# Package 'easyViz'

August 21, 2025

**Title** Easy Visualization of Conditional Effects from Regression Models

**Version** 1.1.0

**Description** Offers a flexible and user-friendly interface for visualizing conditional effects from a broad range of regression models, including mixed-effects and generalized additive (mixed) models. Compatible model types include lm(), rlm(), glm(), glm.nb(), and gam() (from 'mgcv'); nonlinear models via nls(); and generalized least squares via gls(). Mixed-effects models with random intercepts and/or slopes can be fitted using lmer(), glmer(), glmer.nb(), glmmTMB(), or gam() (from 'mgcv', via smooth terms). Plots are rendered using base R graphics with extensive customization options. Approximate confidence intervals for nls() models are computed using the delta method. Robust standard errors for rlm() are computed using the sandwich estimator (Zeileis 2004) <doi:10.18637/jss.v011.i10>. Methods for generalized additive models follow Wood (2017) <doi:10.1201/9781315370279>. For linear mixed-effects models with 'lme4', see Bates et al. (2015) <doi:10.18637/jss.v067.i01>. For mixed models using 'glmmTMB', see Brooks et al. (2017) <doi:10.32614/RJ-2017-066>.

**Maintainer** Luca Corlatti <lucac1980@yahoo.it>

**Imports** stats, utils, graphics, grDevices

**Suggests** nlme, lme4, MASS, glmmTMB, mgcv, numDeriv, sandwich

**License** GPL-3

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**NeedsCompilation** no

**Author** Luca Corlatti [aut, cre]

**Repository** CRAN

**Date/Publication** 2025-08-21 19:42:05 UTC

## Contents

easyViz *Easy Visualization of Conditional Effects from Regression Models*

## Description

easyViz offers a flexible and user-friendly interface for visualizing conditional effects from a broad range of regression and mixed-effects models using base R graphics.

## Usage

```
easyViz(
  model,
  data,
  predictor,
  by = NULL,
  pred_type = "response",
  pred_range_limit = TRUE,
  pred_on_top = FALSE,
  pred_resolution = 101,
  num_conditioning = "median",
  cat_conditioning = "mode",
  fix_values = NULL,
  re.form = NULL,
  backtransform_response = NULL,
  xlim = NULL,
  ylim = NULL,
  xlab = NULL,
  ylab = NULL,
  cat_labels = NULL,
  font_family = "",
  las = 1,
  bty = "o",
  plot_args = list(),
  show_data_points = TRUE,
  binary_data_type = "plain",
  bins = 10,
  jitter_data_points = FALSE,
  point_col = rgb(0, 0, 0, alpha = 0.4),
  point_pch = 16,
  point_cex = 0.75,
  pred_line_col = "black",
  pred_line_lty = c(1, 2, 3, 4),
  pred_line_lwd = 2,
  ci_type = "polygon",
  ci_polygon_col = c("gray", "black", "lightgray", "darkgray"),
  ci_line_col = "black",
  ci_line_lty = c(1, 2, 3, 4),
```

```
  ci_line_lwd = 1,
  pred_point_col = c("black", "gray", "darkgray", "lightgray"),
  pred_point_pch = 16,
  pred_point_cex = 1,
  ci_bar_col = "black",
  ci_bar_lty = 1,
  ci_bar_lwd = 1,
  ci_bar_caps = 0.1,
  add_legend = FALSE,
  legend_position = "top",
  legend_title = NULL,
  legend_labels = NULL,
  legend_title_size = 1,
  legend_label_size = 0.9,
  legend_horiz = FALSE,
  legend_args = list()
)
```

## Arguments

`model` [required] A fitted model object (e.g., `model = your.model`). Supported models include a wide range of regression types, including linear, robust linear, non-linear, generalized least squares, generalized linear, mixed-effects, and generalized additive (mixed) models. Compatible model-fitting functions include: `stats::lm`, `MASS::rlm`, `stats::nls`, `nlme::gls`, `stats::glm`, `MASS::glm.nb`, `lme4::lmer`, `lme4::glmer`, `lme4::glmer.nb`, `glmmTMB::glmmTMB`, and `mgcv::gam`.

`data` [required] The data frame used to fit the model (e.g., `data = your.data`). This data frame is used internally for generating predictions. **All variables used in the model formula (including predictors, offsets, grouping variables, and interaction terms) must be present in this data frame**. If the model was fitted without using a `data` argument (e.g., using variables from the global environment), you must ensure that `data` includes all required variables. Otherwise, prediction may fail or produce incorrect results.

`predictor` [required] The name of the target explanatory variable to be plotted (e.g., `predictor = "x1"`).

`by` The name of an interaction or additional variable for conditioning (e.g., `by = "x2"`). If the variable is:

- continuous, predictions are shown for cross-sections at the 10th, 50th, and 90th percentiles.
- categorical, a separate prediction line or point will be plotted for each level.
- used as a grouping variable in a random effect term (e.g., `(1|group)` or `s(group, bs = "re")`) and `re.form = NULL`, predictions are conditional on each group's estimated random effect.

Although `easyViz` does not natively support direct visualization of three-way interactions in a multi-panel plot, this can be easily achieved by combining the `by` and `fix_values` arguments. For example, if your model includes a term like `x1*x2*x3`, you can visualize the effect of `x1` across levels of `x2` by

setting predictor = "x1", by = "x2", and fixing x3 at a specific value using fix_values = c(x3 = ...). Repeating this with different values of x3 produces multiple plots that can be arranged to visualize the full three-way interaction. See the Examples section for a demonstration of how to apply this approach.

pred_type
: Character string indicating the type of predictions to plot. Either "response" (default), which returns predictions on the original outcome scale by applying the inverse of the model's link function (e.g., probabilities for binary models), or "link", which returns predictions on the linear predictor (link) scale (e.g., log-odds, log-counts, or other transformed scales depending on the model).

pred_range_limit
: Logical. Applies only when the predictor is numeric and a categorical by variable is specified. If TRUE (default), the prediction range for each level of the by variable is limited to the range of the predictor observed within that level. This avoids extrapolating predictions beyond the available data for each subgroup. If FALSE, predictions span the entire range of the predictor across all levels of the by variable. If the by variable is numeric, pred_range_limit is automatically set to FALSE, since numeric by values are treated as continuous rather than grouping factors.

pred_on_top
: Logical. If TRUE, prediction lines (and their confidence intervals) for numeric predictors are drawn after raw data, so they appear on top. Default is FALSE, which draws predictions underneath the data. This has no effect for categorical predictors — for those, predictions are always drawn on top of raw data.

pred_resolution
: Number of prediction points to use for numeric predictors. Defaults to 101, consistent with visreg. The default should work well in most cases. Increasing pred_resolution may be particularly helpful when the predictor spans a wide range or when visualizing nonlinear relationships (e.g., splines or polynomials), to ensure smooth and accurate rendering of the effect. **Note:** A higher value may slightly increase computation time, especially when combined with many levels of a by variable.

num_conditioning
: How to condition non-target numeric predictors. Either "median" (default) or "mean". This determines how numeric variables that are not directly plotted are held constant during prediction, while varying the predictor of interest — a common approach when visualizing effects in multivariable models. To fix specific variables at custom values instead, use the fix_values argument.

cat_conditioning
: How to condition non-target categorical predictors. Either "mode" (default) or "reference". As for "num_conditioning", conditioning means holding these variables constant while varying the predictor of interest. If multiple levels are equally frequent when "mode" is selected, the level chosen will be the first in the factor's level order (which by default is alphabetical and typically coincides with the reference level, unless explicitly re-leveled). This behavior also applies to grouping variables used as random effects when re.form = NULL. To fix categorical variables (including grouping variables) at specific levels, use fix_values.

fix_values
: A named vector or named list specifying fixed values for one or more variables during prediction. Supports both numeric and categorical variables. For

numeric variables, specify a fixed value (e.g., `fix_values = c(x = 1)`). For categorical variables (factors), provide the desired level as a character string or factor (e.g., `fix_values = c(group = "levelA")` or `fix_values = list(group = levels(data$group)[1])`). This overrides the default conditioning behavior specified via `num_conditioning` and `cat_conditioning`. **Note:** This argument also applies to grouping variables used as random effects: when `re.form = NULL`, predictions are conditional on the level specified in `fix_values`; if not specified, the level is chosen based on `cat_conditioning`. This argument is useful for setting offsets, forcing predictions at specific values, or ensuring consistent conditioning across models. For example, it is particularly useful when you want to visualize the effect of a predictor at a specific level of an interacting variable, without conditioning on all levels. E.g., to plot the conditional effect of a continuous predictor `x1` at a specific value of another variable `x2` (numeric or categorical), simply set `fix_values = c(x2 = ...)` and omit the `by` argument. This creates a clean single-effect plot for `x1` at the desired level of `x2`, without plotting multiple lines or groups as `by` would. This argument can also be used to visualize three-way interactions when combined with `by`. See the `by` argument for details, and the Examples section for a demonstration of how to apply this approach.

`re.form` A formula specifying which random effects to include when generating predictions. This argument is relevant for mixed-effects models only (e.g., from `lme4`, `glmmTMB`, or `mgcv::gam()`).

- `re.form = NULL` (default): produces group-specific predictions, conditional on the random-effect levels present in the data. By default, `easyViz` fixes grouping variables at their mode (i.e., the most frequent level), so the prediction reflects the conditional estimate for that group. You can override this by explicitly fixing the grouping variable via `fix_values` (e.g., `fix_values = c(group = "levelA")`). If all levels are equally frequent and no value is specified, the first level (in factor order) is used, which is usually alphabetical unless re-leveled. If `by` corresponds to a grouping variable used in a random effect, predictions are visualized for all group levels (i.e., conditional predictions).
- `re.form = NA` or `re.form = ~0`: produces population-level (i.e., marginal) predictions by excluding random effects from the prediction step. The random effects are still part of the fitted model and influence the estimation of fixed effects and their uncertainty, but they are not included when computing predicted values. This is equivalent to assuming random effects are zero — representing an "average" group or subject.

For `mgcv::gam()` models, random effects are typically modeled using smooth terms like `s(group, bs = "re")`. Although `predict.gam()` does not support a `re.form` argument, `easyViz` emulates its behavior: `re.form = NULL` includes random-effect smooths, while `re.form = NA` or `~0` excludes them via the `exclude` argument in `predict.gam()`. **Note:** For models fitted with `lme4` (e.g., `lmer()`, `glmer()`), standard errors are not available when `re.form = NULL`.

`backtransform_response`

A custom function to back-transform predictions for transformed response variables (e.g., `exp` for log-transformed responses, or `function(x) x^2` for square

root-transformed responses). **Note:** If you wish to model a transformed response, it is recommended to apply the transformation directly in the model formula (e.g., `log(y)`), rather than modifying the response variable in the data set. This ensures that observed data points are correctly plotted on the original (back-transformed) scale. Otherwise, raw data and predicted values may not align properly in the plot.

`xlim` x-axis limits for the plot (e.g., `xlim = c(0, 10)`). Defaults to automatic scaling based on the data range. Applies to both numeric and categorical predictors. For categorical variables, x-axis positions are treated as integer values (e.g., 1, 2, ..., k), and adjusting `xlim` (e.g., `xlim = c(0.5, k + 0.5)`) can control spacing and margins around the plotted levels.

`ylim` y-axis limits for the plot (e.g., `ylim = c(10, 20)`). Defaults to automatic scaling based on the data and prediction range.

`xlab` x-axis labels (e.g., `xlab = "x"`). Defaults to `"predictor"`.

`ylab` y-axis labels (e.g., `ylab = "y"`). Defaults to `"response"`.

`cat_labels` Custom labels for levels of a categorical predictor (e.g., `cat_labels = c("Level A", "Level B", "Level C")`).

`font_family` Font family for the plot. E.g., `"sans"` (default), `"serif"`, `"mono"`.

`las` Text orientation for axis labels (default: `1`).

`bty` Box type around the plot. E.g., `"o"` (default), `"n"`, `"L"`.

`plot_args` A named list of additional graphical parameters passed to base R's `plot()` function. These arguments allow users to override default appearance settings in a flexible way. Common options include axis label size, color, label text, tick mark spacing, and coordinate scaling. **Note:** Only arguments recognized by `plot.default()` are supported. Parameters that must be set via `par()` (such as `mar`, `oma`, `mfrow`, `mgp`) are *not* applied through `plot_args`. If you wish to adjust those settings, set them directly using `par()` before calling `easyViz()`. Many valid parameters are documented in both `?plot.default` and `?par`. In `plot_args`, they are passed to `plot()`, not to `par()`. Common `plot()` parameters you may override:

- Label/Text size and style: `cex.lab`, `cex.axis`, `cex.main`, `font.lab`, `font.axis`, `font.main`
- Colors: `col.lab`, `col.axis`, `col.main`, `col.sub`, `col`, `bg`, `fg`
- Label/Text content: `xlab`, `ylab`, `main`, `sub`
- Box and axis rendering: `bty`, `axes`, `frame.plot`, `ann`
- Coordinate settings and tick spacing: `xlim`, `ylim`, `xaxs`, `yaxs`, `xaxp`, `yaxp`, `asp`, `xlog`, `ylog`

For a full list of supported parameters, see `?plot.default` and `?par`. Example usage:
`plot_args = list(main = "Title", cex.lab = 1.2, col.axis = "gray40", xaxp = c(0, 10, 5))`.

`show_data_points`
Logical. Whether to display raw data points (default: `TRUE`). For binomial models where the response is expressed in the formula as `cbind(successes,`

failures) or as successes / trials, the raw data points plotted on the y-axis are based on the calculated proportions: successes / (successes + failures) or successes / trials, respectively. These proportions are computed internally from the original data and temporarily added to the data set for visualization purposes.

binary_data_type
: For binary responses, how to display raw data points in the plot. Either "plain" (default), which plots each individual 0/1 observation as-is, or "binned", which groups observations into intervals (bins) of the predictor and plots the proportion of 0s and 1s within each bin. This makes it easier to visualize trends in binary outcomes, especially when many points overlap.

bins
: Number of bins for displaying binary response raw data when binary_data_type = "binned" (default: 10).

jitter_data_points
: Logical. If TRUE, raw data points are jittered horizontally to reduce overplotting. Applies to both categorical and numeric predictors. Default is FALSE. For categorical predictors, jittering helps distinguish overlapping points. For numeric predictors, it can be useful when many data points share the same x-value (e.g., integers or rounding).

point_col
: Point color for raw data (default: rgb(0,0,0, alpha = 0.4)). Can be specified as a color name (e.g., "gray"), an integer (e.g., 1), or an RGB (e.g., rgb(0,0,0,alpha=0.4)) or hex string (e.g., "#808080"). Dynamic: accepts multiple values when points are plotted for different values/levels of a variable. **Tip:** For large data sets with many overlapping data points, it is recommended to use semi-transparent colors to reduce overplotting. You can achieve this by setting a low alpha value (e.g., rgb(1,0,0, alpha = 0.1), or by using adjustcolor() with the argument alpha.f (e.g., adjustcolor("red", alpha.f = 0.1)). In such cases, consider setting pred_on_top = TRUE to ensure that prediction lines and confidence intervals remain clearly visible above the dense cloud of raw points.

point_pch
: Point shape for raw data (default: 16). Dynamic: accepts multiple values when points are plotted for different values/levels of a variable.

point_cex
: Point size for raw data (default: 0.75). Dynamic: accepts multiple values when points are plotted for different values/levels of a variable.

pred_line_col
: Color of the predicted line for numerical predictors (default: "black"). Can be specified as a color name, number or RGB/hex string. Dynamic: accepts multiple values (e.g., c("red", "green", "blue")) when multiple lines are plotted (i.e., when by is specified).

pred_line_lty
: Type of the predicted line for numerical predictors (default: 1). Dynamic: accepts multiple values (e.g., c(1, 2, 3)) when multiple lines are plotted (i.e., when by is specified).

pred_line_lwd
: Width of the predicted line for numerical predictors (default: 2). Dynamic: accepts multiple values (e.g., c(1, 2, 3)) when multiple lines are plotted (i.e., when by is specified).

ci_type
: Type of 95 percent confidence intervals for numeric predictors. Either "polygon" (default) to draw shaded confidence bands, "lines" to draw lines, or NULL to

suppress confidence intervals for numeric predictors. **Note:** `ci_type = NULL` does *not* suppress confidence bars for categorical predictors; these are always shown unless manually suppressed via custom logic (e.g., by setting `ci_bar_lwd = 0`).

`ci_polygon_col` Color for 95 percent confidence interval polygon (default: `"gray"`). Requires `ci_type = "polygon"`. Can be specified as a color name, number or RGB/hex string. Dynamic: accepts multiple values (e.g., `c("red", "green", "blue")`) when 95 percent CIs are plotted for multiple lines (i.e., when by is specified).

`ci_line_col` Color for 95 percent confidence interval lines (default: `"black"`). Requires `ci_type = "lines"`. Can be specified as a color name, number or RGB/hex string. Dynamic: accepts multiple values (e.g., `c("red", "green", "blue")`) when 95 percent CIs are plotted for multiple lines (i.e., when by is specified).

`ci_line_lty` Type for 95 percent confidence interval lines (default: `1`). Requires `ci_type = "lines"`. Dynamic: accepts multiple values (e.g., `c(1, 2, 3)`) when 95 percent CIs are plotted for multiple lines (i.e., when by is specified).

`ci_line_lwd` Width for 95 percent confidence interval lines (default: `1`). Requires `ci_type = "lines"`. Dynamic: accepts multiple values (e.g., `c(1, 2, 3)`) when 95 percent CIs are plotted for multiple lines (i.e., when by is specified).

`pred_point_col` Color for predicted point values of categorical predictors (default: `"black"`). Can be specified as a color name, number or RGB/hex string. Dynamic: accepts multiple values (e.g., `c("red", "green", "blue")`) when points are plotted for an interaction (i.e., when by is specified).

`pred_point_pch` Shape for predicted point values of categorical predictors (default: `16`). Dynamic: accepts multiple values (e.g., `c(1, 2, 3)`) when points are plotted for an interaction (i.e., when by is specified).

`pred_point_cex` Size for predicted point values of categorical predictors (default: `1`). Dynamic: accepts multiple values (e.g., `c(1, 2, 3)`) when points are plotted for an interaction (i.e., when by is specified).

`ci_bar_col` Color for 95 percent confidence interval bars (default: `"black"`). Applies only when the predictor is categorical. Can be specified as a color name, number, or RGB/hex string.

`ci_bar_lty` Type for 95 percent confidence interval bars (default: `1`). Applies only when the predictor is categorical.

`ci_bar_lwd` Width for 95 percent confidence interval bars (default: `1`). Applies only when the predictor is categorical. To suppress confidence interval bars, set `ci_bar_lwd = 0` (line width of zero).

`ci_bar_caps` Size of the caps on 95 percent confidence interval bars (default: `0.1`). Increase for more visible caps, set to 0 to remove caps and draw plain vertical bars.

`add_legend` Logical. Whether to add a legend for by variable levels (default: `FALSE`).

`legend_position`

Legend position. Either a named position string (`"top"`, `"bottom"`, `"left"`, `"right"`, `"topleft"`, `"topright"`, `"bottomleft"`, `"bottomright"`), or a numeric vector `c(x, y)` specifying exact coordinates for manual placement. In base `plot()` in R, there is no direct way to place the legend truly outside the plot area (e.g., to the right of the axis and tick marks). A workaround, especially

effective when bty = "n", is to expand the xlim or ylim range, optionally adjust tick spacing using plot_args = list(xaxp = ...) or yaxp = ..., and set legend_position = c(x, y) to manually locate the legend within the expanded plotting coordinates. When placing the legend in an expanded region, note that axis labels may become misaligned; for best results, suppress them with xlab = "" or ylab = "" and add them manually using text() after the plot is drawn. See the Examples section for a demonstration of this approach. Alternatively, you can enable drawing outside the plot region by setting par(xpd = TRUE) and increasing the plot margins using par(mar = ...). In this case, the legend must be added manually *after* the call to easyViz(), as it will fall outside the plot area managed by the function.

legend_title — Optional character string. If specified, this will appear as the title of the legend. In this case, the legend labels will correspond to the levels of the by variable (e.g., "A", "B", "C"). If left as NULL (default), the legend labels will follow standard behavior (e.g., "by = level"). In any case, they can be manually specified via legend_labels.

legend_labels — Custom labels for the legend (e.g., legend_labels = c("Level A", "Level B", "Level C")).

legend_title_size — Numeric. Text size for the legend title (default: 1).

legend_label_size — Numeric. Text size for the legend labels (default: 0.9).

legend_horiz — Logical. If TRUE, the legend is drawn horizontally (side-by-side) instead of vertically (stacked). Defaults to FALSE. Useful for placing the legend above or below the plot in a compact layout.

legend_args — A named list of additional arguments passed to base R's legend() function. These allow fine-tuned control over the appearance and placement of the legend and override the high-level options provided by legend_position, legend_title and other legend_* arguments. For example, you can adjust the legend's box style, border color, spacing, point size, or background color. Common options include:

- Point and line appearance: pch, col, pt.cex, pt.lwd, lty, lwd
- Layout and spacing: ncol, x.intersp, y.intersp, inset, xjust, yjust
- Text style and color: cex, text.col, font, adj
- Box and background: bty, box.lwd, box.col, bg
- Title control: title, title.col, title.cex, title.adj

For a full list of supported parameters, see ?legend. Example usage: legend_args = list(bty = "o", box.col = "black", pt.cex = 1.5). **Tip:** If you're adding a legend outside the plot region using par(xpd = TRUE), you must call legend() manually *after* easyViz() to place it correctly. Use legend_args only for legends drawn *inside* the plot area.

## Details

This function provides an easy-to-use yet highly flexible tool for visualizing conditional effects from a wide range of regression models, including mixed-effects and generalized additive (mixed)

models. Compatible model types include `lm`, `rlm`, `glm`, `glm.nb`, and `mgcv::gam`; nonlinear models via `nls`; and generalized least squares via `gls`. Mixed-effects models with random intercepts and/or slopes can be fitted using `lmer`, `glmer`, `glmer.nb`, `glmmTMB`, or `mgcv::gam` (via smooth terms). The function handles nonlinear relationships (e.g., splines, polynomials), two-way interactions, and supports visualization of three-way interactions via conditional plots. Plots are rendered using base R graphics with extensive customization options available through the `plot_args` and `legend_args` argument. Users can pass any valid graphical parameters accepted by `plot`, `par` or `legend` enabling full control over axis/legend labels, font styles, colors, margins, and more.

**Tip:** To customize plot appearance, look for argument names by prefix: Arguments starting with `point_` control the appearance of raw data. Arguments starting with `pred_` control the appearance of predicted values (lines or points). Arguments beginning with `ci_` adjust the display of confidence intervals (polygons, lines or bars). Arguments beginning with `legend_` control the appearance of the legend. This naming convention simplifies styling: just type the prefix (`point`, `pred`, `ci`, or `legend`) to discover relevant arguments.

The arguments `model`, `data`, and `predictor` are required. The function will return an error if any of them is missing or invalid.

## Value

A base R plot visualizing the conditional effect of a predictor on the response variable. Additionally, a data frame is invisibly returned containing the predictor values, conditioning variables, predicted values (`fit`), and their 95 percent confidence intervals (`lower`, `upper`). To extract prediction data for further use (e.g., custom plotting or tabulation), assign the output to an object: `pred_df <- easyViz(...)`. You can then inspect it using `head(pred_df)` or save it with `write.csv(pred_df, ...)`.

## Examples

```
#----------------------------------------
# Load required packages
#----------------------------------------

library(nlme)
library(MASS)
library(lme4)
library(glmmTMB)
library(mgcv)

#----------------------------------------
# Simulate dataset
#----------------------------------------
set.seed(123)
n <- 100
x1 <- rnorm(n)
x2 <- rnorm(n)
x3 <- runif(n, 0, 5)
x4 <- factor(sample(letters[1:3], n, replace = TRUE))
group_levels <- paste0("G", 1:10)
group <- factor(sample(group_levels, n, replace = TRUE))
```

```
# Generate random intercepts for each group
group_effects <- rnorm(length(group_levels), mean = 0, sd = 2)  # non-zero variance
names(group_effects) <- group_levels
group_intercept <- group_effects[as.character(group)]

# Non-linear continuous response
true_y <- 5 * sin(x3) + 3 * x1 + group_intercept + model.matrix(~x4)[, -1] %*% c(2, -2)
noise <- rnorm(n, sd = 3)
y <- as.vector(true_y + noise)

# Binary response with group effect added to logit
logit_p <- 2 * x1 - 1 + group_intercept
p <- 1 / (1 + exp(-logit_p))
binary_y <- rbinom(n, size = 1, prob = p)

# Binomial response: number of successes and failures
y3 <- sample(10:30, n, replace = TRUE)
logit_p_prop <- -1.5 * scale(x1)
p_prop <- 1 / (1 + exp(-logit_p_prop))
y1 <- rbinom(n, size = y3, prob = p_prop) # successes
y2 <- y3 - y1  # failures

# Count response with group effect in log(mu)
mu_count <- exp(1 + 0.8 * x2 - 0.5 * (x4 == "b") + group_intercept)
size <- 1.2
count_y <- rnbinom(n, size = size, mu = mu_count)
# Offset variable
offset_var <- log(runif(n, 1, 10))

# Assemble dataset
sim_data <- data.frame(x1, x2, x3, x4, group, y, binary_y, y1, y2, y3, count_y, offset_var)

#-----------------------------------------
# 1. Linear model (lm)
#-----------------------------------------
mod_lm <- lm(y ~ x1 + x4,
             data = sim_data)
easyViz(model = mod_lm, data = sim_data, predictor = "x1",
        by = "x4",
        pred_range_limit = FALSE,
        pred_on_top = TRUE,
        bty = "n",
        ylim = c(-12,18),
        xlab = "Predictor x1",
        ylab = "Response y",
        point_col = ifelse(sim_data$x4=="a", "red",
                           ifelse(sim_data$x4=="b", "orange",
                                  "yellow")),
        point_cex = 0.5,
        pred_line_col = c("red", "orange", "yellow"),
        pred_line_lty = 1,
        ci_polygon_col = c(rgb(1,0,0,0.5),
                           rgb(1,0.5,0,0.5),
```

```
                        rgb(1,1,0,0.5)),
        add_legend = TRUE,
        legend_position = "top",
        legend_title = "Predictor x4",
        legend_labels = c("a", "b", "c"),
        legend_horiz = TRUE,
        legend_args = list(pch = 16))

mod_lm2 <- lm(sqrt(x3) ~ x1 * x4,
              data = sim_data)
easyViz(model = mod_lm2, data = sim_data, predictor = "x1",
        by="x4",
        backtransform_response = function(x) x^2,
        ylim = c(0,8),
        show_data_points = FALSE,
        add_legend = TRUE)

mod_lm3 <- lm(y ~ poly(x3, 3),
              data = sim_data)
easyViz(model = mod_lm3, data = sim_data, predictor = "x3",
        pred_on_top = TRUE,
        font_family = "mono",
        point_col = rgb(1,0,0,0.3),
        point_pch = "+",
        ci_type = "lines",
        ci_line_lty = 2)

# Extract prediction data
pred_df <- easyViz(model = mod_lm, data = sim_data, predictor = "x1", by = "x4")
head(pred_df)

#-----------------------------------------
# 2. Robust linear model (rlm)
#-----------------------------------------
mod_rlm <- rlm(y ~ x1 + x4,
               data = sim_data)
easyViz(model = mod_rlm, data = sim_data, predictor = "x1",
        by = "x4",
        pred_on_top = TRUE,
        bty = "n",
        xlim = c(-2.2,3.5), # extend x-axis limits
        xlab = "", # temporarily remove x-axis label
        ylab = "Response y",
        plot_args = list(xaxp=c(-2, 2, 4)), # set tick marks
        point_col = ifelse(sim_data$x4=="a", "red",
                           ifelse(sim_data$x4=="b", "orange",
                                  "yellow")),
        point_cex = 0.5,
        pred_line_col = c("red", "orange", "yellow"),
        pred_line_lty = 1,
        ci_polygon_col = c(rgb(1,0,0,0.5),
                           rgb(1,0.5,0,0.5),
                           rgb(1,1,0,0.5)),
```

```
        add_legend = TRUE,
        legend_position = c(2.25,13),
        legend_title = "Predictor x4",
        legend_title_size = 0.9,
        legend_labels = c("a", "b", "c"),
        legend_horiz = FALSE,
        legend_args = list(pch = 16))
# Then manually add centered x-axis label
text(x = 0, y = -18.2, labels = "Predictor x1", xpd = NA)

#----------------------------------------
# 3. Generalized least squares (gls)
#----------------------------------------
mod_gls <- gls(y ~ x1 + x2 + x4,
               correlation = corAR1(form = ~1|group),
               data = sim_data)
easyViz(model = mod_gls, data = sim_data, predictor = "x4",
        jitter_data_points = TRUE,
        bty = "n",
        xlab = "Predictor x4",
        ylab = "Response y",
        point_col = rgb(0,0,1,0.2),
        pred_point_col = "blue",
        cat_labels = c("group A", "group B", "group C"))

sim_data$x5 <- sample(c(rep("CatA", 50), rep("CatB", 50)))
mod_gls2 <- gls(y ~ x1 + x2 + x4 * x5,
                correlation = corAR1(form = ~1|group),
                data = sim_data)
easyViz(model = mod_gls2, data = sim_data, predictor = "x4",
        by = "x5",
        jitter_data_points = TRUE,
        bty = "n",
        ylim = c(-15,15),
        xlim=c(0.75,4), # extend x-axis limits
        xlab = "", # temporarily remove x-axis label
        ylab = "Response y",
        cat_labels = c("group A", "group B", "group C"),
        point_col = c(rgb(0,0,1,0.2), rgb(1,0,0,0.2)),
        pred_point_col = c("blue", "red"),
        ci_bar_caps = 0,
        add_legend = TRUE,
        legend_position = "topright",
        legend_args = list(title = "Predictor x5",
                           title.cex = 1,
                           legend = c("A", "B"),
                           pt.cex = 1.5,
                           horiz = TRUE))
# Then manually add centered x-axis label
text(x = 2, y = -23.2, labels = "Predictor x4", xpd = NA)

#----------------------------------------
# 4. Nonlinear least squares (nls)
```

```
#----------------------------------------
mod_nls <- nls(y ~ a * sin(b * x3) + c,
               data = sim_data,
               start = list(a = 5, b = 1, c = 0))
summary(mod_nls)
easyViz(model = mod_nls, data = sim_data, predictor = "x3",
        pred_on_top = TRUE,
        font_family = "serif",
        bty = "n",
        xlab = "Predictor x3",
        ylab = "Response y",
        point_col = rgb(0,1,0,0.7),
        point_pch = 1,
        ci_type = "lines",
        ci_line_col = "black",
        ci_line_lty = 2)
text(x = 2.5, y = 11,
     labels = expression(Y %~% 5.31584 %*% sin(1.08158 %*% X[3]) + 0.51338),
     cex = 0.7)

#----------------------------------------
# 5. Generalized linear model (glm)
#----------------------------------------
mod_glm <- glm(binary_y ~ x1 + x4 + offset(log(offset_var)),
               family = binomial(link="cloglog"),
               data = sim_data)
easyViz(model = mod_glm, data = sim_data, predictor = "x1",
        fix_values = list(x4="b", offset_var=1),
        xlab = "Predictor x1",
        ylab = "Response y",
        binary_data_type = "binned",
        point_col = "black",
        ci_polygon_col = "red")

easyViz(model = mod_glm, data = sim_data, predictor = "x4",
        bty = "n",
        xlab = "Predictor x4",
        ylab = "Response y",
        binary_data_type = "plain",
        jitter_data_points = TRUE,
        point_col = "black",
        point_pch = "|",
        point_cex = 0.5)

mod_glm2 <- glm(y1/y3 ~ x1 + x4, weights = y3,
                family = binomial(link="logit"),
                data = sim_data)
easyViz(model = mod_glm2, data = sim_data, predictor = "x1",
        pred_on_top = TRUE,
        xlab = "Predictor x1",
        ylab = "Response y",
        point_col = "black",
        ci_polygon_col = "red")
```

```
#----------------------------------------
# 6. Negative binomial GLM (glm.nb)
#----------------------------------------
mod_glm_nb <- glm.nb(count_y ~ x2,
                     data = sim_data)
easyViz(model = mod_glm_nb, data = sim_data, predictor = "x2",
        font_family = "mono",
        bty = "L",
        plot_args = list(main = "NB model"),
        xlab = "Predictor x2",
        ylab = "Response y",
        ci_polygon_col = "blue")

#----------------------------------------
# 7. Linear mixed-effects model (lmer)
#----------------------------------------
mod_lmer <- lmer(y ~ x1 + x4 + (1 | group),
                 data = sim_data)
easyViz(model = mod_lmer, data = sim_data, predictor = "x1",
        by="group",
        re.form = NULL,
        bty = "n",
        plot_args = list(xaxp = c(round(min(sim_data$x1),1),
                                  round(max(sim_data$x1),1), 5)),
        ylim = c(-15, 15),
        xlab = "Predictor x1",
        ylab = "Response y",
        pred_line_col = "green",
        pred_line_lty = 1,
        pred_line_lwd = 1)
oldpar <- par(new = TRUE)
easyViz(model = mod_lmer, data = sim_data, predictor = "x1",
        re.form = NA,
        bty = "n",
        plot_args = list(xaxp = c(round(min(sim_data$x1),1),
                                  round(max(sim_data$x1),1), 5)),
        show_data_points = FALSE,
        xlab = "Predictor x1",
        ylab = "Response y",
        ylim = c(-15, 15),
        pred_line_col = "red",
        pred_line_lty = 1,
        pred_line_lwd = 2,
        ci_type = NULL)
par(oldpar)

#----------------------------------------
# 8. Generalized linear mixed model (glmer)
#----------------------------------------
mod_glmer <- glmer(binary_y ~ x1 + x4 + (1 | group),
                   family = binomial,
                   data = sim_data)
```

```
easyViz(model = mod_glmer, data = sim_data, predictor = "x1",
        by = "group",
        re.form = NULL,
        cat_conditioning = "reference",
        font_family = "serif",
        xlab = "Predictor x1",
        ylab = "Response y",
        binary_data_type = "binned",
        pred_range_limit = FALSE,
        pred_line_col = "blue",
        pred_line_lty = 1,
        pred_line_lwd = 1)

#----------------------------------------
# 9. GLMM with negative binomial (glmer.nb)
#----------------------------------------
mod_glmer_nb <- glmer.nb(count_y ~ x2 + x4 + (1 | group),
                         data = sim_data)
easyViz(model = mod_glmer_nb, data = sim_data, predictor = "x2",
        re.form = NA,
        bty = "n",
        xlab = "Predictor x2",
        ylab = "Response y",
        ylim = c(0, 120),
        point_pch = 1)

#----------------------------------------
# 10. GLMM using glmmTMB
#----------------------------------------
mod_glmmTMB <- glmmTMB(count_y ~ x2 + x4 + (1 | group),
                       ziformula = ~ x2,
                       family = nbinom2,
                       data = sim_data)
easyViz(model = mod_glmmTMB, data = sim_data, predictor = "x2",
        re.form = NA,
        bty = "n",
        xlab = "Predictor x2",
        ylab = "Response y",
        ylim = c(0, 120),
        point_pch = 1,
        ci_type = NULL)

#----------------------------------------
# 11. GAM with random smooth for group
#----------------------------------------
mod_gam <- gam(y ~ s(x3) + s(group, bs = "re"),
               data = sim_data)
easyViz(model = mod_gam, data = sim_data, predictor = "x3",
        re.form = NA,
        las = 0,
        bty = "n",
        xlab = "Predictor x3",
        ylab = "Response y",
```

```
        point_col = "black",
        point_pch = 1,
        ci_polygon_col = rgb(1,0,0,0.5))

#-----------------------------------------
# 12. Plotting 3-way interaction
#-----------------------------------------
mod_lm_int <- lm(y ~ x1*x2*x3,
                 data = sim_data)

# Check conditional values to use for plotting
quantile(x2, c(0.1,0.5, 0.9))
quantile(x3, c(0.1,0.5, 0.9))

# (optional) Generate a customizable function to add a strip label at the top
add_strip_label <- function(label, bg = "grey90", cex = 1, font = 2, height_mult = 2.5) {
  usr <- par("usr")
  x_left <- usr[1]
  x_right <- usr[2]
  y_top <- usr[4]
  # Estimate strip height using text height
  h <- strheight(label, cex = cex) * height_mult
  # Strip coordinates (extending above the plotting region)
  y_bottom <- y_top + 0.2 * h
  y_top_box <- y_bottom + h
  # Draw the full-width strip
  rect(x_left, y_bottom, x_right, y_top_box, col = bg, border = "black", xpd = NA)
  # Add centered text
  text(x = mean(c(x_left, x_right)),
       y = mean(c(y_bottom, y_top_box)),
       labels = label, cex = cex, font = font, xpd = NA)
}

# par settings for multi-panel plot
old_mfrow <- par(mfrow = c(1, 3))
old_oma <- par(oma = c(4, 4, 2, 1))
old_mar <- par(mar = c(0, 0, 2, 0))

# Panel 1
easyViz(model = mod_lm_int, data = sim_data, predictor = "x1",
        by = "x2",
        fix_values = c(x3 = 0.5750978),
        plot_args = list(xlab = "", ylab = ""),
        show_data_points = FALSE,
        pred_line_col = c(2, 3, 4),
        ci_polygon_col = c(2, 3, 4),
        add_legend = TRUE,
        legend_position = "topleft",
        legend_labels = c("x2 = -1.3", "x2 = -0.2", "x2 = 1.5"))
add_strip_label("x3 = 0.6")
mtext("Response y", side = 2, outer = TRUE, line = 2.5)

# Panel 2
```

```
easyViz(model = mod_lm_int, data = sim_data, predictor = "x1",
        by = "x2",
        fix_values = c(x3 = 2.3095046),
        plot_args = list(yaxt = "n", xlab = "", ylab = ""),
        show_data_points = FALSE,
        pred_line_col = c(2, 3, 4),
        ci_polygon_col = c(2, 3, 4))
add_strip_label("x3 = 2.3")

# Panel 3
easyViz(model = mod_lm_int, data = sim_data, predictor = "x1",
        by = "x2",
        fix_values = c(x3 = 4.4509078),
        plot_args = list(yaxt = "n", xlab = "", ylab = ""),
        show_data_points = FALSE,
        pred_line_col = c(2, 3, 4),
        ci_polygon_col = c(2, 3, 4))
add_strip_label("x3 = 4.5")
mtext("Predictor x1", side = 1, outer = TRUE, line = 2.5)

# Restore original settings
par(old_mfrow)
par(old_oma)
par(old_mar)

#-------------END OF EXAMPLES--------------
```

# Index