Package 'modelbased'

January 13, 2023

Type Package

Title Estimation of Model-Based Predictions, Contrasts and Means

Version 0.8.6

Maintainer Dominique Makowski <dom.makowski@gmail.com>

Description Implements a general interface for model-based estimations for a wide variety of models (see list of supported models using the function 'insight::supported_models()'), used in the computation of marginal means, contrast analysis and predictions.

License GPL-3

URL https://easystats.github.io/modelbased/

BugReports https://github.com/easystats/modelbased/issues

Depends R (>= 3.6)

Imports bayestestR (>= 0.13.0), datawizard (>= 0.6.2), effectsize (>= 0.8.1), insight (>= 0.18.5), parameters (>= 0.19.0), performance (>= 0.10.0), graphics, stats, utils

Suggests brms, correlation, emmeans (>= 1.8.3), gamm4, gganimate, ggplot2, glmmTMB, knitr, lme4, logspline, marginaleffects, mgcv, patchwork, poorman, report, rmarkdown, rstanarm, rtdists, see (>= 0.7.0), testthat

VignetteBuilder knitr

Encoding UTF-8

Language en-US

RoxygenNote 7.2.3

Config/testthat/edition 3

Config/Needs/website rstudio/bslib, r-lib/pkgdown, easystats/easystatstemplate

NeedsCompilation no

Author Dominique Makowski [aut, cre] (<https://orcid.org/0000-0001-5375-9967>, @Dom_Makowski), Daniel Lüdecke [aut] (<https://orcid.org/0000-0002-8895-3206>,

@strengejacke),
Mattan S. Ben-Shachar [aut] (<https://orcid.org/0000-0002-4287-4801>,
@mattansb),
Indrajeet Patil [aut] (<https://orcid.org/0000-0003-1995-6531>,
@patilindrajeets)

Repository CRAN

Date/Publication 2023-01-13 08:50:02 UTC

R topics documented:

`describe_nonlinear` *Describe the smooth term (for GAMs) or non-linear predictors*

Description

This function summarises the smooth term trend in terms of linear segments. Using the approximative derivative, it separates a non-linear vector into quasi-linear segments (in which the trend is either positive or negative). Each of this segment its characterized by its beginning, end, size (in proportion, relative to the total size) trend (the linear regression coefficient) and linearity (the R2 of the linear regression).

Usage

```
describe_nonlinear(data, ...)

## S3 method for class 'data.frame'
describe_nonlinear(data, x = NULL, y = NULL, ...)

estimate_smooth(data, ...)
```

Arguments

data	The data containing the link, as for instance obtained by estimate_relation().
...	Other arguments to be passed to or from.
x, y	The name of the responses variable (y) predicting variable (x).

Value

A dataframe of linear description of non-linear terms.

Examples

```
library(modelbased)

# Create data
data <- data.frame(x = rnorm(200))
data$y <- data$x^2 + rnorm(200, 0, 0.5)

model <- lm(y ~ poly(x, 2), data = data)
link_data <- estimate_relation(model, length = 100)

describe_nonlinear(link_data, x = "x")
```

---

estimate_contrasts *Estimate Marginal Contrasts*

Description

Run a contrast analysis by estimating the differences between each level of a factor. See also other related functions such as estimate_means() and estimate_slopes().

Usage

```
estimate_contrasts(
  model,
  contrast = NULL,
  at = NULL,
  fixed = NULL,
  transform = "none",
  ci = 0.95,
  p_adjust = "holm",
  method = "pairwise",
  adjust = NULL,
  ...
)
```

Arguments

`model`	A statistical model.
`contrast`	A character vector indicating the name of the variable(s) for which to compute the contrasts.
`at`	The predictor variable(s) *at* which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them).
`fixed`	A character vector indicating the names of the predictors to be "fixed" (i.e., maintained), so that the estimation is made at these values.
`transform`	Is passed to the `type` argument in `emmeans::emmeans()`. See this vignette. Can be `"none"` (default for contrasts), `"response"` (default for means), `"mu"`, `"unlink"`, `"log"`. `"none"` will leave the values on scale of the linear predictors. `"response"` will transform them on scale of the response variable. Thus for a logistic model, `"none"` will give estimations expressed in log-odds (probabilities on logit scale) and `"response"` in terms of probabilities.
`ci`	Confidence Interval (CI) level. Default to `0.95` (95%).
`p_adjust`	The p-values adjustment method for frequentist multiple comparisons. Can be one of "holm" (default), "tukey", "hochberg", "hommel", "bonferroni", "BH", "BY", "fdr" or "none". See the p-value adjustment section in the `emmeans::test` documentation.
`method`	Contrast method. See same argument in emmeans::contrast.
`adjust`	Deprecated in favour of `p_adjust`.
`...`	Other arguments passed for instance to `insight::get_datagrid()`.

Details

See the **Details** section below, and don't forget to also check out the Vignettes and README examples for various examples, tutorials and use cases.

The `estimate_slopes()`, `estimate_means()` and `estimate_contrasts()` functions are forming a group, as they are all based on *marginal* estimations (estimations based on a model). All three are also built on the **emmeans** package, so reading its documentation (for instance for `emmeans::emmeans()` and `emmeans::emtrends()`) is recommended to understand the idea behind these types of procedures.

- Model-based **predictions** is the basis for all that follows. Indeed, the first thing to understand is how models can be used to make predictions (see `estimate_link()`). This corresponds to the predicted response (or "outcome variable") given specific predictor values of the predictors (i.e., given a specific data configuration). This is why the concept of `reference grid()` is so important for direct predictions.
- **Marginal "means"**, obtained via `estimate_means()`, are an extension of such predictions, allowing to "average" (collapse) some of the predictors, to obtain the average response value at a specific predictors configuration. This is typically used when some of the predictors of interest are factors. Indeed, the parameters of the model will usually give you the intercept value and then the "effect" of each factor level (how different it is from the intercept). Marginal means can be used to directly give you the mean value of the response variable at all the levels

of a factor. Moreover, it can also be used to control, or average over predictors, which is useful in the case of multiple predictors with or without interactions.

- **Marginal contrasts**, obtained via `estimate_contrasts()`, are themselves at extension of marginal means, in that they allow to investigate the difference (i.e., the contrast) between the marginal means. This is, again, often used to get all pairwise differences between all levels of a factor. It works also for continuous predictors, for instance one could also be interested in whether the difference at two extremes of a continuous predictor is significant.
- Finally, **marginal effects**, obtained via `estimate_slopes()`, are different in that their focus is not values on the response variable, but the model's parameters. The idea is to assess the effect of a predictor at a specific configuration of the other predictors. This is relevant in the case of interactions or non-linear relationships, when the effect of a predictor variable changes depending on the other predictors. Moreover, these effects can also be "averaged" over other predictors, to get for instance the "general trend" of a predictor over different factor levels.

Example: Let's imagine the following model `lm(y ~ condition * x)` where `condition` is a factor with 3 levels A, B and C and `x` a continuous variable (like age for example). One idea is to see how this model performs, and compare the actual response y to the one predicted by the model (using `estimate_response()`). Another idea is evaluate the average mean at each of the condition's levels (using `estimate_means()`), which can be useful to visualize them. Another possibility is to evaluate the difference between these levels (using `estimate_contrasts()`). Finally, one could also estimate the effect of x averaged over all conditions, or instead within each condition (using `[estimate_slopes]`).

Value

A data frame of estimated contrasts.

Examples

```
# Basic usage
model <- lm(Sepal.Width ~ Species, data = iris)
estimate_contrasts(model)

# Dealing with interactions
model <- lm(Sepal.Width ~ Species * Petal.Width, data = iris)

# By default: selects first factor
estimate_contrasts(model)

# Can also run contrasts between points of numeric
estimate_contrasts(model, contrast = "Petal.Width", length = 4)

# Or both
estimate_contrasts(model, contrast = c("Species", "Petal.Width"), length = 2)

# Or with custom specifications
estimate_contrasts(model, contrast = c("Species", "Petal.Width=c(1, 2)"))

# Can fixate the numeric at a specific value
estimate_contrasts(model, fixed = "Petal.Width")
```

```
# Or modulate it
estimate_contrasts(model, at = "Petal.Width", length = 4)

# Standardized differences
estimated <- estimate_contrasts(lm(Sepal.Width ~ Species, data = iris))
standardize(estimated)


# Other models (mixed, Bayesian, ...)
data <- iris
data$Petal.Length_factor <- ifelse(data$Petal.Length < 4.2, "A", "B")

model <- lme4::lmer(Sepal.Width ~ Species + (1 | Petal.Length_factor), data = data)
estimate_contrasts(model)


library(rstanarm)

data <- mtcars
data$cyl <- as.factor(data$cyl)
data$am <- as.factor(data$am)
## Not run:
model <- stan_glm(mpg ~ cyl * am, data = data, refresh = 0)
estimate_contrasts(model)
estimate_contrasts(model, fixed = "am")

model <- stan_glm(mpg ~ cyl * wt, data = data, refresh = 0)
estimate_contrasts(model)
estimate_contrasts(model, fixed = "wt")
estimate_contrasts(model, at = "wt", length = 4)

model <- stan_glm(Sepal.Width ~ Species + Petal.Width + Petal.Length, data = iris, refresh = 0)
estimate_contrasts(model, at = "Petal.Length", test = "bf")

## End(Not run)
```

`estimate_expectation` *Model-based response estimates and uncertainty*

Description

After fitting a model, it is useful generate model-based estimates of the response variables for different combinations of predictor values. Such estimates can be used to make inferences about relationships between variables and to make predictions about individual cases.

Model-based response estimates and uncertainty can be generated for both the conditional average response values (the regression line or expectation) and for predictions about individual cases. See below for details.

Usage

```
estimate_expectation(
  model,
  data = NULL,
  ci = 0.95,
  keep_iterations = FALSE,
  ...
)

estimate_response(...)

estimate_link(model, data = "grid", ci = 0.95, keep_iterations = FALSE, ...)

estimate_prediction(
  model,
  data = NULL,
  ci = 0.95,
  keep_iterations = FALSE,
  ...
)

estimate_relation(
  model,
  data = "grid",
  ci = 0.95,
  keep_iterations = FALSE,
  ...
)
```

Arguments

model	A statistical model.
data	A data frame with model's predictors to estimate the response. If NULL, the model's data is used. If "grid", the model matrix is obtained (through insight::get_datagrid()).
ci	Confidence Interval (CI) level. Default to 0.95 (95%).
keep_iterations	If TRUE, will keep all iterations (draws) of bootstrapped or Bayesian models. They will be added as additional columns named iter_1, iter_2, ... You can reshape them to a long format by running reshape_iterations().
...	You can add all the additional control arguments from insight::get_datagrid() (used when data = "grid") and insight::get_predicted().

Value

A data frame of predicted values and uncertainty intervals, with class "estimate_predicted". Methods for visualisation_recipe() and plot() are available.

Expected (average) values

The most important way that various types of response estimates differ is in terms of what quantity is being estimated and the meaning of the uncertainty intervals. The major choices are **expected values** for uncertainty in the regression line and **predicted values** for uncertainty in the individual case predictions.

Expected values refer to the fitted regression line - the estimated *average* response value (i.e., the "expectation") for individuals with specific predictor values. For example, in a linear model $y = 2 + 3x + 4z + e$, the estimated average y for individuals with $x = 1$ and $z = 2$ is 11.

For expected values, uncertainty intervals refer to uncertainty in the estimated **conditional average** (where might the true regression line actually fall)? Uncertainty intervals for expected values are also called "confidence intervals".

Expected values and their uncertainty intervals are useful for describing the relationship between variables and for describing how precisely a model has been estimated.

For generalized linear models, expected values are reported on one of two scales:

- The **link scale** refers to scale of the fitted regression line, after transformation by the link function. For example, for a logistic regression (logit binomial) model, the link scale gives expected log-odds. For a log-link Poisson model, the link scale gives the expected log-count.
- The **response scale** refers to the original scale of the response variable (i.e., without any link function transformation). Expected values on the link scale are back-transformed to the original response variable metric (e.g., expected probabilities for binomial models, expected counts for Poisson models).

Individual case predictions

In contrast to expected values, **predicted values** refer to predictions for **individual cases**. Predicted values are also called "posterior predictions" or "posterior predictive draws".

For predicted values, uncertainty intervals refer to uncertainty in the **individual response values for each case** (where might any single case actually fall)? Uncertainty intervals for predicted values are also called "prediction intervals" or "posterior predictive intervals".

Predicted values and their uncertainty intervals are useful for forecasting the range of values that might be observed in new data, for making decisions about individual cases, and for checking if model predictions are reasonable ("posterior predictive checks").

Predicted values and intervals are always on the scale of the original response variable (not the link scale).

Functions for estimating predicted values and uncertainty

modelbased provides 4 functions for generating model-based response estimates and their uncertainty:

- `estimate_expectation()`:
  - Generates **expected values** (conditional average) on the **response scale**.
  - The uncertainty interval is a *confidence interval*.
  - By default, values are computed using the data used to fit the model.
- `estimate_link()`:

  - Generates **expected values** (conditional average) on the **link scale**.
  - The uncertainty interval is a *confidence interval*.
  - By default, values are computed using a reference grid spanning the observed range of predictor values (see `visualisation_matrix()`).
- `estimate_prediction()`:
  - Generates **predicted values** (for individual cases) on the **response scale**.
  - The uncertainty interval is a *prediction interval*.
  - By default, values are computed using the data used to fit the model.
- `estimate_relation()`:
  - Like `estimate_expectation()`.
  - Useful for visualizing a model.
  - Generates **expected values** (conditional average) on the **response scale**.
  - The uncertainty interval is a *confidence interval*.
  - By default, values are computed using a reference grid spanning the observed range of predictor values (see `visualisation_matrix()`).

`estimate_response()` is a deprecated alias for `estimate_expectation()`.

Data for predictions

If the `data = NULL`, values are estimated using the data used to fit the model. If `data = "grid"`, values are computed using a reference grid spanning the observed range of predictor values with `visualisation_matrix()`. This can be useful for model visualization. The number of predictor values used for each variable can be controlled with the `length` argument. `data` can also be a data frame containing columns with names matching the model frame (see `insight::get_data()`). This can be used to generate model predictions for specific combinations of predictor values.

Note

These functions are built on top of `insight::get_predicted()` and correspond to different specifications of its parameters. It may be useful to read its documentation, in particular the description of the `predict` argument for additional details on the difference between expected vs. predicted values and link vs. response scales.

Additional control parameters can be used to control results from `insight::get_datagrid()` (when `data = "grid"`) and from `insight::get_predicted()` (the function used internally to compute predictions).

For plotting, check the examples in `visualisation_recipe()`. Also check out the Vignettes and README examples for various examples, tutorials and usecases.

Examples

```
library(modelbased)

# Linear Models
model <- lm(mpg ~ wt, data = mtcars)

# Get predicted and prediction interval (see insight::get_predicted)
```

```
estimate_response(model)

# Get expected values with confidence interval
pred <- estimate_relation(model)
pred

# Visualisation (see visualisation_recipe())
if (require("see")) {
  plot(pred)
}

# Standardize predictions
pred <- estimate_relation(lm(mpg ~ wt + am, data = mtcars))
z <- standardize(pred, include_response = FALSE)
z
unstandardize(z, include_response = FALSE)

# Logistic Models
model <- glm(vs ~ wt, data = mtcars, family = "binomial")
estimate_response(model)
estimate_relation(model)

# Mixed models
if (require("lme4")) {
  model <- lmer(mpg ~ wt + (1 | gear), data = mtcars)
  estimate_response(model)
  estimate_relation(model)
}

# Bayesian models

if (require("rstanarm")) {
  model <- rstanarm::stan_glm(mpg ~ wt, data = mtcars, refresh = 0, iter = 200)
  estimate_response(model)
  estimate_relation(model)
}
```

`estimate_grouplevel` *Group-specific parameters of mixed models random effects*

Description

Extract random parameters of each individual group in the context of mixed models. Can be reshaped to be of the same dimensions as the original data, which can be useful to add the random effects to the original data.

Usage

```
estimate_grouplevel(model, type = "random", ...)
```

```
reshape_grouplevel(x, indices = "all", group = "all", ...)
```

Arguments

model	A mixed model with random effects.
type	If "random" (default), the coefficients are the ones estimated natively by the model (as they are returned by, for instance, lme4::ranef()). They correspond to the deviation of each individual group from their fixed effect. As such, a coefficient close to 0 means that the participants' effect is the same as the population-level effect (in other words, it is "in the norm"). If "total", it will return the sum of the random effect and its corresponding fixed effects. These are known as BLUPs (Best Linear Unbiased Predictions). This argument can be used to reproduce the results given by lme4::ranef() and coef() (see ?coef.merMod). Note that BLUPs currently don't have uncertainty indices (such as SE and CI), as these are not computable.
...	Other arguments passed to or from other methods.
x	The output of estimate_grouplevel().
indices	A list containing the indices to extract (e.g., "Coefficient").
group	A list containing the random factors to select.

Examples

```
# lme4 model
if (require("lme4") && require("see")) {
  model <- lmer(mpg ~ hp + (1 | carb), data = mtcars)
  random <- estimate_grouplevel(model)
  random

  # Visualize random effects
  plot(random)

  # Show group-specific effects
  estimate_grouplevel(model, deviation = FALSE)

  # Reshape to wide data so that it matches the original dataframe...
  reshaped <- reshape_grouplevel(random, indices = c("Coefficient", "SE"))

  # ... and can be easily combined
  alldata <- cbind(mtcars, reshaped)

  # Use summary() to remove duplicated rows
  summary(reshaped)

  # Compute BLUPs
  estimate_grouplevel(model, type = "total")
}

# Bayesian models
```

```
if (require("rstanarm")) {
 model <- rstanarm::stan_lmer(mpg ~ hp + (1 | carb) + (1 | gear), data = mtcars, refresh = 0)
  # Broken estimate_grouplevel(model)
}
```

`estimate_means` *Estimate Marginal Means (Model-based average at each factor level)*

Description

Estimate average value of response variable at each factor levels. For plotting, check the examples in `visualisation_recipe()`. See also other related functions such as `estimate_contrasts()` and `estimate_slopes()`.

Usage

```
estimate_means(
  model,
  at = "auto",
  fixed = NULL,
  transform = "response",
  ci = 0.95,
  backend = "emmeans",
  ...
)
```

Arguments

`model`	A statistical model.
`at`	The predictor variable(s) *at* which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them).
`fixed`	A character vector indicating the names of the predictors to be "fixed" (i.e., maintained), so that the estimation is made at these values.
`transform`	Is passed to the `type` argument in `emmeans::emmeans()`. See this vignette. Can be `"none"` (default for contrasts), `"response"` (default for means), `"mu"`, `"unlink"`, `"log"`. `"none"` will leave the values on scale of the linear predictors. `"response"` will transform them on scale of the response variable. Thus for a logistic model, `"none"` will give estimations expressed in log-odds (probabilities on logit scale) and `"response"` in terms of probabilities.
`ci`	Confidence Interval (CI) level. Default to `0.95` (95%).
`backend`	Whether to use 'emmeans' or 'marginaleffects' as a backend. The latter is experimental and some features might not work.
`...`	Other arguments passed for instance to `insight::get_datagrid()`.

Details

See the **Details** section below, and don't forget to also check out the Vignettes and README examples for various examples, tutorials and use cases.

The `estimate_slopes()`, `estimate_means()` and `estimate_contrasts()` functions are forming a group, as they are all based on *marginal* estimations (estimations based on a model). All three are also built on the **emmeans** package, so reading its documentation (for instance for `emmeans::emmeans()` and `emmeans::emtrends()`) is recommended to understand the idea behind these types of procedures.

- Model-based **predictions** is the basis for all that follows. Indeed, the first thing to understand is how models can be used to make predictions (see `estimate_link()`). This corresponds to the predicted response (or "outcome variable") given specific predictor values of the predictors (i.e., given a specific data configuration). This is why the concept of `reference grid()` is so important for direct predictions.
- **Marginal "means"**, obtained via `estimate_means()`, are an extension of such predictions, allowing to "average" (collapse) some of the predictors, to obtain the average response value at a specific predictors configuration. This is typically used when some of the predictors of interest are factors. Indeed, the parameters of the model will usually give you the intercept value and then the "effect" of each factor level (how different it is from the intercept). Marginal means can be used to directly give you the mean value of the response variable at all the levels of a factor. Moreover, it can also be used to control, or average over predictors, which is useful in the case of multiple predictors with or without interactions.
- **Marginal contrasts**, obtained via `estimate_contrasts()`, are themselves at extension of marginal means, in that they allow to investigate the difference (i.e., the contrast) between the marginal means. This is, again, often used to get all pairwise differences between all levels of a factor. It works also for continuous predictors, for instance one could also be interested in whether the difference at two extremes of a continuous predictor is significant.
- Finally, **marginal effects**, obtained via `estimate_slopes()`, are different in that their focus is not values on the response variable, but the model's parameters. The idea is to assess the effect of a predictor at a specific configuration of the other predictors. This is relevant in the case of interactions or non-linear relationships, when the effect of a predictor variable changes depending on the other predictors. Moreover, these effects can also be "averaged" over other predictors, to get for instance the "general trend" of a predictor over different factor levels.

Example: Let's imagine the following model `lm(y ~ condition * x)` where `condition` is a factor with 3 levels A, B and C and `x` a continuous variable (like age for example). One idea is to see how this model performs, and compare the actual response y to the one predicted by the model (using `estimate_response()`). Another idea is evaluate the average mean at each of the condition's levels (using `estimate_means()`), which can be useful to visualize them. Another possibility is to evaluate the difference between these levels (using `estimate_contrasts()`). Finally, one could also estimate the effect of x averaged over all conditions, or instead within each condition (`using [estimate_slopes]`).

Value

A dataframe of estimated marginal means.

Examples

```
library(modelbased)
if (require("emmeans")) {

# Frequentist models
# -------------------
model <- lm(Petal.Length ~ Sepal.Width * Species, data = iris)

estimate_means(model)
estimate_means(model, fixed = "Sepal.Width")
estimate_means(model, at = c("Species", "Sepal.Width"), length = 2)
estimate_means(model, at = "Species=c('versicolor', 'setosa')")
estimate_means(model, at = "Sepal.Width=c(2, 4)")
estimate_means(model, at = c("Species", "Sepal.Width=0"))
estimate_means(model, at = "Sepal.Width", length = 5)
estimate_means(model, at = "Sepal.Width=c(2, 4)")

# Methods that can be applied to it:
means <- estimate_means(model, fixed = "Sepal.Width")
if (require("see")) {
  plot(means) # which runs visualisation_recipe()
}
standardize(means)

if (require("lme4")) {
  data <- iris
  data$Petal.Length_factor <- ifelse(data$Petal.Length < 4.2, "A", "B")

 model <- lmer(Petal.Length ~ Sepal.Width + Species + (1 | Petal.Length_factor), data = data)
  estimate_means(model)
  estimate_means(model, at = "Sepal.Width", length = 3)
}

}
```

`estimate_slopes` *Estimate Marginal Effects*

Description

Estimate the slopes (i.e., the coefficient) of a predictor over or within different factor levels, or alongside a numeric variable . In other words, to assess the effect of a predictor *at* specific configurations data. Other related functions based on marginal estimations includes `estimate_contrasts()` and `estimate_means()`.

Usage

```
estimate_slopes(model, trend = NULL, at = NULL, ci = 0.95, ...)
```

Arguments

model	A statistical model.
trend	A character indicating the name of the variable for which to compute the slopes.
at	The predictor variable(s) *at* which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them).
ci	Confidence Interval (CI) level. Default to 0.95 (95%).
...	Other arguments passed for instance to insight::get_datagrid().

Details

See the **Details** section below, and don't forget to also check out the Vignettes and README examples for various examples, tutorials and use cases.

The estimate_slopes(), estimate_means() and estimate_contrasts() functions are forming a group, as they are all based on *marginal* estimations (estimations based on a model). All three are also built on the **emmeans** package, so reading its documentation (for instance for emmeans::emmeans() and emmeans::emtrends()) is recommended to understand the idea behind these types of procedures.

- Model-based **predictions** is the basis for all that follows. Indeed, the first thing to understand is how models can be used to make predictions (see estimate_link()). This corresponds to the predicted response (or "outcome variable") given specific predictor values of the predictors (i.e., given a specific data configuration). This is why the concept of reference grid() is so important for direct predictions.
- **Marginal "means"**, obtained via estimate_means(), are an extension of such predictions, allowing to "average" (collapse) some of the predictors, to obtain the average response value at a specific predictors configuration. This is typically used when some of the predictors of interest are factors. Indeed, the parameters of the model will usually give you the intercept value and then the "effect" of each factor level (how different it is from the intercept). Marginal means can be used to directly give you the mean value of the response variable at all the levels of a factor. Moreover, it can also be used to control, or average over predictors, which is useful in the case of multiple predictors with or without interactions.
- **Marginal contrasts**, obtained via estimate_contrasts(), are themselves at extension of marginal means, in that they allow to investigate the difference (i.e., the contrast) between the marginal means. This is, again, often used to get all pairwise differences between all levels of a factor. It works also for continuous predictors, for instance one could also be interested in whether the difference at two extremes of a continuous predictor is significant.
- Finally, **marginal effects**, obtained via estimate_slopes(), are different in that their focus is not values on the response variable, but the model's parameters. The idea is to assess the effect of a predictor at a specific configuration of the other predictors. This is relevant in the case of interactions or non-linear relationships, when the effect of a predictor variable changes depending on the other predictors. Moreover, these effects can also be "averaged" over other predictors, to get for instance the "general trend" of a predictor over different factor levels.

Example: Let's imagine the following model lm(y ~ condition * x) where condition is a factor with 3 levels A, B and C and x a continuous variable (like age for example). One idea is to see how

this model performs, and compare the actual response y to the one predicted by the model (using estimate_response()). Another idea is evaluate the average mean at each of the condition's levels (using estimate_means()), which can be useful to visualize them. Another possibility is to evaluate the difference between these levels (using estimate_contrasts()). Finally, one could also estimate the effect of x averaged over all conditions, or instead within each condition (using [estimate_slopes]).

Value

A data.frame of class estimate_slopes.

Examples

```
if (require("emmeans")) {
  # Get an idea of the data
  if (require("ggplot2")) {
    ggplot(iris, aes(x = Petal.Length, y = Sepal.Width)) +
      geom_point(aes(color = Species)) +
      geom_smooth(color = "black", se = FALSE) +
      geom_smooth(aes(color = Species), linetype = "dotted", se = FALSE) +
      geom_smooth(aes(color = Species), method = "lm", se = FALSE)
  }

  # Model it
  model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)
  # Compute the marginal effect of Petal.Length at each level of Species
  slopes <- estimate_slopes(model, trend = "Petal.Length", at = "Species")
  slopes
  if (require("see")) {
    plot(slopes)
  }
  standardize(slopes)

  ## Not run:
  # TODO: fails with latest emmeans (1.8.0)
  if (require("mgcv") && require("see")) {
    model <- mgcv::gam(Sepal.Width ~ s(Petal.Length), data = iris)
    slopes <- estimate_slopes(model, at = "Petal.Length", length = 50)
    summary(slopes)
    plot(slopes)

    model <- mgcv::gam(Sepal.Width ~ s(Petal.Length, by = Species), data = iris)
    slopes <- estimate_slopes(model,
      trend = "Petal.Length",
      at = c("Petal.Length", "Species"), length = 20
    )
    summary(slopes)
    plot(slopes)
  }
## End(Not run)
}
```

find_inversions *Find points of inversion*

Description

Find points of inversion of a curve.

Usage

```
find_inversions(x)
```

Arguments

x	A numeric vector.

Value

Vector of inversion points.

Examples

```
x <- sin(seq(0, 4 * pi, length.out = 100))
plot(x, type = "b")
find_inversions(x)
```

get_emcontrasts *Easy 'emmeans' and 'emtrends'*

Description

The `get_emmeans()` function is a wrapper to facilitate the usage of `emmeans::emmeans()` and `emmeans::emtrends()`, providing a somewhat simpler and intuitive API to find the specifications and variables of interest. It is meanly made to for the developers to facilitate the organization and debugging, and end-users should rather use the `estimate_*()` series of functions.

Usage

```
get_emcontrasts(
  model,
  contrast = NULL,
  at = NULL,
  fixed = NULL,
  transform = "none",
  method = "pairwise",
  ...
)
```

```
model_emcontrasts(
  model,
  contrast = NULL,
  at = NULL,
  fixed = NULL,
  transform = "none",
  method = "pairwise",
  ...
)

get_emmeans(
  model,
  at = "auto",
  fixed = NULL,
  transform = "response",
  levels = NULL,
  modulate = NULL,
  ...
)

model_emmeans(
  model,
  at = "auto",
  fixed = NULL,
  transform = "response",
  levels = NULL,
  modulate = NULL,
  ...
)

get_emtrends(
  model,
  trend = NULL,
  at = NULL,
  fixed = NULL,
  levels = NULL,
  modulate = NULL,
  ...
)

model_emtrends(
  model,
  trend = NULL,
  at = NULL,
  fixed = NULL,
  levels = NULL,
  modulate = NULL,
```

```
  ...
)
```

Arguments

model	A statistical model.
contrast	A character vector indicating the name of the variable(s) for which to compute the contrasts.
at	The predictor variable(s) *at* which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them).
fixed	A character vector indicating the names of the predictors to be "fixed" (i.e., maintained), so that the estimation is made at these values.
transform	Is passed to the type argument in emmeans::emmeans(). See this vignette. Can be "none" (default for contrasts), "response" (default for means), "mu", "unlink", "log". "none" will leave the values on scale of the linear predictors. "response" will transform them on scale of the response variable. Thus for a logistic model, "none" will give estimations expressed in log-odds (probabilities on logit scale) and "response" in terms of probabilities.
method	Contrast method. See same argument in emmeans::contrast.
...	Other arguments passed for instance to insight::get_datagrid().
levels, modulate	Deprecated, use at instead.
trend	A character indicating the name of the variable for which to compute the slopes.

Examples

```
if (require("emmeans", quietly = TRUE)) {
  # Basic usage
  model <- lm(Sepal.Width ~ Species, data = iris)
  get_emcontrasts(model)

  # Dealing with interactions
  model <- lm(Sepal.Width ~ Species * Petal.Width, data = iris)
  # By default: selects first factor
  get_emcontrasts(model)
  # Can also run contrasts between points of numeric
  get_emcontrasts(model, contrast = "Petal.Width", length = 3)
  # Or both
  get_emcontrasts(model, contrast = c("Species", "Petal.Width"), length = 2)
  # Or with custom specifications
  estimate_contrasts(model, contrast = c("Species", "Petal.Width=c(1, 2)"))
  # Can fixate the numeric at a specific value
  get_emcontrasts(model, fixed = "Petal.Width")
  # Or modulate it
  get_emcontrasts(model, at = "Petal.Width", length = 4)
}
model <- lm(Sepal.Length ~ Species + Petal.Width, data = iris)
```

```
if (require("emmeans", quietly = TRUE)) {
  # By default, 'at' is set to "Species"
  get_emmeans(model)

  # Overall mean (close to 'mean(iris$Sepal.Length)')
  get_emmeans(model, at = NULL)

  # One can estimate marginal means at several values of a 'modulate' variable
  get_emmeans(model, at = "Petal.Width", length = 3)

  # Interactions
  model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

  get_emmeans(model)
  get_emmeans(model, at = c("Species", "Petal.Length"), length = 2)
  get_emmeans(model, at = c("Species", "Petal.Length = c(1, 3, 5)"), length = 2)
}
if (require("emmeans")) {
  model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

  get_emtrends(model)
  get_emtrends(model, at = "Species")
  get_emtrends(model, at = "Petal.Length")
  get_emtrends(model, at = c("Species", "Petal.Length"))

  model <- lm(Petal.Length ~ poly(Sepal.Width, 4), data = iris)
  get_emtrends(model)
  get_emtrends(model, at = "Sepal.Width")
}
```

`get_marginaleffects` *Easy marginaleffects*

Description

Modelbased-like API to create **marginaleffects** objects. This is Work-in-progress.

Usage

```
get_marginaleffects(model, trend = NULL, at = NULL, fixed = NULL, ...)
```

Arguments

`model`	A statistical model.
`trend`	A character indicating the name of the variable for which to compute the slopes.
`at`	The predictor variable(s) *at* which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them).

fixed	A character vector indicating the names of the predictors to be "fixed" (i.e., maintained), so that the estimation is made at these values.
...	Other arguments passed for instance to insight::get_datagrid().

Examples

```
if (require("marginaleffects")) {
  model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

  get_marginaleffects(model, trend = "Petal.Length", at = "Species")
  get_marginaleffects(model, trend = "Petal.Length", at = "Petal.Length")
  get_marginaleffects(model, trend = "Petal.Length", at = c("Species", "Petal.Length"))
}
```

---

smoothing *Smoothing a vector or a time series*

Description

Smoothing a vector or a time series. For data.frames, the function will smooth all numeric variables stratified by factor levels (i.e., will smooth within each factor level combination).

Usage

```
smoothing(x, method = "loess", strength = 0.25, ...)
```

Arguments

x	A numeric vector.
method	Can be "loess" (default) or "smooth". A loess smoothing can be slow.
strength	This argument only applies when method = "loess". Degree of smoothing passed to span (see loess()).
...	Arguments passed to or from other methods.

Value

A smoothed vector or data frame.

Examples

```
x <- sin(seq(0, 4 * pi, length.out = 100)) + rnorm(100, 0, 0.2)
plot(x, type = "l")
lines(smoothing(x, method = "smooth"), type = "l", col = "blue")
lines(smoothing(x, method = "loess"), type = "l", col = "red")

x <- sin(seq(0, 4 * pi, length.out = 10000)) + rnorm(10000, 0, 0.2)
plot(x, type = "l")
lines(smoothing(x, method = "smooth"), type = "l", col = "blue")
lines(smoothing(x, method = "loess"), type = "l", col = "red")
```

`visualisation_matrix` *Create a reference grid*

Description

This function is an alias (another name) for the `insight::get_datagrid()` function. Same arguments apply.

Usage

```
visualisation_matrix(x, ...)

## S3 method for class 'data.frame'
visualisation_matrix(
  x,
  at = "all",
  target = NULL,
  factors = "reference",
  numerics = "mean",
  preserve_range = FALSE,
  reference = x,
  ...
)

## S3 method for class 'numeric'
visualisation_matrix(x, ...)

## S3 method for class 'factor'
visualisation_matrix(x, ...)
```

Arguments

`x`	An object from which to construct the reference grid.
`...`	Arguments passed to or from other methods (for instance, `length` or `range` to control the spread of numeric variables.).
`at`	Indicates the *focal predictors* (variables) for the reference grid and at which values focal predictors should be represented. If not specified otherwise, representative values for numeric variables or predictors are evenly distributed from the minimum to the maximum, with a total number of `length` values covering that range (see 'Examples'). Possible options for `at` are: • `"all"`, which will include all variables or predictors. • a character vector of one or more variable or predictor names, like `c("Species", "Sepal.Width")`, which will create a grid of all combinations of unique values. For factors, will use all levels, for numeric variables, will use a range of length `length` (evenly spread from minimum to maximum) and for character vectors, will use all unique values.

- a list of named elements, indicating focal predictors and their representative values, e.g. `at = list(Sepal.Length = c(2, 4), Species = "setosa")`.
- a string with assignments, e.g. `at = "Sepal.Length = 2"` or `at = c("Sepal.Length = 2", "Species = 'setosa'")` - note the usage of single and double quotes to assign strings within strings.

There is a special handling of assignments with *brackets*, i.e. values defined inside `[` and `]`.For **numeric** variables, the value(s) inside the brackets should either be

- two values, indicating minimum and maximum (e.g. `at = "Sepal.Length = [0, 5]"`), for which a range of length `length` (evenly spread from given minimum to maximum) is created.
- more than two numeric values `at = "Sepal.Length = [2,3,4,5]"`, in which case these values are used as representative values.
- a "token" that creates pre-defined representative values:
  - for mean and -/+ 1 SD around the mean: `"x = [sd]"`
  - for median and -/+ 1 MAD around the median: `"x = [mad]"`
  - for Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum): `"x = [fivenum]"`
  - for terciles, including minimum and maximum: `"x = [terciles]"`
  - for terciles, excluding minimum and maximum: `"x = [terciles2]"`
  - for quartiles, including minimum and maximum: `"x = [quartiles]"`
  - for quartiles, excluding minimum and maximum: `"x = [quartiles2]"`
  - for minimum and maximum value: `"x = [minmax]"`
  - for 0 and the maximum value: `"x = [zeromax]"`

For **factor** variables, the value(s) inside the brackets should indicate one or more factor levels, like `at = "Species = [setosa, versicolor]"`. **Note**: the `length` argument will be ignored when using brackets-tokens.

The remaining variables not specified in `at` will be fixed (see also arguments `factors` and `numerics`).

`target` Deprecated name. Please use `at` instead.

`factors` Type of summary for factors. Can be `"reference"` (set at the reference level), `"mode"` (set at the most common level) or `"all"` to keep all levels.

`numerics` Type of summary for numeric values. Can be `"all"` (will duplicate the grid for all unique values), any function (`"mean"`, `"median"`, ...) or a value (e.g., `numerics = 0`).

`preserve_range` In the case of combinations between numeric variables and factors, setting `preserve_range = TRUE` will drop the observations where the value of the numeric variable is originally not present in the range of its factor level. This leads to an unbalanced grid. Also, if you want the minimum and the maximum to closely match the actual ranges, you should increase the `length` argument.

`reference` The reference vector from which to compute the mean and SD. Used when standardizing or unstandardizing the grid using `effectsize::standardize`.

Value

Reference grid data frame.

Examples

```
library(modelbased)

# Add one row to change the "mode" of Species
data <- rbind(iris, iris[149, ], make.row.names = FALSE)

# Single variable is of interest; all others are "fixed"
visualisation_matrix(data, at = "Sepal.Length")
visualisation_matrix(data, at = "Sepal.Length", length = 3)
visualisation_matrix(data, at = "Sepal.Length", range = "ci", ci = 0.90)
visualisation_matrix(data, at = "Sepal.Length", factors = "mode")

# Multiple variables are of interest, creating a combination
visualisation_matrix(data, at = c("Sepal.Length", "Species"), length = 3)
visualisation_matrix(data, at = c(1, 3), length = 3)
visualisation_matrix(data, at = c("Sepal.Length", "Species"), preserve_range = TRUE)
visualisation_matrix(data, at = c("Sepal.Length", "Species"), numerics = 0)
visualisation_matrix(data, at = c("Sepal.Length = 3", "Species"))
visualisation_matrix(data, at = c("Sepal.Length = c(3, 1)", "Species = 'setosa'"))

# with list-style at-argument
visualisation_matrix(data, at = list(Sepal.Length = c(1, 3), Species = "setosa"))

# Standardize
vizdata <- visualisation_matrix(data, at = "Sepal.Length")
standardize(vizdata)
```

---

visualisation_recipe.estimate_grouplevel

Visualisation Recipe for 'modelbased' Objects

Description

Visualisation Recipe for 'modelbased' Objects

Usage

```
## S3 method for class 'estimate_grouplevel'
visualisation_recipe(
  x,
  hline = NULL,
  pointrange = NULL,
  facet_wrap = NULL,
  labs = NULL,
  ...
)

## S3 method for class 'estimate_means'
```

```
visualisation_recipe(
  x,
  show_data = "jitter",
  point = NULL,
  jitter = point,
  boxplot = NULL,
  violin = NULL,
  line = NULL,
  pointrange = NULL,
  labs = NULL,
  ...
)

## S3 method for class 'estimate_predicted'
visualisation_recipe(
  x,
  show_data = "points",
  point = NULL,
  density_2d = NULL,
  line = NULL,
  ribbon = NULL,
  labs = NULL,
  ...
)

## S3 method for class 'estimate_slopes'
visualisation_recipe(
  x,
  hline = NULL,
  line = NULL,
  pointrange = NULL,
  ribbon = NULL,
  labs = NULL,
  facet_wrap = NULL,
  ...
)
```

Arguments

x	A modelbased object.
...	Other arguments passed to other functions.
show_data	Display the "raw" data as a background to the model-based estimation. Can be set to "none" to remove it. When input is the result of estimate_means, show_data can be "points" (the jittered observation points), "boxplot", "violin" a combination of them (see examples). When input is the result of estimate_response or estimate_relation, show_data can be "points" (the points of the original data corresponding to the x and y axes), "density_2d", "density_2d_filled", "density_2d_polygon" or "density_2d_raster".

point, jitter, boxplot, violin, pointrange, density_2d, line, hline, ribbon, labs, facet_wrap
: Additional aesthetics and parameters for the geoms (see customization example).

Examples

```
# ================================================
# estimate_grouplevel
# ================================================
if (require("see") && require("lme4")) {
  data <- lme4::sleepstudy
  data <- rbind(data, data)
  data$Newfactor <- rep(c("A", "B", "C", "D"))

  # 1 random intercept
  model <- lmer(Reaction ~ Days + (1 | Subject), data = data)
  x <- estimate_grouplevel(model)
  layers <- visualisation_recipe(x)
  layers
  plot(layers)
}

if (require("see") && require("lme4")) {
  # 2 random intercepts
  model <- lmer(Reaction ~ Days + (1 | Subject) + (1 | Newfactor), data = data)
  x <- estimate_grouplevel(model)
  plot(visualisation_recipe(x))


  model <- lmer(Reaction ~ Days + (1 + Days | Subject) + (1 | Newfactor), data = data)
  x <- estimate_grouplevel(model)
  plot(visualisation_recipe(x))
}

# ================================================
# estimate_means
# ================================================
if (require("ggplot2")) {
  # Simple Model ---------------
  x <- estimate_means(lm(Sepal.Width ~ Species, data = iris))
  layers <- visualisation_recipe(x)
  layers
  plot(layers)
}

if (require("ggplot2")) {
  # Customize aesthetics
  layers <- visualisation_recipe(x,
    jitter = list(width = 0.03, color = "red"),
    line = list(linetype = "dashed")
  )
  plot(layers)
```

```
  # Customize raw data
  plot(visualisation_recipe(x, show_data = c("violin", "boxplot", "jitter")))

  # Two levels ---------------
  data <- mtcars
  data$cyl <- as.factor(data$cyl)
  data$new_factor <- as.factor(rep(c("A", "B"), length.out = nrow(mtcars)))

  model <- lm(mpg ~ new_factor * cyl * wt, data = data)
  x <- estimate_means(model, at = c("new_factor", "cyl"))
  plot(visualisation_recipe(x))

  # Modulations --------------
  x <- estimate_means(model, at = c("new_factor", "wt"))
  plot(visualisation_recipe(x))

  x <- estimate_means(model, at = c("new_factor", "cyl", "wt"))
  plot(visualisation_recipe(x))

  #'   # GLMs ---------------------
  data <- data.frame(vs = mtcars$vs, cyl = as.factor(mtcars$cyl))
  x <- estimate_means(glm(vs ~ cyl, data = data, family = "binomial"))
  plot(visualisation_recipe(x))
}

# ================================================
# estimate_relation, estimate_response, ...
# ================================================
if (require("ggplot2")) {
  # Simple Model ---------------
  x <- estimate_relation(lm(mpg ~ wt, data = mtcars))
  layers <- visualisation_recipe(x)
  layers
  plot(layers)
}

if (require("ggplot2")) {
  # Customize aesthetics ----------

  layers <- visualisation_recipe(x,
    point = list(color = "red", alpha = 0.6, size = 3),
    line = list(color = "blue", size = 3),
    ribbon = list(fill = "green", alpha = 0.7),
    labs = list(subtitle = "Oh yeah!")
  )
  layers
  plot(layers)

  # Customize raw data -------------

  plot(visualisation_recipe(x, show_data = "none"))
  plot(visualisation_recipe(x, show_data = c("density_2d", "points")))
  plot(visualisation_recipe(x, show_data = "density_2d_filled"))
```

```
plot(visualisation_recipe(x, show_data = "density_2d_polygon"))
plot(visualisation_recipe(x, show_data = "density_2d_raster")) +
  scale_x_continuous(expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0))

# Single predictors examples -----------

plot(estimate_relation(lm(Sepal.Length ~ Sepal.Width, data = iris)))
plot(estimate_relation(lm(Sepal.Length ~ Species, data = iris)))

# 2-ways interaction ------------

# Numeric * numeric
x <- estimate_relation(lm(mpg ~ wt * qsec, data = mtcars))
layers <- visualisation_recipe(x)
plot(layers)

# Numeric * factor
x <- estimate_relation(lm(Sepal.Width ~ Sepal.Length * Species, data = iris))
layers <- visualisation_recipe(x)
plot(layers)

# Factor * numeric
x <- estimate_relation(lm(Sepal.Width ~ Species * Sepal.Length, data = iris))
layers <- visualisation_recipe(x)
plot(layers)

# 3-ways interaction ------------

data <- mtcars
data$vs <- as.factor(data$vs)
data$cyl <- as.factor(data$cyl)
data$new_factor <- as.factor(rep(c("A", "B"), length.out = nrow(mtcars)))

# Numeric * numeric * numeric
x <- estimate_relation(lm(mpg ~ wt * qsec * hp, data = data), length = c(5, 3, 20))
layers <- visualisation_recipe(x)
plot(layers)

# Numeric * numeric * factor
x <- estimate_relation(lm(mpg ~ wt * am * vs, data = data))
layers <- visualisation_recipe(x)
plot(layers)

# Numeric * factor * factor
x <- estimate_relation(lm(mpg ~ wt * cyl * new_factor, data = data))
layers <- visualisation_recipe(x)
plot(layers)

# Factor * numeric * numeric
x <- estimate_relation(lm(mpg ~ cyl * qsec * hp, data = data))
layers <- visualisation_recipe(x)
plot(layers) +
```

```
    scale_size_continuous(range = c(0.2, 1))

  # GLMs ---------------------
  x <- estimate_relation(glm(vs ~ mpg, data = mtcars, family = "binomial"))
  plot(visualisation_recipe(x))
  plot(visualisation_recipe(x, show_data = "jitter", point = list(height = 0.03)))

  # Multiple CIs ---------------------
  plot(estimate_relation(lm(mpg ~ disp, data = mtcars),
    ci = c(.50, .80, .95)
  ))
  plot(estimate_relation(lm(Sepal.Length ~ Species, data = iris),
    ci = c(0.5, 0.7, 0.95)
  ))
}

# Bayesian models ---------------------
if (require("ggplot2") && require("rstanarm")) {
  model <- rstanarm::stan_glm(mpg ~ wt, data = mtcars, refresh = 0)

  # Plot individual draws instead of regular ribbon
  x <- estimate_relation(model, keep_iterations = 100)
  layers <- visualisation_recipe(x, ribbon = list(color = "red"))
  plot(layers)

 model <- rstanarm::stan_glm(Sepal.Width ~ Species * Sepal.Length, data = iris, refresh = 0)
  plot(estimate_relation(model, keep_iterations = 100))
}

# ============================================
# estimate_slopes
# ============================================
if (require("ggplot2")) {
  model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)
  x <- estimate_slopes(model, trend = "Petal.Length", at = "Species")

  layers <- visualisation_recipe(x)
  layers
  plot(layers)

  model <- lm(Petal.Length ~ poly(Sepal.Width, 4), data = iris)
  x <- estimate_slopes(model, at = "Sepal.Width", length = 20)
  plot(visualisation_recipe(x))

  model <- lm(Petal.Length ~ Species * poly(Sepal.Width, 3), data = iris)
  x <- estimate_slopes(model, at = c("Sepal.Width", "Species"))
  plot(visualisation_recipe(x))
}
## Not run:
# TODO: fails with latest emmeans (1.8.0)
if (require("mgcv")) {
  data <- iris
  data$Petal.Length <- data$Petal.Length^2
```

```
  model <- mgcv::gam(Sepal.Width ~ t2(Petal.Width, Petal.Length), data = data)
  x <- estimate_slopes(model, at = c("Petal.Width", "Petal.Length"), length = 20)
  plot(visualisation_recipe(x))

 model <- mgcv::gam(Sepal.Width ~ t2(Petal.Width, Petal.Length, by = Species), data = data)
 x <- estimate_slopes(model, at = c("Petal.Width", "Petal.Length", "Species"), length = 10)
  plot(visualisation_recipe(x))
}

## End(Not run)
```

`zero_crossings` *Find zero crossings of a vector*

Description

Find zero crossings of a vector, i.e., indices when the numeric variable crosses 0.

Usage

```
zero_crossings(x)
```

Arguments

x	A numeric vector.

Value

Vector of zero crossings.

See Also

Based on the `uniroot.all` function from the rootSolve package.

Examples

```
x <- sin(seq(0, 4 * pi, length.out = 100))
plot(x)
zero_crossings(x)
```

Index