Package: Gmisc (via r-universe)

November 1, 2024

Version 3.0.3

Title Descriptive Statistics, Transition Plots, and More

Author Max Gordon <`max@gforge.se`>

Maintainer Max Gordon <`max@gforge.se`>

Description Tools for making the descriptive ``Table 1'' used in medical articles, a transition plot for showing changes between categories (also known as a Sankey diagram), flow charts by extending the grid package, a method for variable selection based on the SVD, Bézier lines with arrows complementing the ones in the 'grid' package, and more.

License GPL (>= 3)

URL `https://gforge.se`

BugReports `https://github.com/gforge/Gmisc/issues`

Biarch yes

Depends R (>= 4.1.0), Rcpp (>= 0.11.4), htmlTable (>= 2.0.0)

Imports abind, checkmate, forestplot, Hmisc, glue, grid, grDevices, graphics, knitr, lattice, lubridate, magrittr, methods, rlang, rmarkdown, stringr, stats, XML, yaml, utils

Suggests datasets, dplyr, jsonlite, testthat, tidyselect

Encoding UTF-8

NeedsCompilation no

VignetteBuilder knitr

LinkingTo Rcpp

RoxygenNote 7.2.2

Repository https://gforge.r-universe.dev

RemoteUrl https://github.com/gforge/gmisc

RemoteRef HEAD

RemoteSha 968d4edfd84453c33c483d5a1f1f0825fae98c8e

Contents

`Gmisc-package` *Collection of functions for plotting relations, generating tables, and more.*

Description

This is a collection of functions that I've found useful in my research. The package is inspired by Frank Harrell's **Hmisc** package. The main focus is on tables, plots, and **knitr**-integration.

Awesome tables

The `getDescriptionStatsBy` is a straight forward function that aims at helping you to generate descriptive table stratified by different variables. In other words, the function returns everything you need for generating a *Table 1* ready for publication. This function is accompanied by the `describeMean`, `describeMedian`, `describeProp`, and `describeFactors` functions.

The `mergeDesc` allows you to merge a set of outputs `getDescriptionStatsBy` into a `htmlTable` with the rgroup arguments automatically generated, see `vignette("descriptives", package = "Gmisc")` for a detailed workflow description.

Some fancy plots

The transition plot function, `transitionPlot`, is for descriptive purposes. It tries to illustrate the size of change between one state and the next, i.e. a transition. This is basically a graph of based upon `table(var1, var2)`.

The Singular value decomposition is a common method for reducing the number of variables. Unfortunately this compression can reduce the interpretability of the model. The `getSvdMostInfluential` function tries to remedy that by identifying the most influential elements from the V-matrix.

Other stuff

The `insertRowAndKeepAttr` simply adds a row while remembering all the attributes previously set by using the `copyAllNewAttributes`. The `mergeLists` tries to merge lists that do not have identical elements.

`align` *Align boxes*

Description

Aligns a set of boxGrob/boxPropGrob according to the first positional argument.

Usage

```
alignVertical(reference, ..., .position = c("center", "top", "bottom"))

alignHorizontal(
  reference,
  ...,
  .position = c("center", "left", "right"),
  .sub_position = c("none", "left", "right")
)
```

Arguments

`reference`	A boxGrob/boxPropGrob/coords object or a `unit` or a numerical value that can be converted into a `unit` of npc type.
`...`	A set of boxes.
`.position`	How to align the boxes, differs slightly for vertical and horizontal alignment see the accepted arguments
`.sub_position`	When the box is a boxPropGrob it not only has the general `.positions` but also `left` and `right` which can be viewed as separate boxes that have simply been merged.

Value

`list` with the boxes that are to be aligned

See Also

Other flowchart components: `boxGrob()`, `boxPropGrob()`, `connectGrob()`, `coords()`, `distance()`, `moveBox()`, `spread`

Examples

```
library(grid)
grid.newpage()

box <- boxGrob("A cool\nreference\nbox",
               x = .5, y = .8,
               box_gp = gpar(fill = "#ADB5C7"))
another_box <- boxGrob("A horizontal box", x = .1, y =  .5)
```

```
yet_another_box <- boxGrob("Another horizontal box", x = .8, y = .3)

alignedBoxes <- alignHorizontal(box,
                                another_box,
                                yet_another_box,
                                .position = "right")

box
for (b in alignedBoxes) {
  print(b)
}


vert_box <- boxGrob("Vert",
                    x = .8, y = .3,
                    box_gp = gpar(fill = "darkgreen"),
                    txt_gp = gpar(col = "white"))
another_vert_box <- boxGrob("Another vertical",
                            x = .1, y =  .5,
                            box_gp = gpar(fill = "darkgreen"),
                            txt_gp = gpar(col = "white"))

alignedBoxes <- alignVertical(box,
                              vert_box,
                              another_vert_box,
                              .position = "bottom")
for (b in alignedBoxes) {
  print(b)
}
```

`bezierArrowGradient` *A bezier arrow with gradient*

Description

This is an experimental addition to the original `bezierArrowSmpl` with the addition of a gradient in the center of the arrow that fades.

Usage

```
bezierArrowGradient(
  x = c(0.2, 0.7, 0.3, 0.9),
  y = c(0.2, 0.2, 0.9, 0.9),
  width = 0.05,
  clr = "#000000",
  default.units = "npc",
  align_2_axis = TRUE,
  grdt_type = c("triangle", "rectangle"),
  grdt_prop = 0.8,
```

```
  grdt_decrease_prop = 0.5,
  grdt_clr_prop = 0.7,
  grdt_line_width,
  grdt_clr = "#2F4F2F",
  vp = NULL,
  gp = gpar(),
  rm_intersect = 3L,
  ...
)
```

Arguments

`x`	A numeric vector or unit object specifying x-locations of spline control points.
`y`	A numeric vector or unit object specifying y-locations of spline control points.
`width`	The width of the arrow, either a numeric single number or a unit. **Note:** The arrow does not rely on `lwd` but on actual width.
`clr`	The color of the arrow. This is the main color of the arrow and not the gradient color.
`default.units`	A string indicating the default units to use if `x` or `y` are only given as numeric vectors.
`align_2_axis`	Indicates if the arrow should be vertically/horizontally aligned. This is useful for instance if the arrow attaches to a box.
`grdt_type`	The type of growth and gradient that is to be used, currently it only supports triangle (I'm considering adding bezier curves but currently I'm a little tired of coding)
`grdt_prop`	The proportion of the full length that should be a the gradient. The gradient consists of three things: (1) the central band, (2) the slimming of the central band, (3) the color shift into the arrow color. *Note* that the the slimming and color proportions can be overlapping.
`grdt_decrease_prop`	The proportion of the gradient that should be decreasing, i.e. narrowing according to the `grdt_type` argument.
`grdt_clr_prop`	The proportion of the gradient that should be converging to the arrow color.
`grdt_line_width`	The width of the border line. If not specified it defaults to 5 % of the original width, note the gradient's width is thus 90 %.
`grdt_clr`	The color of the gradient.
`vp`	A Grid viewport object (or NULL).
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`rm_intersect`	Set to 0 if you want to skip intersection removal, 1 only to remove left or 2 to only remove right. See details for why. @section Remove intersections: When the line is wide and the arrow has a narrow curve there may appear an empty triangle due to polygon cancellation (two polygons within the same are

	cancel out). This behaviour may be ugly and the function therefor tries to remove these. *Note:* it is expensive to check if there are the lineas may intersect at one point, remove those unexpected, and then adjust the line to the new situation so that the top and bottom lines match. It can also cause some unexpected behaviour why you may want to remove this feature if the arrow behaves erratically.
`...`	Passed on to `bezierArrowSmpl`

Value

A grob of `gList`-type

Note

The triangle section of the arrow is not currently included in the gradient.

Examples

```
library(grid)
grid.newpage()
arrowGrob <- bezierArrowGradient(
  x = c(.1, .3, .6, .9),
  y = c(0.2, 0.2, 0.9, 0.9)
)
grid.draw(arrowGrob)
```

`bezierArrowSmpl` *A simple bezier arrow*

Description

This is an alternative to the grid packages `bezierGrob` with the advantage that it allows you to draw an arrow with a specific unit width. Note, it has only a end-arrow at this point.

Usage

```
bezierArrowSmpl(
  x = c(0.2, 0.7, 0.3, 0.9),
  y = c(0.2, 0.2, 0.9, 0.9),
  width = 0.05,
  clr = "#000000",
  default.units = "npc",
  arrow = list(),
  rez = 200,
  align_2_axis = TRUE,
  name = NULL,
  rm_intersect = 3L,
  gp = gpar(),
```

```
  vp = NULL
)
```

Arguments

`x`	A numeric vector or unit object specifying x-locations of spline control points.
`y`	A numeric vector or unit object specifying y-locations of spline control points.
`width`	The width of the arrow, either a numeric single number or a unit. **Note:** The arrow does not rely on `lwd` but on actual width.
`clr`	The color of the arrow.
`default.units`	A string indicating the default units to use if `x` or `y` are only given as numeric vectors.
`arrow`	This is a list with all the **base** (width) and the desired **length** for the arrow. **Note:** This differs from the original `bezierGrob` function.
`rez`	The resolution of the arrow. This specifies how many points to retrieve from the `gnrlBezierPoints` function. Defaults to 200.
`align_2_axis`	Indicates if the arrow should be vertically/horizontally aligned. This is useful for instance if the arrow attaches to a box.
`name`	A character identifier.
`rm_intersect`	Set to 0 if you want to skip intersection removal, 1 only to remove left or 2 to only remove right. See details for why. @section Remove intersections: When the line is wide and the arrow has a narrow curve there may appear an empty triangle due to polygon cancellation (two polygons within the same are cancel out). This behaviour may be ugly and the function therefor tries to remove these. *Note:* it is expensive to check if there are the lineas may intersect at one point, remove those unexpected, and then adjust the line to the new situation so that the top and bottom lines match. It can also cause some unexpected behaviour why you may want to remove this feature if the arrow behaves erratically.
`gp`	An object of class `"gpar"`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`vp`	A Grid viewport object (or NULL).

Value

`grid::grob` A grob of the class polygonGrob with attributes that correspond to the bezier points.

Examples

```
library(grid)
grid.newpage()
arrowGrob <- bezierArrowSmpl(
  x = c(.1, .3, .6, .9),
  y = c(0.2, 0.2, 0.9, 0.9)
)
grid.draw(arrowGrob)
```

boxGrob *Create a box with text*

Description

Creates a `grob` box with text inside it.

Usage

```
boxGrob(
  label,
  y = unit(0.5, "npc"),
  x = unit(0.5, "npc"),
  width,
  height,
  just = "center",
  bjust = "center",
  txt_gp = getOption("boxGrobTxt", default = gpar(color = "black", cex = 1)),
  box_gp = getOption("boxGrob", default = gpar(fill = "white")),
  box_fn = roundrectGrob,
  name = NULL
)

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

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

## S3 method for class 'box'
widthDetails(x)

## S3 method for class 'box'
heightDetails(x)
```

Arguments

`label`	The label to print - should be a number, text or expression.
`y`	The y position to put the box at. Can be either in npc (i.e. 0-1) or a `unit`.
`x`	The x position to put the box at. Can be either in npc (i.e. 0-1) or a `unit`.
`width`	The box automatically adapts the size but you can force by specifying the width
`height`	The box automatically adapts the size but you can force by specifying the height
`just`	The justification for the text: left, center or right.
`bjust`	The justification for the box: left, center, right, top or bottom. See the `just` option for the `viewport`

`txt_gp`	The `gpar` style to apply to the text. Set `boxGrobTxt` option if you want to customize all the boxes at once.
`box_gp`	The `gpar` style to apply to the box function of ‘box_fn‘ below.
`box_fn`	Function to create box for the text. Parameters of ‘x=0.5‘, ‘y=0.5‘ and ‘box_gp‘ will be passed to this function and return a `grob` object.
`name`	a character identifier for the `grob`. Used to find the `grob` on the display list and/or as a child of another grob.
`...`	Passed to `grid.draw`

Value

A grob

The `plot`/`print`

To output the `grob` objects to the plot either call `plot` on the object or `print` it. Note that R automatically prints any object that is outputted to the console. The function calls in turn the `grid.draw` function on the object.

S3 from the grid package

Width and height functions address the `coords` attribute for the corresponding information. The `widthDetails` and `heightDetails` that provide information on an object.

See Also

Other flowchart components: `align`, `boxPropGrob()`, `connectGrob()`, `coords()`, `distance()`, `moveBox()`, `spread`

Examples

```
library(grid)
grid.newpage()
boxGrob("My box")
```

`boxPropGrob` *Create a box with a color split*

Description

Creates a grob box with text inside it and a color split in the horizontal axes that allow indicating different proportions. The box can also have a title that spanse the two color areas and that has its own background.

Usage

```
boxPropGrob(
  label,
  label_left,
  label_right,
  prop,
  y = unit(0.5, "npc"),
  x = unit(0.5, "npc"),
  width,
  height,
  just = "center",
  bjust = "center",
  txt_gp = getOption("boxPropGrobTxt", default = gpar(color = "black")),
  txt_left_gp = getOption("boxPropGrobLeftTxt", default = gpar(col = "black")),
  txt_right_gp = getOption("boxPropGrobRightTxt", default = gpar(col = "black")),
  box_left_gp = getOption("boxPropGrobLeft", default = gpar(fill = "#E6E8EF")),
  box_right_gp = getOption("boxPropGrobRight", default = gpar(fill = "#FFFDF6")),
  box_highlight_gp = getOption("boxPropGrobHighlight", default = gpar(fill = "#ffffff55",
    col = NA)),
  name = NULL
)
```

Arguments

`label`	The label to print - should be a number, text or expression.
`label_left`	The label for the left area
`label_right`	The label for the right area
`prop`	The proportion to split along
`y`	The y position to put the box at. Can be either in npc (i.e. 0-1) or a `unit`.
`x`	The x position to put the box at. Can be either in npc (i.e. 0-1) or a `unit`.
`width`	The box automatically adapts the size but you can force by specifying the width
`height`	The box automatically adapts the size but you can force by specifying the height
`just`	The justification for the text: left, center or right.
`bjust`	The justification for the box: left, center, right, top or bottom. See the `just` option for the `viewport`
`txt_gp`	The `gpar` style to apply to the text. Set `boxPropGrobTxt` option if you want to customize all the boxes at once.
`txt_left_gp`	The `gpar` style to apply to the left text. Set `boxPropGrobLeftTxt` option if you want to customize all the boxes at once.
`txt_right_gp`	The `gpar` style to apply to the right text. Set `boxPropGrobRightTxt` option if you want to customize all the boxes at once.
`box_left_gp`	The `gpar` style to apply to the left box. Set `boxPropGrobLeft` option if you want to customize all the boxes at once.

`box_right_gp`	The `gpar` style to apply to the right box. Set `boxPropGrobRight` option if you want to customize all the boxes at once.
`box_highlight_gp`	The `gpar` style to apply to the background of the main label. Set `boxPropGrobHighlight` option if you want to customize all the boxes at once.
`name`	a character identifier for the `grob`. Used to find the `grob` on the display list and/or as a child of another grob.

Value

A box grob

See Also

Other flowchart components: `align`, `boxGrob()`, `connectGrob()`, `coords()`, `distance()`, `moveBox()`, `spread`

Examples

```
library(grid)
grid.newpage()
boxPropGrob("Main label", "Left text", "Right text", prop = .3)
```

`calculateLinesAndArrow`

Gets offsetted lines

Description

The function calculates new points according to the offset that lie to the left/right of the provided line.

Usage

```
calculateLinesAndArrow(
  x,
  y,
  offset,
  end_x = -1,
  end_y = -1,
  arrow_offset = -1,
  rm_intersect = 3L
)
```

Arguments

`x`	A numeric vector containing all the x-elements
`y`	A numeric vector containing all the y-elements
`offset`	The offset to add to the line, can be a vector if you want to use different offsets.
`end_x`	The x end of the line where the arrow occurrs (if < 0 arrow is skipped)
`end_y`	The y end of the line where the arrow occurrs (if < 0 arrow is skipped)
`arrow_offset`	The offset to add to the arrow section if any (if <= 0 arrow is skipped)
`rm_intersect`	Set to 0 if you want to skip intersection removal, 1 only to remove left or 2 to only remove right. See details for why. @section Remove intersections: When the line is wide and the arrow has a narrow curve there may appear an empty triangle due to polygon cancellation (two polygons within the same are cancel out). This behaviour may be ugly and the function therefor tries to remove these. *Note:* it is expensive to check if there are the lineas may intersect at one point, remove those unexpected, and then adjust the line to the new situation so that the top and bottom lines match. It can also cause some unexpected behaviour why you may want to remove this feature if the arrow behaves erratically.

Value

`list(list(x = ..., y = ...))` Returns a list with the right/left lines that in turn lists with *x* and *y* elements

---

`connectGrob` *Connect boxes with an arrow*

Description

The function creates a grob that links two boxes together. It looks for which side it should attach the arrow, e.g. if the start is on top of the bottom it should attach to the bottom edge of ther start box and then to the top at the end.

Usage

```
connectGrob(
  start,
  end,
  type = c("vertical", "horizontal", "L", "-", "Z", "N"),
  subelmnt = c("right", "left"),
  lty_gp = getOption("connectGrob", default = gpar(fill = "black")),
  arrow_obj = getOption("connectGrobArrow", default = arrow(ends = "last", type =
    "closed"))
)
```

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

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

Arguments

`start`	The start box
`end`	The end box
`type`	How the boxes are stacked. The `L` alternative generates a straight line up/down and then turns to righT/left for connecting with the end. The `-` generates a straight horizontal arrow. The `Z` creates a horizontal line that looks like a `Z` with 90 degree turns. The option `N` allows for vertical lines.
`subelmnt`	If we have a split box we can specify the right/left x as the connector point.
`lty_gp`	The `gpar` for the line. Set `connectGrob` option if you want to customize all the arrows at once.
`arrow_obj`	The arrow spec according to `arrow`. Set `connectGrobArrow` option if you want to customize all the arrows at once.
`x`	The grob to print/plot
`...`	Passed to `grid.draw`

Details

The exact positions of the line is stored at the `attr(..., "line")`. If you want to draw your own custom line all you need to do is check which `attr(my_line, "line")$x` and `attr(my_line, "line")$y` you want to attach to and then create your own custom `linesGrob`.

Value

grob with an arrow

See Also

Other flowchart components: `align`, `boxGrob()`, `boxPropGrob()`, `coords()`, `distance()`, `moveBox()`, `spread`

Examples

```
library(grid)
grid.newpage()

# Initiate the boxes that we want to connect
start <- boxGrob("Top", x = .5, y = .8)
end <- boxGrob("Bottom", x = .5, y = .2)
side <- boxPropGrob("Side", "Left", "Right", prop = .3, x = .2, y = .8)
sub_side_left <- boxGrob("Left", x = attr(side, "coords")$left_x, y = .5)
```

```
sub_side_right <- boxGrob("Right", x = attr(side, "coords")$right_x, y = .5)
exclude <- boxGrob("Exclude:\n - Too sick\n - Prev. surgery", x = .8, y = .5, just = "left")

# Connect the boxes and print/plot them
connectGrob(start, end, "vertical")
connectGrob(start, side, "horizontal")
connectGrob(side, sub_side_left, "v", "l")
connectGrob(side, sub_side_right, "v", "r")
connectGrob(start, exclude, "L")

# Print the grobs
start
end
side
exclude
sub_side_left
sub_side_right
```

`convertShowMissing` *A function for converting a show_missing variable.*

Description

The variable is supposed to be directly compatible with `table`(..., useNA = show_missing). It throws an error if not compatible. It is mostly useful for custom describe functions.

Usage

```
convertShowMissing(show_missing)
```

Arguments

`show_missing`	Boolean or "no", "ifany", "always"

Details

Deprecated: This function will be deprecated as all functions now use the useNA style in order to comply with standard R naming.

Value

string

`coords` *Get the box coordinates*

Description

Retrieves the boxes `"coords"` attribute.

Usage

```
coords(box)
```

Arguments

`box`	The `boxGrob` or `boxPropGrob`

Value

A list with the coordinates

See Also

Other flowchart components: `align`, `boxGrob()`, `boxPropGrob()`, `connectGrob()`, `distance()`, `moveBox()`, `spread`

Examples

```
box <- boxGrob("A test box")
coords(box)
```

`copyAllNewAttributes` *A simple thing to keep the attributes*

Description

Skips the attributes that the to object already has to avoid overwriting dim and other important attributes

Usage

```
copyAllNewAttributes(from, to, attr2skip = c(), attr2force = c())
```

Arguments

`from`	The from object
`to`	The to object
`attr2skip`	An optional lists of attributes that you may want to avoid having copied
`attr2force`	An optional lists of attributes that you may want to force copy even if they already exist in the new object

Value

`object` The `to` argument object

Examples

```
a <- "test"
attr(a, 'wow') <- 1000
b <- a
b <- copyAllNewAttributes(a, b)
print(attr(b, 'wow'))
```

`descGetMissing` *Get statistics for missing data*

Description

This function calculates the amount of missing per row for `describeMean`, `describeMedian` and custom description functions. It will return invisibly when no missing values are present.

Usage

```
descGetMissing(
  x,
  html = TRUE,
  number_first = TRUE,
  percentage_sign = TRUE,
  language = "en",
  useNA.digits = 1,
  ...
)
```

Arguments

`x`	The variable that you want the statistics for
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`number_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses (). This is only used together with the useNA variable.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable. Note, this is only used when combined with the missing information.

`language`	The ISO-639-1 two-letter code for the language of interest. Currently only english is distinguished from the ISO format using a ',' as the separator in the `txtInt` function.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`...`	Passed on to `describeFactors`

Value

`vector` A vector with the missing estimate

`describeFactors` *Describes factor variables*

Description

A function that returns a description of proportions in a factor that contains the number of times a level occurs and the percentage

Usage

```
describeFactors(
  x,
  html = TRUE,
  digits = 1,
  digits.nonzero = NA,
  number_first = TRUE,
  useNA = c("ifany", "no", "always"),
  useNA.digits = digits,
  horizontal_proportions,
  percentage_sign = TRUE,
  language = "en",
  ...
)
```

Arguments

`x`	The variable that you want the statistics for
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`number_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses (). This is only used together with the useNA variable.

`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`horizontal_proportions`	Is only active if useNA since this is the only case of a proportion among continuous variables. This is default NULL and indicates that the proportions are to be interpreted in a vertical manner. If we want the data to be horizontal, i.e. the total should be shown and then how these differ in the different groups then supply the function with the total number in each group, i.e. if done in a by manner as in `getDescriptionStatsBy` it needs to provide the number before the by() command.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable. Note, this is only used when combined with the missing information.
`language`	The ISO-639-1 two-letter code for the language of interest. Currently only english is distinguished from the ISO format using a ',' as the separator in the `txtInt` function.
`...`	Passed on to `txtInt`

Value

A string formatted for printing either latex by HTML

See Also

`getDescriptionStatsBy`

Other descriptive functions: `describeMean()`, `describeMedian()`, `describeProp()`, `getDescriptionStatsBy()`, `getPvalWilcox()`

Examples

```
set.seed(1)
describeFactors(sample(50, x = c("A", "B", "C"), replace = TRUE))

n <- 500
my_var <- factor(sample(size = n, x = c("A", "B", "C", NA), replace = TRUE))
my_exp <- rbinom(n = n, size = 1, prob = 0.2)
total <- table(my_var, useNA = "ifany")
by(my_var,
  INDICES = my_exp,
  FUN = describeFactors,
  useNA = "ifany",
  horizontal_proportions = total
)
```

`describeMean` *Describe the mean*

Description

A function that returns a description of a continuous variable using the mean together with the standard deviation. The standard deviation is used as it is "industry standard" to use mean with standard deviation and not because it's the only option.

Usage

```
describeMean(
  x,
  html = TRUE,
  digits = 1,
  digits.nonzero = NA,
  number_first = TRUE,
  useNA = c("ifany", "no", "always"),
  useNA.digits = digits,
  percentage_sign = TRUE,
  plusmin_str,
  language = "en",
  ...
)
```

Arguments

`x`	The variable that you want the statistics for
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`number_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses (). This is only used together with the useNA variable.
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable. Note, this is only used when combined with the missing information.

`plusmin_str`	Provide if you want anything other than the plus minus sign suited for the given output format.
`language`	The ISO-639-1 two-letter code for the language of interest. Currently only english is distinguished from the ISO format using a ',' as the separator in the `txtInt` function.
`...`	Passed on to `describeFactors`

Value

`string` Returns a string formatted for either LaTeX or HTML

See Also

`getDescriptionStatsBy`

Other descriptive functions: `describeFactors()`, `describeMedian()`, `describeProp()`, `getDescriptionStatsBy()`, `getPvalWilcox()`

Examples

```
describeMean(1:10)
describeMean(c(1:10, NA), useNA = "always")
describeMean(c(1:10, NA), useNA = "no")
```

`describeMedian` *A function that returns a description median that contains the interquartile range or the full range*

Description

A function that returns a description median that contains the interquartile range or the full range

Usage

```
describeMedian(
  x,
  iqr = TRUE,
  html = TRUE,
  digits = 1,
  digits.nonzero = NA,
  number_first = TRUE,
  useNA = c("ifany", "no", "always"),
  useNA.digits = digits,
  percentage_sign = TRUE,
  language = "en",
  ...
)
```

Arguments

`x`	The variable that you want the statistics for
`iqr`	If interquartile range should be used
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`number_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses (). This is only used together with the useNA variable.
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable. Note, this is only used when combined with the missing information.
`language`	The ISO-639-1 two-letter code for the language of interest. Currently only english is distinguished from the ISO format using a ',' as the separator in the `txtInt` function.
`...`	Passed on to `describeFactors`

Value

`string` A string formatted for either LaTeX or HTML

See Also

`getDescriptionStatsBy`

Other descriptive functions: `describeFactors()`, `describeMean()`, `describeProp()`, `getDescriptionStatsBy()`, `getPvalWilcox()`

Examples

```
describeMedian(1:10)
describeMedian(c(1:10, NA), useNA = "ifany")
```

`describeProp` *A function that returns a description proportion that contains the number and the percentage*

Description

A function that returns a description proportion that contains the number and the percentage

Usage

```
describeProp(
  x,
  html = TRUE,
  digits = 1,
  digits.nonzero = NA,
  number_first = TRUE,
  useNA = c("ifany", "no", "always"),
  useNA.digits = digits,
  default_ref = NULL,
  percentage_sign = TRUE,
  language = "en",
  ...
)
```

Arguments

`x`	The variable that you want the statistics for
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`number_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses (). This is only used together with the useNA variable.
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`default_ref`	The default reference, either first, the level name or a number within the levels. If left out it defaults to the first value.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable. Note, this is only used when combined with the missing information.

`language`	The ISO-639-1 two-letter code for the language of interest. Currently only english is distinguished from the ISO format using a ',' as the separator in the `txtInt` function.
`...`	Passed on to `describeFactors`

Value

`string` A string formatted for either LaTeX or HTML

See Also

Other descriptive functions: `describeFactors()`, `describeMean()`, `describeMedian()`, `getDescriptionStatsBy()`, `getPvalWilcox()`

Examples

```
describeProp(factor(sample(50, x = c("A", "B", NA), replace = TRUE)))
```

`distance` *Get the distance between grid objects*

Description

Retrieves the distance between two boxes as absolute `"mm"` units. The function also accepts `coords` objects as well as a `unit` or a numeric input.

Usage

```
distance(
  box1,
  box2,
  type = c("vertical", "horizontal", "euclidean"),
  half = FALSE,
  center = FALSE
)

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

Arguments

`box1`	The first `boxGrob`. Can also be a `coords` object, a `unit` or a numeric. The latter is evaluated to a `unit` with `units="npc"`.
`box2`	The second object to calculate the distance to. Same type as for `box1`.
`type`	Whether we should retrieve the vertical, horizontal or euclidean distance
`half`	If set to true it returns half the distance. This is convenient when positioning boxes between each other.

`center`	Calculate the distance from the center of each object
`x`	A unit with from the `distance` function
`...`	Passed on to print

Value

A `unit` in `"mm"` with an absolute value. The attribute `positive` indicates the direction of the value, i.e. if it is `TRUE` the distance was calculated from the first to the second, otherwise it is `FALSE`. For `euclidean` distance the `positive` attribute is `NA`. There is also the `from` and `to` attributes that has the coordinates that were used for the calculations, for `euclidean` distance this is `NA`.

See Also

Other flowchart components: `align`, `boxGrob()`, `boxPropGrob()`, `connectGrob()`, `coords()`, `moveBox()`, `spread`

Examples

```
box1 <- boxGrob("A test box", y = .8)
box2 <- boxGrob("Another test box", y = .2)
distance(box1, box2, "v")
```

`docx_document` *Formatter wrapper for `html_document`, facilitates easier porting to docx*

Description

This function adds the option of having adaptations needed for seemless integration with MS Word for importing html-documents in the .docx-format. The advantage of html documents is the ability to create advanced formatting frequently needed in medical publications and that is available in the `htmlTable` function. You can view the series for more details regarding how to achieve fast-track-publishing (ftp) together with knitr.

Usage

```
docx_document(
  ...,
  self_contained = FALSE,
  mathjax = NULL,
  theme = NULL,
  highlight = NULL,
  css = "rmarkdown/docx.css",
  h1_style = "margin: 24pt 0pt 0pt 0pt;",
  other_h_style = "margin: 10pt 0pt 0pt 0pt;",
  remove_scripts = TRUE,
  force_captions = FALSE,
  css_max_width
)
```

Arguments

`...`	Passed onto `html_document`.
`self_contained`	Overrides the default `TRUE` for `html_document` to `FALSE` as LibreOffice hangs on long lines such as the base64 images included in the self-contained version.
`mathjax`	The advanced mathjax does not work with with Word/LibreOffice.
`theme`	No theme should be used for the output as the custom CSS should take care of everything.
`highlight`	By default turn off highlighting as scripts are difficult to import. This does though work somewhat OK when copy-pasting from the web-browser.
`css`	The CSS if other that the default within the package
`h1_style`	You can choose any css style formatting here that you want to be applied to all h1 elements. Note: this is only applied if LibreOffice_adapt is `TRUE`.
`other_h_style`	This is the formatting applied to any other h elements not included to the first. Note: this is only applied if LibreOffice_adapt is `TRUE`.
`remove_scripts`	`TRUE` if <script></script> tags are to be removed. These are usually not compatible with Word-processors and should therefore in most cases be stripped from the document.
`force_captions`	Since out.width and out.height remove the option of having captions this allows a workaround through some processing via the XML-package
`css_max_width`	The max width of the body element. Defaults to "40em" if not specified. Any CSS-compliant width format works.

Details

If you want to get equations into Word the currently best way is to use the `word_document` format.

Value

R Markdown output format to pass to `render`

Author(s)

Max Gordon

Examples

```
# Possible yaml configuration at the top of the Rmd doc
## Not run:
---
title: "Test"
author: "Max Gordon"
output:
  Gmisc::docx_document
---

## End(Not run)
```

`fastDoCall` *An alternative to the internal* `do.call`

Description

The `do.call` can be somewhat slow, especially when working with large objects. This function is based upon the suggestions from Hadley Wickham on the R mailing list. Also thanks to *Tommy* at StackOverflow for suggesting how to handle double and triple colon operators, `::`, further enhancing the function.

Usage

```
fastDoCall(what, args, quote = FALSE, envir = parent.frame())
```

Arguments

`what`	either a function or a non-empty character string naming the function to be called.
`args`	a *list* of arguments to the function call. The `names` attribute of `args` gives the argument names.
`quote`	a logical value indicating whether to quote the arguments.
`envir`	an environment within which to evaluate the call. This will be most useful if `what` is a character string and the arguments are symbols or quoted expressions.

Note

While the function attempts to do most of what `do.call` can it has limitations. It can currently not parse the example code from the original function: `do.call(paste, list(as.name("A"), as.name("B")), quote = TRUE)` and the functionality of `quote` has not been thoroughly tested.

Examples

```
fastDoCall("complex", list(imaginary = 1:3))

## if we already have a list (e.g. a data frame)
## we need c() to add further arguments
tmp <- expand.grid(letters[1:2], 1:3, c("+", "-"))
fastDoCall("paste", c(tmp, sep = ""))

## examples of where objects will be found.
A <- 2
f <- function(x) print(x^2)
env <- new.env()
assign("A", 10, envir = env)
assign("f", f, envir = env)
f <- function(x) print(x)
f(A)                                      # 2
fastDoCall("f", list(A))                     # 2
```

```
fastDoCall("f", list(A), envir = env)        # 4
fastDoCall(f, list(A), envir = env)          # 2
fastDoCall("f", list(quote(A)), envir = env) # 100
fastDoCall(f, list(quote(A)), envir = env)   # 10
fastDoCall("f", list(as.name("A")), envir = env) # 100

eval(call("f", A))                      # 2
eval(call("f", quote(A)))               # 2
eval(call("f", A), envir = env)         # 4
eval(call("f", quote(A)), envir = env)  # 100
```

`figCapNo` *Adds a figure caption number*

Description

The function relies on `options("fig_caption_no")` in order to keep track of the last number. If you want to force the caption function to skip captions while still using it in the knitr fig.cap option then simply set `options(fig_caption_no = FALSE)`

Usage

```
figCapNo(
  str,
  roman = getOption("fig_caption_no_roman", default = FALSE),
  sprintf_str = getOption("fig_caption_no_sprintf", default = "Fig. %s: %s")
)
```

Arguments

`str`	The string that is to be prepended with string
`roman`	Whether or not to use roman numbers instead of Arabic. Can also be set through `options(fig_caption_no_roman = TRUE)`
`sprintf_str`	An `sprintf` formatted string where the first argument is reserved for the string generated by the counter and the second one is for the caption text. Can also be set through `options(fig_caption_no_sprintf = TRUE)`

See Also

Other figure caption functions: `figCapNoLast()`, `figCapNoNext()`

Examples

~~~
## Not run:
```{r, fig.cap = pigCapNo("My nice plot")}
plot(1:10 + rnorm(10), 1:10)
~~~
```
```

```
## End(Not run)
org_opts <- options(fig_caption_no = 2,
                    fig_caption_no_sprintf = "Figure %s: %s")
figCapNo("A plot with caption number = 3")

org_opts <- options(fig_caption_no = TRUE)
figCapNo("A plot with caption number = 1")

# Use default setting
options(fig_caption_no_sprintf = NULL)
figCapNo("A plot with caption number = 2")

# Return the original settings
options(org_opts)
```

`figCapNoLast` *Gets the last figure caption number*

Description

The function relies on `options("fig_caption_no")` in order to keep track of the last number.

Usage

```
figCapNoLast(roman = getOption("fig_caption_no_roman", FALSE))
```

Arguments

`roman`	Whether or not to use roman numbers instead of Arabic. Can also be set through `options(fig_caption_no_roman = TRUE)`

See Also

Other figure caption functions: `figCapNoNext()`, `figCapNo()`

Examples

```
org_opts <- options(fig_caption_no = 1)
figCapNoLast()
options(org_opts)
```

`figCapNoNext` *Gets the next figure caption number*

Description

The function relies on `options("fig_caption_no")` in order to keep track of the last number.

Usage

```
figCapNoNext(roman = getOption("fig_caption_no_roman", default = FALSE))
```

Arguments

`roman`	Whether or not to use roman numbers instead of Arabic. Can also be set through `options(fig_caption_no_roman = TRUE)`

See Also

Other figure caption functions: `figCapNoLast()`, `figCapNo()`

Examples

```
org_opts <- options(fig_caption_no = 1)
figCapNoNext()
options(org_opts)
```

`getBezierAdj4Arrw` *Gets the bezier points adjusted for an arrow*

Description

Gets the bezier points adjusted for an arrow

Usage

```
getBezierAdj4Arrw(x, y, arrow_length, length_out = 100)
```

Arguments

`x`	The x start and end points
`y`	The spline control points
`arrow_length`	The desired length of the arrow
`length_out`	Increases the resolution for the final bezier points, i.e. generating more fine-grained intervals

Value

list

getDescriptionStatsBy *Creating of description statistics*

Description

A function that returns a description statistic that can be used for creating a publication "table 1" when you want it by groups. The function identifies if the variable is a continuous, binary or a factored variable. The format is inspired by NEJM, Lancet & BMJ.

Usage

```
getDescriptionStatsBy(
  x,
  ...,
  by,
  digits = 1,
  digits.nonzero = NA,
  html = TRUE,
  numbers_first = TRUE,
  statistics = FALSE,
  statistics.sig_lim = 10^-4,
  statistics.two_dec_lim = 10^-2,
  statistics.suppress_warnings = TRUE,
  useNA = c("ifany", "no", "always"),
  useNA.digits = digits,
  continuous_fn = describeMean,
  prop_fn = describeProp,
  factor_fn = describeFactors,
  show_all_values = FALSE,
  hrzl_prop = FALSE,
  add_total_col,
  total_col_show_perc = TRUE,
  use_units = FALSE,
  units_column_name = "Units",
  default_ref = NULL,
  NEJMstyle = FALSE,
  percentage_sign = TRUE,
  header_count = NULL,
  missing_value = "-",
  names_of_missing = NULL
)

## S3 method for class 'Gmisc_getDescriptionStatsBy'
htmlTable(x, ...)

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

```
## S3 method for class 'Gmisc_getDescriptionStatsBy'
knit_print(x, ...)

## S3 method for class 'Gmisc_getDescriptionStatsBy'
length(x)
```

Arguments

`x`	If a data.frame it will be used as the data source for the variables in the `...` parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`...`	The variables that you want you statistic for. In the print all thes parameters are passed on as [htmlTable::htmlTable] arguments.
`by`	The variable that you want to split into different columns
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`numbers_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses ().
`statistics`	Add statistics, fisher test for proportions and Wilcoxon for continuous variables. See details below for more customization.
`statistics.sig_lim`	The significance limit for < sign, i.e. p-value 0.0000312 should be < 0.0001 with the default setting.
`statistics.two_dec_lim`	The limit for showing two decimals. E.g. the p-value may be 0.056 and we may want to keep the two decimals in order to emphasize the proximity to the all-mighty 0.05 p-value and set this to $10^{-}2$. This allows that a value of 0.0056 is rounded to 0.006 and this makes intuitive sense as the 0.0056 level as this is well below the 0.05 value and thus not as interesting to know the exact proximity to 0.05. *Disclaimer:* The 0.05-limit is really silly and debated, unfortunately it remains a standard and this package tries to adapt to the current standards in order to limit publication associated issues.
`statistics.suppress_warnings`	Hide warnings from the statistics function.
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`continuous_fn`	The method to describe continuous variables. The default is `describeMean`.
`prop_fn`	The method used to describe proportions, see `describeProp`.
`factor_fn`	The method used to describe factors, see `describeFactors`.

`show_all_values`	Show all values in proportions. For factors with only two values it is most sane to only show one option as the other one will just be a complement to the first, i.e. we want to convey a proportion. For instance sex - if you know gender then automatically you know the distribution of the other sex as it's 100 % - other %. To choose which one you want to show then set the `default_ref` parameter.
`hrzl_prop`	This is default FALSE and indicates that the proportions are to be interpreted in a vertical manner. If we want the data to be horizontal, i.e. the total should be shown and then how these differ in the different groups then set this to TRUE.
`add_total_col`	This adds a total column to the resulting table. You can also specify if you want the total column "first" or "last" in the column order.
`total_col_show_perc`	This is by default true but if requested the percentages are suppressed as this sometimes may be confusing.
`use_units`	If the Hmisc package's units() function has been employed it may be interesting to have a column at the far right that indicates the unit measurement. If this column is specified then the total column will appear before the units (if specified as last). You can also set the value to `"name"` and the units will be added to the name as a parenthesis, e.g. Age (years).
`units_column_name`	The name of the units column. Used if use_units = TRUE
`default_ref`	The default reference when dealing with proportions. When using 'dplyr' syntax ('tidyselect') you can specify a named vector/list for each column name.
`NEJMstyle`	Adds - no (%) at the end to proportions
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable.
`header_count`	Set to `TRUE` if you want to add a header count, e.g. Smoking; No. 25 observations, where there is a new line after the factor name. If you want a different text for the second line you can specifically use the `sprintf` formatting, e.g. "No. %s patients".
`missing_value`	Value that is substituted for empty cells. Defaults to "-"
`names_of_missing`	Optional character vector containing the names of returned statistics, in case all returned values for a given by level are missing. Defaults to NULL

Value

Returns `matrix` if a single value was provided, otherwise a `list` of matrices with the class `"Gmisc_getDescriptionStatsBy`

Customizing statistics

You can specify what function that you want for statistic by providing a function that takes two arguments `x` and `by` and returns a p-value. There are a few functions already prepared for this see `getPvalAnova`, `getPvalChiSq` `getPvalFisher` `getPvalKruskal` `getPvalWilcox`. The default functions used are `getPvalFisher` and `getPvalWilcox` (unless the by argument has more than three unique levels where it defaults to `getPvalAnova`).

If you want the function to select functions depending on the type of input you can provide a list with the names `'continuous'`, `'proportion'`, `'factor'` and the function will choose accordingly. If you fail to define a certain category it will default to the above.

You can also use a custom function that returns a string with the attribute `'colname'` set that will be appended to the results instead of the p-value column.

See Also

Other descriptive functions: `describeFactors()`, `describeMean()`, `describeMedian()`, `describeProp()`, `getPvalWilcox()`

Examples

```
library(magrittr)
library(dplyr)
library(htmlTable)

data(mtcars)
mtcars %<>%
  mutate(am = factor(am, levels = 0:1, labels = c("Automatic", "Manual")),
         vs = factor(vs, levels = 0:1, labels = c("V-shaped", "straight")),
         drat_prop = drat > median(drat),
         drat_prop = factor(drat_prop,
                            levels = c(FALSE, TRUE),
                            labels = c("High ratio", "Low ratio")),
         carb_prop = carb > 2,
         carb_prop = factor(carb_prop,
                            levels = c(FALSE, TRUE),
                            labels = c("&le; 2", "&gt; 2")),
         across(c(gear, carb, cyl), factor))

# A simple bare-bone example
mtcars %>%
  getDescriptionStatsBy(`Miles per gallon` = mpg,
                        Weight = wt,
                        `Carborators &le; 2` = carb_prop,
                        by = am) %>%
  htmlTable(caption  = "Basic continuous stats from the mtcars dataset")
invisible(readline(prompt = "Press [enter] to continue"))

# For labeling & units we use set_column_labels/set_column_unit that use
# the Hmisc package annotation functions
mtcars %<>%
  set_column_labels(am = "Transmission",
                    mpg = "Gas",
                    wt = "Weight",
                    gear = "Gears",
                    disp = "Displacement",
                    vs = "Engine type",
```

```
                    drat_prop = "Rear axel ratio",
                    carb_prop = "Carburetors") %>%
  set_column_units(mpg = "Miles/(US) gallon",
                   wt = "10<sup>3</sup> lbs",
                   disp = "cu.in.")

mtcars %>%
  getDescriptionStatsBy(mpg,
                        wt,
                        `Gear&dagger;` = gear,
                        drat_prop,
                        carb_prop,
                        vs,
                        by = am,
                        header_count = TRUE,
                        use_units = TRUE,
                        show_all_values = TRUE)  %>%
  addHtmlTableStyle(pos.caption = "bottom") %>%
  htmlTable(caption  = "Stats from the mtcars dataset",
            tfoot = "&dagger; Number of forward gears")
invisible(readline(prompt = "Press [enter] to continue"))

# Using the default parameter we can
mtcars %>%
  getDescriptionStatsBy(mpg,
                        wt,
                        `Gear&dagger;` = gear,
                        drat_prop,
                        carb_prop,
                        vs,
                        by = am,
                        header_count = TRUE,
                        use_units = TRUE,
                        default_ref = c(drat_prop = "Low ratio",
                                        carb_prop = "&gt; 2"))  %>%
  addHtmlTableStyle(pos.caption = "bottom") %>%
  htmlTable(caption  = "Stats from the mtcars dataset",
            tfoot = "&dagger; Number of forward gears")
invisible(readline(prompt = "Press [enter] to continue"))

# We can also use lists
tll <- list()
tll[["Gear (3 to 5)"]] <- getDescriptionStatsBy(mtcars$gear, mtcars$am)
tll <- c(tll,
         list(getDescriptionStatsBy(mtcars$disp, mtcars$am)))

mergeDesc(tll,
          htmlTable_args = list(caption  = "Factored variables")) %>%
  htmlTable::addHtmlTableStyle(css.rgroup = "")
invisible(readline(prompt = "Press [enter] to continue"))

tl_no_units <- list()
tl_no_units[["Gas (mile/gallons)"]] <-
```

```
  getDescriptionStatsBy(mtcars$mpg, mtcars$am,
                        header_count = TRUE)
tl_no_units[["Weight (10<sup>3</sup> kg)"]] <-
  getDescriptionStatsBy(mtcars$wt, mtcars$am,
                        header_count = TRUE)
mergeDesc(tl_no_units,
          tll) %>%
  htmlTable::addHtmlTableStyle(css.rgroup = "")
invisible(readline(prompt = "Press [enter] to continue"))

# Other settings
mtcars$mpg[sample(1:NROW(mtcars), size = 5)] <- NA
getDescriptionStatsBy(mtcars$mpg,
                      mtcars$am,
                      statistics = TRUE)
invisible(readline(prompt = "Press [enter] to continue"))

# Do the horizontal version
getDescriptionStatsBy(mtcars$gear,
                      mtcars$am,
                      statistics = TRUE,
                      hrzl_prop = TRUE)
invisible(readline(prompt = "Press [enter] to continue"))

mtcars$wt_with_missing <- mtcars$wt
mtcars$wt_with_missing[sample(1:NROW(mtcars), size = 8)] <- NA
getDescriptionStatsBy(mtcars$wt_with_missing, mtcars$am, statistics = TRUE,
                      hrzl_prop = TRUE, total_col_show_perc = FALSE)
invisible(readline(prompt = "Press [enter] to continue"))

## Not run:
  ## There is also a LaTeX wrapper
  tll <- list(
    getDescriptionStatsBy(mtcars$gear, mtcars$am),
    getDescriptionStatsBy(mtcars$col, mtcars$am))

  latex(mergeDesc(tll),
        caption  = "Factored variables",
        file = "")

## End(Not run)
```

`getPvalWilcox` *P-value extractors for* `getDescriptionStatsBy`

Description

These functions are the base functions for getting the description p-values. You can provide your own functions but all functions should take two arguments and return a p-value (numeric, non-formatted)

Usage

```
getPvalWilcox(x, by)

getPvalAnova(x, by)

getPvalFisher(x, by)

getPvalChiSq(x, by)

getPvalKruskal(x, by)
```

Arguments

`x`	The main variable of interest
`by`	The variable for the stratification

Value

`numeric` Returns the p-value from that particular test

getPvalWilcox

Performs a two-sample two-sided Wilcoxon test (also known as the Mann-Whitney test), see `wilcox.test`.

getPvalAnova

Performs a standard Analysis of Variance model through `anova(lm(x ~ by))`

getPvalFisher

Performs Fisher's exact test through the `fisher.test`.

getPvalChiSq

Performs a standard Chi-Squares analysis through `chisq.test`

getPvalKruskal

Performs a Kruskal-Wallis rank sum test through `kruskal.test`

See Also

Other descriptive functions: `describeFactors()`, `describeMean()`, `describeMedian()`, `describeProp()`, `getDescriptionStatsBy()`

Examples

```
set.seed(123)
getPvalFisher(
  sample(letters[1:3], size = 100, replace = TRUE),
  sample(LETTERS[1:3], size = 100, replace = TRUE)
)
getPvalWilcox(
  rnorm(100),
  sample(LETTERS[1:2], size = 100, replace = TRUE)
)
```

`getSvdMostInfluential` *Gets the maximum contributor variables from svd()*

Description

This function is inspired by Jeff Leeks Data Analysis course where he suggests that one way to use the `svd` is to look at the most influential rows for first columns in the V matrix.

Usage

```
getSvdMostInfluential(
  mtrx,
  quantile,
  similarity_threshold,
  plot_selection = TRUE,
  plot_threshold = 0.05,
  varnames = NULL
)
```

Arguments

`mtrx`	A matrix or data frame with the variables. Note: if it contains missing variables make sure to impute prior to this function as the `svd` can't handle missing values.
`quantile`	The SVD D-matrix gives an estimate for the amount that is explained. This parameter is used for selecting the columns that have that quantile of explanation.
`similarity_threshold`	A quantile for how close other variables have to be in value to maximum contributor of that particular column. If you only want the maximum value then set this value to 1.
`plot_selection`	As this is all about variable exploring it is often interesting to see how the variables were distributed among the vectors
`plot_threshold`	The threshold of the plotted bars, measured as percent explained by the D-matrix. By default it is set to 0.05.
`varnames`	A vector with alternative names to the colnames

Details

This function expands on that idea and adds the option of choosing more than just the most contributing variable for each row. For instance two variables may have a major impact on a certain component where the second variable has 95 important in that particular component it makes sense to include it in the selection.

It is of course useful when you have many continuous variables and you want to determine a subgroup to look at, i.e. finding the needle in the haystack.

Value

Returns a list with vector with the column numbers that were picked in the "most_influential" variable and the svd caluclation in the "svd"

Examples

```
org_par <- par(ask = TRUE)
set.seed(1345)
# Simulate data with a pattern
dataMatrix <- matrix(rnorm(15 * 160), ncol = 15)
colnames(dataMatrix) <- c(
    paste("Pos.3:", 1:3, sep = " #"),
    paste("Neg.Decr:", 4:6, sep = " #"),
    paste("No pattern:", 7:8, sep = " #"),
    paste("Pos.Incr:", 9:11, sep = " #"),
    paste("No pattern:", 12:15, sep = " #"))

for (i in 1:nrow(dataMatrix)) {
  # flip a coin
  coinFlip1 <- rbinom(1, size = 1, prob = 0.5)
  coinFlip2 <- rbinom(1, size = 1, prob = 0.5)
  coinFlip3 <- rbinom(1, size = 1, prob = 0.5)

  # if coin is heads add a common pattern to that row
  if (coinFlip1) {
    cols <- grep("Pos.3", colnames(dataMatrix))
    dataMatrix[i, cols] <- dataMatrix[i, cols] + 3
  }

  if (coinFlip2) {
    cols <- grep("Neg.Decr", colnames(dataMatrix))
   dataMatrix[i, cols] <- dataMatrix[i, cols] - seq(from = 5, to = 15, length.out = length(cols))
  }

  if (coinFlip3) {
    cols <- grep("Pos.Incr", colnames(dataMatrix))
   dataMatrix[i, cols] <- dataMatrix[i, cols] + seq(from = 3, to = 15, length.out = length(cols))
  }
}

# Illustrate data
heatmap(dataMatrix, Colv = NA, Rowv = NA, margins = c(7, 2), labRow = "")
```

```
svd_out <- svd(scale(dataMatrix))

library(lattice)
b_clr <- c("steelblue", "darkred")
key <- simpleKey(
  rectangles = TRUE, space = "top", points = FALSE,
  text = c("Positive", "Negative")
)
key$rectangles$col <- b_clr

b1 <- barchart(as.table(svd_out$v[, 1]),
               main = "First column",
               horizontal = FALSE, col = ifelse(svd_out$v[, 1] > 0,
                                                b_clr[1], b_clr[2]
               ),
               ylab = "Impact value",
             scales = list(x = list(rot = 55, labels = colnames(dataMatrix), cex = 1.1)),
               key = key
)

b2 <- barchart(as.table(svd_out$v[, 2]),
               main = "Second column",
               horizontal = FALSE, col = ifelse(svd_out$v[, 2] > 0,
                                                b_clr[1], b_clr[2]
               ),
               ylab = "Impact value",
             scales = list(x = list(rot = 55, labels = colnames(dataMatrix), cex = 1.1)),
               key = key
)

b3 <- barchart(as.table(svd_out$v[, 3]),
               main = "Third column",
               horizontal = FALSE, col = ifelse(svd_out$v[, 3] > 0,
                                                b_clr[1], b_clr[2]
               ),
               ylab = "Impact value",
             scales = list(x = list(rot = 55, labels = colnames(dataMatrix), cex = 1.1)),
               key = key
)

b4 <- barchart(as.table(svd_out$v[, 4]),
               main = "Fourth column",
               horizontal = FALSE, col = ifelse(svd_out$v[, 4] > 0,
                                                b_clr[1], b_clr[2]
               ),
               ylab = "Impact value",
             scales = list(x = list(rot = 55, labels = colnames(dataMatrix), cex = 1.1)),
               key = key
)

# Note that the fourth has the no pattern columns as the
# chosen pattern, probably partly because of the previous
```

```
# patterns already had been identified
print(b1, position = c(0, 0.5, .5, 1), more = TRUE)
print(b2, position = c(0.5, 0.5, 1, 1), more = TRUE)
print(b3, position = c(0, 0, .5, .5), more = TRUE)
print(b4, position = c(0.5, 0, 1, .5))

# Let's look at how well the SVD identifies
# the most influential columns
getSvdMostInfluential(dataMatrix,
                      quantile = .8,
                      similarity_threshold = .9,
                      plot_threshold = .05,
                      plot_selection = TRUE)
par(org_par)
```

`gnrlBezierPoints` *Generates a generalized Bézier line*

Description

This is a general form of bezier line that can be used for cubic, quadratic, and more advanced Bézier lines.

Usage

```
gnrlBezierPoints(ctrl_points, length_out = 100L)
```

Arguments

`ctrl_points`	The ctrl_points for the bezier control points. This should either be a `matrix` or a `data.frame`.
`length_out`	The length of the return points, i.e. how fine detailed the points should be.

Examples

```
library(grid)
grid.newpage()
l <- gnrlBezierPoints(data.frame(x = c(.1, -.1, .7, 1, 1, 0.1),
                                 y = c(.9, 0, 1, .8, .4, .1)),
                      length_out = 100)
grid.lines(l[,1], l[,2], gp=gpar(col="#550000", lwd = 4))

out_sizes <- 4:20
clrs <- colorRampPalette(c("orange", "darkblue"))(length(out_sizes))
for (i in out_sizes){
   l <- gnrlBezierPoints(data.frame(x = c(.1, -.1, .7, 1, 1, 0.1),
                                    y = c(.9, 0, 1, .8, .4, .1)),
                         length_out = i)
   grid.lines(l[,1], l[,2],
   gp=gpar(col=clrs[which(i == out_sizes)]))
}
```

`has` *An R alternative to the lodash* `has` *in JavaScript*

Description

This is a handy function for checking if item exist in a nested structure

Usage

```
has(sourceList, path)
```

Arguments

`sourceList`	The `list()`/`c()` that is to be searched for the element
`path`	A string that can be separated by [,] or ., the string `"elementname1.1.elementname"` the validity of the path - it only separates and tries to address that element with '[[]]'.

Value

Returns a boolean.

See Also

Other lodash similar functions: `retrieve()`

Examples

```
has(list(a = list(b = 1)), "a.b")
```

`insertRowAndKeepAttr` *Insert a row into a matrix*

Description

Inserts a row and keeps the attributes `copyAllNewAttributes`

Usage

```
insertRowAndKeepAttr(m, r, v = NA, rName = "")
```

Arguments

`m`	matrix
`r`	row number where the new row should be inserted
`v`	optional values for the new row
`rName`	optional character string: the name of the new row.

Value

`matrix` Returns a matrix with one more row than the provided matrix `m`

Author(s)

Max Gordon, Arne Henningsen

Examples

```
test <- matrix(1:4, ncol = 2)
attr(test, "wow") <- 1000
test <- insertRowAndKeepAttr(test, 2)
print(attr(test, "wow"))
```

`mergeDesc` *Prepares a matrix for* `htmlTable` *from a list*

Description

By putting all the output from the `getDescriptionStatsBy` into a list, naming each element that we want in an `rgroup` we can automatically merge everything and create an object ready for the `htmlTable`.

Usage

```
mergeDesc(..., htmlTable_args = list())
```

Arguments

`...`	One or more elements coming from `getDescriptionStatsBy`. You can also provide pure output from the `getDescriptionStatsBy` function and have the function merge this together with the `...` argument. *Note* that all elements must have the same by argument or you will not be able to merge it into a list.
`htmlTable_args`	Any arguments that should be passed to `htmlTable` function. The default is to remove any css formatting for the `rgroup`.

Value

`matrix` Returns a matrix object of class descList

The `rgroup` value

The value for the `rgroup` is by default the name of the list element. If you have passed a list without a name for that particular element or if you have passed a matrix it will look for a label set by the **Hmisc**`::label` function. For those elements that have only one row no `rgroup` is set, and the naming sequence is the same as above but with an additional `rownames` if the previous two turn out empty. All this behavior is exemplified in the example.

The `rgroup` value can be overridden by simply specifying a custom `rgroup` when calling the `htmlTable` function.

The `colnames` of the matrix

The function chooses the `colnames` from the first element in the `tlist`.

Examples

```
library(magrittr)
library(dplyr)
library(htmlTable)

data(mtcars)
mtcars %<>%
  mutate(am = factor(am, levels = 0:1, labels = c("Automatic", "Manual")),
         vs = factor(vs, levels = 0:1, labels = c("V-shaped", "straight")),
         drat_prop = drat > median(drat),
         drat_prop = factor(drat_prop,
                            levels = c(FALSE, TRUE),
                            labels = c("High ratio", "Low ratio")),
         carb_prop = carb > 2,
         carb_prop = factor(carb_prop,
                            levels = c(FALSE, TRUE),
                            labels = c("&le; 2", "&gt; 2")),
         across(c(gear, carb, cyl), factor))

# A simple bare-bone example
mtcars %>%
  getDescriptionStatsBy(`Miles per gallon` = mpg,
                        Weight = wt,
                        `Carborators &le; 2` = carb_prop,
                        by = am) %>%
  htmlTable(caption  = "Basic continuous stats from the mtcars dataset")
invisible(readline(prompt = "Press [enter] to continue"))

# For labeling & units we use set_column_labels/set_column_unit that use
# the Hmisc package annotation functions
mtcars %<>%
  set_column_labels(am = "Transmission",
                    mpg = "Gas",
                    wt = "Weight",
                    gear = "Gears",
                    disp = "Displacement",
                    vs = "Engine type",
```

```
                    drat_prop = "Rear axel ratio",
                    carb_prop = "Carburetors") %>%
  set_column_units(mpg = "Miles/(US) gallon",
                   wt = "10<sup>3</sup> lbs",
                   disp = "cu.in.")

mtcars %>%
  getDescriptionStatsBy(mpg,
                        wt,
                        `Gear&dagger;` = gear,
                        drat_prop,
                        carb_prop,
                        vs,
                        by = am,
                        header_count = TRUE,
                        use_units = TRUE,
                        show_all_values = TRUE)  %>%
  addHtmlTableStyle(pos.caption = "bottom") %>%
  htmlTable(caption  = "Stats from the mtcars dataset",
            tfoot = "&dagger; Number of forward gears")
invisible(readline(prompt = "Press [enter] to continue"))

# Using the default parameter we can
mtcars %>%
  getDescriptionStatsBy(mpg,
                        wt,
                        `Gear&dagger;` = gear,
                        drat_prop,
                        carb_prop,
                        vs,
                        by = am,
                        header_count = TRUE,
                        use_units = TRUE,
                        default_ref = c(drat_prop = "Low ratio",
                                        carb_prop = "&gt; 2"))  %>%
  addHtmlTableStyle(pos.caption = "bottom") %>%
  htmlTable(caption  = "Stats from the mtcars dataset",
            tfoot = "&dagger; Number of forward gears")
invisible(readline(prompt = "Press [enter] to continue"))

# We can also use lists
tll <- list()
tll[["Gear (3 to 5)"]] <- getDescriptionStatsBy(mtcars$gear, mtcars$am)
tll <- c(tll,
         list(getDescriptionStatsBy(mtcars$disp, mtcars$am)))

mergeDesc(tll,
          htmlTable_args = list(caption  = "Factored variables")) %>%
  htmlTable::addHtmlTableStyle(css.rgroup = "")
invisible(readline(prompt = "Press [enter] to continue"))

tl_no_units <- list()
tl_no_units[["Gas (mile/gallons)"]] <-
```

```
  getDescriptionStatsBy(mtcars$mpg, mtcars$am,
                        header_count = TRUE)
tl_no_units[["Weight (10<sup>3</sup> kg)"]] <-
  getDescriptionStatsBy(mtcars$wt, mtcars$am,
                        header_count = TRUE)
mergeDesc(tl_no_units,
          tll) %>%
  htmlTable::addHtmlTableStyle(css.rgroup = "")
invisible(readline(prompt = "Press [enter] to continue"))

# Other settings
mtcars$mpg[sample(1:NROW(mtcars), size = 5)] <- NA
getDescriptionStatsBy(mtcars$mpg,
                      mtcars$am,
                      statistics = TRUE)
invisible(readline(prompt = "Press [enter] to continue"))

# Do the horizontal version
getDescriptionStatsBy(mtcars$gear,
                      mtcars$am,
                      statistics = TRUE,
                      hrzl_prop = TRUE)
invisible(readline(prompt = "Press [enter] to continue"))

mtcars$wt_with_missing <- mtcars$wt
mtcars$wt_with_missing[sample(1:NROW(mtcars), size = 8)] <- NA
getDescriptionStatsBy(mtcars$wt_with_missing, mtcars$am, statistics = TRUE,
                      hrzl_prop = TRUE, total_col_show_perc = FALSE)
invisible(readline(prompt = "Press [enter] to continue"))

## Not run:
  ## There is also a LaTeX wrapper
  tll <- list(
    getDescriptionStatsBy(mtcars$gear, mtcars$am),
    getDescriptionStatsBy(mtcars$col, mtcars$am))

  latex(mergeDesc(tll),
        caption  = "Factored variables",
        file = "")

## End(Not run)
```

`mergeLists` *Merging of multiple lists*

Description

The merge allows for a recursive component where the lists are compared on the subelement. If one does not contain that element it will get NA in for those parameters.

Usage

```
mergeLists(
  ...,
  lapplyOutput = NULL,
  sortNames = getOption("Gmisc.mergeList.sort", default = TRUE)
)
```

Arguments

`...`	Any number of lists that you want to merge
`lapplyOutput`	The `lapply` function outputs a number of lists and this is for specifically merging all of those.
`sortNames`	Set to false if you don't want the names to be sorted. This can also be done via the option 'Gmisc.mergeList.sort'.

Value

Returns a list with all the given lists.

Examples

```
v1 <- list("a" = c(1, 2), b = "test 1", sublist = list(one = 20:21, two = 21:22))
v2 <- list("a" = c(3, 4), b = "test 2", sublist = list(one = 10:11, two = 11:12, three = 1:2))
mergeLists(v1, v2)
```

`moveBox` *Move a boxGrob*

Description

Moves a `boxGrob`/`boxPropGrob` by modifying it's `viewport`. This can be useful if you want to create a series of boxes whose position are relative to each other and depend on each box's width/height.

Usage

```
moveBox(
  element,
  x = NULL,
  y = NULL,
  space = c("absolute", "relative"),
  just = NULL
)
```

Arguments

`element`	A `boxGrob`/`boxPropGrob` object.
`x`	A `unit` element or a numeric that can be converted to a `npc` unit object.
`y`	A `unit` element or a numeric that can be converted to a `npc` unit object.
`space`	We can provide `absolute` that confers the box absolute position within the parent `viewport`. If `relative` the movement is related to the current position.
`just`	The justification of an argument as used by `viewport` some tiny differences: (1) you only want to change the justification in the vertical direction you can retain the existing justification by using `NA`, e.g. `c(NA, 'top')`, (2) if you specify only one string and that string is either `top` or `bottom` it will assume vertical justification.

Value

The element with updated

See Also

Other flowchart components: `align`, `boxGrob()`, `boxPropGrob()`, `connectGrob()`, `coords()`, `distance()`, `spread`

Examples

```
library(grid)
grid.newpage()

box <- boxGrob("A simple box", x = .5, y = .8)
moveBox(box, x = -.2, space = "relative")
```

`pathJoin` *A path join function*

Description

This function joins strings into a valid path. It is a simple version of python's `os.path.join` and fixes simple problems such as having/not having trailing / in each section.

Usage

```
pathJoin(...)
```

Arguments

`...`	A set of strings to join. Each may be a single string or a vector. If you provide vectors they can either be all of the same length or where there are two lengths where one is equal to 1.

Value

`string` A string with the merged path

Examples

```
pathJoin("my_base_path/helpers", "superfunction.R")
# 'my_base_path/helpers/superfunction.R'

base_dir <- "/home/tester/images"
out <- data.frame(filename = c("file1.png", "file2.png", "file3.png")) |>
  dplyr::mutate(full_path = pathJoin(base_dir, filename))
```

`prAddDescStats` *Add a p-value column to the results*

Description

Add a p-value column to the results

Usage

```
prAddDescStats(
  results,
  x,
  by,
  statistics,
  statistics.suppress_warnings,
  statistics.sig_lim,
  statistics.two_dec_lim,
  html
)
```

Arguments

`results`	The results that we want to add the column to
`x`	If a data.frame it will be used as the data source for the variables in the ... parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`by`	The variable that you want to split into different columns
`statistics`	Add statistics, fisher test for proportions and Wilcoxon for continuous variables. See details below for more customization.
`statistics.suppress_warnings`	Hide warnings from the statistics function.
`statistics.sig_lim`	The significance limit for < sign, i.e. p-value 0.0000312 should be < 0.0001 with the default setting.

`statistics.two_dec_lim`	The limit for showing two decimals. E.g. the p-value may be 0.056 and we may want to keep the two decimals in order to emphasize the proximity to the all-mighty 0.05 p-value and set this to 10^-2. This allows that a value of 0.0056 is rounded to 0.006 and this makes intuitive sense as the 0.0056 level as this is well below the 0.05 value and thus not as interesting to know the exact proximity to 0.05. *Disclaimer:* The 0.05-limit is really silly and debated, unfortunately it remains a standard and this package tries to adapt to the current standards in order to limit publication associated issues.
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting

Value

results with added column

`prAddDescUnitColumn` *Add a units column to the results*

Description

Add a units column to the results

Usage

```
prAddDescUnitColumn(results, x, use_units, units_column_name)
```

Arguments

`results`	The results that we want to add the column to
`x`	If a data.frame it will be used as the data source for the variables in the ... parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`use_units`	If the Hmisc package's units() function has been employed it may be interesting to have a column at the far right that indicates the unit measurement. If this column is specified then the total column will appear before the units (if specified as last). You can also set the value to `"name"` and the units will be added to the name as a parenthesis, e.g. Age (years).
`units_column_name`	The name of the units column. Used if use_units = TRUE

Value

results with added column

`prAddEmptyVals` *Convert the by-list into a matrix compatible format*

Description

Helper for [getDescriptionStatsBy] that fixes empty values in matrix so that they are compatible with the matrix

Usage

```
prAddEmptyVals(t, missing_value)
```

Arguments

`t`	Output from [prNumericDescs], [prPropDescs], or [prFactorDescs].
`missing_value`	Value that is substituted for empty cells. Defaults to "-"

Value

A fixed list

`prAddTotalDescColumn` *Add a total column to the results*

Description

Add a total column to the results

Usage

```
prAddTotalDescColumn(
  results,
  x,
  by,
  numbers_first,
  total_col_show_perc,
  show_all_values,
  useNA,
  useNA.digits,
  html,
  digits,
  continuous_fn,
  factor_fn,
  prop_fn,
  percentage_sign,
```

```
  default_ref,
  header_count = NULL,
  add_total_col
)
```

Arguments

`results`	The results that we want to add the column to
`x`	If a data.frame it will be used as the data source for the variables in the `...` parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`by`	The variable that you want to split into different columns
`numbers_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses ().
`total_col_show_perc`	This is by default true but if requested the percentages are suppressed as this sometimes may be confusing.
`show_all_values`	Show all values in proportions. For factors with only two values it is most sane to only show one option as the other one will just be a complement to the first, i.e. we want to convey a proportion. For instance sex - if you know gender then automatically you know the distribution of the other sex as it's 100 % - other %. To choose which one you want to show then set the `default_ref` parameter.
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`continuous_fn`	The method to describe continuous variables. The default is `describeMean`.
`factor_fn`	The method used to describe factors, see `describeFactors`.
`prop_fn`	The method used to describe proportions, see `describeProp`.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable.
`default_ref`	The default reference when dealing with proportions. When using 'dplyr' syntax ('tidyselect') you can specify a named vector/list for each column name.
`header_count`	Set to `TRUE` if you want to add a header count, e.g. Smoking; No. 25 observations, where there is a new line after the factor name. If you want a different text for the second line you can specifically use the `sprintf` formatting, e.g. "No. %s patients".
`add_total_col`	This adds a total column to the resulting table. You can also specify if you want the total column "first" or "last" in the column order.

Value

results with added column

`prBuildSubLabel` *Add a sub-label to* `boxPropGrob`

Description

Add a sub-label to `boxPropGrob`

Usage

```
prBuildSubLabel(label, prop, txt_gp, side = c("left", "right"))
```

Arguments

`label`	The text of the label
`prop`	The proportion
`txt_gp`	The style as defined by `gpar()`
`side`	The side that the label belongs to

Value

A `textGrob` with he additional attributes `width` and `height`.

`prConvert2Coords` *Converts an object to coordinates*

Description

Sometimes we have an object that can be either a box, a coprdinate, a unit or a numerical value and all we want is a `list` of coordinates that we can use for calculating distance, alignment and other things.

Usage

```
prConvert2Coords(obj)
```

Arguments

`obj`	A `boxGrob`, `boxPropGrob`, `coords` output, `unit` or a number ranging to be converted to a npc `unit`

Value

A `list` with all the points that `coords` returns

`prCreateBoxCoordinates` *Creates coordinates for box*

Description

Creates coordinates for box

Usage

```
prCreateBoxCoordinates(viewport_data, extra_coordinate_functions = NULL)
```

Arguments

`viewport_data`	The arguments that will be used for generating the `viewport`
`extra_coordinate_functions`	A list with named functions if we want additional parameters

Value

`list` of class `coords`

`prFactorDescs` *Helper to [getDescriptionStatsBy()]*

Description

Helper to [getDescriptionStatsBy()]

Usage

```
prFactorDescs(
  x,
  by,
  factor_fn,
  hrzl_prop,
  html,
  digits,
  digits.nonzero,
  numbers_first,
  useNA,
  useNA.digits,
  percentage_sign,
  missing_value,
  names_of_missing
)
```

Arguments

`x`	If a data.frame it will be used as the data source for the variables in the `...` parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`by`	The variable that you want to split into different columns
`factor_fn`	The method used to describe factors, see `describeFactors`.
`hrzl_prop`	This is default FALSE and indicates that the proportions are to be interpreted in a vertical manner. If we want the data to be horizontal, i.e. the total should be shown and then how these differ in the different groups then set this to TRUE.
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`numbers_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses ().
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable.
`missing_value`	Value that is substituted for empty cells. Defaults to "-"
`names_of_missing`	Optional character vector containing the names of returned statistics, in case all returned values for a given by level are missing. Defaults to NULL

Value

A [base::by] list

`prFixDescRownames` *Fix rownames for descriptive results*

Description

Helper for [getDescriptionStatsBy] that fixes row names

Usage

```
prFixDescRownames(results, t, name)
```

Arguments

`results`	A matrix with the results
`t`	The [base::by()] output
`name`	Name if row names are missing or the results is a single row

Value

The results with fixed names

`prGetDescHeader` *Retrieve basic description stats by header*

Description

Helper for [getDescriptionStatsBy] that retrieves the basic header names.

Usage

```
prGetDescHeader(by, html, header_count, already_table_format = FALSE)
```

Arguments

`by`	The variable that you want to split into different columns
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`header_count`	Set to `TRUE` if you want to add a header count, e.g. Smoking; No. 25 observations, where there is a new line after the factor name. If you want a different text for the second line you can specifically use the `sprintf` formatting, e.g. "No. %s patients".
`already_table_format`	Just a boolean as we use this in the total column

Value

A vector with basic headers

`print.Gmisc_list_of_boxes` *Output boxes*

Description

Outputs a list of boxes as produced by either the spread or align functions for boxGrobs.

Usage

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

Arguments

`x`	A list of a set of [‘boxGrob‘]/[‘boxPropGrob‘] to plot
`...`	Ignored argument

`prNumericDescs` *Helper to [getDescriptionStatsBy()]*

Description

Helper to [getDescriptionStatsBy()]

Usage

```
prNumericDescs(
  x,
  by,
  hrzl_prop,
  continuous_fn,
  html,
  digits,
  digits.nonzero,
  numbers_first,
  useNA,
  useNA.digits,
  percentage_sign,
  missing_value,
  names_of_missing
)
```

Arguments

`x`	If a data.frame it will be used as the data source for the variables in the ... parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`by`	The variable that you want to split into different columns
`hrzl_prop`	This is default FALSE and indicates that the proportions are to be interpreted in a vertical manner. If we want the data to be horizontal, i.e. the total should be shown and then how these differ in the different groups then set this to TRUE.
`continuous_fn`	The method to describe continuous variables. The default is `describeMean`.
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero
`numbers_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses ().
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable.
`missing_value`	Value that is substituted for empty cells. Defaults to "-"
`names_of_missing`	Optional character vector containing the names of returned statistics, in case all returned values for a given by level are missing. Defaults to NULL

Value

A [base::by] list

`prPasteVec` *Collapses a vector for throwing errors*

Description

The function collapses a vector into an output useful when throwing errors, e.g. 1:3 becomes '1', '2', '3'

Usage

```
prPasteVec(x)
```

Arguments

`x`	The vector

`prPropDescs` *Helper to [getDescriptionStatsBy()]*

Description

Helper to [getDescriptionStatsBy()]

Usage

```
prPropDescs(
  x,
  by,
  name,
  default_ref,
  prop_fn,
  html,
  digits,
  digits.nonzero,
  numbers_first,
  useNA,
  useNA.digits,
  percentage_sign,
  missing_value,
  names_of_missing,
  NEJMstyle
)
```

Arguments

`x`	If a data.frame it will be used as the data source for the variables in the `...` parameter. If it is a single variable it will be the core value that want the statistics for. In the print this is equivalent to the output of this function.
`by`	The variable that you want to split into different columns
`name`	The name of the row
`default_ref`	The default reference when dealing with proportions. When using ‘dplyr‘ syntax (‘tidyselect‘) you can specify a named vector/list for each column name.
`prop_fn`	The method used to describe proportions, see `describeProp`.
`html`	If HTML compatible output should be used. If `FALSE` it outputs LaTeX formatting
`digits`	The number of decimals used
`digits.nonzero`	The number of decimals used for values that are close to zero

`numbers_first`	If the number should be given or if the percentage should be presented first. The second is encapsulated in parentheses ().
`useNA`	This indicates if missing should be added as a separate row below all other. See `table` for `useNA`-options. *Note:* defaults to ifany and not "no" as `table` does.
`useNA.digits`	The number of digits to use for the missing percentage, defaults to the overall `digits`.
`percentage_sign`	If you want to suppress the percentage sign you can set this variable to FALSE. You can also choose something else that the default % if you so wish by setting this variable.
`missing_value`	Value that is substituted for empty cells. Defaults to "-"
`names_of_missing`	Optional character vector containing the names of returned statistics, in case all returned values for a given by level are missing. Defaults to NULL
`NEJMstyle`	Adds - no (%) at the end to proportions

Value

A [base::by] list

`retrieve` *An R alternative to the lodash* `get` *in JavaScript*

Description

This is a handy function for retrieving items deep in a nested structure without causing error if not found

Usage

```
retrieve(sourceList, path, default = NA)
```

Arguments

`sourceList`	The `list()/c()` that is to be searched for the element
`path`	A string that can be separated by [,] or ., the string `"elementname1.1.elementname"` is equivalent to `"elementname1[[1]]elementname"`. Note that the function doesn't check the validity of the path - it only separates and tries to address that element with '[[]]'.
`default`	The value to return if the element isn't found

Value

Returns a sub-element from `sourceList` or the `default` value.

See Also

Other lodash similar functions: `has()`

Examples

```
source <- list(a = list(b = 1, `odd.name` = 'I hate . in names', c(1,2,3)))
retrieve(source, "a.b")
retrieve(source, "a.b.1")
retrieve(source, "a.odd\\.name")
retrieve(source, "a.not_in_list")
```

`set_column_labels` *Add [Hmisc::label()] to multiple columns*

Description

Add label attribute using ‘dplyr‘ syntax using the [Hmisc::label()]

Usage

```
set_column_labels(x, ...)
```

Arguments

`x`	The data frame that we want to label
`...`	Variable names with their intended label, e.g. ‘mpg = "Miles per gallon"‘.

Value

The original data.frame

See Also

Other Hmisc helpers: `set_column_units()`

Examples

```
library(magrittr)
data(mtcars)
mtcars_with_labels <- mtcars %>%
  set_column_labels(mpg = "Gas",
                    cyl = "Cylinders",
                    hp = "Strength")
Hmisc::label(mtcars_with_labels$mpg)
```

`set_column_units` *Add [Hmisc::unit()] to multiple columns*

Description

Add label attribute using ‘dplyr‘ syntax using the [Hmisc::unit()]

Usage

```
set_column_units(x, ...)
```

Arguments

`x`	The data frame that we want to define units on
`...`	Variable names with their intended unit, e.g. ‘hp = "Hp"‘.

Value

The original data.frame

See Also

Other Hmisc helpers: `set_column_labels()`

Examples

```
library(magrittr)
data(mtcars)
mtcars_with_units <- mtcars %>%
  set_column_units(wt = "1000 lbs")
Hmisc::units(mtcars_with_units$wt)
```

`spread` *Spread boxes*

Description

Spreads a set of `boxGrob`/`boxPropGrob` in either horizontal or vertical direction.

Usage

```
spreadVertical(..., .from = NULL, .to = NULL, .type = c("between", "center"))

spreadHorizontal(..., .from = NULL, .to = NULL, .type = c("between", "center"))
```

Arguments

`...`	A set of boxes to spread. Can also be a `list` of boxes.
`.from`	A box that the spread originates from. If left empty the entire `viewport` will be used.
`.to`	A box that the spread ends at. If left empty the entire `viewport` will be used.
`.type`	If `between` the space between the boxes will be identical while `center` has each box's center is equally distributed.

Value

`list` with the boxes that have been spread

See Also

Other flowchart components: `align`, `boxGrob()`, `boxPropGrob()`, `connectGrob()`, `coords()`, `distance()`, `moveBox()`

Examples

```
library(grid)
grid.newpage()

box1 <- boxGrob("B1", x = .2, y = .8)
box2 <- boxGrob("B2\n\n\neach\nbox\neven\nspace\nbetween", x = .2, y = .8)
box3 <- boxGrob("B3", x = .2, y = .8)
box4 <- boxGrob("B4", x = .2, y = .8)
box5 <- boxGrob("B5", x = .2, y = .8)

spread_boxes <- spreadVertical(box1,
                               box2,
                               box3,
                               a = box4,
                               box5,
                               .type = "between")
for (b in spread_boxes) {
  print(b)
}

box1 <- boxGrob("B1\n\nanother group\ncenter oriented", x = .6, y = .8)
box2 <- boxGrob("B2", x = .6, y = .8)
box3 <- boxGrob("B3", x = .6, y = .8)
box4 <- boxGrob("B4", x = .6, y = .8)
box5 <- boxGrob("B5", x = .6, y = .8)

spread_boxes <- spreadVertical(box1,
                               box2,
                               box3,
                               a = box4,
                               box5,
                               .type = "center")
```

```
for (b in spread_boxes) {
  print(b)
}
```

`time2spanTxt` *A dense time-span text*

Description

When adding a time span text we often don't want to write *3 jun - 10 jun* but shorten it to *3 - 10 jun* while retaining month and year info only if the span crosses between months or years.

Usage

```
time2spanTxt(
  times,
  day_month_glue_txt = getOption("Gmisc_time2spanTxt_day_month", default =
    "{mday(time)} {month(time, label = TRUE)}"),
  full_year_format = getOption("Gmisc_time2spanTxt_full_year", default =
    "{mday(time)} {month(time, label = TRUE)} {year(time)}"),
  start_stop_glue_txt = getOption("Gmisc_time2spanTxt_template", default =
    "{start} to {stop}")
)
```

Arguments

`times`	The dates or POSIX timestamps to used for time span
`day_month_glue_txt`	The `glue` string to format days and months with `time` as the time input
`full_year_format`	The `glue` string to format the full year with `time` as the time input
`start_stop_glue_txt`	The string used in the `glue` for putting the start and stop dates together into one string

Details

There are options that can be set using the `options`:

- `Gmisc_time2spanTxt_day_month` The date with day + month as formatted by `glue` where the time is passed as `time`.
- `Gmisc_time2spanTxt_full_year` The full date with day + month + year as formatted by `glue` where the time is passed as `time`.
- `Gmisc_time2spanTxt_template` The merge of the stop & start elements using `glue`.

Value

`string` A string describing the time span

Examples

```
time2spanTxt(as.POSIXct(c("2020-01-02", "2020-03-01", NA)))
# 2 jan to 1 mar
```

`Transition-class` *A reference class for generating transition plots*

Description

This class simplifies the creating of transition plots. It also allows for advanced multi-column transitions.

Details

Transition plots are a type of *Sankey diagrams*. These are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. See Wikipedia for details.

Fields

`id` Optional id. The render uses named viewports that require a unique id if multiple transition plots are combined. In order to avoid having overlapping graphs we need to generate a unique id for each viewport and thus this variable exists. If left empty it will create a counter that is stored in the `options` (`"Gmisc.transitionClassCounter"`) and each viewport will have the name preceded with `tc_[0-9]+`. Set this if you intend to use `seekViewport`.

`transitions` This is a >= 3 dimensional array with the transitions. Should not be directly accessed.

`box_width` The box width

`box_txt` The texts of each box

`box_label` Box labels

`box_label_pos` The label's positions, either "top"/"bottom"

`box_label_cex` The size of the box labels

`box_cex` The font-size multiplier for the text within the boxes

`arrow_type` The type of arrow to use, defaults to "gradient", but can also be "simple". The corresponding functions are `bezierArrowGradient`, and `bezierArrowSmpl`. *Note* The `bezierGrob` has been deprecated as it is no longer faster than the bezier arrows and there is a difference in design.

`arrow_clr` The arrow color

`arrow_rez` The resolution of the arrow

`vertical_space` The space between the boxes

`fill_clr` The box fill color

`clr_bar` Shows a color bar if there are proportions. Can be `"none"`, `"top"`, `"bottom"`

`clr_bar_clrs` Extracts the colors for the color bar from the `fill_clr` if none is provided `clr_bar_cex` The size of the ticks in the color bar

`clr_bar_subspace` If little or no difference exists at the low/high proportions of the spectrum then it can be of interest to focus the color change to the center leaving the tails constant

`clr_bar_labels` The labels of the color bars. Defaults to the dim names for the proportions.

`txt_clr` The text color within the boxes

`txt_gpar` Similar to ‘txt_clr‘ but for more advanced styling with fontfamily (see [grid::gpar()]). *Note* that col & cex are overridden.

`title` The plot title if any

`title_cex` The font-size multiplier for the title

`skip_shadows` Skip the shadow effect on the boxes

`mar` The margins for the plot.

`min_lwd` The minimum line width that is still shown. The pixels will most likely not have the same fine resolution as the data and you therefore may want to hide lines that are smaller than a certain amount.

`max_lwd` The maximum line width to show

`lwd_prop_type` The line can either be proportional to the `"set"` of transitions (group of two box columns), to `"all"` transitions, or to each `"box"`. It defaults to `"all"`.

`data` Internal storage variable. Should not be accessed directly.

Methods

`addBoxStyle(fill, txt, gpar)` Adds colors or extends existing one so that they match the transition matrix. The fill corresponds to the fill_clr and txt corresponds to the txt_clr. If the colors are missing and the transitions consist of only two columns the default colors will be used. If the matrix is being extended and these values are missing the values from the previous last column will be used for the default columns.

`addTransitions(mtrx, label, txt, fill_clr, txt_clr, txt_gpar)` Add a transition matrix. The input has to be a numerical matrix between 2 and 3 dimensions. If you don’t provide the txt field the box’ text field will be deduced from the transition matrix’ dimnames. The fill_clr and txt_clr are passed on to the `addBoxStyle` function.

`arrowWidths(set_no, add_width)` Retrieves the details regarding arrow sizes for each arrow within the transition group

`boxPositions(col)` The box positions as a list with scalars for the positions:

1. *x* The center x-position
2. *y* The center y-position
3. *right* The right edge
4. *left* The left edge
5. *top* The top edge
6. *bottom* The bottom edge
7. *height* The box height
8. *width* The box width
9. *unit* The unit used for the values (npc)

`boxSizes(col)` Gets the size of the boxes. The `col` argument should be either an integer or 'last'

`getDim()` Gets the current dimensions of the transitions

`getTransitionSet(no, reduce_dim = FALSE)` Gets a specific set of transitions. If the `reduce_dim` is set to TRUE it will only return a 2-dimensional matrix even if the original has a 3rd proportions dimension

`getYProps(col)` Gets the proportions after removing the `vertical_space` between the boxes

`initialize(transitions, label, txt, fill_clr, txt_clr, txt_gpar, id, ...)` Set up a Transition object. The `transitions` should be a 2D or 3D matrix as defined in the `$addTransitions` section and not as later internally stored.

`noCols()` Gets the number of columns, i.e. the number of transitions

`noRows(no)` Gets the number of boxes in each row. If multiple rows the number of rows may differ betwen each transition matrix we therefore need to specify what transitions that we refer to. If no value is specified it returns all of them.

`render(new_page = TRUE)` Call this to render the full graph. The `new_page` argument is for creating a new plot, set this to `FALSE` if you want to combine this plot with another or if you have additional viewports that you intend to use.

`trnstnSizes(set_no)` Gets the transitions per box as a 2D matrix. For the proportions it also adds an attribute `attr('props', prop_mtrx)` that is a 2D matrix with the corresponding proportions.

Examples

```
# Transitions
set.seed(1)
n <- 10
my_data <-
  data.frame(
    Var_a = sample(c(
      "Test 1",
      "Test 2",
      "Test 3"
    ),
    size = n,
    replace = TRUE,
    prob = 3:1
    ),
    Var_b = sample(c(
      "Test 1",
      "Test 2",
      "Test 3"
    ),
    size = n,
    replace = TRUE,
    prob = 1:3
    )
  )
mtrx <- with(
  my_data,
  table(Var_a, Var_b)
```

```
)

# Initialize the transition plot
transitions <- getRefClass("Transition")$new(mtrx,
                                             label = c("Before", "After"))

# Render the plot
transitions$render()
```

`transitionPlot` *A transition plot*

Description

This plot's purpose is to illustrate how states change before and after. In my research I use it before surgery and after surgery but it can be used in any situation where you have a change from one state to another

Usage

```
transitionPlot(
  transition_flow,
  type_of_arrow = c("grid", "simple", "gradient"),
  box_txt = rownames(transition_flow),
  tot_spacing = 0.2,
  box_width = 1/4,
  fill_start_box = "darkgreen",
  txt_start_clr = "white",
  fill_end_box = fill_start_box,
  txt_end_clr = txt_start_clr,
  cex = 2,
  min_lwd = if (type_of_arrow == "grid") 1 else unit(0.1, "mm"),
  max_lwd = if (type_of_arrow == "grid") 6 else unit(5, "mm"),
  lwd_prop_total = TRUE,
  arrow_clr = "#000000",
  abs_arrow_width = FALSE,
  overlap_bg_clr = "#FFFFFF",
  overlap_order = 1:nrow(transition_flow),
  overlap_add_width = if (type_of_arrow == "grid") 1.5 else unit(1, "mm"),
  box_prop,
  mar = unit(rep(3, times = 4), "mm"),
  main = NULL,
  box_label = NULL,
  box_label_pos = "top",
  box_label_cex = cex,
  color_bar = TRUE,
  color_bar_cex = cex * 0.33,
  color_bar_labels,
```

```
  color_bar_subspace = NULL,
  new_page = FALSE
)
```

Arguments

`transition_flow`	This should be a matrix with the size of the transitions. The unit for each cell should be number of observations, row/column-proportions will show incorrect sizes. The matrix needs to be square. The best way to generate this matrix is probably just do a `table(starting_state, end_state)`. The rows represent the starting positions, while the columns the end positions. I.e. the first rows third column is the number of observations that go from the first class to the third class.
`type_of_arrow`	The types of arrow may be grid, simple, or gradient. Simple grid arrows are the `bezierGrob` arrows (not that pretty), simple is the `bezierArrowSmpl` that I've created to get a more exact control of the arrow position and width, while gradient corresponds to `bezierArrowGradient` allowing the arrow to have a fill color that slowly turns into the color of the arrow.
`box_txt`	The text to appear inside of the boxes. If you need line breaks then you need to manually add a \n inside the string.
`tot_spacing`	The proportion of the vertical space that is to be left empty. It is then split evenly between the boxes.
`box_width`	The width of the box. By default the box is one fourth of the plot width.
`fill_start_box`	The fill color of the start boxes. This can either be a single value or a vector if you desire different colors for each box. If you specify box_prop then this has to be a 2 column matrix.
`txt_start_clr`	The text color of the start boxes. This can either be a single value or a vector if you desire different colors for each box. If you specify box_prop then this has to be a 2 column matrix.
`fill_end_box`	The fill color of the end boxes. This can either be a single value or a vector if you desire different colors for each box. If you specify box_prop then this has to be a 2 column matrix.
`txt_end_clr`	The text color of the end boxes. This can either be a single value or a vector if you desire different colors for each box. If you specify box_prop then this has to be a 2 column matrix.
`cex`	The cex `gpar` of the text
`min_lwd`	The minimum width of the line that we want to illustrate the tranisition with.
`max_lwd`	The maximum width of the line that we want to illustrate the tranisition with.
`lwd_prop_total`	The width of the lines may be proportional to either the other flows from that box, or they may be related to all flows. This is a boolean parameter that is set to true by default, i.e. relating to all flows.
`arrow_clr`	The color of the arrows. Usually black, can be a vector indicating each arrow from first to last arrow (counting from the top). If the vector is of the same length as the boxes then all box arrows will have the same color (that is all the arrows stemming from the left boxes)

`abs_arrow_width`	The width can either be absolute, i.e. each arrow headed for a box has the exact same width. The alternative is that the width is related to the line width.
`overlap_bg_clr`	In order to enhance the 3D perspective and to make it easier to follow arrows the arrows have a background color to separate them from those underneath.
`overlap_order`	The order from first->last for the lines. This means that the last line will be on top while the first one will appear at the bottom. This should be provided as a vector.
`overlap_add_width`	The width of the white cross-over line. You can specify this as a scalar multiplication of the current line width. In case of non-grid arrows then you can also have this as a unit which is recommended as it looks better. If the scalar is < 1 then the overlap is ignored.
`box_prop`	If you want the boxes to have proportions indicating some other factors then input a matrix with quantiles for the proportions. Note the size must be `nrow(transition_flow) x 2`.
`mar`	A numerical vector of the form c(bottom, left, top, right) of the type `unit()`
`main`	The title of the plot if any, default `NULL`
`box_label`	A vector of length 2 if you want to label each box column
`box_label_pos`	The position of the label, either `'top'` or `'bottom'`
`box_label_cex`	The cex of the label, defaults to the default cex
`color_bar`	If you have proportions inside the transition_flow variable then the color_bar will automatically appear at the bottom unless you set this to `FALSE`
`color_bar_cex`	The size of the tick labels for the color bar
`color_bar_labels`	The labels of the two proportions that make up the color bar. Defaults to the labels of the third dimension for the `transition_flow` argument.
`color_bar_subspace`	If there is little or no difference at the low/high proportions of the spectrum then it can be of interest to focus the color change to the center leaving the tails constant
`new_page`	If you want the plot to appear on a new blank page then set this to `TRUE`, by default it is `FALSE`.

Value

void

Examples

```
# This example does not run since it
# takes a little while to assemble the
# arrows and RMD Check complains that this
# is more than allowed for
library(grid)
par_org <- par(ask = TRUE)
```

```
# Settings
no_boxes <- 3
# Generate test setting
transition_matrix <- matrix(NA, nrow = no_boxes, ncol = no_boxes)
transition_matrix[1, ] <- 200 * c(.5, .25, .25)
transition_matrix[2, ] <- 540 * c(.75, .10, .15)
transition_matrix[3, ] <- 340 * c(0, .2, .80)

grid.newpage()
transitionPlot(transition_matrix,
  box_txt = c("First", "Second", "Third"),
  type_of_arrow = "simple",
  min_lwd = unit(1, "mm"),
  max_lwd = unit(6, "mm"),
  overlap_add_width = unit(1, "mm")
)


# Setup proportions
box_prop <- cbind(c(1, 0, 0.5), c(.52, .2, .8))
# From the Set2 Colorbrewer
start_box_clr <- c("#8DA0CB", "#FC8D62")
# Darken the colors slightly
end_box_clr <- c(
  colorRampPalette(c(start_box_clr[1], "#000000"))(10)[2],
  colorRampPalette(c(start_box_clr[2], "#000000"))(10)[2]
)
# Create a new grid
grid.newpage()
transitionPlot(transition_matrix,
  box_prop = box_prop,
  fill_start_box = start_box_clr, fill_end_box = end_box_clr,
  txt_start_clr = c("#FFFFFF", "#000000"), txt_end_clr = c("#FFFFFF", "#000000"),
  box_txt = c("First", "Second", "Third"),
  type_of_arrow = "gradient",
  min_lwd = unit(1, "mm"),
  max_lwd = unit(10, "mm"),
  overlap_add_width = unit(1, "mm")
)
par(par_org)
```

`yamlDump` *Outputs an object*

Description

Manually viewing a list object can be tricky where the natural print can be hard to work through. The config format *yaml* is increadibly dense and useful not only for writing configs but also viewing them which ‘yamlDump‘ helps with.

Usage

```
yamlDump(x)
```

Arguments

`x`	An object that `as.yaml` accepts

Value

```
void
```

Examples

```
some_fancy_list <- list(complex = list(some_data = 1:3,
                                       other_data = list(name = ”Max”)),
                        simple = ”awesome overview”)
yamlDump(some_fancy_list)
#complex:
#  some_data:
#  - 1
#  - 2
#  - 3
#  other_data:
#    name: Max
#simple: awesome overview

# If you got a character json you can also input it directly
# and the function will automatically convert it to a list
yamlDump('{”a”:{”b”:[”1”]}}')
```

Index