Package: dRiftDM (via r-universe)

January 8, 2025

Type Package

Title Estimating (Time-Dependent) Drift Diffusion Models

Version 0.2.1

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 7.3.2

Imports withr, parallel, DEoptim, dfoptim, Rcpp, Rdpack, progress, stats

LinkingTo Rcpp

Depends R (>= 3.5.0)

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 [ctb]

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

Repository CRAN

Date/Publication 2025-01-08 16:30:05 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, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_ids_dm` (see load_fits_ids).
`...`	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_ids()
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 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,
  verbose = 0,
  average = FALSE,
  split_by_ID = TRUE,
  b_coding = NULL
)

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

## S3 method for class 'fits_ids_dm'
calc_stats(object, type, ..., verbose = 1, average = FALSE)
```

Arguments

`object`	an object for which statistics are calculated. This can be a data.frame of observed data, a drift_dm object, or a `fits_ids_dm` object (see estimate_model_ids).
`type`	a character vector, specifying the statistics to calculate. Supported values include `"cafs"`, `"quantiles"`, `"delta_funs"`, 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`.
`verbose`	integer, indicating if information about the progress should be displayed. 0 -> no information, 1 -> a progress bar. Default is 0.
`average`	logical. If `TRUE`, averages the statistics across individuals where applicable. Default is `FALSE`.
`split_by_ID`	logical. If `TRUE`, statistics are calculated separately for each individual ID in `object` (when `object` is a data.frame). Default is `TRUE`.
`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.

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.

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.

Fit Statistics:

Calculates the Akaike and Bayesian Information Criteria (AIC and BIC). Users can provide a `k` argument to penalize the AIC statistic (see stats::AIC and AIC.fits_ids_dm)

Value

If `type` is a single character string, then a data.frame is returned. If `type` contains multiple character strings (i.e., is a character vector) a list with the calculated statistics (with entries being data.frames) is returned.

Each returned data.frame has a certain class label and may store additional attributes required for the custom `plot()` functions. If a list is returned, then that list will have the class label `list_stats_dm` (to easily create multiple panels using the respective `plot()` method).

Examples

```
# Example 1: Calculate CAFs and Quantiles from a model ---------------------
# get a model for demonstration purpose
a_model <- ssp_dm(dx = .0025, dt = .0025, t_max = 2)
# and then calculate cafs and quantiles
some_stats <- calc_stats(a_model, type = c("cafs", "quantiles"))
head(some_stats$cafs)
head(some_stats$quantiles)
```

```
# 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"
)
head(some_stats)


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

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

coef<-   *Convenient Coefficients Access*

Description

Extract or set the coefficients/parameters of drift_dm or `fits_ids_dm` objects

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)

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

Arguments

`object`	an object of type drift_dm or `fits_ids_dm` (see load_fits_ids).

`...`	additional arguments passed to the respective method
`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.

Details

`coef()` are methods for the generic `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 = TRUE`, dRiftDM tries to provide unique parameter labels.

Value

For objects of type drift_dm, `coefs()` returns either a named numeric vector for `select_unique = TRUE`, or the `prms_matrix` matrix for `select_unique = FALSE`. If custom parameters exist, they 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`. The returned data.frame has the class label `coefs_dm` to easily plot histograms for each parameter (see hist.coefs_dm).

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(t_max = 1.5, dx = .0025, dt = .0025)
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("use_ddm_models", "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, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_ids_dm` (see load_fits_ids).
`...`	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("use_ddm_models", "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 the pre-print/vignette 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_ids()
names(comp_funs(fits))
```

---

conds *The Conditions of an Object*

Description

Extract the conditions from a (supported) object.

Usage

```
conds(object, ...)

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

## S3 method for class 'fits_ids_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.

Details

`conds()` is a generic accessor function. The default methods get the "conditions" that are present in an object. Currently supported objects:

- drift_dm
- `fits_ids_dm` (see load_fits_ids)
- data.frame
- `traces_dm_list` (see simulate_traces)

Value

`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).

Note

There is no respective replacement function for `conds()`. If users want to modify the conditions of a drift_dm model, they should create a new flex_prms object and subsequently set it to the model. This is because there is no meaningful way to know for the package how the model shall behave for the newly introduced condition(s).

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_ids()
conds(fits)

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

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 = 0.001,
  dx = 0.001,
  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 additional "instructions", see `modify_flex_prms()` and the Details below.
`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. 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().

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 with a more coarse
# discretization
my_model <- dmc_dm(
  var_start = FALSE,
  t_max = 1.5,
  dx = .0025,
  dt = .0025
)
```

---

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. Its structure is the backbone of the dRiftDM package and every child of the drift_dm class must have the attributes of the parent class. Typically, users will not want to create an object of drift_dm alone, as its use is very limited. Rather, they will want an object of one of its child classes. See `vignette("use_ddm_models", "dRiftDM")` for more information on how to create/use/modify child classes.

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",
  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., `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. Default is `kfe`, which provides access to the numerical discretization of the Kolmogorov Forward Equation.
`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 function, which are document 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 `vignette("use_ddm_models", "dRiftDM")` and `vignette("use_ddm_models", "dRiftDM")` for more information.

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

- ... the log likelihood; can be addressed via `logLik.drift_dm()`.
- ... the PDFs of the first passage time; can be addressed via `drift_dm_obj$pdfs`.

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()`

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_model` *Estimate the Parameters of a drift_dm Model*

Description

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("use_ddm_models", "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 `Unique Parameters` 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

Examples

```
# the example uses a simple model and the Nelder-Mead minimization
# routine to ensure that it runs in a couple of seconds.

# get a model and attach data to the model
my_model <- ratcliff_dm(t_max = 1.5, dx = .005, dt = .005)
obs_data(my_model) <- ratcliff_synth_data # this data set comes with dRiftDM
```

```
# set the search space
lower <- c(muc = 1, b = 0.2, non_dec = 0.1)
upper <- c(muc = 7, b = 1.0, non_dec = 0.6)

# then fit the data to the model using Nelder-Mead after setting some start
# values
coef(my_model) <- c(muc = 2, b = 0.5, non_dec = 0.4)
my_model <- estimate_model(
  drift_dm_obj = my_model, # (starting values are those set to the model)
  lower = lower, # lower and upper parameter ranges
  upper = upper,
  use_de_optim = FALSE, # don't use the default diff. evol. algorithm
  use_nmkb = TRUE # but Nelder-Mead (faster, but way less robust)
)

# show the result
print(my_model)
```

`estimate_model_ids` *Fit Multiple Individuals and Save Results*

Description

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` which indicates that saved files
`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 be found here `vignette("use_ddm_models", "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

Examples

```
# We'll provide a somewhat unrealistic example, trimmed for speed.
# In practice, users likely employ more complex models and more individuals.
# However, a more realistic example would take minutes (and maybe even hours)
# and is therefore not suitable for an example.

# Fit the Ratcliff model to synthetic data --------------------------------
# get the model (pre-built by dRiftDM)
model <- ratcliff_dm(t_max = 2.0, dx = .005, dt = .005)

# define an upper and lower boundary for the parameter space
lower <- c(muc = 1, b = 0.2, non_dec = 0.1)
upper <- c(muc = 7, b = 1.0, non_dec = 0.6)

# simulate synthetic data for demonstration purpose
synth_data_prms <- simulate_data(
  model,
  n = 100, k = 2, lower = lower, upper = upper, seed = 1
)
synth_data <- synth_data_prms$synth_data

# finally, call the fit procedure. To increase speed, we'll use the
# Nelder-Mead minimization routine. Note: We'll save the fits in tempdir()
# to avoid writing to a user's file directory without explicit permission.
estimate_model_ids(
  drift_dm_obj = model, # which model (the Ratcliff model)
  obs_data_ids = synth_data, # which data (the synthetic data set)
  lower = lower, # the lower and upper parameter/search space
  upper = upper,
  fit_procedure_name = "example", # a label for the fit procedure
  fit_path = tempdir(), # temporary directory (replace, e.g., with getwd())
  use_nmkb = TRUE, # use Nelder-Mead (fast, but less robust)
  use_de_optim = FALSE # and not differential evolution
)
```

`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 ushered 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_ids *Auxiliary Function to create a fits_ids object*

Description

This function is merely a helper function to create an object of type `fits_ids_dm`. It is used for example code.

Usage

```
get_example_fits_ids()
```

Details

The returned fit object comprises DMC (see `dmc_dm()`) fitted to three subjects of the ulrich_flanker_data.

Value

An object of type `fits_ids_dm`, mimicking a result from calling `load_fits_ids()`.

Examples

```
fits <- get_example_fits_ids()
```

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,
  ...,
  separate_plots = TRUE,
  alpha = 0.5,
  main = NULL,
  colors = NULL,
  xlab = "values"
)
```

Arguments

`x`	an object of class `coefs_dm` (see coef.fits_ids_dm)
`...`	additional arguments passed to the graphics::hist function.
`separate_plots`	logical, indicating whether to display separate panels for each parameter in a single plot layout (`TRUE`), or to plot them sequentially (`FALSE`).
`alpha`	numeric, specifying the transparency level for histogram colors when conditions are present, with values between 0 (fully transparent) and 1 (fully opaque).
`main`	character vector, specifying titles for each parameter histogram. Defaults to parameter names.
`colors`	character vector, specifying colors for each condition if conditions are present. Defaults to a rainbow color palette. If `NULL` and no conditions are present, the default color is `"skyblue"`.
`xlab`	character, specifying the label for the x-axis.

Details

The `hist.coefs_dm` function is designed for visualizing parameter distributions for a single fit procedure.

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

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

Value

Nothing (`NULL`; invisibly)

Examples

```
# get an auxiliary fit procedure result (see the function load_fits_ids)
all_fits <- get_example_fits_ids()
hist(coef(all_fits)) # only three participants in this fit_ids object

# allows for some customization
hist(coef(all_fits), colors = "lightgreen")
```

`load_fits_ids` *Load Estimates of a Fit Procedure*

Description

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`.

Usage

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

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

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.
`x`	an object of type `fits_ids_dm`, created when calling `load_fits_ids`
`...`	additional arguments

Details

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()`

Examples

```
# -------------------------------------------------------------------------
# We stored a fit procedure (matching with the example for
# estimate_model_ids()) within the package to easily access it here.
# -------------------------------------------------------------------------

# get the path to the fit procedures' location
# -> if a user saved fit procedures in their working directory,
#    path_to would just be "drift_dm_fits" (see the default value of path)
path_to <- file.path(
  system.file(package = "dRiftDM"), "drift_dm_fits"
)

# then load all the fits of a fit procedure
all_fits <- load_fits_ids(path = path_to, fit_procedure_name = "example")
print(all_fits)
summary(all_fits)
```

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

Description

This method extracts the log-likelihood for a `drift_dm` object, ensuring data is available and evaluating the model if necessary.

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 observed data is not available.

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(t_max = 1.5, dx = .0025, dt = .0025)
obs_data(a_model) <- dmc_synth_data

# calculate the log-likelihood
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
`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

A data.frame containing the respective statistic in one column (named `Log_Like`, `AIC`, or `BIC`) and a corresponding `ID` column.

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_ids()

# 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 ushered 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 wish 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 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 wish 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 to manually calculate this, users can define "custom" parameter combinations with the ":=" operation:
- An examplary instruction might look like this: "peak_l := (a - 2) * 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 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`.

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, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_ids_dm` (see load_fits_ids).
`...`	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 ushered 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 ushered to remind the user that the returned data.frame may be sorted differently than expected.

Value

For obs_data() a (re-assembled) data.frame of the observed data. A message is ushered to remind the user that the returned data.frame may be 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_ids()
head(obs_data(fits))
```

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

Description

This function generates a plot of Conditional Accuracy Functions (CAFs). It can display observed and predicted values, making it useful for assessing model fit or exploring observed data.

Usage

```
## S3 method for class 'cafs'
plot(
  x,
  ...,
  conds = NULL,
  col = NULL,
  xlim = NULL,
  ylim = c(0, 1),
  xlab = "Bins",
  ylab = NULL,
  pch = 21,
  lty = 1,
  type = "l",
  legend = NULL,
  legend_pos = "bottomright"
)
```

Arguments

`x`	a data.frame, containing CAFs, typically resulting from a call to calc_stats.
`...`	additional arguments passed to the plot, graphics::points, and graphics::legend functions. Oftentimes, this will (unfortunately) lead to an error due to a clash of arguments.

`conds`	character vector, specifying the conditions to plot. Defaults to all unique conditions.
`col`	Character vector, specifying colors for each condition. If a single color is provided, it will be repeated for each condition.
`xlim`, `ylim`	numeric vectors of length 2, specifying the x and y axis limits.
`xlab`, `ylab`	character, labels for the x and y axes.
`pch`	integer, specifying the plotting symbol for observed data points.
`lty`	integer, line type for the predicted CAFs.
`type`	character, type of plot for the predicted CAFs.
`legend`	character vector, specifying legend labels corresponding to the conditions in the CAFs. Defaults to the condition names.
`legend_pos`	character, specifying the position of the legend on the plot.

Details

The `plot.cafs` function allows for a quick investigation of CAFs, including options for color, symbols, and line types for different data sources (observed vs. predicted). When the supplied data.frame includes multiple IDs, CAFs are aggregated across IDs before plotting.

Value

Nothing (`NULL`; invisibly)

Examples

```
# Example 1: Only model predictions ----------------------------------------
# get a cafs data.frame for demonstration purpose
a_model <- dmc_dm(t_max = 1.5, dt = .0025, dx = .0025)
cafs <- calc_stats(a_model, type = "cafs")

# call the plot function with default values
plot(cafs)

# make the plot a little bit more pretty
plot(cafs,
  col = c("green", "red"),
  ylim = c(0.5, 1)
)

# Example 2: Model predictions and observed data --------------------------
obs_data(a_model) <- dmc_synth_data
cafs <- calc_stats(a_model, type = "cafs")
plot(cafs)
# Note: The model was not fitted to the data set, thus observed data and
# model predictions don't match


# Example 3: Only observed data -------------------------------------------
cafs <- calc_stats(dmc_synth_data, type = "cafs")
```

```
plot(cafs)
```

`plot.delta_funs` *Plot Delta Functions*

Description

This function generates a plot of delta functions, displaying observed and predicted values, which can be useful for evaluating model fit or exploring data characteristics.

If the data contains multiple IDs, delta functions are aggregated across IDs before plotting.

Usage

```
## S3 method for class 'delta_funs'
plot(
  x,
  ...,
  dv = NULL,
  col = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "RT [s]",
  ylab = expression(Delta),
  pch = 21,
  lty = 1,
  type = "l",
  legend = NULL,
  legend_pos = "topright"
)
```

Arguments

`x`	a data.frame, containing delta functions, typically resulting from a call to calc_stats.
`...`	additional arguments passed to the plot, graphics::points, and graphics::legend functions. Oftentimes, this will (unfortunately) lead to an error due to a clash of arguments.
`dv`	character vector, specifying the delta functions to plot. Defaults to all columns beginning with "Delta_" in `x`.
`col`	character vector, specifying colors for each delta function. If a single color is provided, it will be repeated for each function.
`xlim`, `ylim`	numeric vectors of length 2, specifying the x and y axis limits.
`xlab`, `ylab`	character, labels for the x and y axes.
`pch`	integer, specifying the plotting symbol for observed data points.
`lty`	integer, line type for the predicted delta functions.

`type`	character, type of plot for the predicted delta functions.
`legend`	character vector, specifying legend labels corresponding to the delta functions. Defaults to the way functions were derived.
`legend_pos`	character, specifying the position of the legend on the plot.

Details

The `plot.delta_funs` function provides an easy way to investigate delta functions, allowing for customization in color, symbols, and line types for different data sources (observed vs. predicted). If multiple IDs are present in the data, delta functions are aggregated across IDs before plotting. By default, `ylim` is set to twice the range of the delta values to provide more context.

Value

Nothing (`NULL`; invisibly)

Examples

```
# Example 1: Only model predictions ---------------------------------------
# get a delta function data.frame for demonstration purpose
a_model <- dmc_dm(t_max = 1.5, dt = .0025, dx = .0025)
deltas <- calc_stats(
  a_model,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)

# call the plot function with default values
plot(deltas)

# modify the plot
plot(deltas,
  col = c("black"),
  lty = 2,
  xlim = c(0.2, 0.65)
)

# Example 2: Model predictions and observed data --------------------------
obs_data(a_model) <- dmc_synth_data
deltas <- calc_stats(
  a_model,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)
plot(deltas)
# Note: The model was not fitted to the data set, thus observed data and
# model predictions don't match
# Example 3: Only observed data ------------------------------------------
deltas <- calc_stats(
  dmc_synth_data,
  type = "delta_funs",
  minuends = "incomp",
  subtrahends = "comp"
)
plot(deltas)
```

`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,
  legend = NULL,
  legend_pos = "topright"
)
```

Arguments

`x`	an object of class drift_dm
`...`	additional arguments passed forward.
`conds`	character vector, specifying conditions to plot. Defaults to all conditions in `x`.
`col`	character vector, specifying colors for each condition. If a single color is provided, it will be repeated for each condition.
`xlim`	numeric vector of length 2, specifying the x-axis limits for components related to the time space.
`legend`	character vector, specifying legend labels corresponding to the conditions.
`legend_pos`	character, specifying the position of the legend on the plot (e.g., `"topright"`).

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.
- `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.

For each component, if multiple conditions are specified, they will be plotted using different colors as specified in `color`.

When the evaluation of a model component fails, the respective component will not be plotted, but no warning is ushered.

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.list_stats_dm` *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 flexible arrangement of multiple plots on a single graphics device.

Usage

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

Arguments

`x`	an object of type `list_stats_dm`, which is essentially a list 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.list_stats_dm()` function is "merely" a wrapper. All plotting is done by the respective `plot()` methods. 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()`, `calc_stats()`

Examples

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

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

`plot.quantiles` *Plot Quantiles*

Description

This function generates a plot of quantiles. It can display observed and predicted values, making it useful for assessing model fit or exploring observed data distributions.

If the data contains multiple IDs, quantiles are aggregated across IDs before plotting.

Usage

```
## S3 method for class 'quantiles'
plot(
  x,
  ...,
  conds = NULL,
  dv = NULL,
  col = NULL,
  xlim = NULL,
  ylim = c(0, 1),
  xlab = "RT [s]",
  ylab = "F(RT)",
  pch = 21,
  lty = 1,
  type = "l",
  legend = NULL,
  legend_pos = "bottomright"
)
```

Arguments

`x`	a data.frame, containing quantiles, typically resulting from a call to calc_stats.
`...`	additional arguments passed to the plot, graphics::points, and graphics::legend functions. Oftentimes, this will (unfortunately) lead to an error due to a clash of arguments.
`conds`	character vector, specifying the conditions to plot. Defaults to all unique conditions.
`dv`	character, specifying the quantiles to plot. Defaults to quantiles derived from the upper boundary.
`col`	character vector, specifying colors for each condition. If a single color is provided, it will be repeated for each condition.
`xlim`, `ylim`	numeric vectors of length 2, specifying the x and y axis limits.
`xlab`, `ylab`	character, labels for the x and y axes.
`pch`	integer, specifying the plotting symbol for observed data points.
`lty`	integer, line type for the predicted quantiles.
`type`	character, type of plot for the predicted quantiles.
`legend`	character vector, specifying legend labels corresponding to the conditions in the quantiles. Defaults to the condition names.
`legend_pos`	character, specifying the position of the legend on the plot.

Details

The `plot.quantiles` function allows for a quick investigation of quantiles, including options for color, symbols, and line types for different data sources (observed vs. predicted). When the supplied data.frame includes multiple IDs, quantiles are aggregated across IDs before plotting.

Value

Nothing (`NULL`; invisibly)

Examples

```
# Example 1: Only model predictions ---------------------------------------
# get a quantiles data.frame for demonstration purpose
a_model <- dmc_dm(t_max = 1.5, dt = .0025, dx = .0025)
quantiles <- calc_stats(a_model, type = "quantiles")

# call the plot function with default values
plot(quantiles)

# make the plot a little bit more pretty
plot(quantiles,
  col = c("green", "red"),
  xlim = c(0.2, 0.6),
  ylab = "Quantile Level",
  xlab = "Response Times [s]"
)

# Example 2: Model predictions and observed data --------------------------
obs_data(a_model) <- dmc_synth_data
quantiles <- calc_stats(a_model, type = "quantiles")
plot(quantiles)
# Note: The model was not fitted to the data set, thus observed data and
# model predictions don't match


# Example 3: Only observed data -------------------------------------------
quantiles <- calc_stats(dmc_synth_data, type = "quantiles")
plot(quantiles)
```

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

Description

Creates a basic plot showing simulated traces (simulated evidence accumulation processes) from a drift diffusion model. Such plots are useful for exploring and testing model behavior, allowing users to visualize the traces.

Usage

```
## S3 method for class 'traces_dm_list'
plot(
  x,
  ...,
  col = NULL,
  col_b = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "Time",
  ylab = "Evidence",
  lty = 1,
  type = "l",
  legend = NULL,
  legend_pos = "topright"
)

## S3 method for class 'traces_dm'
plot(
  x,
  ...,
  col = NULL,
  col_b = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = "Time",
  ylab = "Evidence",
  lty = 1,
  type = "l"
)
```

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 arguments passed to the plot, graphics::points, and graphics::legend functions. Oftentimes, this will (unfortunately) lead to an error due to a clash of arguments.
`col`	character, vector of colors for the evidence accumulation traces, one per condition. Defaults to a rainbow palette if not specified.
`col_b`	character, a vector of colors for the boundary lines. Defaults to black for all conditions.
`xlim, ylim`	numeric vectors of length 2, specifying the x and y axis limits.
`xlab, ylab`	character, labels for the x and y axes.
`lty`	integer, line type for both the traces and boundary lines.
`type`	character, type of plot to use for traces and boundaries.
`legend`	character vector, specifying legend labels, corresponding to the conditions in the traces. Defaults to the condition names.
`legend_pos`	character, specifying the position of the legend on the plot.

Details

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

`plot_traces_dm` only plots the traces provided (i.e., traces for one condition)

Boundaries and traces are color-coded according to `col` and `col_b`. The function automatically generates the upper and lower boundaries based on the information stored within `x`.

Value

Nothing (`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 generally 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])
)
```

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, ..., round_digits = drift_dm_default_rounding())

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

Arguments

`x`	an object of class `summary.fits_ids_dm`.
`...`	additional arguments
`round_digits`	integer, specifying the number of decimal places for rounding in the printed summary. Default is set to 3.
`object`	an object of class `fits_ids_dm`, generated by a call to load_fits_ids.

Details

The `summary.fits_ids_dm` function creates a summary object containing:

- **fit_procedure_name**: The name of the fit procedure used.
- **time_call**: Timestamp of the last fit procedure call.
- **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_ids()
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, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_ids_dm` (see load_fits_ids).
`...`	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 ushered).

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 (i.e., 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_ids()
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 = 0.001,
  dx = 0.001,
  solver = "kfe",
  b_coding = NULL
)
```

Arguments

var_non_dec, var_start, var_drift	logical, indicating whether the model should have a (uniform) variable non-decision time, starting point, or (normally-distributed) variable drift rate. (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().

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 a more coarse
# discretization
my_model <- ratcliff_dm(
  var_non_dec = TRUE,
  t_max = 1.5,
  dx = .005,
  dt = .005
)
```

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 obs_data are set to the model, the log-likelihood 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 log-likelihood value 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 log-likelihood.

- the PDFs an be addressed via `drift_dm_obj$pdfs`
- the log-likelihood can be addressed via `drift_dm_obj$log_like_val`

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

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(t_max = 1.5, dx = .005, dt = .005)

# 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 log_likelihood, 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$pdfs)
print(my_model$log_like_val)
```

`set_default_colors` *Set Default Colors*

Description

This function assigns default colors if none are provided or adjusts the color vector to match the number of conditions.

Usage

```
set_default_colors(colors, unique_conds, default_colors)
```

Arguments

`colors`	character vector, specifying colors for conditions. If NULL, `default_colors` is used.
`unique_conds`	character vector, listing unique conditions to match color assignments (only the length counts).
`default_colors`	character vector, default colors to use if `colors` is not provided.

Value

A character vector of colors, matching the length of `unique_conds`.

`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,
  k = 1,
  lower = NULL,
  upper = NULL,
  df_prms = NULL,
  seed = NULL,
  verbose = 1
)
```

Arguments

`object`	an object inheriting from drift_dm.
`...`	further arguments passed on to other functions, including the function simulate_values. 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.

`k`	numeric larger than 0, indicating how many data sets shall be simulated. If > 1, then it is only effective when specifying `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
`verbose`	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 `Unique Parameters` 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(t_max = 1.5, dx = .005, dt = .005)

# 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, verbose = 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(t_max = 1.5, dx = .005, dt = .005, 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, verbose = 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 '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 or `fits_ids_dm` (see load_fits_ids).
`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_traces 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 a list of type traces_dm_list, or directly the plain traces as matrices across conditions (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, and the added starting values.

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_traces().

See Also

unpack_traces(), 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_ids()
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).

`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, ...)
```

Arguments

`object`	an object of type drift_dm or `fits_ids_dm` (see load_fits_ids).
`...`	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_ids()
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(
  instr = NULL,
  obs_data = NULL,
  sigma = 1,
  t_max = 3,
  dt = 0.001,
  dx = 0.001,
  b_coding = NULL
)
```

Arguments

`instr`	optional string with additional "instructions", see `modify_flex_prms()` and the Details below.
`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 attentional 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 attentional 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()`.

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 more coarse discretization
my_model <- ssp_dm(
  t_max = 1.5,
  dx = .0025,
  dt = .0025
)
```

summary.drift_dm *Summary for* drift_dm *Objects*

Description

Summary and printing methods for objects of the class drift_dm, resulting from 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 to the respective method
x	an object of type summary.drift_dm
round_digits	Integer specifying the number of decimal places for rounding in the printed summary. Default is 3.

Details

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

- **class**: The class type of the drift_dm object.
- **summary_flex_prms**: A 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**: The solver used for model fitting.
- **obs_data**: A summary table of observed response time data, if available, by response type (upper/lower boundary responses). Includes sample size, mean, and quantiles.
- **fit_stats**: Fit statistics, if available, including log-likelihood, AIC, and BIC values.

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

Value

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

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

Examples

```
# get a pre-built model for demonstration purpose
a_model <- dmc_dm(t_max = 1.5, dx = .0025, dt = .0025)
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)

# finally: fit indices are provided 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)
```

`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_traces` *Unpack/Destroy Traces Objects*

Description

`simulate_traces()` provides a list of type `traces_dm_list`, containing arrays of type `traces_dm`. The respective classes were created to ensure convenient plotting and printing, but they are not really necessary. If users want to create their own figures or access the values of the simulated traces, the data types can even mask the underlying properties.

The goal of `unpack_traces()` is to provide a convenient way to strip away the attributes of `traces_dm_list` and `traces_dm` objects.

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 drift_dm or `fits_ids_dm` (see load_fits_ids)
`...`	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()` is 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.

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)
# and then unpack them to get the underlying arrays
str(unpack_traces(some_traces))
```

Index