = FALSE
)

## S3 method for class 'fits_ids_dm'
simulate_traces(object, k, ...)

## S3 method for class 'fits_agg_dm'
simulate_traces(object, k, ...)

## S3 method for class 'traces_dm_list'
print(x, ..., round_digits = drift_dm_default_rounding(), print_steps = 5)

## S3 method for class 'traces_dm'
print(
  x,
  ...,
  round_digits = drift_dm_default_rounding(),
  print_steps = 5,
  print_k = 4
)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`k`	numeric, the number of traces to simulate per condition. Can be a named numeric vector, to specify different number of traces per condition.
`...`	additional arguments passed forward to the respective method.
`conds`	optional character vector, conditions for which traces shall be simulated. If `NULL`, then traces for all conditions are simulated.
`add_x`	logical, indicating whether traces should contain a variable starting point. If `TRUE`, samples from `x_fun` (see comp_vals) are added to each trace. Default is `FALSE`.
`sigma`	optional numeric, providing a value >= 0 for the diffusion constant "sigma" to temporally override prms_solve. Useful for exploring the model without noise.
`seed`	optional numerical, a seed for reproducible sampling
`unpack`	logical, indicating if the traces shall be "unpacked" (see also unpack_obj and the return value below).
`x`	an object of type `traces_dm_list` or `traces_dm`, resulting from a call to `simulate_traces`.
`round_digits`	integer, indicating the number of decimal places (round) to be used when printing out the traces (default is 3).
`print_steps`	integer, indicating the number of steps to show when printing out traces (default is 5).
`print_k`	integer, indicating how many traces shall be shown when printing out traces (default is 4).

Details

`simulate_traces()` is a generic function, applicable to objects of type drift_dm or `fits_ids_dm` (see load_fits_ids).

For drift_dm objects, `simulate_traces()` performs the simulation on the parameter values currently set (see `coef.drift_dm()`).

For `fits_ids_dm` objects, `simulate_traces()` first extracts the model and all parameter values for all IDs (see `coef.fits_ids_dm()`). Subsequently, simulations are based on the averaged parameter values.

The algorithm for simulating traces is forward euler. See Richter et al. (2023) and Ulrich et al. (2015) (Appendix A) for more information.

Value

`simulate_traces()` returns either an object of type `traces_dm_list`, or directly a list of matrices across conditions, containing the traces (if `unpack = TRUE`). If the model has only one condition (and `unpack = TRUE`), then the matrix of traces for this one condition is directly returned.

The returned list has as many entries as conditions requested. For example, if only one condition is requested via the `conds` argument, then the list is of length 1 (if `unpack = FALSE`). If `conds` is set to `NULL` (default), then the list will have as many entries as conditions specified in the supplied `object` (see also conds). If `unpack = FALSE`, the list contains an additional attribute with the time space.

Each matrix of traces has `k` rows and `nt + 1` columns, stored as an array of size (`k`, `nt + 1`). Note that `nt` is the number of steps in the discretization of time; see drift_dm. If `unpack = FALSE`, the array is of type `traces_dm`. It contains some additional attributes about the time space, the drift rate, the boundary, the added starting values, if starting values were added, the original model class and parameters, the boundary coding, and the solver settings.

The print methods `print.traces_dm_list()` and `print.traces_dm()` each invisibly return the supplied object `x`.

Note

Evidence values with traces beyond the boundary of the model are set to NA before passing them back.

The reason why `simulate_traces` passes back an object of type `traces_dm_list` (instead of simply a list of arrays) is to provide a plot.traces_dm_list and print.traces_dm_list function.

Users can unpack the traces even after calling `simulate_traces()` using `unpack_obj()`.

See Also

`unpack_obj()`, `plot.traces_dm_list()`

Examples

```
# get a pre-built model to demonstrate the function
my_model <- dmc_dm()
some_traces <- simulate_traces(my_model, k = 1, seed = 1)
print(some_traces)
```

```
# a method is also available for fits_ids_dm objects
# (see estimate_model_ids)
# get an exemplary fits_ids_dm object
fits <- get_example_fits("fits_ids_dm")
some_traces <- simulate_traces(fits, k = 1, seed = 1)
print(some_traces)

# we can also print only the traces of one condition
print(some_traces$comp)
```

`simulate_traces_one_cond` *Simulate Traces for One Conditions*

Description

The function simulates traces with forward Euler. It is the backend function to `simulate_traces`.

Usage

```
simulate_traces_one_cond(drift_dm_obj, k, one_cond, add_x, sigma)
```

Arguments

`drift_dm_obj`	a model of type drift_dm
`k`	a single numeric, the number of traces to simulate
`one_cond`	a single character string, specifying which condition shall be simulated
`add_x`	a single logical, indicating if starting values shall be added or not. Sometimes, when visualizing the model, one does not want to have the starting values.
`sigma`	a single numeric, to override the "sigma" in prms_solve

Value

An array of size k times `nt + 1`. The array becomes an object of type `traces_dm`, which allows for easier printing with print.traces_dm. Furthermore, each object has the additional attributes:

- "t_vec" -> the time space from 0 to t_max
- "mu_vals" -> the drift rate values by mu_fun
- "b_vals" -> the boundary values by b_fun
- "samp_x" -> the values of the starting points (which are always added to the traces in the array.
- "add_x" -> boolean, indicating if the starting values were added or not
- "orig_model_class" -> the class label of the original model
- "orig_prms" -> the parameters with which the traces were simulated (for the respective condition)
- "b_coding" -> the boundary coding
- "prms_solve" -> the solver settings with which the traces were simulated

---

`simulate_values` *Simulate Values*

Description

Draw values, most likely model parameters.

Usage

```
simulate_values(
  lower,
  upper,
  k,
  distr = NULL,
  cast_to_data_frame = TRUE,
  add_id_column = "numeric",
  seed = NULL,
  ...
)
```

Arguments

`lower`, `upper`	Numeric vectors, indicating the lower/upper boundary of the drawn values.
`k`	Numeric, the number of values to be drawn for each value pair of lower/upper. If named numeric, the labels are used for the column names of the returned object
`distr`	Character, indicating which distribution to draw from. Currently available are: `"unif"` for a uniform distribution or `"tnorm"` for a truncated normal distribution. `NUll` will lead to `"unif"` (default).
`cast_to_data_frame`	Logical, controls whether the returned object is of type data.frame (TRUE) or matrix (FALSE). Default is TRUE
`add_id_column`	Character, controls whether an ID column should be added. Options are "numeric", "character", or "none". If "numeric" or "character" the column ID provides values from 1 to k of the respective type. If none, no column is added. Note that "character" casts all simulated values to character if the argument `cast_to_data_frame` is set to FALSE.
`seed`	Numeric, optional seed for making the simulation reproducable (see details)
`...`	Further arguments relevant for the distribution to draw from

Details

When drawing from a truncated normal distribution, users must provide values for the arguments `means` and `sds`. These are numeric vectors of the same size as `lower` and `upper`, and indicate the mean and the standard deviation of the normal distributions.

Value

If `cast_to_data_frame` is TRUE, a data.frame with `k` rows and at least `length(lower);length(upper)` columns. Otherwise a matrix with the same number of rows and columns. Columns are labeled either from V1 to Vk or in case `lower` and `upper` are named numeric vectors using the labels of both vectors.

If `add_id_column` is not "none", an ID column is provided of the respective data type.

The data type of the parameters will be numeric, unless `add_id_column` is "character" and `cast_to_data_frame` is FALSE. In this case the returned matrix will be of type character.

Examples

```
# Example 1: Draw from uniform distributions ------------------------------
lower <- c(a = 1, b = 1, c = 1)
upper <- c(a = 3, b = 4, c = 5)
values <- simulate_values(
  lower = lower,
  upper = upper,
  k = 50,
  add_id_column = "none"
)
summary(values)

# Example 2: Draw from truncated normal distributions ---------------------
lower <- c(a = 1, b = 1, c = 1)
upper <- c(a = 3, b = 4, c = 5)
means <- c(a = 2, b = 2.5, c = 3)
sds <- c(a = 0.5, b = 0.5, c = 0.5)
values <- simulate_values(
  lower = lower,
  upper = upper,
  distr = "tnorm",
  k = 5000,
  add_id_column = "none",
  means = means,
  sds = sds
)
quantile(values$a, probs = c(0.025, 0.5, 0.975))
quantile(values$b, probs = c(0.025, 0.5, 0.975))
quantile(values$c, probs = c(0.025, 0.5, 0.975))
```

`solver<-` *The Solver for Deriving Model Predictions*

Description

Functions to get or set the "solver" of an object. The "solver" controls the method for deriving the model's first passage time (i.e., its predicted PDFs).

Usage

```
solver(object, ...) <- value

## S3 replacement method for class 'drift_dm'
solver(object, ..., eval_model = FALSE) <- value

solver(object, ...)

## S3 method for class 'drift_dm'
solver(object, ...)

## S3 method for class 'fits_ids_dm'
solver(object, ...)

## S3 method for class 'fits_agg_dm'
solver(object, ...)
```

Arguments

`object`	an object of type drift_dm, `fits_ids_dm`, or `fits_agg_dm` (see `estimate_dm()`).
`...`	additional arguments (i.e., `eval_model`).
`value`	a single character string, providing the new "solver" (i.e., approach to derive the first passage time; see `drift_dm()`).
`eval_model`	logical, indicating if the model should be re-evaluated or not when updating the solver (see re_evaluate_model). Default is `FALSE`.

Details

`solver()` is a generic accessor function, and `solver<-()` is a generic replacement function. The default methods get and set the "solver".

The "solver" indicates the approach with which the PDFs of a model are calculated. Supported options are "kfe" and "im_zero" (method based on the Kolmogorov-Forward-Equation or on integral equations, respectively). Note that "im_zero" is only supported for models that assume a fixed starting point from 0.

Value

For `solve()` the string `solver` (see `drift_dm()`).

For `solver<-()` the updated drift_dm object.

Note

There is only a replacement function for drift_dm objects. This is because replacing the approach for deriving PDFs after the model has been fitted (i.e., for a `fits_ids_dm` object) doesn't make sense.

See Also

`drift_dm()`

Examples

```
# get some default model to demonstrate the solver() functions
my_model <- ratcliff_dm()
solver(my_model)
# change to the integral approach
solver(my_model) <- "im_zero"
solver(my_model)

# accessor method also available for fits_ids_dm objects
# (see estimate_model_ids)
# get an exemplary fits_ids_dm object
fits <- get_example_fits("fits_ids_dm")
solver(fits)
```

`ssp_dm` *Create the Shrinking Spotlight Model*

Description

This function creates a drift_dm object that corresponds to a simple version of the shrinking spotlight model by White et al. (2011).

Usage

```
ssp_dm(
  var_non_dec = TRUE,
  var_start = FALSE,
  instr = NULL,
  obs_data = NULL,
  sigma = 1,
  t_max = 3,
  dt = 0.005,
  dx = 0.02,
  b_coding = NULL
)
```

Arguments

`var_non_dec`, `var_start`	logical, indicating whether the model should have a variable non-decision time or starting point (see also `nt_uniform` and `x_uniform` in component_shelf
`instr`	optional string with "instructions", see `modify_flex_prms()`.
`obs_data`	data.frame, an optional data.frame with the observed data. See obs_data.

`sigma, t_max, dt, dx`	numeric, providing the settings for the diffusion constant and discretization (see drift_dm)
`b_coding`	list, an optional list with the boundary encoding (see b_coding)

Details

The shrinking spotlight model is a model developed for the flanker task.

It has the following properties (see component_shelf):

- a constant boundary (parameter `b`)
- a constant starting point in between the decision boundaries
- an evidence accumulation process that is driven by an attention spotlight that covers both the flankers and the target. The area that covers the flankers and target is modeled by normal distribution with mean 0:
  - At the beginning of the trial attention is wide-spread, and the width at t=0 is the standard deviation `sd_0`
  - As the trial progresses in time, the attention spotlight narrows, reflected by a linear decline of the standard deviation with rate `r` (to a minimum of 0.001).
  - the attention attributed to both the flankers and the target is scaled by `p` which controls the strength of evidence accumulation
- A non-decision time that follows a truncated normal distribution with mean `non_dec` and standard deviation `sd_non_dec`.
- The model also contains the auxiliary parameter `sign`, which is used to control the influence of the flankers across conditions. It is not really a parameter and should not be estimated!

Per default, the parameter `r` is assumed to be fixed (i.e., is not estimated freely). The model also contains the custom parameter `interf_t`, quantifying the interference time (`sd_0 / r`).

Value

An object of type `drift_dm` (parent class) and `ssp_dm` (child class), created by the function `drift_dm()`.

Note

The parameters of SSP in `dRiftDM` differ in their size from the original publication of White et al. (2011). `dRiftDM` uses symmetrical boundaries around zero and a diffusion constant of 1. In the original publication, SSP was parameterized with boundaries ranging from zero to `a` and a diffusion constant of 0.1.

Thus, in `dRiftDM`, the boundary `b` corresponds to $b = a/2 \cdot 10$. Additionally, `p` in `dRiftDM` is 10 times larger than `p` in the original publication. Finally, `r` is expressed in seconds, and thus `r` is 1000 times larger in `dRiftDM` than in the original publication.

References

White CN, Ratcliff R, Starns JJ (2011). "Diffusion models of the flanker task: Discrete versus gradual attentional selection." *Cognitive psychology*, **63**(4), 210–238. doi:10.1016/j.cogpsych.2011.08.001.

Examples

```
# the model with default settings
my_model <- ssp_dm()

# the model with a finer discretization
my_model <- ssp_dm(dt = .0025, dx = .01)
```

`ssp_synth_data` *A synthetic data set with two conditions*

Description

This dataset was simulated by using the Shrinking Spotlight Model (see `ssp_dm()`) with parameter settings that are typical for a Flanker task.

Usage

```
ssp_synth_data
```

Format

A data frame with 600 rows and 3 columns:

RT Response Times

Error Error Coding (Error Response = 1; Correct Response = 0)

Cond Condition ('comp' and 'incomp')

`summary.coefs_dm` *Summary for* `coefs_dm` *Objects*

Description

Summary and corresponding printing methods for `coefs_dm` objects. These objects result from a call to `coef.fits_ids_dm()` (i.e., when calling `coef()` with an object of type `fits_ids_dm`).

Usage

```
## S3 method for class 'coefs_dm'
summary(object, ..., round_digits = drift_dm_default_rounding())

## S3 method for class 'summary.coefs_dm'
print(x, ..., show_header = TRUE)
```

Arguments

`object`	an object of type `coefs_dm`.
`...`	additional arguments passed forward.
`round_digits`	integer, specifying the number of decimal places for rounding the summary of the underlying data.frame. Default is 3.
`x`	an object of class `summary.coefs_dm`.
`show_header`	logical. If `TRUE`, a header specifying the type of statistic will be displayed.

Details

`summary.coefs_dm()` summarizes `coefs_dm` objects, returning the type, a summary of the underlying data.frame (`summary_dataframe`), and the number of unique IDs (`n_ids`).

Value

For `summary.coefs_dm()` a summary object of class `summary.coefs_dm`.

For `print.summary.coefs_dm()`, the supplied object is returned invisibly.

Examples

```
# get a fits_ids object for demonstration purpose
fits_ids <- get_example_fits("fits_ids_dm")
coefs <- coef(fits_ids)
summary(coefs)
```

`summary.drift_dm` *Summary for* `drift_dm` *objects*

Description

summary and corresponding printing methods for objects of class `drift_dm`, created by a call to `drift_dm()`.

Usage

```
## S3 method for class 'drift_dm'
summary(object, ...)

## S3 method for class 'summary.drift_dm'
print(x, ..., round_digits = drift_dm_default_rounding())
```

Arguments

`object`	an object of class `drift_dm`.
`...`	additional arguments passed forward (currently not used).
`x`	an object of class `summary.drift_dm`.
`round_digits`	integer, specifying the number of decimal places for rounding in the printed summary. Default is 3.

Details

`summary.drift_dm()` constructs a summary list with information about the `drift_dm` object. The returned list has class `summary.drift_dm` and can include the following entries:

- **class**: Class vector of the `drift_dm` object.
- **summary_flex_prms**: Summary of the flex_prms object in the model (see summary.flex_prms).
- **prms_solve**: Parameters used for solving the model (see prms_solve).
- **solver**: Solver used for generating model predictions.
- **b_coding**: Boundary coding for the model (see b_coding).
- **obs_data**: Summary table of observed response time data, if available, by response type (upper/lower boundary). rows correspond to upper first then lower responses; row names are prefixed by the boundary names from `b_coding`. columns (all lower-case) are: `min`, `1st qu.`, `median`, `mean`, `3rd qu.`, `max`, and `n`.
- **cost_function**: Name (or descriptor) of the cost function used during estimation.
- **fit_stats**: Fit statistics, if available. we return a named atomic vector created via `unlist(unpack_obj(calc_stats(...` `type = "fit_stats")))`.
- **estimate_info**: Additional information about the estimation procedure.

`print.summary.drift_dm()` displays this summary in a formatted way.

Value

`summary.drift_dm()` returns a list of class `summary.drift_dm` (see details for the entries).

`print.summary.drift_dm()` returns invisibly the `summary.drift_dm` object.

Examples

```
# get a pre-built model for demonstration
a_model <- dmc_dm()
sum_obj <- summary(a_model)
print(sum_obj, round_digits = 2)

# more information is provided when we add data to the model
obs_data(a_model) <- dmc_synth_data # (data set comes with dRiftDM)
summary(a_model)

# fit indices are added once we evaluate the model
a_model <- re_evaluate_model(a_model)
summary(a_model)
```

`summary.flex_prms` *Summarizing Flex Parameters*

Description

summary method for class "flex_prms".

Usage

```
## S3 method for class 'flex_prms'
summary(object, ...)

## S3 method for class 'summary.flex_prms'
print(
  x,
  ...,
  round_digits = drift_dm_default_rounding(),
  dependencies = TRUE,
  cust_parameters = TRUE
)
```

Arguments

`object`	an object of class `flex_prms`, resulting from a call to flex_prms.
`...`	additional arguments passed forward to the respective method
`x`	an object of class `summary.flex_prms`; a result of a call to `summary.flex_prms()`.
`round_digits`	integer, indicating the number of decimal places (round) to be used (default is 3).
`dependencies`	logical, controlling if a summary of the special dependencies shall be printed (see the "special dependency instruction" in the details of flex_prms)
`cust_parameters`	logical, controlling if a summary of the custom parameters shall be printed (see the "additional/custom parameter instruction" in the details of flex_prms)

Details

The `summary.flex_prms()` function creates a summary object containing:

- **prms_matrix**: All parameter values across all conditions.
- **unique_matrix**: A character matrix, showing how parameters relate across conditions.
- **depend_strings**: Special Dependencies, formatted as a string.
- **cust_prms_matrix**: (if they exist), a matrix containing all custom parameters.

The `print.summary.flex_prms()` function displays the summary object in a formatted manner.

Value

`summary.flex_prms()` returns a list of class `summary.flex_prms` (see the Details section summarizing each entry of this list).

`print.summary.flex_prms()` returns invisibly the `summary.flex_prms` object.

Examples

```
# create a flex_prms object
flex_obj <- flex_prms(c(a = 1, b = 2), conds = c("foo", "bar"))

sum_obj <- summary(flex_obj)
print(sum_obj)

# the print function for the summary object is identical to the print
# function of the flex_prms object
print(flex_obj)
```

`summary.mcmc_dm` *Summary for* `mcmc_dm` *Objects*

Description

Summary and corresponding print methods for objects of the class `mcmc_dm`, resulting from a call to `estimate_bayesian()`. `mcmc_dm` objects contain MCMC samples for Bayesian parameter estimation of `drift_dm()` objects. The summary includes basic parameter statistics, quantiles, Gelman-Rubin diagnostics, and effective sample sizes.

Usage

```
## S3 method for class 'mcmc_dm'
summary(object, ..., id = NULL)

## S3 method for class 'summary.mcmc_dm'
print(
  x,
  ...,
  round_digits = drift_dm_default_rounding(),
  show_statistics = TRUE,
  show_quantiles = FALSE,
  show_gr = TRUE,
  show_eff_n = TRUE
)
```

Arguments

`object`	an object of class `mcmc_dm`, as returned by `estimate_bayesian()`
`...`	additional arguments passed forward to `coda::summary.mcmc.list()`.
`id`	optional single numeric or character, specifying one or more participant IDs to subset `object` in the hierarchical case. Note that `id` will be converted to character, because dimension names of the chains stored in `object` are character. If `NULL`, then the function is applied to group-level parameters.
`x`	an object of class `summary.mcmc_dm`, as returned by `summary.mcmc_dm()`.
`round_digits`	an integer, defining the number of digits for rounding the output.
`show_statistics`	a logical, if `TRUE`, print basic parameter statistics (means, SDs, standard errors).
`show_quantiles`	a logical, if `TRUE`, print quantile summary.
`show_gr`	a logical; if `TRUE`, print Gelman-Rubin convergence diagnostics for each parameter.
`show_eff_n`	a logical, if `TRUE`, print effective sample sizes for each parameter.

Details

The summary and diagnostic statistics of the MCMC chains are obtained using the R package `coda`.

Value

`summary.mcmc_dm()` returns an object of class `summary.mcmc_dm`, which is a list with the following entries:

- `general`: General information about the MCMC run.
- `statistics`: Basic parameter summary statistics.
- `quantiles`: Quantiles for each parameter.
- `gr`: Gelman-Rubin diagnostics.
- `eff_n`: Effective sample sizes.

`print.summary.mcmc_dm()` prints selected summary components and returns the input object invisibly.

See Also

`coda::gelman.diag()`, `coda::effectiveSize()`, `coda::summary.mcmc.list()`

Examples

```
mcmc_obj <- get_example_fits("mcmc_dm")
print(mcmc_obj)
summary(mcmc_obj)
```

`summary.stats_dm` *Summary for* `stats_dm` *Objects*

Description

Summary and corresponding printing methods for objects of the classes `stats_dm`, `basic_stats`, `cafs`, `quantiles`, `delta_funs`, `fit_stats`, `sum_dist`, and `stats_dm_list`. These object types result from a call to `calc_stats()`.

Usage

```
## S3 method for class 'stats_dm'
summary(object, ..., round_digits = drift_dm_default_rounding())

## S3 method for class 'basic_stats'
summary(object, ...)

## S3 method for class 'cafs'
summary(object, ...)

## S3 method for class 'quantiles'
summary(object, ...)

## S3 method for class 'delta_funs'
summary(object, ...)

## S3 method for class 'fit_stats'
summary(object, ...)

## S3 method for class 'sum_dist'
summary(object, ...)

## S3 method for class 'stats_dm_list'
summary(object, ...)

## S3 method for class 'summary.stats_dm'
print(x, ..., show_header = TRUE, drop_cols = NULL)

## S3 method for class 'summary.basic_stats'
print(x, ...)

## S3 method for class 'summary.cafs'
print(x, ...)

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

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

## S3 method for class 'summary.fit_stats'
print(x, ...)

## S3 method for class 'summary.sum_dist'
print(x, ...)

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

Arguments

`object`	an object of the respective class
`...`	additional arguments passed forward.
`round_digits`	integer, specifying the number of decimal places for rounding the summary of the underlying data.frame. Default is 3.
`x`	an object of the respective class.
`show_header`	logical. If `TRUE`, a header specifying the type of statistic will be displayed.
`drop_cols`	character vector, specifying which columns of the table summarizing the underlying data.frame should not be displayed.

Details

- `summary.stats_dm()`: Summarizes `stats_dm` objects, returning the type, a summary of the underlying data.frame (`summary_dataframe`), and, if possible, the number of unique IDs (`n_ids`).
- `summary.sum_dist()`: Extends `summary.stats_dm()` with additional information about the source (`source`).
- `summary.basic_stats()`: Extends `summary.sum_dist()` with additional information about the conditions (`conds`).
- `summary.cafs()`: Extends `summary.sum_dist()` with additional information about the bins (`bins`) and conditions (`conds`).
- `summary.quantiles()`: Extends `summary.sum_dist()` with additional information about the quantile levels (`probs`) and conditions (`conds`).
- `summary.delta_funs()`: Extends `summary.sum_dist()` with additional information about the quantile levels (`probs`).
- `summary.fit_stats()`: Identical to `summary.stats_dm`.
- `summary.stats_dm_list()`: Applies the summary function to each element of the list and returns a list of the respective summary objects.

Note the following class relationships and properties:

- `basic_stats`, `cafs`, `quantiles`, and `delta_funs` are all inheriting from `sum_dist`.
- All `sum_dist` and `fit_stats` objects are inheriting from `stats_dm`.
- Each `stats_dm_list` object is just a list containing instances of `stats_dm`.

Value

For `summary.*()` methods, a summary object of class corresponding to the input class.

For `print.*()` methods, the respective object is returned invisibly

Examples

```
# get a model with data for demonstration purpose
a_model <- dmc_dm()
obs_data(a_model) <- dmc_synth_data

# now get some statistics and call the summary functions
some_stats <- calc_stats(a_model, type = c("quantiles", "fit_stats"))
summary(some_stats) # summary.stats_dm_list
summary(some_stats$quantiles) # summary.quantiles
```

`summary.traces_dm` *Summary for traces_dm and traces_dm_list Objects*

Description

Summary and corresponding printing methods for `traces_dm` and `traces_dm_list` objects, resulting from a call to `simulate_traces()`. Here, `traces_dm` objects are entries of the returned list.

Usage

```
## S3 method for class 'traces_dm'
summary(object, ...)

## S3 method for class 'summary.traces_dm'
print(x, ..., round_digits = drift_dm_default_rounding())

## S3 method for class 'traces_dm_list'
summary(object, ...)

## S3 method for class 'summary.traces_dm_list'
print(x, ..., round_digits = drift_dm_default_rounding())
```

Arguments

`object`	an object of class `traces_dm` or `traces_dm_list`.
`...`	additional arguments passed forward.
`x`	an object of type `summary.traces_dm` or `summary.traces_dm_list`.
`round_digits`	integer, specifying the number of decimal places for rounding in the printed summary. Default is 3.

Details

The `summary.traces_dm()` function constructs a summary list with information about the `traces_dm` object, including:

- **k**: The number of traces in the object.
- **add_x**: A logical, indicating whether starting values were added.
- **orig_model_class**: The class label of the original model.
- **orig_prms**: The parameters with which the traces were simulated (for the respective condition)
- **prms_solve**: The solver settings with which the traces were simulated.
- **fpt_desc**: A summary of the first passage times, including mean, standard deviation, and response probabilities for upper and lower boundaries.

The `summary.traces_dm_list()` function constructs a summary list with information about the `traces_dm_list` object, including:

- **k**: A numeric vector, providing the number of traces per condition.
- **add_x**: A logical vector, indicating whether starting values were added for each condition.
- **orig_prms**: A matrix, containing the original parameter values per condition, with which the traces were simulated.
- **orig_model_class**: The class label of the original model
- **prms_solve**: A matrix of solver settings per condition.
- **fpt_desc**: A summary of the first passage times per condition, including mean, standard deviation, and response probabilities for the upper or lower boundary.

The `print.summary.traces_dm()` and `print.summary.traces_dm_list()` functions display the summary in a formatted way.

Value

`summary.traces_dm()` returns a list of class `summary.traces_dm` (see the Details section summarizing each entry of this list).

`summary.traces_dm_list()` returns a list of class `summary.traces_dm_list` (see the Details section summarizing each entry of this list).

`print.summary.traces_dm()` returns the `summary.traces_dm` object invisibly.

`print.summary.traces_dm_list()` returns the `summary.traces_dm_list` object invisibly.

Examples

```
# get a couple of traces a cross conditions
traces <- simulate_traces(dmc_dm(), k = c(5, 10))
summary(traces)

# get a single traces object
one_traces_obj <- traces[[1]]
summary(one_traces_obj)
```

`ulrich_flanker_data` *Exemplary Flanker Data*

Description

Data of the Flanker task collected in the course of the study by Ulrich et al. (2015).

Usage

```
ulrich_flanker_data
```

Format

A data.frame with 16 individuals and the following columns:

ID Individual IDs

RT Response Times

Error Error Coding (Error Response = 1; Correct Response = 0)

Cond Condition ('comp' and 'incomp')

`ulrich_simon_data` *Exemplary Simon Data*

Description

Data of the Simon task collected in the course of the study by Ulrich et al. (2015).

Usage

```
ulrich_simon_data
```

Format

A data.frame with 16 individuals and the following columns:

ID Individual IDs

RT Response Times

Error Error Coding (Error Response = 1; Correct Response = 0)

Cond Condition ('comp' and 'incomp')

`unpack_obj` *Unpack/Destroy dRiftDM Objects*

Description

When calling `simulate_traces()`, calc_stats, or coef.fits_ids_dm the returned objects will be custom objects (e.g., subclasses of list or data.frame). The respective subclasses were created to provide convenient plotting and printing, but they don't really provide any additional functionality.

The goal of `unpack_obj()` is to provide a convenient way to strip away the attributes of the respective objects (revealing them as standard arrays, data.frames, or lists).

Usage

```
unpack_obj(object, ...)

## S3 method for class 'traces_dm'
unpack_obj(object, ..., unpack_elements = TRUE)

## S3 method for class 'traces_dm_list'
unpack_obj(object, ..., unpack_elements = TRUE, conds = NULL)

## S3 method for class 'stats_dm'
unpack_obj(object, ..., unpack_elements = TRUE)

## S3 method for class 'stats_dm_list'
unpack_obj(object, ..., unpack_elements = TRUE, type = NULL)

## S3 method for class 'coefs_dm'
unpack_obj(object, ..., unpack_elements = TRUE)
```

Arguments

`object`	an object of type `stats_dm`, `stats_dm_list`, `traces_dm`, `traces_dm_list`, or `coefs_dm`
`...`	further arguments passed on to the respective method.
`unpack_elements`	logical, indicating if the `traces_dm`, `stats_dm`, or `coefs_dm` objects shall be unpacked. Default is `TRUE`.
`conds`	optional character vector, indicating specific condition(s). The default `NULL` will lead to `conds = conds(object)`. Thus, per default all conditions are addressed
`type`	optional character vector, indicating specific type(s) of statistics. The default `NULL` will access all types of statics.

Details

`unpack_obj()` is a generic function to strip away the custom information and class labels of `stats_dm`, `stats_dm_list`, `traces_dm`, `traces_dm_list`, and `coefs_dm` objects. These objects are created when calling `simulate_traces()`, calc_stats, or coef.fits_ids_dm.

For `traces_dm_list`, `unpack_obj()` returns the requested conditions (see the argument `conds`). The result contains objects of type `traces_dm` if `unpack_elements = FALSE`. For `unpack_elements = TRUE`, the result contains the plain arrays with the traces.

For `stats_dm_list`, `unpack_obj()` returns the requested statistics (see the argument `type`). The result contains objects of type `stats_dm` if `unpack_elements = FALSE`. For `unpack_elements = TRUE`, the result contains the plain data.frames with the statistics.

Value

For `traces_dm_list`, the returned value is a list, if `conds` specifies more than one condition. For example, if `conds = c("foo", "bar")`, then the returned value is a list with the two (named) entries "foo" and "bar". If the returned list would only have one entry (either because the `traces_dm_list` has only one condition, see conds, or because a user explicitly requested only one condition), then the underlying array or `traces_dm` object is returned directly.

For `stats_dm_list`, the returned value is a list, if `type` specifies more than one condition. If the returned list would only have one entry, then the underlying data.frame or `stats_dm` object is returned directly.

For `traces_dm`, `unpack_obj()` returns an array with the traces, if `unpack=TRUE`. If `unpack=FALSE`, the unmodified object is returned.

For `stats_dm`, `unpack_obj()` returns a data.frame with the respective statistic, if `unpack=TRUE`. If `unpack=FALSE`, the unmodified object is returned.

For `coefs_dm`, `unpack_obj()` returns a data.frame with the parameters, if `unpack=TRUE`. If `unpack=FALSE`, the unmodified object is returned.

Examples

```
# get a pre-built model to demonstrate the function
my_model <- dmc_dm()

# get some traces ...
some_traces <- simulate_traces(my_model, k = 2, seed = 1)
some_traces <- some_traces$comp
class(some_traces)
# ... unpack them to get the underlying arrays
class(unpack_obj(some_traces))

# get some statistics ...
some_stats <- calc_stats(my_model, type = "cafs")
class(some_stats)
class(unpack_obj(some_stats))

# get some parameters ...
some_coefs <- coef(get_example_fits("fits_ids_dm"))
class(some_coefs)
```

```
class(unpack_obj(some_coefs))
```

`unpack_traces` *Unpack/Destroy Traces Objects*

Description

[Deprecated]

`unpack_traces()` is deprecated. Please use the more general `unpack_obj()` function.

Usage

```
unpack_traces(object, ...)

## S3 method for class 'traces_dm'
unpack_traces(object, ..., unpack = TRUE)

## S3 method for class 'traces_dm_list'
unpack_traces(object, ..., unpack = TRUE, conds = NULL)
```

Arguments

`object`	an object of type `traces_dm` or `traces_dm_list` (see `simulate_traces()`)
`...`	further arguments passed on to the respective method.
`unpack`	logical, indicating if the `traces_dm` objects shall be unpacked. Default is `TRUE`.
`conds`	optional character, indicating specific condition(s). The default `NULL` will lead to `conds = conds(object)`. Thus, per default all conditions are accessed.

Details

`unpack_traces()` was a generic function to strip away the "unnecessary" information of `traces_dm_list` and `traces_dm` objects. These objects are created when calling `simulate_traces()`.

For `traces_dm_list`, `unpack_traces()` returns the requested conditions (see the argument `conds`). The result contains objects of type `traces_dm` if `unpack = FALSE`. For `unpack = TRUE`, the result contains the plain arrays with the traces.

Value

For `traces_dm_list`, the returned value is a list, if `conds` specifies more than one condition. For example, if `conds = c("foo", "bar")`, then the returned value is a list with the two (named) entries "foo" and "bar". If the returned list would only have one entry (either because the `traces_dm_list` has only one condition, see conds, or because a user explicitly requested only one condition), then the underlying array or `traces_dm` object is returned directly.

For `traces_dm`, `unpack_traces()` returns an array with the traces, if `unpack=TRUE`. If `unpack=FALSE`, the unmodified object is returned.

Index