Package 'rearrr'

March 3, 2025

Title Rearranging Data

Version 0.3.5

Description Arrange data by a set of methods. Use rearrangers to reorder data points and mutators to change their values. From basic utilities, to centering the greatest value, to swirling in 3-dimensional space, 'rearrr' enables creativity when plotting and experimenting with data.

License MIT + file LICENSE

URL `https://github.com/ludvigolsen/rearrr`

BugReports `https://github.com/ludvigolsen/rearrr/issues`

Depends R (>= 3.5)

Imports checkmate (>= 2.0.0), dplyr (>= 0.8.5), lifecycle, plyr, purrr (>= 0.3.4), rlang (>= 0.4.7), R6, stats, tibble, utils

Suggests covr, ggplot2, knitr, testthat, tidyr, xpectr (>= 0.4.3)

RdMacros lifecycle

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Ludvig Renbo Olsen [aut, cre] (<`https://orcid.org/0009-0006-6798-7454`>)

Maintainer Ludvig Renbo Olsen <`r-pkgs@ludvigolsen.dk`>

Repository CRAN

Date/Publication 2025-03-03 17:10:02 UTC

Contents

`angle` *Calculate the angle to an origin*

Description

[Experimental]

Calculates the angle between each data point $(x2, y2)$ and the origin $(x1, y1)$ with:

$$atan2(y2 - y1, x2 - x1)$$

And converts to degrees `[0-360)`, measured counterclockwise from the `{x > x1, y = y1}` line.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and finding the angle to e.g. the centroid of each group.

Usage

```
angle(
  data,
  x_col = NULL,
  y_col = NULL,
  origin = NULL,
  origin_fn = NULL,
  degrees_col_name = ".degrees",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`x_col`	Name of x column in `` `data` ``. If `NULL` and `` `data` `` is a `vector`, the index of `` `data` `` is used. If `` `data` `` is a `data.frame`, it must be specified.
`y_col`	Name of y column in `` `data` ``. If `` `data` `` is a `data.frame`, it must be specified.
`origin`	Coordinates of the origin to calculate angle to. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`

`degrees_col_name`	Name of new column with the degrees.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with the additional columns (degrees and origin coordinates).

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other measuring functions: `distance()`, `vector_length()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)

# Calculate angles in the two dimensions (x and y)
# With the origin at x=0.5, y=0.5
df_angles <- angle(
  data = df,
  x_col = "x",
  y_col = "y",
  origin = c(0.5, 0.5)
)
df_angles

# Plot points with degrees
# Degrees are measured counterclockwise around the
# positive side of the x-axis
if (has_ggplot){
  df_angles %>%
    ggplot(aes(x = x, y = y, color = .degrees)) +
    geom_segment(aes(x = 0.5, xend = 1, y = 0.5, yend = 0.5), color = "magenta") +
    geom_point() +
    theme_minimal()
}

# Calculate angles to the centroid for each group in 'g'
angle(
  data = dplyr::group_by(df, g),
  x_col = "x",
  y_col = "y",
  origin_fn = centroid
)
```

---

apply_transformation_matrix

Apply transformation matrix to a set of columns

Description

[Experimental]

Perform matrix multiplication with a transformation matrix and a set of `data.frame` columns.

The data points in `` `data` `` are moved prior to the transformation, to bring the origin to `0` in all dimensions. After the transformation, the inverse move is applied to bring the origin back to its original position. See `` `Details` `` section.

The columns in `` `data` `` are transposed, making the operation (without the origin movement):

$$mat data[, cols]^T$$

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and transforming around e.g. the centroid of each group.

Usage

```
apply_transformation_matrix(
  data,
  mat,
  cols,
  origin = NULL,
  origin_fn = NULL,
  suffix = "_transformed",
  keep_original = TRUE,
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`mat`	Transformation `matrix`. Must have the same number of columns as `` `cols` ``.
`cols`	Columns to mutate values of. Must be specified when `` `data` `` is a `data.frame`.
`origin`	Coordinates of the origin. `Vector` with the same number of elements as `` `cols` `` (i.e. origin_x, origin_y, ...). Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Details

Example with 2 columns (`x`, `y`) and a 2x2 transformation matrix:

- Move origin to `(0, 0)`:
  `x = x - origin_x`
  `y = y - origin_y`
- Convert to transposed matrix:
  `data_mat = rbind(x, y)`
- Matrix multiplication:
  `transformed = mat %*% data_mat`
- Move origin to original position (after extraction from `transformed`):
  `x = x + origin_x`
  `y = y + origin_y`

Value

`data.frame` (`tibble`) with the new, transformed columns and the origin coordinates.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: cluster_groups(), dim_values(), expand_distances(), expand_distances_each(), flip_values(), roll_values(), rotate_2d(), rotate_3d(), shear_2d(), shear_3d(), swirl_2d(), swirl_3d()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(3)

# Create a data frame
df <- data.frame(
  "x" = 1:12,
  "y" = 13:24,
  "z" = runif(12),
  "g" = c(
    1, 1, 1, 1, 2, 2,
    2, 2, 3, 3, 3, 3
  )
)

# Apply identity matrix
mat <- matrix(c(1, 0, 0, 0, 1, 0, 0, 0, 1), nrow = 3)
apply_transformation_matrix(
  data = df,
  mat = mat,
  cols = c("x", "y", "z"),
  origin = c(0, 0, 0)
)

# Apply rotation matrix
# 90 degrees around z-axis
# Origin is the most centered point
mat <- matrix(c(0, 1, 0, -1, 0, 0, 0, 0, 1), nrow = 3)
res <- apply_transformation_matrix(
  data = df,
  mat = mat,
  cols = c("x", "y", "z"),
  origin_fn = most_centered
)

# Plot the rotation
# z wasn't changed so we plot x and y
if (has_ggplot){
  res %>%
    ggplot(aes(x = x, y = y)) +
    geom_point() +
    geom_point(aes(x = x_transformed, y = y_transformed)) +
    theme_minimal()
}

# Apply rotation matrix to grouped data frame
# Around centroids
# Same matrix as before
res <- apply_transformation_matrix(
  data = dplyr::group_by(df, g),
  mat = mat,
  cols = c("x", "y", "z"),
  origin_fn = centroid
)

# Plot the rotation
if (has_ggplot){
  res %>%
    ggplot(aes(x = x, y = y, color = g)) +
    geom_point() +
    geom_point(aes(x = x_transformed, y = y_transformed)) +
    theme_minimal()
}
```

`center_max` *Centers the highest value with values decreasing around it*

Description

[Experimental]

The highest value is positioned in the middle with the other values decreasing around it.

Example:

The column values:

`c(1, 2, 3, 4, 5)`

are **ordered as**:

`c(1, 3, 5, 4, 2)`

Usage

```
center_max(data, col = NULL, shuffle_sides = FALSE)
```

Arguments

`data`	`data.frame` or `vector`.
`col`	Column to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`shuffle_sides`	Whether to shuffle which elements are left and right of the center. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other rearrange functions: `center_min()`, `closest_to()`, `furthest_from()`, `pair_extremes()`, `position_max()`, `position_min()`, `rev_windows()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "C" = LETTERS[1:10],
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Center by the index (row numbers)
center_max(df)

# Center by each of the columns
center_max(df, col = "A")
center_max(df, col = "B")
center_max(df, col = "C")

# Randomize which elements are left and right of the center
center_max(df, col = "A", shuffle_sides = TRUE)

# Grouped by G
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  center_max(col = "A")
```

```
# Plot the centered values
plot(x = 1:10, y = center_max(df, col = "B")$B)
plot(x = 1:10, y = center_max(df, col = "B", shuffle_sides = TRUE)$B)
```

`center_min` *Centers the lowest value with values increasing around it*

Description

[Experimental]

The lowest value is positioned in the middle with the other values increasing around it.

Example:

The column values:

`c(1, 2, 3, 4, 5)`

are **ordered as**:

`c(5, 3, 1, 2, 4)`

Usage

```
center_min(data, col = NULL, shuffle_sides = FALSE)
```

Arguments

`data`	`data.frame` or `vector`.
`col`	Column to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`shuffle_sides`	Whether to shuffle which elements are left and right of the center. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other rearrange functions: `center_max()`, `closest_to()`, `furthest_from()`, `pair_extremes()`, `position_max()`, `position_min()`, `rev_windows()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "C" = LETTERS[1:10],
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Center by the index (row numbers)
center_min(df)

# Center by each of the columns
center_min(df, col = "A")
center_min(df, col = "B")
center_min(df, col = "C")

# Randomize which elements are left and right of the center
center_min(df, col = "A", shuffle_sides = TRUE)

# Grouped by G
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  center_min(col = "A")

# Plot the centered values
plot(x = 1:10, y = center_min(df, col = "B")$B)
plot(x = 1:10, y = center_min(df, col = "B", shuffle_sides = TRUE)$B)
```

`centroid` *Find the coordinates for the centroid*

Description

[Experimental]

Calculates the mean of each passed `vector`/`column`.

Usage

```
centroid(..., cols = NULL, na.rm = FALSE)
```

Arguments

`...`	`Numeric` `vectors` or a single `data.frame`.
`cols`	Names of columns to use when `` `...` `` is a single `data.frame`.
`na.rm`	Whether to ignore missing values when calculating means. (Logical)

Value

Means of the supplied `vectors/columns`. Either as a `vector` or a `data.frame`.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other coordinate functions: `create_origin_fn()`, `is_most_centered()`, `midrange()`, `most_centered()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create three vectors
x <- runif(10)
y <- runif(10)
z <- runif(10)

# Find centroid coordinates
# Aka. the means of each vector
centroid(x, y, z)

#
# For data.frames
#

# Create data frame
df <- data.frame(
  "x" = x,
  "y" = y,
  "z" = z,
  "g" = rep(1:2, each = 5)
)
```

```
# Find centroid coordinates
# Aka. the means of each column
centroid(df, cols = c("x", "y", "z"))

# When 'df' is grouped
df %>%
  dplyr::group_by(g) %>%
  centroid(cols = c("x", "y", "z"))
```

---

circularize *Create x-coordinates so the points form a circle*

Description

[Experimental]

Create the x-coordinates for a `vector` of y-coordinates such that they form a circle.

This will likely look most like a circle when the y-coordinates are somewhat equally distributed, e.g. a uniform distribution.

Usage

```
circularize(
  data,
  y_col = NULL,
  .min = NULL,
  .max = NULL,
  offset_x = 0,
  keep_original = TRUE,
  x_col_name = ".circle_x",
  degrees_col_name = ".degrees",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`y_col`	Name of column in `` `data` `` with y-coordinates to create x-coordinates for.
`.min`	Minimum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`.max`	Maximum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`offset_x`	Value to offset the x-coordinates by.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`x_col_name`	Name of new column with the x-coordinates.

`degrees_col_name`	Name of new column with the angles in degrees. If `NULL`, no column is added. Angling is counterclockwise around `(0, 0)` and starts at `(max(x), 0)`.
`origin_col_name`	Name of new column with the origin coordinates (center of circle). If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with the added x-coordinates and the angle in degrees.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other forming functions: `hexagonalize()`, `square()`, `triangularize()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "y" = runif(200),
  "g" = factor(rep(1:5, each = 40))
)

# Circularize 'y'
df_circ <- circularize(df, y_col = "y")
df_circ

# Plot circle
if (has_ggplot){
  df_circ %>%
    ggplot(aes(x = .circle_x, y = y, color = .degrees)) +
    geom_point() +
    theme_minimal()
}

#
# Grouped circularization
```

```
#

# Circularize 'y' for each group
# First cluster the groups a bit to move the
# circles away from each other
df_circ <- df %>%
  cluster_groups(
    cols = "y",
    group_cols = "g",
    suffix = "",
    overwrite = TRUE
  ) %>%
  dplyr::group_by(g) %>%
  circularize(
    y_col = "y",
    overwrite = TRUE
  )

# Plot circles
if (has_ggplot){
  df_circ %>%
    ggplot(aes(x = .circle_x, y = y, color = g)) +
    geom_point() +
    theme_minimal()
}

#
# Specifying minimum value
#

# Specify minimum value manually
df_circ <- circularize(df, y_col = "y", .min = -2)
df_circ

# Plot circle
if (has_ggplot){
  df_circ %>%
    ggplot(aes(x = .circle_x, y = y, color = .degrees)) +
    geom_point() +
    theme_minimal()
}

#
# Multiple circles by contraction
#

# Start by circularizing 'y'
df_circ <- circularize(df, y_col = "y")

# Contract '.circle_x' and 'y' towards the centroid
# To contract with multiple multipliers at once,
# we wrap the call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = 1:10 / 10,
  .f = function(mult) {
    expand_distances(
      data = df_circ,
      cols = c(".circle_x", "y"),
      multiplier = mult,
      origin_fn = centroid,
      overwrite = TRUE
    )
  }
)
df_expanded

if (has_ggplot){
  df_expanded %>%
    ggplot(aes(
      x = .circle_x_expanded, y = y_expanded,
      color = .degrees, alpha = .multiplier
    )) +
    geom_point() +
    theme_minimal()
}
```

---

closest_to *Orders values by shortest distance to an origin*

Description

[Experimental]

Values are ordered by how close they are to the origin.

In 1d (when `` `cols` `` has length 1), the origin can be thought of as a target value. In *n* dimensions, the origin can be thought of as coordinates.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and ordering the rows by their distance to the centroid of each group.

The `*_vec()` version takes and returns a `vector`.

Example:

The column values:

`c(1, 2, 3, 4, 5)`

and `origin` = `2`

are **ordered as**:

`c(2, 1, 3, 4, 5)`

Usage

```
closest_to(
  data,
  cols = NULL,
  origin = NULL,
  origin_fn = NULL,
  shuffle_ties = FALSE,
  origin_col_name = ".origin",
  distance_col_name = ".distance",
  overwrite = FALSE
)

closest_to_vec(data, origin = NULL, origin_fn = NULL, shuffle_ties = FALSE)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Column(s) to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`origin`	Coordinates of the origin to calculate distances to. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `` `NULL` ``.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`shuffle_ties`	Whether to shuffle elements with the same distance to the origin. (Logical)
`origin_col_name`	Name of new column with the origin coordinates. If `` `NULL` ``, no column is added.
`distance_col_name`	Name of new column with the distances to the origin. If `` `NULL` ``, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other rearrange functions: `center_max()`, `center_min()`, `furthest_from()`, `pair_extremes()`, `position_max()`, `position_min()`, `rev_windows()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Other distance functions: `dim_values()`, `distance()`, `expand_distances()`, `expand_distances_each()`, `furthest_from()`, `swirl_2d()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Closest to 3 in a vector
closest_to_vec(1:10, origin = 3)

# Closest to the third row (index of data.frame)
closest_to(df, origin = 3)$index

# By each of the columns
closest_to(df, cols = "A", origin = 3)$A
closest_to(df, cols = "A", origin_fn = most_centered)$A
closest_to(df, cols = "B", origin = 0.5)$B
closest_to(df, cols = "B", origin_fn = centroid)$B

# Shuffle the elements with the same distance to the origin
closest_to(df,
  cols = "A",
  origin_fn = create_origin_fn(median),
  shuffle_ties = TRUE
)$A

# Grouped by G
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
    closest_to(
      cols = "A",
      origin_fn = create_origin_fn(median)
    )

# Plot the rearranged values
plot(
  x = 1:10,
  y = closest_to(df,
    cols = "B",
    origin_fn = create_origin_fn(median)
  )$B,
  xlab = "Position",
  ylab = "B"
)
plot(
  x = 1:10,
  y = closest_to(df,
    cols = "A",
    origin_fn = create_origin_fn(median),
    shuffle_ties = TRUE
  )$A,
  xlab = "Position",
  ylab = "A"
)

# In multiple dimensions
df %>%
  closest_to(cols = c("A", "B"), origin_fn = most_centered)
```

`cluster_groups` *Move data points into clusters*

Description

[Experimental]

Transform values such that the elements in each group move closer to their centroid.

Usage

```
cluster_groups(
  data,
  cols,
  group_cols = NULL,
  scale_min_fn = function(x) {
     quantile(x, 0.025)
 },
  scale_max_fn = function(x) {
     quantile(x, 0.975)
  },
  keep_centroids = FALSE,
  multiplier = 0.05,
  suffix = "_clustered",
  keep_original = TRUE,
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame`. If `` `group_cols` `` is `NULL`, it must be grouped with `dplyr::group_by()`.
`cols`	Names of columns in `` `data` `` to mutate. Each column is considered a dimension to contract distances in.
`group_cols`	Names of grouping columns in `` `data` ``. Must be distinct from the names in `` `cols` ``. If `NULL` and `` `data` `` is grouped, those groups are used instead.
`scale_min_fn, scale_max_fn`	Function to find the minimum/maximum value in the original data when rescaling the contracted data. **Input**: A `numeric vector`. **Output**: A `numeric scalar`.
`keep_centroids`	Whether to ensure the clusters have their original centroids. (Logical)
`multiplier`	Numeric constant to multiply the distance to the group centroid by. A smaller value makes the clusters more compact and vice versa.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Details

- Contracts the distance from each data point to the centroid of its group.
- Performs MinMax scaling such that the scale of the data points is *similar* to the original data.
- If enabled (not default), the centroids are moved to the original centroids.

Value

`data.frame` (`tibble`) with the clustered columns.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: `apply_transformation_matrix()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Other clustering functions: `generate_clusters()`, `transfer_centroids()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(2)

# Create a data frame
df <- data.frame(
  "x" = runif(50),
  "y" = runif(50),
  "z" = runif(50),
  "g" = rep(c(1, 2, 3, 4, 5), each = 10)
)

# Move the data points into clusters
cluster_groups(df,
  cols = c("x", "y"),
  group_col = "g"
)
cluster_groups(df,
  cols = c("x", "y"),
  group_col = "g",
  multiplier = 0.1
)
cluster_groups(df,
  cols = c("x"),
  group_col = "g",
  multiplier = 0.1
)

#
# Plotting clusters
#

# Cluster x and y for each group in g
df_clustered <- cluster_groups(
  data = df,
  cols = c("x", "y"),
  group_col = "g"
)
```

```
# Plot the clusters over the original data points
# As we work with random data, the cluster might overlap
if (has_ggplot){
  ggplot(
    df_clustered,
    aes(x = x_clustered, y = y_clustered, color = factor(g))
  ) +
    # Original data
    geom_point(aes(x = x, y = y), alpha = 0.3, size = 0.8) +
    # Clustered data
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "g")
}

#
# Maintain original group centroids
#

df_clustered <- cluster_groups(
  data = df,
  cols = c("x", "y"),
  group_col = "g",
  keep_centroids = TRUE
)

# Plot the clusters over the original data points
# As we work with random data, the cluster might overlap
if (has_ggplot){
  ggplot(
    df_clustered,
    aes(x = x_clustered, y = y_clustered, color = factor(g))
  ) +
    # Original data
    geom_point(aes(x = x, y = y), alpha = 0.3, size = 0.8) +
    # Clustered data
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "g")
}

#
# Three dimensions
#

# Cluster in 3d
df_clustered <- cluster_groups(
  data = df,
  cols = c("x", "y", "z"),
  group_col = "g"
)

## Not run:
```

```
# Plot 3d with plotly
plotly::plot_ly(
  x = df_clustered$x_clustered,
  y = df_clustered$y_clustered,
  z = df_clustered$z_clustered,
  type = "scatter3d",
  mode = "markers",
  color = df_clustered$g
)

## End(Not run)
```

`create_dimming_fn` *Create dimming_fn function*

Description

[Experimental]

Creates a function that takes 2 inputs (`` `x` ``, `` `d` ``) and performs the operation:

$$x * (numerator/((add_t o_d istance + d)^e xponent))$$

Here, `` `x` `` is the current value and `` `d` `` is its distance to an origin. The greater the distance, the more we will dim the value of `` `x` ``.

With the default values, the returned function is:

```
function(x, d){
  x * (1 / ((1 + d) ^ 2))
}
```

Usage

```
create_dimming_fn(numerator = 1, exponent = 2, add_to_distance = 1)
```

Arguments

`numerator`	The numerator. Defaults to `1`.
`exponent`	The exponent. Defaults to `2`.
`add_to_distance`	Constant to add to the distance before exponentiation. Ensures dimming even when the distance (d) is below `1`. Defaults to `1`.

Value

Function with the arguments `x` and `d`, with both expected to be `numeric` `vector`s. More specifically:

```
function(x, d){
  x * (numerator / ((add_to_distance + d) ^ exponent))
}
```

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other function creators: `create_n_fn()`, `create_origin_fn()`

Examples

```
# Attach packages
library(rearrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create two vectors
x <- runif(10)
d <- runif(10, max = 0.5)

# Create dimming_fn with an add_to_distance of 0
# Note: In practice this risks zero-division
non_smoothed_dimming_fn <- create_dimming_fn(add_to_distance = 0)
non_smoothed_dimming_fn
as.list(environment(non_smoothed_dimming_fn))

# Use median_origin_fn
non_smoothed_dimming_fn(x, d)

# Plotting the dimming

# Create data.frame with distance-based dimming
df <- data.frame(
  "x" = 1,
  "d" = 1:10
)
df$x_dimmed <- non_smoothed_dimming_fn(df$x, df$d)

# Plot the dimming
if (has_ggplot){
  ggplot(df, aes(x=d, y=x_dimmed)) +
    geom_point() +
    geom_line() +
    theme_minimal()
}
```

`create_n_fn` *Create n_fn function*

Description

[Experimental]

Creates a function that applies a supplied function to all input vectors, or their indices, and rounds the results.

As used with `roll_elements()`. E.g. to find the the median index in a subset of a grouped `data.frame`.

Usage

```
create_n_fn(fn, use_index = FALSE, negate = FALSE, round_fn = round, ...)
```

Arguments

`fn`	Function to apply to each dimension. Should return a numeric scalar.
`use_index`	Whether to apply ``fn`` to the *indices* of the vectors. (Logical) The indices are created with `seq_along(x)`.
`negate`	Whether to negate the result. I.e. to multiply it with `-1`. (Logical)
`round_fn`	Function for rounding results of ``fn``. Rounding is done *prior* to negation. E.g. `round`, `floor`, or `ceiling`. To avoid rounding, supply `identity`.
`...`	Arguments for ``fn``. E.g. ``na.rm = TRUE``.

Value

Function with the dots (``...``) argument that applies the ``fn`` function to each element in ``...`` (or indices thereof) (usually one vector per dimension). The results are rounded with ``round_fn``.

Note: The dots argument in the generated function should not to be confused with the dots argument in `create_n_fn()`).

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other n functions: `median_index()`

Other function creators: `create_dimming_fn()`, `create_origin_fn()`

Examples

```
# Attach packages
library(rearrr)

# Set seed
set.seed(1)
```

```
# Create three vectors
x <- runif(10)
y <- runif(10)
z <- runif(10)

# Create n_fn that gets the median index
# and rounds down with floor()
median_index_fn <- create_n_fn(median, use_index = TRUE, round_fn = floor)

# Use median_index_fn
median_index_fn(x, y, z)

# Create n_fn that gets the median of each dimension
median_n_fn <- create_n_fn(median)

# Use median_origin_fn
median_n_fn(x, y, z)

# Should be the same as
round(c(median(x), median(y), median(z)))

# Use mean and ignore missing values
mean_n_fn <- create_n_fn(mean, na.rm = TRUE)

# Add missing values
x[[2]] <- NA
y[[5]] <- NA

# Use mean_n_fn
mean_n_fn(x, y, z)

# Should be the same as
round(c(
  mean(x, na.rm = TRUE),
  mean(y, na.rm = TRUE),
  mean(z, na.rm = TRUE)
))
```

`create_origin_fn` *Create origin_fn function*

Description

[Experimental]

Creates a function that applies a supplied function to all input vectors.

Usage

```
create_origin_fn(fn, ...)
```

Arguments

`fn`	Function to apply to each dimension. Should return a numeric scalar.
`...`	Arguments for \``fn`\`. E.g. \``na.rm = TRUE`\`.

Value

Function with the dots (`...`) argument that applies the \``fn`\` function to each element in `...` (usually one vector per dimension).

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other coordinate functions: `centroid()`, `is_most_centered()`, `midrange()`, `most_centered()`

Other function creators: `create_dimming_fn()`, `create_n_fn()`

Examples

```
# Attach packages
library(rearrr)

# Set seed
set.seed(1)

# Create three vectors
x <- runif(10)
y <- runif(10)
z <- runif(10)

# Create origin_fn that gets the median of each dimension
median_origin_fn <- create_origin_fn(median)

# Use median_origin_fn
median_origin_fn(x, y, z)

# Should be the same as
c(median(x), median(y), median(z))

# Use mean and ignore missing values
mean_origin_fn <- create_origin_fn(mean, na.rm = TRUE)

# Add missing values
x[[2]] <- NA
y[[5]] <- NA

# Use mean_origin_fn
mean_origin_fn(x, y, z)

# Should be the same as
```

```
c(mean(x, na.rm = TRUE),
  mean(y, na.rm = TRUE),
  mean(z, na.rm = TRUE)
)
```

---

degrees_to_radians *Conversion between radians and degrees*

Description

[Experimental]

Convert degrees to radians or radians to degrees.

Usage

```
degrees_to_radians(degrees)

radians_to_degrees(radians)
```

Arguments

`degrees`	`vector` of degrees to convert to radians with $$`degrees` * (\pi/180)$$
`radians`	`vector` of radians to convert to degrees with $$`radians`/(\pi/180)$$

Value

`vector` with converted degrees/radians.

Missing values (`NA`s) are returned as they are.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Degrees to radians
degrees_to_radians(c(90, 180, 270))

# Radians to degrees
radians_to_degrees(c(pi / 2, pi, 1.5 * pi))
```

```
# Get back the original degrees
radians_to_degrees(degrees_to_radians(c(90, 180, 270)))
```

dim_values *Dim values of a dimension based on the distance to an n-dimensional origin*

Description

[Experimental]

Dims the values in the dimming dimension (last by default) based on the data point's distance to the origin.

Distance is calculated as:

$$d(P1, P2) = sqrt((x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2 + ...)$$

The default `dimming_fn` multiplies by the inverse-square of $1 + distance$ and is calculated as:

$$dimming_fn(x, d) = x * (1/(1 + d)^2)$$

Where x is the value in the dimming dimension. The $+1$ is added to ensure that values are dimmed even when the distance is below `1`. The quickest way to change the exponent or the $+1$ is with `create_dimming_fn()`.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and dimming around e.g. the centroid of each group.

Usage

```
dim_values(
  data,
  cols,
  dimming_fn = create_dimming_fn(numerator = 1, exponent = 2, add_to_distance = 1),
  origin = NULL,
  origin_fn = NULL,
  dim_col = tail(cols, 1),
  suffix = "_dimmed",
  keep_original = TRUE,
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data` — `data.frame` or `vector`.

`cols` — Names of columns in `` `data` `` to calculate distances from. The dimming column (`` `dim_col` ``) is dimmed based on all the columns. Each column is considered a dimension.

N.B. when the dimming dimension is included in `` `cols` ``, it is used in the distance calculation as well.

`dimming_fn` — `Function` for calculating the dimmed values.

Input: Two (2) input arguments:

1. A `numeric vector` with the values in the dimming dimension.
2. A `numeric vector` with corresponding distances to the origin.

Output: A `numeric vector` with the same length as the input vectors.

E.g.:

```
function(x, d){
  x * (1 / ((1 + d) ^ 2))
}
```

This kind of dimming function can be created with `create_dimming_fn()`, which for instance makes it easy to change the exponent (the `2` above).

`origin` — Coordinates of the origin to dim around. A scalar to use in all dimensions or a `vector` with one scalar per dimension.

N.B. Ignored when `` `origin_fn` `` is not `NULL`.

`origin_fn` — Function for finding the origin coordinates.

Input: Each column will be passed as a `vector` in the order of `` `cols` ``.

Output: A `vector` with one scalar per dimension.

Can be created with `create_origin_fn()` if you want to apply the same function to each dimension.

E.g. `` `create_origin_fn(median)` `` would find the median of each column.

Built-in functions are `centroid()`, `most_centered()`, and `midrange()`

`dim_col` — Name of column to dim. Default is the last column in `` `cols` ``.

When the `` `dim_col` `` is not present in `` `cols` ``, it is not used in the distance calculation.

`suffix` — Suffix to add to the names of the generated columns.

Use an empty string (i.e. `""`) to overwrite the original columns.

`keep_original` — Whether to keep the original columns. (Logical)

Some columns may have been overwritten, in which case only the newest versions are returned.

`origin_col_name` — Name of new column with the origin coordinates. If `NULL`, no column is added.

`overwrite` — Whether to allow overwriting of existing columns. (Logical)

Details

- Calculates distances to origin with:

$$d(P1, P2) = sqrt((x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2 + ...)$$

- Applies the `` `dimming_fn` `` to the `` `dim_col` `` based on the distances.

Value

`data.frame` (`tibble`) with the dimmed column, along with the origin coordinates.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Other distance functions: `closest_to()`, `distance()`, `expand_distances()`, `expand_distances_each()`, `furthest_from()`, `swirl_2d()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(7)

# Create a data frame with clusters
df <- generate_clusters(
  num_rows = 70,
  num_cols = 3,
  num_clusters = 5,
  compactness = 1.6
) %>%
  dplyr::rename(x = D1, y = D2, z = D3) %>%
  dplyr::mutate(o = 1)

# Dim the values in the z column
dim_values(
  data = df,
  cols = c("x", "y", "z"),
  origin = c(0.5, 0.5, 0.5)
)
```

```
# Dim the values in the `o` column (all 1s)
# around the centroid
dim_values(
  data = df,
  cols = c("x", "y"),
  dim_col = "o",
  origin_fn = centroid
)

# Specify dimming_fn
# around the centroid
dim_values(
  data = df,
  cols = c("x", "y"),
  dim_col = "o",
  origin_fn = centroid,
  dimming_fn = function(x, d) {
    x * 1 / (2^(1 + d))
  }
)

#
# Dim cluster-wise
#

# Group-wise dimming
df_dimmed <- df %>%
  dplyr::group_by(.cluster) %>%
  dim_values(
    cols = c("x", "y"),
    dim_col = "o",
    origin_fn = centroid
  )

# Plot the dimmed data such that the alpha (opacity) is
# controlled by the dimming
# (Note: This works because the `o` column is 1 for all values)
if (has_ggplot){
  ggplot(
    data = df_dimmed,
    aes(x = x, y = y, alpha = o_dimmed, color = .cluster)
  ) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "Cluster", alpha = "o_dimmed")
}
```

`distance` *Calculate the distance to an origin*

Description

[Experimental]

Calculates the distance to the specified origin with:

$$d(P1, P2) = sqrt((x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2 + ...)$$

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and finding the distance to e.g. the centroid of each group.

Usage

```
distance(
  data,
  cols = NULL,
  origin = NULL,
  origin_fn = NULL,
  distance_col_name = ".distance",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Names of columns in `` `data` `` to measure distance in. Each column is considered a dimension.
`origin`	Coordinates of the origin to calculate distances to. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`distance_col_name`	Name of new column with the distances.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with the additional columns (distances and origin coordinates).

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other measuring functions: angle(), vector_length()

Other distance functions: closest_to(), dim_values(), expand_distances(), expand_distances_each(), furthest_from(), swirl_2d(), swirl_3d()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)

# Calculate distances in the two dimensions (x and y)
# With the origin at x=0.5, y=0.5
distance(
  data = df,
  cols = c("x", "y"),
  origin = c(0.5, 0.5)
)

# Calculate distances to the centroid for each group in 'g'
distance(
  data = dplyr::group_by(df, g),
  cols = c("x", "y"),
  origin_fn = centroid
)
```

---

expand_distances *Expand the distances to an origin*

Description

[Experimental]

Moves the data points in n-dimensional space such that their distance to a specified origin is increased/decreased. A `multiplier` greater than 1 leads to expansion, while a positive `multiplier` lower than 1 leads to contraction.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and expanding around e.g. the centroid of each group.

The multiplier/exponent can be supplied as a constant or as a function that returns a constant. The latter can be useful when supplying a grouped `data.frame` and the multiplier/exponent depends on the data in the groups.

For expansion in each dimension separately, use `expand_distances_each()`.

NOTE: When exponentiating, the default is to first add `1` to the distances, to ensure expansion even when the distance is between `0` and `1`. If you need the purely exponentiated distances, disable `` `add_one_exp` ``.

Usage

```
expand_distances(
  data,
  cols = NULL,
  multiplier = NULL,
  multiplier_fn = NULL,
  origin = NULL,
  origin_fn = NULL,
  exponentiate = FALSE,
  add_one_exp = TRUE,
  suffix = "_expanded",
  keep_original = TRUE,
  mult_col_name = ifelse(isTRUE(exponentiate), ".exponent", ".multiplier"),
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Names of columns in `` `data` `` to expand coordinates of. Each column is considered a dimension.
`multiplier`	Constant to multiply/exponentiate the distances to the origin by. **N.B.** When `` `exponentiate` `` is `TRUE`, the `` `multiplier` `` becomes an *exponent*.
`multiplier_fn`	Function for finding the `` `multiplier` ``. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `numeric scalar`.
`origin`	Coordinates of the origin to expand around. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension.

	Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`exponentiate`	Whether to exponentiate instead of multiplying. (Logical)

`add_one_exp`	Whether to add `1` to the distances before exponentiating to ensure they don't contract when between `0` and `1`. The added value is subtracted after the exponentiation. (Logical)
	The distances to the origin (`` `d` ``) are exponentiated as such:
	`d <- d + 1`
	`d <- d ^ multiplier`
	`d <- d - 1`
	N.B. Ignored when `` `exponentiate` `` is `FALSE`.

`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`mult_col_name`	Name of new column with the `` `multiplier` ``. If `NULL`, no column is added.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Details

Increases the distance to the origin in n-dimensional space by multiplying or exponentiating it by the multiplier.

We first move the origin to the zero-coordinates (e.g. `c(0, 0, 0)`) and normalize each vector to unit length. We then multiply this unit vector by the multiplied/exponentiated distance and moves the origin back to its original coordinates.

The distance to the specified origin is calculated with:

$$d(P1, P2) = sqrt((x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2 + ...)$$

Note: By default (when `` `add_one_exp` `` is `TRUE`), we add `1` to the distance before the exponentiation and subtract it afterwards. See `` `add_one_exp` ``.

Value

`data.frame` (`tibble`) with the expanded columns, along with the applied multiplier/exponent and origin coordinates.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: apply_transformation_matrix(), cluster_groups(), dim_values(), expand_distances_each(), flip_values(), roll_values(), rotate_2d(), rotate_3d(), shear_2d(), shear_3d(), swirl_2d(), swirl_3d()

Other expander functions: expand_distances_each()

Other distance functions: closest_to(), dim_values(), distance(), expand_distances_each(), furthest_from(), swirl_2d(), swirl_3d()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)

# Expand distances in the two dimensions (x and y)
# With the origin at x=0.5, y=0.5
# We multiply the distances by 2
expand_distances(
  data = df,
  cols = c("x", "y"),
  multiplier = 2,
  origin = c(0.5, 0.5)
)

# Expand distances in the two dimensions (x and y)
# With the origin at x=0.5, y=0.5
# We exponentiate the distances by 2
expand_distances(
  data = df,
  cols = c("x", "y"),
  multiplier = 2,
  exponentiate = TRUE,
  origin = 0.5
)

# Expand values in one dimension (x)
# With the origin at x=0.5
# We exponentiate the distances by 3
expand_distances(
  data = df,
  cols = c("x"),
  multiplier = 3,
  exponentiate = TRUE,
  origin = 0.5
)

# Expand x and y around the centroid
# We use exponentiation for a more drastic effect
# The add_one_exp makes sure it expands
# even when x or y is in the range [0, <1]
# To compare multiple exponents, we wrap the
# call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = c(1, 3, 5),
  .f = function(exponent) {
    expand_distances(
      data = df,
      cols = c("x", "y"),
      multiplier = exponent,
      origin_fn = centroid,
      exponentiate = TRUE,
      add_one_exp = TRUE
    )
  }
)
df_expanded

# Plot the expansions of x and y around the overall centroid
if (has_ggplot){
  ggplot(df_expanded, aes(x = x_expanded, y = y_expanded, color = factor(.exponent))) +
    geom_vline(
      xintercept = df_expanded[[".origin"]][[1]][[1]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_hline(
      yintercept = df_expanded[[".origin"]][[1]][[2]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_path(size = 0.2) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "Exponent")
}

# Expand x and y around the centroid using multiplication
# To compare multiple multipliers, we wrap the
# call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = c(1, 3, 5),
  .f = function(multiplier) {
    expand_distances(df,
      cols = c("x", "y"),
      multiplier = multiplier,
      origin_fn = centroid,
      exponentiate = FALSE
    )
  }
)
df_expanded

# Plot the expansions of x and y around the overall centroid
if (has_ggplot){
  ggplot(df_expanded, aes(x = x_expanded, y = y_expanded, color = factor(.multiplier))) +
    geom_vline(
      xintercept = df_expanded[[".origin"]][[1]][[1]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_hline(
      yintercept = df_expanded[[".origin"]][[1]][[2]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_path(size = 0.2, alpha = .8) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "Multiplier")
}

#
# Contraction
#

# Group-wise contraction to create clusters
df_contracted <- df %>%
  dplyr::group_by(g) %>%
  expand_distances(
    cols = c("x", "y"),
    multiplier = 0.07,
    suffix = "_contracted",
    origin_fn = centroid
  )

# Plot the clustered data point on top of the original data points
if (has_ggplot){
  ggplot(df_contracted, aes(x = x_contracted, y = y_contracted, color = factor(g))) +
    geom_point(aes(x = x, y = y, color = factor(g)), alpha = 0.3, shape = 16) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "g")
}
```

`expand_distances_each` *Expand the distances to an origin in each dimension*

Description

[Experimental]

Moves the data points in n-dimensional space such that their distance to the specified origin is increased/decreased *in each dimension separately*. A `` `multiplier` `` greater than 1 leads to expansion, while a positive `` `multiplier` `` lower than 1 leads to contraction.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and expanding around e.g. the centroid of each group.

The multipliers/exponents can be supplied as constant(s) or as a function that returns constants. The latter can be useful when supplying a grouped `data.frame` and the multiplier/exponent depends on the data in the groups. If supplying multiple constants, there must be one per dimension (length of `` `cols` ``).

For expansion of the *multidimensional* distance, use `expand_distances()`.

NOTE: When exponentiating, the default is to first add `1` or `-1` (depending on the sign of the distance) to the distances, to ensure expansion even when the distance is between `-1` and `1`. If you need the purely exponentiated distances, disable `` `add_one_exp` ``.

Usage

```
expand_distances_each(
  data,
  cols = NULL,
  multipliers = NULL,
  multipliers_fn = NULL,
  origin = NULL,
  origin_fn = NULL,
  exponentiate = FALSE,
  add_one_exp = TRUE,
  suffix = "_expanded",
  keep_original = TRUE,
  mult_col_name = ifelse(isTRUE(exponentiate), ".exponents", ".multipliers"),
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Names of columns in `` `data` `` to expand. Each column is considered a dimension to expand in.
`multipliers`	Constant(s) to multiply/exponentiate the distance to the origin by. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** When `` `exponentiate` `` is `TRUE`, the `` `multipliers` `` become *exponents*.
`multipliers_fn`	Function for finding the `` `multipliers` ``. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``.

Output: A `numeric` `vector` with one element per dimension.

Just as for `` `origin_fn` ``, it can be created with `create_origin_fn()` if you want to apply the same function to each dimension. See `` `origin_fn` ``.

`origin`	Coordinates of the origin to expand around. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`exponentiate`	Whether to exponentiate instead of multiplying. (Logical)

`add_one_exp`	Whether to add the `sign` (either `1` or `-1`) before exponentiating to ensure the values don't contract. The added value is subtracted after the exponentiation. (Logical)
	Exponentiation becomes:
	`x <- x + sign(x)`
	`x <- sign(x) * abs(x) ^ multiplier`
	`x <- x - sign(x)`
	N.B. Ignored when `` `exponentiate` `` is `FALSE`.

`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`mult_col_name`	Name of new column with the multiplier(s). If `NULL`, no column is added.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Details

For each value of each dimension (column), either multiply or exponentiate by the multiplier:

```
# Multiplication
x <- x * multiplier
# Exponentiation
x <- sign(x) * abs(x) ^ multiplier
```

Note: By default (when `` `add_one_exp` `` is `TRUE`), we add the sign (`1 / -1`) of the value before the exponentiation and subtract it afterwards. See `` `add_one_exp` ``.

Value

`data.frame` (`tibble`) with the expanded columns, along with the applied multiplier/exponent and origin coordinates.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Other expander functions: `expand_distances()`

Other distance functions: `closest_to()`, `dim_values()`, `distance()`, `expand_distances()`, `furthest_from()`, `swirl_2d()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)

# Expand values in the two dimensions (x and y)
# With the origin at x=0.5, y=0.5
# We expand x by 2 and y by 4
expand_distances_each(
  data = df,
  cols = c("x", "y"),
  multipliers = c(2, 4),
  origin = c(0.5, 0.5)
)

# Expand values in the two dimensions (x and y)
# With the origin at x=0.5, y=0.5
# We expand both by 3
expand_distances_each(
  data = df,
  cols = c("x", "y"),
  multipliers = 3,
  origin = 0.5
)

# Expand values in one dimension (x)
# With the origin at x=0.5
# We expand by 3
expand_distances_each(
  data = df,
  cols = c("x"),
  multipliers = 3,
  origin = 0.5
)

# Expand x and y around the centroid
# We use exponentiation for a more drastic effect
# The add_one_exp makes sure it expands
# even when x or y is in the range [>-1, <1]
# To compare multiple exponents, we wrap the
# call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = c(1, 2.0, 3.0, 4.0),
  .f = function(exponent) {
    expand_distances_each(
      data = df,
      cols = c("x", "y"),
      multipliers = exponent,
      origin_fn = centroid,
      exponentiate = TRUE,
      add_one_exp = TRUE
    )
  }
)
df_expanded

# Plot the expansions of x and y around the overall centroid
if (has_ggplot){
  ggplot(df_expanded, aes(x = x_expanded, y = y_expanded, color = factor(.exponents))) +
    geom_vline(
      xintercept = df_expanded[[".origin"]][[1]][[1]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_hline(
      yintercept = df_expanded[[".origin"]][[1]][[2]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "Exponent")
}

# Expand x and y around the centroid using multiplication
# To compare multiple multipliers, we wrap the
```

```
# call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = c(1, 2.0, 3.0, 4.0),
  .f = function(multiplier) {
    expand_distances_each(df,
      cols = c("x", "y"),
      multipliers = multiplier,
      origin_fn = centroid,
      exponentiate = FALSE
    )
  }
)
df_expanded

# Plot the expansions of x and y around the overall centroid
if (has_ggplot){
ggplot(df_expanded, aes(x = x_expanded, y = y_expanded, color = factor(.multipliers))) +
    geom_vline(
      xintercept = df_expanded[[".origin"]][[1]][[1]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_hline(
      yintercept = df_expanded[[".origin"]][[1]][[2]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "Multiplier")
}

# Expand x and y with different multipliers
# around the centroid using multiplication
df_expanded <- expand_distances_each(
  df,
  cols = c("x", "y"),
  multipliers = c(1.25, 10),
  origin_fn = centroid,
  exponentiate = FALSE
)
df_expanded

# Plot the expansions of x and y around the overall centroid
# Note how the y axis is expanded a lot more than the x-axis
if (has_ggplot){
  ggplot(df_expanded, aes(x = x_expanded, y = y_expanded)) +
    geom_vline(
      xintercept = df_expanded[[".origin"]][[1]][[1]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_hline(
      yintercept = df_expanded[[".origin"]][[1]][[2]],
      size = 0.2, alpha = .4, linetype = "dashed"
    ) +
    geom_line(aes(color = "Expanded")) +
    geom_point(aes(color = "Expanded")) +
    geom_line(aes(x = x, y = y, color = "Original")) +
    geom_point(aes(x = x, y = y, color = "Original")) +
    theme_minimal() +
    labs(x = "x", y = "y", color = "Multiplier")
}

#
# Contraction
#

# Group-wise contraction to create clusters
df_contracted <- df %>%
  dplyr::group_by(g) %>%
  expand_distances_each(
    cols = c("x", "y"),
    multipliers = 0.07,
    suffix = "_contracted",
    origin_fn = centroid
  )

# Plot the clustered data point on top of the original data points
if (has_ggplot){
  ggplot(df_contracted, aes(x = x_contracted, y = y_contracted, color = factor(g))) +
    geom_point(aes(x = x, y = y, color = factor(g)), alpha = 0.3, shape = 16) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "g")
}
```

`FixedGroupsPipeline` *Chain multiple transformations with different argument values per group*

Description

[Experimental]

Build a pipeline of transformations to be applied sequentially.

Specify different argument values for each group in a fixed set of groups. E.g. if your `data.frame` contains 5 groups, you provide 5 argument values for each of the non-constant arguments (see `` `var_args` ``).

The number of expected groups is specified during initialization and the input `` `data` `` must be grouped such that it contains that exact number of groups.

Transformations are applied to groups separately, why the given transformation function only receives the subset of `` `data` `` belonging to the current group.

Standard workflow: Instantiate pipeline -> Add transformations -> Apply to data

To apply the same arguments to all groups, see `Pipeline`.

To apply generated argument values to an arbitrary number of groups, see `GeneratedPipeline`.

Super class

`rearrr::Pipeline` -> `FixedGroupsPipeline`

Public fields

`transformations` `list` of transformations to apply.

`names` Names of the transformations.

`num_groups` Number of groups the pipeline will be applied to.

Methods

Public methods:

- `FixedGroupsPipeline$new()`
- `FixedGroupsPipeline$add_transformation()`
- `FixedGroupsPipeline$apply()`
- `FixedGroupsPipeline$print()`
- `FixedGroupsPipeline$clone()`

Method `new()`**:** Initialize the pipeline with the number of groups the pipeline will be applied to.

Usage:

```
FixedGroupsPipeline$new(num_groups)
```

Arguments:

`num_groups` Number of groups the pipeline will be applied to.

Method `add_transformation()`**:** Add a transformation to the pipeline.

Usage:

```
FixedGroupsPipeline$add_transformation(fn, args, var_args, name)
```

Arguments:

`fn` Function that performs the transformation.

`args` Named `list` with arguments for the `` `fn` `` function.

`var_args` Named `list` of arguments with `list` of differing values for each group.
E.g. `list("a" = list(1, 2, 3), "b" = list("a", "b", "c"))` given 3 groups.
By adding `".apply"` with a list of `TRUE`/`FALSE` flags, the transformation can be disabled for a specific group.
E.g. `list(".apply" = list(TRUE, FALSE, TRUE), ...`.

`name` Name of the transformation step. Must be unique.

Returns: The pipeline. To allow chaining of methods.

Method `apply()`**:** Apply the pipeline to a `data.frame`.

Usage:

```
FixedGroupsPipeline$apply(data, verbose = FALSE)
```

Arguments:

`data` `data.frame` with the same number of groups as pre-registered in the pipeline.
You can find the number of groups in `` `data` `` with `` `dplyr::n_groups(data)` ``. The number of groups expected by the pipeline can be accessed with `` `pipe$num_groups` ``.

`verbose` Whether to print the progress.

Returns: Transformed version of `` `data` ``.

Method `print()`**:** Print an overview of the pipeline.

Usage:

```
FixedGroupsPipeline$print(...)
```

Arguments:

`...` further arguments passed to or from other methods.

Returns: The pipeline. To allow chaining of methods.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FixedGroupsPipeline$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other pipelines: `GeneratedPipeline`, `Pipeline`

Examples

```
# Attach package
library(rearrr)
library(dplyr)

# Create a data frame
# We group it by G so we have 3 groups
df <- data.frame(
  "Index" = 1:12,
  "A" = c(1:4, 9:12, 15:18),
  "G" = rep(1:3, each = 4)
) %>%
  dplyr::group_by(G)

# Create new pipeline
pipe <- FixedGroupsPipeline$new(num_groups = 3)

# Add 2D rotation transformation
pipe$add_transformation(
  fn = rotate_2d,
  args = list(
    x_col = "Index",
    y_col = "A",
    suffix = "",
    overwrite = TRUE
  ),
  var_args = list(
    degrees = list(45, 90, 180),
    origin = list(c(0, 0), c(1, 2), c(-1, 0))
  ),
  name = "Rotate"
)

# Add the `cluster_group` transformation
# As the function is fed an ungrouped subset of `data`,
# i.e. the rows of that group, we need to specify `group_cols` in `args`
# That is specific to `cluster_groups()` though
# Also note `.apply` in `var_args` which tells the pipeline *not*
# to apply this transformation to the second group
pipe$add_transformation(
  fn = cluster_groups,
  args = list(
    cols = c("Index", "A"),
    suffix = "",
    overwrite = TRUE,
    group_cols = "G"
  ),
  var_args = list(
    multiplier = list(0.5, 1, 5),
    .apply = list(TRUE, FALSE, TRUE)
  ),
  name = "Cluster"
)

# Check pipeline object
pipe

# Apply pipeline to already grouped data.frame
# Enable `verbose` to print progress
pipe$apply(df, verbose = TRUE)
```

`FixedGroupsTransformation`

FixedGroupsTransformation

Description

[Experimental]

Container for the type of transformation used in `FixedGroupsPipeline`.

Note: For internal use.

Super class

`rearrr::Transformation` -> `FixedGroupsTransformation`

Public fields

`name` Name of transformation.

`fn` Transformation function.

`args` `list` of constant arguments for `` `fn` ``.

`var_args` `list` of arguments for `` `fn` `` with different values per group.

`num_groups` Number of groups that the transformation expects.

`apply_arg` `list` of `TRUE`/`FALSE` flags indicating whether the transformation should be applied to each of the groups.
When `` `NULL` ``, the transformation is applied to all groups.

Methods

Public methods:

- `FixedGroupsTransformation$new()`
- `FixedGroupsTransformation$get_group_args()`
- `FixedGroupsTransformation$apply()`
- `FixedGroupsTransformation$print()`
- `FixedGroupsTransformation$clone()`

Method `new()`**:** Initialize transformation.

Usage:

```
FixedGroupsTransformation$new(fn, args, var_args, name = NULL)
```

Arguments:

`fn` Transformation function.

`args` `list` of constant arguments for `` `fn` ``.

`var_args` `list` of arguments for `` `fn` `` with different values per group. Each argument should have a list of values (one per group).
By adding `".apply"` with a list of `TRUE`/`FALSE` flags, the transformation can be disabled for a specific group.
E.g. `list(".apply" = list(TRUE, FALSE, TRUE), ...`.

`name` Name of transformation.

Method `get_group_args()`**:** Get arguments for specific group ID.

Usage:

```
FixedGroupsTransformation$get_group_args(group_id)
```

Arguments:

`group_id` ID of the group to get arguments for.

Returns: `list` of arguments.

Method `apply()`**:** Apply the transformation to a `data.frame`.

Usage:

`FixedGroupsTransformation$apply(data)`

Arguments:

`data` `data.frame` with the expected number of groups.

Returns: Transformed version of `` `data` ``.

Method `print()`**:** Print an overview of the transformation.

Usage:

`FixedGroupsTransformation$print(..., indent = 0, show_class = TRUE)`

Arguments:

`...` further arguments passed to or from other methods.

`indent` How many spaces to indent when printing.

`show_class` Whether to print the transformation class name.

Returns: The pipeline. To allow chaining of methods.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`FixedGroupsTransformation$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other transformation classes: `GeneratedTransformation`, `Transformation`

`flip_values` *Flip the values around an origin*

Description

[Experimental]

The values are flipped with the formula ‘$x = 2 * c - x$‘ where x is the value and c is the origin coordinate to flip the values around.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and flipping around e.g. the centroid of each group. By default the median value in each dimension is used.

The `*_vec()` version take and return a vector.

Example:

The column values:

`c(5, 2, 7, 4, 3, 1)`

and the `origin_fn` = `create_origin_fn(median)`

Changes the values to :

`c(2, 5, 0, 3, 4, 6)`

Usage

```
flip_values(
  data,
  cols = NULL,
  origin = NULL,
  origin_fn = create_origin_fn(median),
  suffix = "_flipped",
  keep_original = TRUE,
  origin_col_name = ".origin",
  overwrite = FALSE
)

flip_values_vec(data, origin = NULL, origin_fn = create_origin_fn(median))
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Names of columns in `` `data` `` to flip values of.
`origin`	Coordinates of the origin to flip around. A scalar to use in all dimensions (columns) or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `NULL`. Remember to set it to `NULL` when passing origin coordinates manually!
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other mutate functions: apply_transformation_matrix(), cluster_groups(), dim_values(), expand_distances(), expand_distances_each(), roll_values(), rotate_2d(), rotate_3d(), shear_2d(), shear_3d(), swirl_2d(), swirl_3d()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "Index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Flip values of the columns
flip_values(df$A)
flip_values(df, cols = "A")
flip_values(df, cols = "B", origin = 0.3, origin_fn = NULL, keep_original = FALSE)
flip_values(df,
  cols = c("A", "B"),
  origin = c(3, 0.3),
  origin_fn = NULL,
  suffix = "",
  keep_original = FALSE,
  overwrite = TRUE
)
flip_values(df, cols = c("A", "B"), origin_fn = create_origin_fn(max))

# Grouped by G
df %>%
  dplyr::group_by(G) %>%
  flip_values(
    cols = c("A", "B"),
    origin_fn = create_origin_fn(median),
    keep_original = FALSE
  )

# Plot A and flipped A

# First flip A around the median and then around the value 3.
df <- df %>%
  flip_values(cols = "A", suffix = "_flip_median", origin_col_name = NULL) %>%
  flip_values(cols = "A", suffix = "_flip_3", origin = 3,
              origin_fn = NULL, origin_col_name = NULL)

# Plot A and A flipped around its median
if (has_ggplot){
  ggplot(df, aes(x = Index, y = A)) +
    geom_line(aes(color = "A")) +
    geom_line(aes(y = A_flip_median, color = "Flipped A (median)")) +
    geom_hline(aes(color = "Median A", yintercept = median(A))) +
    theme_minimal()
}

# Plot A and A flipped around the value 3
if (has_ggplot){
  ggplot(df, aes(x = Index, y = A)) +
    geom_line(aes(color = "A")) +
    geom_line(aes(y = A_flip_3, color = "Flipped A (3)")) +
    geom_hline(aes(color = "3", yintercept = 3)) +
    theme_minimal()
}
```

`furthest_from` *Orders values by longest distance to an origin*

Description

[Experimental]

Values are ordered by how far they are from the origin.

In 1d (when `` `cols` `` has length 1), the origin can be thought of as a target value. In *n* dimensions, the origin can be thought of as coordinates.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and ordering the rows by their distance to the centroid of each group.

The `*_vec()` version takes and returns a `vector`.

Example:

The column values:

`c(1, 2, 3, 4, 5)`

and `origin` = 2

are **ordered as**:

`c(5, 4, 1, 3, 2)`

Usage

```
furthest_from(
  data,
  cols = NULL,
  origin = NULL,
  origin_fn = NULL,
  shuffle_ties = FALSE,
  origin_col_name = ".origin",
  distance_col_name = ".distance",
  overwrite = FALSE
)

furthest_from_vec(data, origin = NULL, origin_fn = NULL, shuffle_ties = FALSE)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Column(s) to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`origin`	Coordinates of the origin to calculate distances to. A scalar to use in all dimensions or a `vector` with one scalar per dimension. **N.B.** Ignored when `` `origin_fn` `` is not `` `NULL` ``.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`shuffle_ties`	Whether to shuffle elements with the same distance to the origin. (Logical)
`origin_col_name`	Name of new column with the origin coordinates. If `` `NULL` ``, no column is added.
`distance_col_name`	Name of new column with the distances to the origin. If `` `NULL` ``, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other rearrange functions: center_max(), center_min(), closest_to(), pair_extremes(), position_max(), position_min(), rev_windows(), roll_elements(), shuffle_hierarchy(), triplet_extremes()

Other distance functions: closest_to(), dim_values(), distance(), expand_distances(), expand_distances_each(), swirl_2d(), swirl_3d()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Furthest from 3 in a vector
furthest_from_vec(1:10, origin = 3)

# Furthest from the third row (index of data.frame)
furthest_from(df, origin = 3)$index

# By each of the columns
furthest_from(df, cols = "A", origin = 3)$A
furthest_from(df, cols = "A", origin_fn = most_centered)$A
furthest_from(df, cols = "B", origin = 0.5)$B
furthest_from(df, cols = "B", origin_fn = centroid)$B

# Shuffle the elements with the same distance to the origin
furthest_from(df,
  cols = "A",
  origin_fn = create_origin_fn(median),
  shuffle_ties = TRUE
)$A

# Grouped by G
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  furthest_from(
    cols = "A",
    origin_fn = create_origin_fn(median)
  )

# Plot the rearranged values
plot(
  x = 1:10,
  y = furthest_from(df,
    cols = "B",
    origin_fn = create_origin_fn(median)
  )$B,
  xlab = "Position", ylab = "B"
)
plot(
  x = 1:10,
  y = furthest_from(df,
    cols = "A",
    origin_fn = create_origin_fn(median),
    shuffle_ties = TRUE
  )$A,
  xlab = "Position", ylab = "A"
)

# In multiple dimensions
df %>%
  furthest_from(cols = c("A", "B"), origin_fn = most_centered)
```

`GeneratedPipeline` *Chain multiple transformations and generate argument values per group*

Description

[Experimental]

Build a pipeline of transformations to be applied sequentially.

Generate argument values for selected arguments with a given set of generators. E.g. randomly generate argument values for each group in a `data.frame`.

Groupings are reset between each transformation. See `group_cols`.

Standard workflow: Instantiate pipeline -> Add transformations -> Apply to data

To apply the same arguments to all groups, see `Pipeline`.

To apply different but specified argument values to a fixed set of groups, see `FixedGroupsPipeline`.

Super class

`rearrr::Pipeline` -> `GeneratedPipeline`

Public fields

`transformations` `list` of transformations to apply.

`names` Names of the transformations.

Methods

Public methods:

- `GeneratedPipeline$add_transformation()`
- `GeneratedPipeline$print()`
- `GeneratedPipeline$clone()`

Method `add_transformation()`**:** Add a transformation to the pipeline.

Usage:

```
GeneratedPipeline$add_transformation(
  fn,
  args,
  generators,
  name,
  group_cols = NULL
)
```

Arguments:

`fn` Function that performs the transformation.

`args` Named `list` with arguments for the `` `fn` `` function.

`generators` Named `list` of functions for generating argument values for a single call of `` `fn` ``.
It is possible to include an *apply generator* for deciding whether the transformation should be applied to the current group or not. This is done by adding a function with the name `` `.apply` `` to the `` `generators` `` list. E.g. `".apply" = function(){sample(c(TRUE, FALSE), 1)}`.

`name` Name of the transformation step. Must be unique.

`group_cols` Names of the columns to group the input data by before applying the transformation.
Note that the transformation function is applied separately to each group (subset). If the `` `fn` `` function requires access to the entire `data.frame`, the grouping columns should be specified as part of `` `args` `` and handled by the `` `fn` `` function.

Returns: The pipeline. To allow chaining of methods.

Examples:

```
# `generators` is a list of functions for generating
# argument values for a chosen set of arguments
# `.apply` can be used to disable the transformation
generators = list(degrees = function(){sample.int(360, 1)},
                  origin = function(){rnorm(2)},
                  .apply = function(){sample(c(TRUE, FALSE), 1)})
```

Method `print()`**:** Print an overview of the pipeline.

Usage:

```
GeneratedPipeline$print(...)
```

Arguments:

`...` further arguments passed to or from other methods.

Returns: The pipeline. To allow chaining of methods.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GeneratedPipeline$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other pipelines: `FixedGroupsPipeline`, `Pipeline`

Examples

```
# Attach package
library(rearrr)

# Create a data frame
df <- data.frame(
  "Index" = 1:12,
  "A" = c(1:4, 9:12, 15:18),
  "G" = rep(1:3, each = 4)
)

# Create new pipeline
pipe <- GeneratedPipeline$new()

# Add 2D rotation transformation
# Note that we specify the grouping via `group_cols`
pipe$add_transformation(
  fn = rotate_2d,
  args = list(
    x_col = "Index",
    y_col = "A",
    suffix = "",
    overwrite = TRUE
  ),
  generators = list(degrees = function(){sample.int(360, 1)},
                    origin = function(){rnorm(2)}),
  name = "Rotate",
  group_cols = "G"
)
```

```
# Add the `cluster_group` transformation
# Note that this function requires the entire input data
# to properly scale the groups. We therefore specify `group_cols`
# as part of `args`. This works as `cluster_groups()` accepts that
# argument.
# Also note the `.apply` generator which generates a TRUE/FALSE scalar
# for whether the transformation should be applied to the current group
pipe$add_transformation(
  fn = cluster_groups,
  args = list(
    cols = c("Index", "A"),
    suffix = "",
    overwrite = TRUE,
    group_cols = "G"
  ),
  generators = list(
    multiplier = function() {
      0.1 * runif(1) * 3 ^ sample.int(5, 1)
    },
    .apply = function(){sample(c(TRUE, FALSE), 1)}
  ),
  name = "Cluster"
)

# Check pipeline object
pipe

# Apply pipeline to data.frame
# Enable `verbose` to print progress
pipe$apply(df, verbose = TRUE)


## ------------------------------------------------
## Method `GeneratedPipeline$add_transformation`
## ------------------------------------------------

# `generators` is a list of functions for generating
# argument values for a chosen set of arguments
# `.apply` can be used to disable the transformation
generators = list(degrees = function(){sample.int(360, 1)},
                  origin = function(){rnorm(2)},
                  .apply = function(){sample(c(TRUE, FALSE), 1)})
```

`GeneratedTransformation`

GeneratedTransformation

Description

[Experimental]

Container for the type of transformation used in `GeneratedPipeline`.

Note: For internal use.

Super class

`rearrr::Transformation` -> `GeneratedTransformation`

Public fields

`name` Name of transformation.

`fn` Transformation function.

`args` `list` of constant arguments for `` `fn` ``.

`generators` `list` of generator functions for generating argument values.

`apply_generator` Generator function for deciding whether to apply the transformation to the current group.

Methods

Public methods:

- `GeneratedTransformation$new()`
- `GeneratedTransformation$get_group_args()`
- `GeneratedTransformation$generate_args()`
- `GeneratedTransformation$print()`
- `GeneratedTransformation$clone()`

Method `new()`**:** Initialize transformation.

Usage:

```
GeneratedTransformation$new(
  fn,
  args,
  generators,
  name = NULL,
  group_cols = NULL
)
```

Arguments:

`fn` Transformation function.

`args` `list` of constant arguments for `` `fn` ``.

`generators` Named `list` of functions for generating argument values for a single call of `` `fn` ``. It is possible to include an *apply generator* for deciding whether the transformation should be applied to the current group or not. This is done by adding a function with the name `` `.apply` `` to the `` `generators` `` list. E.g. `".apply" = function(){sample(c(TRUE, FALSE), 1)}`.

`name` Name of transformation.

`group_cols` Names of columns to group `data.frame` by before applying `` `fn` ``. When `` `NULL` ``, the `data.frame` is not grouped.

Method `get_group_args()`**:** Get arguments for a group.

Usage:

```
GeneratedTransformation$get_group_args()
```

Returns: `list` of arguments (both constant and generated).

Method `generate_args()`**:** Generate arguments for a group with the `` `generators` ``.

Usage:

```
GeneratedTransformation$generate_args()
```

Returns: `list` of generated arguments.
Does not include the constant arguments.

Method `print()`**:** Print an overview of the transformation.

Usage:

```
GeneratedTransformation$print(..., indent = 0, show_class = TRUE)
```

Arguments:

`...` further arguments passed to or from other methods.

`indent` How many spaces to indent when printing.

`show_class` Whether to print the transformation class name.

Returns: The pipeline. To allow chaining of methods.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GeneratedTransformation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other transformation classes: `FixedGroupsTransformation`, `Transformation`

generate_clusters *Generate n-dimensional clusters*

Description

[Experimental]

Generates `data.frame` (`tibble`) with clustered groups.

Usage

```
generate_clusters(
  num_rows,
  num_cols,
  num_clusters,
  compactness = 1.6,
  generator = runif,
  name_prefix = "D",
  cluster_col_name = ".cluster"
)
```

Arguments

`num_rows`	Number of rows.
`num_cols`	Number of columns (dimensions).
`num_clusters`	Number of clusters.
`compactness`	How compact the clusters should be. A larger value leads to more compact clusters (on average). Technically, it is passed to the `` `multiplier` `` argument in `cluster_groups()` as '$0.1/compactness$'.
`generator`	Function to generate the numeric values. Must have the *number of values to generate* as its first (and only required) argument, as that is the only argument we pass to it.
`name_prefix`	Prefix string for naming columns.
`cluster_col_name`	Name of cluster factor.

Details

- Generates `data.frame` with random values using the `` `generator` ``.
- Divides the rows into groups (the clusters).
- Contracts the distance from each data point to the centroid of its group.
- Performs MinMax scaling such that the scale of the data points is similar to the generated data.

Value

`data.frame` (`tibble`) with the clustered columns and the cluster grouping factor.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other clustering functions: `cluster_groups()`, `transfer_centroids()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(10)

# Generate clusters
generate_clusters(num_rows = 20, num_cols = 3, num_clusters = 3, compactness = 1.6)
generate_clusters(num_rows = 20, num_cols = 5, num_clusters = 6, compactness = 2.5)

# Generate clusters and plot them
# Tip: Call this multiple times
# to see the behavior of `generate_clusters()`
if (has_ggplot){
  generate_clusters(
    num_rows = 50, num_cols = 2,
    num_clusters = 5, compactness = 1.6
  ) %>%
    ggplot(
      aes(x = D1, y = D2, color = .cluster)
    ) +
    geom_point() +
    theme_minimal() +
    labs(x = "D1", y = "D2", color = "Cluster")
}

#
# Plot clusters in 3d view
#

# Generate clusters
clusters <- generate_clusters(
  num_rows = 50, num_cols = 3,
  num_clusters = 5, compactness = 1.6
)

## Not run:
```

```
# Plot 3d with plotly
plotly::plot_ly(
  x = clusters$D1,
  y = clusters$D2,
  z = clusters$D3,
  type = "scatter3d",
  mode = "markers",
  color = clusters$.cluster
)

## End(Not run)
```

`hexagonalize` *Create x-coordinates so the points form a hexagon*

Description

[Experimental]

Create the x-coordinates for a `vector` of y-coordinates such that they form a hexagon.

This will likely look most like a hexagon when the y-coordinates are somewhat equally distributed, e.g. a uniform distribution.

Usage

```
hexagonalize(
  data,
  y_col = NULL,
  .min = NULL,
  .max = NULL,
  offset_x = 0,
  keep_original = TRUE,
  x_col_name = ".hexagon_x",
  edge_col_name = ".edge",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`y_col`	Name of column in `` `data` `` with y-coordinates to create x-coordinates for.
`.min`	Minimum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`.max`	Maximum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`offset_x`	Value to offset the x-coordinates by.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.

`x_col_name`	Name of new column with the x-coordinates.
`edge_col_name`	Name of new column with the edge identifiers. If `NULL`, no column is added. Numbering is clockwise and starts at the upper-right edge.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with the added x-coordinates and an identifier for the edge the data point is a part of.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other forming functions: `circularize()`, `square()`, `triangularize()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "y" = runif(200),
  "g" = factor(rep(1:5, each = 40))
)

# Hexagonalize 'y'
df_hex <- hexagonalize(df, y_col = "y")
df_hex

# Plot hexagon
if (has_ggplot){
  df_hex %>%
    ggplot(aes(x = .hexagon_x, y = y, color = .edge)) +
    geom_point() +
    theme_minimal()
}

#
# Grouped hexagonalization
#
```

```
# Hexagonalize 'y' for each group
# First cluster the groups a bit to move the
# hexagons away from each other
df_hex <- df %>%
  cluster_groups(
    cols = "y",
    group_cols = "g",
    suffix = "",
    overwrite = TRUE
  ) %>%
  dplyr::group_by(g) %>%
  hexagonalize(
    y_col = "y",
    overwrite = TRUE
  )

# Plot hexagons
if (has_ggplot){
  df_hex %>%
    ggplot(aes(x = .hexagon_x, y = y, color = g)) +
    geom_point() +
    theme_minimal()
}

#
# Specifying minimum value
#

# Specify minimum value manually
df_hex <- hexagonalize(df, y_col = "y", .min = -2)
df_hex

# Plot hexagon
if (has_ggplot){
  df_hex %>%
    ggplot(aes(x = .hexagon_x, y = y, color = .edge)) +
    geom_point() +
    theme_minimal()
}

#
# Multiple hexagons by contraction
#

# Start by hexagonalizing 'y'
df_hex <- hexagonalize(df, y_col = "y")

# Contract '.hexagon_x' and 'y' towards the centroid
# To contract with multiple multipliers at once,
# we wrap the call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = c(1, 0.75, 0.5, 0.25, 0.125),
  .f = function(mult) {
    expand_distances(
      data = df_hex,
      cols = c(".hexagon_x", "y"),
      multiplier = mult,
      origin_fn = centroid,
      overwrite = TRUE
    )
  }
)
df_expanded

if (has_ggplot){
  df_expanded %>%
    ggplot(aes(
      x = .hexagon_x_expanded, y = y_expanded,
      color = .edge, alpha = .multiplier
    )) +
    geom_point() +
    theme_minimal()
}
```

`is_most_centered` *Find which data point is closest to the centroid*

Description

[Experimental]

Finds the data point with the shortest distance to the centroid.

To get the coordinates of the most centered data point, use `most_centered()` instead.

Usage

```
is_most_centered(..., na.rm = FALSE)
```

Arguments

`...`	`Numeric` vectors.
`na.rm`	Whether to ignore missing values. At least one data point must be complete. (Logical)

Value

Logical vector (`TRUE`/`FALSE`) indicating if a data point is the most centered.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other coordinate functions: `centroid()`, `create_origin_fn()`, `midrange()`, `most_centered()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create three vectors
x <- runif(10)
y <- runif(10)
z <- runif(10)

# Find the data point closest to the centroid
is_most_centered(x, y, z)

# Compare to coordinates for the most centered
most_centered(x, y, z)

#
# For data.frames
#

# Create data frame
df <- data.frame(
  "x" = x,
  "y" = y,
  "z" = z,
  "g" = rep(1:2, each = 5)
)

# Filter the data points
# closest to the centroid
df %>%
  dplyr::filter(is_most_centered(x, y, z))

# When 'df' is grouped
df %>%
  dplyr::group_by(g) %>%
  dplyr::filter(is_most_centered(x, y, z))

# Add as column
df %>%
  dplyr::group_by(g) %>%
  dplyr::mutate(mc = is_most_centered(x, y, z))
```

`median_index` *Find index of interest for each vector*

Description

[Experimental]

Applies function to the indices of each `vector` in `` `...` ``.

These functions were created with `create_n_fn()`.

Usage

```
median_index(..., negate = FALSE, round_fn = round)

quantile_index(..., prob, type = 7, negate = FALSE, round_fn = round)
```

Arguments

`...`	Numeric `vector`s.
`negate`	Whether to negate the result. I.e. to multiply it with `-1`. (Logical)
`round_fn`	Function for rounding output. Rounding is done *prior* to negation. E.g. `round`, `floor`, or `ceiling`.
`prob`	Probability in `[0,1]` for `quantile()`.
`type`	Quantile algorithm to use. See `quantile()`.

Value

`numeric vector` with one element per supplied vector.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other n functions: `create_n_fn()`

Examples

```
# Attach packages
library(rearrr)

# Set seed
set.seed(1)

# Create three vectors
x <- runif(10)
```

```
y <- runif(15)
z <- runif(20)

median_index(x, y, z)
quantile_index(x, y, z, prob = 0.2)

# Negate result
median_index(x, y, z, negate = TRUE)
```

`midrange` *Find the midrange values*

Description

[Experimental]

Calculates the midrange for each of the passed `vectors/columns`.

Midrange is defined as:

$$(max x + min x)/2$$

Usage

```
midrange(..., cols = NULL, na.rm = FALSE)
```

Arguments

`...`	`Numeric vectors` or a single `data.frame`.
`cols`	Names of columns to use when `` `...` `` is a single `data.frame`.
`na.rm`	Whether to ignore missing values when calculating `min` and `max` values. (Logical)

Value

Either a `vector` with the midrange of each supplied `vector` or a `data.frame` with the midrange of each supplied column along with any grouping variables.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other coordinate functions: `centroid()`, `create_origin_fn()`, `is_most_centered()`, `most_centered()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create three vectors
x <- runif(10)
y <- runif(10)
z <- runif(10)

# Find midrange for each vector
midrange(x, y, z)

#
# For data.frames
#

# Create data frame
df <- data.frame(
  "x" = x,
  "y" = y,
  "z" = z,
  "g" = rep(1:2, each = 5)
)

# Find midrange for each column
midrange(df, cols = c("x", "y", "z"))

# When 'df' is grouped
df %>%
  dplyr::group_by(g) %>%
  midrange(cols = c("x", "y", "z"))
```

`min_max_scale` *Scale to a range*

Description

[Experimental]

Scales the values to a range with MinMax scaling.

Usage

```
min_max_scale(
  x,
  new_min,
  new_max,
  old_min = NULL,
  old_max = NULL,
  na.rm = FALSE
)
```

Arguments

`x`	Numeric `vector` to scale.
`new_min`	Minimum value of target range.
`new_max`	Maximum value of target range.
`old_min`	Minimum value of original range. If `NULL`, this is the minimum value in `` `x` ``.
`old_max`	Maximum value of original range. If `NULL`, this is the maximum value in `` `x` ``.
`na.rm`	Whether missing values should be removed when calculating `` `old_min` `` and/or `` `old_max` ``. **N.B.** Ignored when both `` `old_min` `` and `` `old_max` `` are `NULL`.

Value

Scaled version of `` `x` ``.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other scaling functions: `to_unit_length()`

Examples

```
# Attach packages
library(rearrr)

# Set seed
set.seed(1)

# Create numeric vector
x <- runif(10)

# Scale
min_max_scale(x, new_min = -1, new_max = 0)
min_max_scale(x, new_min = -1, new_max = 0, old_max = 3)
```

`most_centered` *Find the coordinates for the data point closest to the centroid*

Description

[Experimental]

Returns the coordinates for the data point with the shortest distance to the `centroid`.

To get a logical vector (`TRUE/FALSE`) indicating whether a data point is the most centered, use `is_most_centered()`.

Usage

```
most_centered(..., cols = NULL, na.rm = FALSE)
```

Arguments

`...`	`Numeric vectors` or a single `data.frame`.
`cols`	Names of columns to use when `` `...` `` is a single `data.frame`.
`na.rm`	Whether to ignore missing values. At least one data point must be complete. (Logical)

Value

The coordinates for the data point closest to the centroid. Either as a `vector` or a `data.frame`.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other coordinate functions: `centroid()`, `create_origin_fn()`, `is_most_centered()`, `midrange()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create three vectors
x <- runif(10)
y <- runif(10)
z <- runif(10)

# Find coordinates of the data point
# closest to the centroid
most_centered(x, y, z)

# Compare to centroid coordinates
centroid(x, y, z)

#
# For data.frames
#

# Create data frame
df <- data.frame(
  "x" = x,
  "y" = y,
  "z" = z,
  "g" = rep(1:2, each = 5)
)

# Find coordinates of the data point
# closest to the centroid
most_centered(df, cols = c("x", "y", "z"))

# When 'df' is grouped
df %>%
  dplyr::group_by(g) %>%
  most_centered(cols = c("x", "y", "z"))

# Filter to only include most centered data points
df %>%
  dplyr::group_by(g) %>%
  dplyr::filter(is_most_centered(x, y, z))
```

`pair_extremes` *Pair extreme values and sort by the pairs*

Description

[Experimental]

The values are paired/grouped such that the lowest and highest values form the first group, the second lowest and the second highest values form the second group, and so on. The values are then sorted by these groups/pairs.

When `` `data` `` has an uneven number of rows, the `` `unequal_method` `` determines which group should have only 1 element.

The `*_vec()` version takes and returns a `vector`.

Example:

The column values:

`c(1, 2, 3, 4, 5, 6)`

Creates the **sorting factor**:

`c(1, 2, 3, 3, 2, 1)`

And are **ordered as**:

`c(1, 6, 2, 5, 3, 4)`

Usage

```
pair_extremes(
  data,
  col = NULL,
  unequal_method = "middle",
  num_pairings = 1,
  balance = "mean",
  order_by_aggregates = FALSE,
  shuffle_members = FALSE,
  shuffle_pairs = FALSE,
  factor_name = ifelse(num_pairings == 1, ".pair", ".pairing"),
  overwrite = FALSE
)

pair_extremes_vec(
  data,
  unequal_method = "middle",
  num_pairings = 1,
  balance = "mean",
  order_by_aggregates = FALSE,
  shuffle_members = FALSE,
  shuffle_pairs = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`col`	Column to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`unequal_method`	Method for dealing with an unequal number of rows/elements in `` `data` ``.

One of: `first`, `middle` or `last`

first: The first group will have size `1`.
Example:
The ordered column values:
`c(1, 2, 3, 4, 5)`
Creates the **sorting factor**:
`c(1, 2, 3, 3, 2)`
And are **ordered as**:
`c(1, 2, 5, 3, 4)`

middle: The middle group will have size `1`.

Example:
The ordered column values:
`c(1, 2, 3, 4, 5)`
Creates the **sorting factor**:
`c(1, 3, 2, 3, 1)`
And are **ordered as**:
`c(1, 5, 3, 2, 4)`

last: The last group will have size `1`.
Example:
The ordered column values:
`c(1, 2, 3, 4, 5)`
Creates the **sorting factor**:
`c(1, 2, 2, 1, 3)`
And are **ordered as**:
`c(1, 4, 2, 3, 5)`

`num_pairings` Number of pairings to perform (recursively). At least `1`.

Based on `` `balance` ``, the secondary pairings perform extreme pairing on either the *sum*, *absolute difference*, *min*, or *max* of the pair elements.

`balance` What to balance pairs for in a given *secondary* pairing. Either `"mean"`, `"spread"`, `"min"`, or `"max"`. Can be a single string used for all secondary pairings or one for each secondary pairing (`` `num_pairings` - 1 ``).

The first pairing always pairs the actual element values.

mean: Pairs have similar means. The values in the pairs from the previous pairing are aggregated with `` `sum()` `` and paired.

spread: Pairs have similar spread (e.g. standard deviations). The values in the pairs from the previous pairing are aggregated with `` `sum(abs(diff()))` `` and paired.

min / max: Pairs have similar minimum / maximum values. The values in the pairs from the previous pairing are aggregated with `` `min()` `` / `` `max()` `` and paired.

`order_by_aggregates`
Whether to order the pairs from initial pairings (first `` `num_pairings` - 1 ``) by their aggregate values instead of their pair identifiers.

N.B. Only used when `` `num_pairings` > 1 ``.

`shuffle_members`
Whether to shuffle the order of the group members within the groups. (Logical)

`shuffle_pairs` Whether to shuffle the order of the pairs. Pair members remain together. (Logical)

`factor_name` Name of new column with the sorting factor. If `` `NULL` ``, no column is added.

`overwrite` Whether to allow overwriting of existing columns. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`. Optionally with the sorting factor added.

When `` `data` `` is a `vector` and `` `keep_factors` `` is `FALSE`, the output will be a `vector`. Otherwise, a `data.frame`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other rearrange functions: `center_max()`, `center_min()`, `closest_to()`, `furthest_from()`, `position_max()`, `position_min()`, `rev_windows()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "C" = LETTERS[1:10],
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Pair extreme indices (row numbers)
pair_extremes(df)

# Pair extremes in each of the columns
pair_extremes(df, col = "A")$A
pair_extremes(df, col = "B")$B
pair_extremes(df, col = "C")$C

# Shuffle the members pair-wise
# The rows within each pair are shuffled
# while the `.pair` column maintains it order
pair_extremes(df, col = "A", shuffle_members = TRUE)

# Shuffle the order of the pairs
# The rows within each pair maintain their order
# and stay together but the `.pair` column is shuffled
pair_extremes(df, col = "A", shuffle_pairs = TRUE)

# Use recursive pairing
# Mostly meaningful with much larger datasets
# Order initial grouping by pair identifiers
```

```
pair_extremes(df, col = "A", num_pairings = 2)
# Order initial grouping by aggregate values
pair_extremes(df, col = "A", num_pairings = 2, order_by_aggregates = TRUE)

# Grouped by G
# Each G group only has 3 elements
# so it only creates 1 pair and a group
# with the single excessive element
# per G group
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  pair_extremes(col = "A")

# Plot the extreme pairs
plot(
  x = 1:10,
  y = pair_extremes(df, col = "B")$B,
  col = as.character(rep(1:5, each = 2))
)
# With shuffled pair members (run a few times)
plot(
  x = 1:10,
  y = pair_extremes(df, col = "B", shuffle_members = TRUE)$B,
  col = as.character(rep(1:5, each = 2))
)
# With shuffled pairs (run a few times)
plot(
  x = rep(1:5, each = 2),
  y = pair_extremes(df, col = "B", shuffle_pairs = TRUE)$B,
  col = as.character(rep(1:5, each = 2))
)
```

Pipeline *Chain multiple transformations*

Description

[Experimental]

Build a pipeline of transformations to be applied sequentially.

Uses the same arguments for all groups in `` `data` ``.

Groupings are reset between each transformation. See `group_cols`.

Standard workflow: Instantiate pipeline -> Add transformations -> Apply to data

To apply different argument values to each group, see `GeneratedPipeline` for generating argument values for an arbitrary number of groups and `FixedGroupsPipeline` for specifying specific values for a fixed set of groups.

Public fields

`transformations` `list` of transformations to apply.

`names` Names of the transformations.

Methods

Public methods:

- `Pipeline$add_transformation()`
- `Pipeline$apply()`
- `Pipeline$print()`
- `Pipeline$clone()`

Method `add_transformation()`**:** Add a transformation to the pipeline.

Usage:

```
Pipeline$add_transformation(fn, args, name, group_cols = NULL)
```

Arguments:

`fn` Function that performs the transformation.

`args` Named `list` with arguments for the `` `fn` `` function.

`name` Name of the transformation step. Must be unique.

`group_cols` Names of the columns to group the input data by before applying the transformation.

Note that the transformation function is applied separately to each group (subset). If the `` `fn` `` function requires access to the entire `data.frame`, the grouping columns should be specified as part of `` `args` `` and handled by the `` `fn` `` function.

Returns: The pipeline. To allow chaining of methods.

Method `apply()`**:** Apply the pipeline to a `data.frame`.

Usage:

```
Pipeline$apply(data, verbose = FALSE)
```

Arguments:

`data` `data.frame`.

A grouped `data.frame` will raise a warning and the grouping will be ignored. Use the `` `group_cols` `` argument in the `` `add_transformation` `` method to specify how `` `data` `` should be grouped for each transformation.

`verbose` Whether to print the progress.

Returns: Transformed version of `` `data` ``.

Method `print()`**:** Print an overview of the pipeline.

Usage:

```
Pipeline$print(...)
```

Arguments:

`...` further arguments passed to or from other methods.

Returns: The pipeline. To allow chaining of methods.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`Pipeline$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other pipelines: `FixedGroupsPipeline`, `GeneratedPipeline`

Examples

```
# Attach package
library(rearrr)

# Create a data frame
df <- data.frame(
  "Index" = 1:12,
  "A" = c(1:4, 9:12, 15:18),
  "G" = rep(1:3, each = 4)
)

# Create new pipeline
pipe <- Pipeline$new()

# Add 2D rotation transformation
# Note that we specify the grouping via `group_cols`
pipe$add_transformation(
  fn = rotate_2d,
  args = list(
    x_col = "Index",
    y_col = "A",
    origin = c(0, 0),
    degrees = 45,
    suffix = "",
    overwrite = TRUE
  ),
  name = "Rotate",
  group_cols = "G"
)

# Add the `cluster_group` transformation
# Note that this function requires the entire input data
# to properly scale the groups. We therefore specify `group_cols`
# as part of `args`. This works as `cluster_groups()` accepts that
# argument.
pipe$add_transformation(
  fn = cluster_groups,
  args = list(
    cols = c("Index", "A"),
    suffix = "",
    overwrite = TRUE,
    multiplier = 0.05,
    group_cols = "G"
  ),
  name = "Cluster"
)

# Check pipeline object
pipe

# Apply pipeline to data.frame
# Enable `verbose` to print progress
pipe$apply(df, verbose = TRUE)
```

---

position_max *Positions the highest values with values decreasing around it*

Description

[Experimental]

The highest value is positioned at the given index/quantile with the other values decreasing around it.

Example:

The column values:

`c(1, 2, 3, 4, 5)`

and `position = 2`

are **ordered as**:

`c(3, 5, 4, 2, 1)`

Usage

```
position_max(data, col = NULL, position = NULL, shuffle_sides = FALSE)
```

Arguments

`data`	`data.frame` or `vector`.
`col`	Column to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`position`	Index or quantile (in `0-1`) at which to position the element of interest.
`shuffle_sides`	Whether to shuffle which elements are left and right of the position. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other rearrange functions: `center_max()`, `center_min()`, `closest_to()`, `furthest_from()`, `pair_extremes()`, `position_min()`, `rev_windows()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "C" = LETTERS[1:10],
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Position the highest index (row number)
position_max(df, position = 3)$index
position_max(df, position = 8)$index

# Position the maximum value in each of the columns
position_max(df, col = "A", position = 3)$A
position_max(df, col = "B", position = 3)$B
position_max(df, col = "C", position = 3)$C

# Randomize which elements are left and right of the position
position_max(df, col = "A", position = 3, shuffle_sides = TRUE)$A

# Grouped by G
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  position_max(col = "A", position = 2)
```

```
# Plot the rearranged values
plot(x = 1:10, y = position_max(df, col = "B", position = 3)$B)
plot(x = 1:10, y = position_max(df, col = "B", position = 3, shuffle_sides = TRUE)$B)
```

`position_min` *Positions the lowest value with values increasing around it*

Description

[Experimental]

The lowest value is positioned at the given index/quantile with the other values increasing around it.

Example:

The column values:

`c(1, 2, 3, 4, 5)`

and `position` = `2`

are **ordered as**:

`c(3, 1, 2, 4, 5)`

Usage

```
position_min(data, col = NULL, position = NULL, shuffle_sides = FALSE)
```

Arguments

`data`	`data.frame` or `vector`.
`col`	Column to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.
`position`	Index or quantile (in `0-1`) at which to position the element of interest.
`shuffle_sides`	Whether to shuffle which elements are left and right of the position. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other rearrange functions: `center_max()`, `center_min()`, `closest_to()`, `furthest_from()`, `pair_extremes()`, `position_max()`, `rev_windows()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "C" = LETTERS[1:10],
  "G" = c(
    1, 1, 1, 2, 2,
    2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Position the smallest index (row number)
position_min(df, position = 3)$index
position_min(df, position = 8)$index

# Position the minimum value in each of the columns
position_min(df, col = "A", position = 3)$A
position_min(df, col = "B", position = 3)$B
position_min(df, col = "C", position = 3)$C

# Randomize which elements are left and right of the position
position_min(df, col = "A", position = 3, shuffle_sides = TRUE)$A

# Grouped by G
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  position_min(col = "A", position = 2)

# Plot the rearranged values
plot(x = 1:10, y = position_min(df, col = "B", position = 3)$B)
plot(x = 1:10, y = position_min(df, col = "B", position = 3, shuffle_sides = TRUE)$B)
```

---

rev_windows *Reverse order window-wise*

Description

[Experimental]

The values are windowed and reversed within windows.

The `*_vec()` version takes and returns a `vector`.

Example:

The column values:

`c(1, 2, 3, 4, 5, 6)`

With `window_size` = `3`

Are **ordered as**:

`c(3, 2, 1, 6, 4, 5)`

Usage

```
rev_windows(data, window_size, factor_name = ".window", overwrite = FALSE)

rev_windows_vec(data, window_size)
```

Arguments

`data`	`data.frame` or `vector`.
`window_size`	Size of the windows. (Logical)
`factor_name`	Name of the factor with window identifiers. If `` `NULL` ``, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`. Optionally with the windows factor added.

When `` `data` `` is a `vector` and `` `keep_windows` `` is `FALSE`, the output will be a `vector`. Otherwise, a `data.frame`.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other rearrange functions: `center_max()`, `center_min()`, `closest_to()`, `furthest_from()`, `pair_extremes()`, `position_max()`, `position_min()`, `roll_elements()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)
```

```
# Create a data frame
df <- data.frame(
  "index" = 1:10,
  "A" = sample(1:10),
  "B" = runif(10),
  "C" = LETTERS[1:10],
  "G" = rep(1:2, each = 5),
  stringsAsFactors = FALSE
)

# For vector
rev_windows_vec(1:10, window_size = 3)

# For data frame
rev_windows(df, window_size = 3)
rev_windows(df, window_size = 3, factor_name = NULL)

# Grouped by G
df %>%
  dplyr::select(G, index) %>% # For clarity
  dplyr::group_by(G) %>%
  rev_windows(window_size = 3)

# Plot the extreme pairs
plot(
  x = 1:10,
  y = rev_windows_vec(1:10, window_size = 3)
)
```

`roll_elements` *Roll elements*

Description

[Experimental]

Rolls positions of elements.

Example:

Rolling `c(1, 2, 3, 4, 5)` with `` `n = 2` `` becomes:

`c(3, 4, 5, 1, 2)`

`roll_elements_vec()` takes and returns a `vector`.

Should not be confused with `roll_values()`, which changes the *values* of the elements and wraps to a given range.

Usage

```
roll_elements(
  data,
  cols = NULL,
  n = NULL,
  n_fn = NULL,
  n_col_name = ".n",
  overwrite = FALSE,
  ...
)

roll_elements_vec(data, n = NULL, n_fn = NULL, ...)
```

Arguments

`data`	`vector` or `data.frame` to roll elements of. When a `data.frame` is grouped, the rolling is applied group-wise.
`cols`	Names of columns in `` `data` `` to roll. If `NULL`, the *index* is rolled and used to reorder `` `data` ``. **N.B.** only used when `` `data` `` is a `data.frame`.
`n`	Number of positions to roll. A positive number rolls *left/up*. A negative number rolls *right/down*.
`n_fn`	Function to find `` `n` ``. Useful when `` `data` `` is a grouped `data.frame` and `` `n` `` should depend on the rows in the group. **Input**: Each specified `vector/column` in `` `data` `` is passed to the function as a separate argument. **Output**: It should return either a `vector` with one `integer-like scalar` *per column* or a single `integer-like scalar` to use for all columns. Can be created with `create_n_fn()`. See also `median_index()` and `quantile_index()`.
`n_col_name`	Name of new column with the applied `` `n` `` values. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of columns with the same name as `` `n_col_name` ``. (Logical)
`...`	Extra arguments for `` `n_fn` ``.

Value

Rolled `` `data` ``.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other roll functions: `roll_values()`

Other rearrange functions: `center_max()`, `center_min()`, `closest_to()`, `furthest_from()`, `pair_extremes()`, `position_max()`, `position_min()`, `rev_windows()`, `shuffle_hierarchy()`, `triplet_extremes()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Roll vector left
roll_elements(1:10, n = 2)

# Roll vector right and return the vector
roll_elements_vec(1:10, n = -2)

# Roll vector left by median index (rounded to 6)
roll_elements(3:12, n_fn = median_index)

# Roll vector right by median value (rounded to 8)
roll_elements(3:12, n_fn = create_n_fn(median, negate = TRUE))

# Pass extra arguments (here 'prob') to 'n_fn' via '...'
roll_elements(
  1:10,
  n_fn = quantile_index,
  prob = 0.2
)

#
# Roll data.frame
#

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = 1:20,
  "y" = runif(20) * 10,
  "g" = rep(1:4, each = 5)
)

# Roll rows left/up
roll_elements(df, n = 2)

# Roll rows right/down
roll_elements(df, n = -2)

# Roll 'x' column right/down
roll_elements(df, cols = "x", n = -2)

# Roll rows right by median index in each group
# Specify 'negate' for the 'median_index' function
roll_elements(
  df %>% dplyr::group_by(g),
  n_fn = median_index,
  negate = TRUE
)
```

roll_values *Shift values and wrap to range*

Description

[Experimental]

Adds a specified value to each element in the vector and wraps the values around the min-max range with:

$(x - .min)$ % $(.max - .min + between) + .min$

Useful when adding to the degrees of a circle, where the values should remain in the `0-360` range. A value larger than `360` will start over from `0`, e.g. $365- > 5$, while a value smaller than `0` would become e.g. $-5- > 355$. Here, `0` and `360` are considered the same angle. If we were instead adding days to the weekdays `1-7`, where `1` and `7` are separate days, we can set `` `between = 1` `` to have one day in-between them.

`wrap_to_range()` is a wrapper with `` `add = 0` ``.

The `*_vec()` versions take and return a vector.

Should not be confused with `roll_elements()`, which changes the *positions* of the elements.

Usage

```
roll_values(
  data,
  cols = NULL,
  add = 0,
  .min = NULL,
  .max = NULL,
  between = 0,
  na.rm = FALSE,
  suffix = "_rolled",
  keep_original = TRUE,
  range_col_name = ".range",
  overwrite = FALSE
)

wrap_to_range(
  data,
  cols = NULL,
  .min = NULL,
  .max = NULL,
  between = 0,
  na.rm = FALSE,
  suffix = "_wrapped",
  keep_original = TRUE,
  range_col_name = ".range",
  overwrite = FALSE
)

roll_values_vec(
  data,
  add = 0,
  .min = NULL,
  .max = NULL,
  between = 0,
  na.rm = FALSE
)

wrap_to_range_vec(data, .min = NULL, .max = NULL, between = 0, na.rm = FALSE)
```

Arguments

`data`	`vector` or `data.frame` to roll/wrap values of. When a `data.frame` is grouped, the rolling/wrapping is applied group-wise.
`cols`	Names of columns to roll/wrap in `` `data` ``. Must be specified when `` `data` `` is a `data.frame`.
`add`	Amount to add to each element. (`numeric scalar`) When `0`, the wrapping is applied without any rolling.
`.min`	Minimum value allowed. If `NULL`, the minimum value in the `vector/column` is used.
`.max`	Maximum value allowed. If `NULL`, the maximum value in the `vector/column` is used.
`between`	The wrapping distance between `` `.max` `` and `` `.min` ``. When `0`, they are considered the same. I.e. ‘$.max == .min$‘. When `1`, `` `x` `` can be greater than `` `.max` `` by up to `1`, why `` `.min` `` and `` `.max` `` are two separate values with `1` in-between them. I.e. ‘$.max + 1 == .min$‘.
`na.rm`	Whether to remove missing values (`NA`s) when finding the `` `.min` `` and `` `.max` `` values.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`range_col_name`	Name of new column with the min-max range. If `NULL`, no column is added. **N.B.** Ignored when `` `data` `` is a `vector`.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`` `data` `` with new columns with values in the specified min-max range(s) and columns with the applied ranges.

The `*_vec()` versions return a `vector`.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other roll functions: `roll_elements()`

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)

# Add 90 to all degrees
# Note that 0 and 360 is considered the same angle
# why there is no distance between the two
roll_values(c(0:360), add = 90)

# Get as vector
roll_values_vec(c(0:360), add = 90)

# Change limits to 0-180
# so e.g. 270 becomes 90
roll_values(c(0:360), .min = 0, .max = 180)

# Change values first, then wrap to range
x <- c(1:7)
x <- x^2
wrap_to_range(x, .min = 1, .max = 7)

# With 1 in-between .min and .max
wrap_to_range(x, .min = 1, .max = 7, between = 1)

# Get as vector
wrap_to_range_vec(x, .min = 1, .max = 7, between = 1)

#
# Roll data.frame
#

# Set seed
set.seed(1)
```

```
# Create a data frame
df <- data.frame(
  "w" = 1:7,
  "d" = c(0, 45, 90, 135, 180, 270, 360)
)

# Roll weekdays by 1 day
roll_values(
  df,
  cols = "w",
  add = 1,
  .min = 1,
  .max = 7,
  between = 1
)

# Roll degrees by -90 degrees
roll_values(
  df,
  cols = "d",
  add = -90,
  .min = 0,
  .max = 360,
  between = 0
)

# Roll both weekdays and degrees by 1
# We don't specify .min and .max, so they
# are based on the values in the columns
# Note: This is not that meaningful but shows
# the option
roll_values(
  df,
  cols = c("w", "d"),
  add = 1
)

# Wrap weekdays to 2-5 range
wrap_to_range(
  df,
  cols = "w",
  .min = 2,
  .max = 5,
  between = 1
)
```

`rotate_2d` *Rotate the values around an origin in 2 dimensions*

Description

[Experimental]

The values are rotated counterclockwise around a specified origin.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and rotating around e.g. the centroid of each group.

Usage

```
rotate_2d(
  data,
  degrees,
  x_col = NULL,
  y_col = NULL,
  suffix = "_rotated",
  origin = NULL,
  origin_fn = NULL,
  keep_original = TRUE,
  degrees_col_name = ".degrees",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`degrees`	Degrees to rotate values counterclockwise. In `[-360, 360]`. Can be a `vector` with multiple degrees.
`x_col`	Name of x column in `` `data` ``. If `NULL` and `` `data` `` is a `vector`, the index of `` `data` `` is used. If `` `data` `` is a `data.frame`, it must be specified.
`y_col`	Name of y column in `` `data` ``. If `` `data` `` is a `data.frame`, it must be specified.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`origin`	Coordinates of the origin to rotate around. A `vector` with 2 elements (i.e. origin_x, origin_y). Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.

`degrees_col_name`
: Name of new column with the degrees. If `NULL`, no column is added.

`origin_col_name`
: Name of new column with the origin coordinates. If `NULL`, no column is added.

`overwrite`
: Whether to allow overwriting of existing columns. (Logical)

Details

Applies the following rotation matrix:

$$\begin{matrix} [\ cos\theta & , -sin\theta &] \\ [\ sin\theta & , cos\theta &] \end{matrix}$$

That is:

$x' = x cos\theta - y sin\theta$

$y' = x sin\theta + y cos\theta$

Where θ is the angle in radians.

As specified at Wikipedia/Rotation_matrix.

Value

`data.frame` (`tibble`) with seven new columns containing the rotated x-,y- and z-values and the degrees, radiuses and origin coordinates.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Other rotation functions: `rotate_3d()`, `swirl_2d()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "Index" = 1:12,
  "A" = c(1:4, 9:12, 15:18),
  "G" = rep(1:3, each = 4)
)

# Rotate values around (0, 0)
rotate_2d(df, degrees = 45, x_col = "Index", y_col = "A", origin = c(0, 0))

# Rotate A around the centroid
df_rotated <- df %>%
  rotate_2d(
    x_col = "Index",
    y_col = "A",
    degrees = c(0, 120, 240),
    origin_fn = centroid
  )
df_rotated

# Plot A and A rotated around overall centroid
if (has_ggplot){
  ggplot(df_rotated, aes(x = Index_rotated, y = A_rotated, color = factor(.degrees))) +
    geom_hline(yintercept = mean(df$A), size = 0.2, alpha = .4, linetype = "dashed") +
   geom_vline(xintercept = mean(df$Index), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_line(alpha = .4) +
    geom_point() +
    theme_minimal() +
    labs(x = "Index", y = "Value", color = "Degrees")
}

# Rotate around group centroids
df_grouped <- df %>%
  dplyr::group_by(G) %>%
  rotate_2d(
    x_col = "Index",
    y_col = "A",
    degrees = c(0, 120, 240),
    origin_fn = centroid
  )
df_grouped

# Plot A and A rotated around group centroids
if (has_ggplot){
  ggplot(df_grouped, aes(x = Index_rotated, y = A_rotated, color = factor(.degrees))) +
    geom_point() +
    theme_minimal() +
    labs(x = "Index", y = "Value", color = "Degrees")
}
```

`rotate_3d` *Rotate the values around an origin in 3 dimensions*

Description

[Experimental]

The values are rotated counterclockwise around a specified origin.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and rotating around e.g. the centroid of each group.

Usage

```
rotate_3d(
  data,
  x_col,
  y_col,
  z_col,
  x_deg = 0,
  y_deg = 0,
  z_deg = 0,
  suffix = "_rotated",
  origin = NULL,
  origin_fn = NULL,
  keep_original = TRUE,
  degrees_col_name = ".degrees",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`x_col, y_col, z_col`	Name of x/y/z column in `` `data` ``. All must be specified.
`x_deg, y_deg, z_deg`	Degrees to rotate values around the x/y/z-axis counterclockwise. In `[-360, 360]`. Can be `vector`s with multiple degrees. `` `x_deg` `` is *roll*. `` `y_deg` `` is *pitch*. `` `z_deg` `` is *yaw*.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`origin`	Coordinates of the origin to rotate around. `Vector` with 3 elements (i.e. origin_x, origin_y, origin_z). Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`

`keep_original` Whether to keep the original columns. (Logical)

Some columns may have been overwritten, in which case only the newest versions are returned.

`degrees_col_name` Name of new column with the degrees. If `NULL`, no column is added.

Also adds a string version with the same name + `"_str"`, making it easier to group by the degrees when plotting multiple rotations.

`origin_col_name` Name of new column with the origin coordinates. If `NULL`, no column is added.

`overwrite` Whether to allow overwriting of existing columns. (Logical)

Details

Applies the following rotation matrix:

$$\begin{array}{llll} [\ cos\alpha cos\beta & , cos\alpha sin\beta sin\gamma - sin\alpha cos\gamma & , cos\alpha sin\beta cos\gamma + sin\alpha sin\gamma &] \\ [\ sin\alpha cos\beta & , sin\alpha sin\beta sin\gamma + cos\alpha cos\gamma & , sin\alpha sin\beta cos\gamma - cos\alpha sin\gamma &] \\ [\ -sin\beta & , cos\beta sin\gamma & , cos\beta cos\gamma &] \end{array}$$

Where $\alpha =$ `` `z_deg` `` in radians, $\beta =$ `` `y_deg` `` in radians, $\gamma =$ `` `x_deg` `` in radians.

As specified at Wikipedia/Rotation_matrix.

Value

`data.frame` (`tibble`) with six new columns containing the rotated x-,y- and z-values and the degrees and origin coordinates.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Other rotation functions: `rotate_2d()`, `swirl_2d()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(3)
```

```
# Create a data frame
df <- data.frame(
  "x" = 1:12,
  "y" = c(1:4, 9:12, 15:18),
  "z" = runif(12),
  "g" = rep(1:3, each = 4)
)

# Rotate values 45 degrees around x-axis at (0, 0, 0)
rotate_3d(df, x_col = "x", y_col = "y", z_col = "z", x_deg = 45, origin = c(0, 0, 0))

# Rotate all axes around the centroid
df_rotated <- df %>%
  rotate_3d(
    x_col = "x",
    y_col = "y",
    z_col = "z",
    x_deg = c(0, 72, 144, 216, 288),
    y_deg = c(0, 72, 144, 216, 288),
    z_deg = c(0, 72, 144, 216, 288),
    origin_fn = centroid
  )
df_rotated

# Plot rotations
if (has_ggplot){
 ggplot(df_rotated, aes(x = x_rotated, y = y_rotated, color = .degrees_str, alpha = z_rotated)) +
    geom_vline(xintercept = mean(df$x), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_hline(yintercept = mean(df$y), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_line(alpha = .4) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "degrees", alpha = "z (opacity)")
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_rotated$x_rotated,
  y = df_rotated$y_rotated,
  z = df_rotated$z_rotated,
  type = "scatter3d",
  mode = "markers",
  color = df_rotated$.degrees_str
)

## End(Not run)


# Rotate randomly around all axes
df_rotated <- df %>%
  rotate_3d(
    x_col = "x",
    y_col = "y",
    z_col = "z",
    x_deg = round(runif(10, min = 0, max = 360)),
    y_deg = round(runif(10, min = 0, max = 360)),
    z_deg = round(runif(10, min = 0, max = 360)),
    origin_fn = centroid
  )
df_rotated

# Plot rotations
if (has_ggplot){
 ggplot(df_rotated, aes(x = x_rotated, y = y_rotated, color = .degrees_str, alpha = z_rotated)) +
    geom_vline(xintercept = mean(df$x), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_hline(yintercept = mean(df$y), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_line(alpha = .4) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "degrees", alpha = "z (opacity)")
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_rotated$x_rotated,
  y = df_rotated$y_rotated,
  z = df_rotated$z_rotated,
  type = "scatter3d",
  mode = "markers",
  color = df_rotated$.degrees_str
)

## End(Not run)


# Rotate around group centroids
df_grouped <- df %>%
  dplyr::group_by(g) %>%
  rotate_3d(
    x_col = "x",
    y_col = "y",
    z_col = "z",
    x_deg = c(0, 72, 144, 216, 288),
    y_deg = c(0, 72, 144, 216, 288),
    z_deg = c(0, 72, 144, 216, 288),
    origin_fn = centroid
  )

# Plot A and A rotated around group centroids
if (has_ggplot){
 ggplot(df_grouped, aes(x = x_rotated, y = y_rotated, color = .degrees_str, alpha = z_rotated)) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "degrees", alpha = "z (opacity)")
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_grouped$x_rotated,
  y = df_grouped$y_rotated,
  z = df_grouped$z_rotated,
  type = "scatter3d",
  mode = "markers",
  color = df_grouped$.degrees_str
)

## End(Not run)
```

`shear_2d` *Shear the values around an origin in 2 dimensions*

Description

[Experimental]

Shear a set of 2d points around an origin. The shearing formulas (excluding the origin movements) is:

$$x' = x + x_shear * y$$

$$y' = y + y_shear * x$$

The data points in `` `data` `` are moved prior to the shearing, to bring the origin to `0` in all dimensions. After the shearing, the inverse move is applied to bring the origin back to its original position.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and shearing around e.g. the centroid of each group.

Usage

```
shear_2d(
  data,
  x_shear,
  y_shear = 0,
  x_col = NULL,
  y_col = NULL,
  suffix = "_sheared",
  origin = NULL,
  origin_fn = NULL,
  keep_original = TRUE,
  shear_col_name = ".shear",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`x_shear`	Shear factor for the x dimension (`numeric`). Decides the amount of shearing. Can be a `vector` with multiple shear factors.
`y_shear`	Shear factor for the y dimension (`numeric`). Decides the amount of shearing. Can be a `vector` with multiple shear factors.
`x_col`	Name of x column in `` `data` ``.
`y_col`	Name of y column in `` `data` ``.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`origin`	Coordinates of the origin to shear around. `Vector` with 2 elements (origin_x, origin_y). Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`shear_col_name`	Name of new column with the shearing factors. If `NULL`, no column is added. Also adds a string version with the same name + `"_str"`, making it easier to group by the shearing factors when plotting multiple shearings.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with sheared columns, the shearing factors and the origin coordinates.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_3d()`, `swirl_2d()`, `swirl_3d()`

Other shearing functions: `shear_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Create a data frame
df <- data.frame(
  "x" = rep(1:6, each = 2),
  "y" = rep(c(1, 4), 6),
  "g" = rep(1:2, each = 6)
)

# Shear the x variable with regards to y
# around the centroid
df_sheared <- shear_2d(
  data = df,
  x_shear = 2.5,
  x_col = "x",
  y_col = "y",
  origin_fn = centroid
)

# Plot sheared data
# Black: original points
# Red: sheared points
if (has_ggplot){
  df_sheared %>%
    ggplot(aes(x = x, y = y)) +
    geom_point() +
    geom_point(aes(x = x_sheared, y = y_sheared, color = "red")) +
    theme_minimal()
}

# Shear in both dimensions
df_sheared <- shear_2d(
  data = df,
  x_shear = 2.5,
  y_shear = 2.5,
  x_col = "x",
  y_col = "y",
  origin_fn = centroid
)

# Plot sheared data
# Black: original points
# Red: sheared points
if (has_ggplot){
  df_sheared %>%
    ggplot(aes(x = x, y = y)) +
    geom_point() +
    geom_point(aes(x = x_sheared,y = y_sheared, color = "red")) +
    theme_minimal()
}

# Shear grouped data frame
# Affects the calculated origin
df_sheared <- shear_2d(
  data = dplyr::group_by(df, g),
  x_shear = 2.5,
  x_col = "x",
  y_col = "y",
  origin_fn = centroid
)

# Plot sheared data
# Black: original points
# Red: sheared points
if (has_ggplot){
  df_sheared %>%
    ggplot(aes(x = x, y = y)) +
    geom_point() +
    geom_point(aes(x = x_sheared, y = y_sheared, color = "red")) +
    theme_minimal()
}

# Shear a vector with multiple shearing factors
shear_2d(
  data = c(1:10),
  x_shear = c(1, 1.5, 2, 2.5),
  origin = c(0, 0)
)
```

`shear_3d` *Shear values around an origin in 3 dimensions*

Description

[Experimental]

Shears points around an origin in 3-dimensional space. See applied shearing matrices under **Details**.

The data points in `` `data` `` are moved prior to the shearing, to bring the origin to `0` in all dimensions. After the shearing, the inverse move is applied to bring the origin back to its original position.

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and shearing around e.g. the centroid of each group.

Usage

```
shear_3d(
  data,
  x_col,
  y_col,
  z_col,
  x_shear = NULL,
  y_shear = NULL,
  z_shear = NULL,
  suffix = "_sheared",
  origin = NULL,
  origin_fn = NULL,
  keep_original = TRUE,
  shear_col_name = ".shear",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`x_col, y_col, z_col`	Name of x/y/z column in `` `data` ``. All must be specified.
`x_shear, y_shear, z_shear`	Shear factor for the x/y/z dimension (`numeric`). Decides the amount of shearing. Can be `vectors` with multiple shear factors. **N.B.** Exactly **2** of the dimensions must have shear factors specified.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`origin`	Coordinates of the origin to shear around. `Vector` with 3 elements (i.e. origin_x, origin_y, origin_z). Ignored when `` `origin_fn` `` is not `NULL`.
`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`shear_col_name`	Name of new column with the shearing amounts. If `NULL`, no column is added. Also adds a string version with the same name + `"_str"`, making it easier to group by the shearing amounts when plotting multiple shears.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Details

Applies one of the following transformation matrices, depending on which two shearing amounts are specified:

Given `x_shear` and `y_shear`:

```
[ 1   , 0   , x_shear   ]
[ 0   , 1   , y_shear   ]
[ 0   , 0   , 1         ]
```

Given `x_shear` and `z_shear`:

```
[ 1   , x_shear   , 0   ]
[ 0   , 1         , 0   ]
[ 0   , z_shear   , 1   ]
```

Given `y_shear` and `z_shear`:

```
[ 1         , 0   , 0   ]
[ y_shear   , 1   , 0   ]
[ z_shear   , 0   , 1   ]
```

Value

`data.frame` (`tibble`) with six new columns containing the sheared x-, y- and z-values and the shearing amounts and origin coordinates.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `swirl_2d()`, `swirl_3d()`

Other shearing functions: `shear_2d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
```

```
set.seed(1)

df_square <- square(runif(100)) %>%
  rename(x = .square_x,
         y = Value) %>%
  mutate(z = 1)

# Shear the x and z axes
# around the centroid
df_sheared <- shear_3d(
  data = df_square,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  x_shear = 2,
  z_shear = 4,
  origin_fn = centroid
)

# Plot sheared data
# Black: original points
# Red: sheared points
if (has_ggplot){
  df_sheared %>%
    ggplot(aes(x = x, y = y)) +
    geom_point() +
    geom_point(aes(x = x_sheared, y = y_sheared, color = "red")) +
    theme_minimal()
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_sheared$x_sheared,
  y = df_sheared$y_sheared,
  z = df_sheared$z_sheared,
  type = "scatter3d",
  mode = "markers",
  color = df_sheared$.shear_str
)

## End(Not run)

# Shear the y and z axes
# around the centroid
df_sheared <- shear_3d(
  data = df_square,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  y_shear = 2,
  z_shear = 4,
  origin_fn = centroid
```

```
)

# Plot sheared data
# Black: original points
# Red: sheared points
if (has_ggplot){
  df_sheared %>%
    ggplot(aes(x = x, y = y)) +
    geom_point() +
    geom_point(aes(x = x_sheared, y = y_sheared, color = "red")) +
    theme_minimal()
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_sheared$x_sheared,
  y = df_sheared$y_sheared,
  z = df_sheared$z_sheared,
  type = "scatter3d",
  mode = "markers",
  color = df_sheared$.shear_str
)

## End(Not run)

# Shear the y and z axes with multiple amounts at once
# around the centroid
df_sheared <- shear_3d(
  data = df_square,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  y_shear = c(0, 2, 4),
  z_shear = c(0, 4, 6),
  origin_fn = centroid
)

# Plot sheared data
if (has_ggplot){
  df_sheared %>%
    ggplot(aes(x = x_sheared, y = y_sheared, color = .shear_str)) +
    geom_point() +
    theme_minimal()
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_sheared$x_sheared,
  y = df_sheared$y_sheared,
  z = df_sheared$z_sheared,
  type = "scatter3d",
  mode = "markers",
  color = df_sheared$.shear_str
)

## End(Not run)
```

---

shuffle_hierarchy *Shuffle multi-column hierarchy of groups*

Description

[Experimental]

Shuffles a tree/hierarchy of groups, one column at a time. The levels in the last ("leaf") column are shuffled first, then the second-last column, and so on. Elements of the same group are ordered sequentially.

Usage

```
shuffle_hierarchy(
  data,
  group_cols,
  cols_to_shuffle = group_cols,
  leaf_has_groups = TRUE
)
```

Arguments

`data`	`data.frame`.
`group_cols`	Names of columns making up the group hierarchy. The last column is the *leaf* and is shuffled first (if also in `` `cols_to_shuffle` ``).
`cols_to_shuffle`	Names of columns to shuffle hierarchically. By default, all the `` `group_cols` `` are shuffled.
`leaf_has_groups`	Whether the leaf column contains groups or values. (Logical) When the elements are *group identifiers*, they are ordered sequentially and shuffled together. When the elements are *values*, they are simply shuffled.

Value

The shuffled `data.frame` (`tibble`).

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other rearrange functions: center_max(), center_min(), closest_to(), furthest_from(), pair_extremes(), position_max(), position_min(), rev_windows(), roll_elements(), triplet_extremes()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

df <- data.frame(
  'a' = rep(1:4, each = 4),
  'b' = rep(1:8, each = 2),
  'c' = 1:16
)

# Set seed for reproducibility
set.seed(2)

# Shuffle all columns
shuffle_hierarchy(df, group_cols = c('a', 'b', 'c'))

# Don't shuffle 'b' but keep grouping by it
# So 'c' will be shuffled within each group in 'b'
shuffle_hierarchy(
  data = df,
  group_cols = c('a', 'b', 'c'),
  cols_to_shuffle = c('a', 'c')
)

# Shuffle 'b' as if it's not a group column
# so elements are independent within their group
# (i.e. same-valued elements are not necessarily ordered sequentially)
shuffle_hierarchy(df, group_cols = c('a', 'b'), leaf_has_groups = FALSE)
```

---

square *Create x-coordinates so the points form a square*

Description

[Experimental]

Create the x-coordinates for a `vector` of y-coordinates such that they form a rotated square.

This will likely look most like a square when the y-coordinates are somewhat equally distributed, e.g. a uniform distribution.

Usage

```
square(
  data,
  y_col = NULL,
  .min = NULL,
  .max = NULL,
  offset_x = 0,
  keep_original = TRUE,
  x_col_name = ".square_x",
  edge_col_name = ".edge",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`y_col`	Name of column in `` `data` `` with y-coordinates to create x-coordinates for.
`.min`	Minimum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`.max`	Maximum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`offset_x`	Value to offset the x-coordinates by.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`x_col_name`	Name of new column with the x-coordinates.
`edge_col_name`	Name of new column with the edge identifiers. If `NULL`, no column is added. Numbering is clockwise and starts at the upper-right edge.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with the added x-coordinates and an identifier for the edge the data point is a part of.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other forming functions: `circularize()`, `hexagonalize()`, `triangularize()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
```

```
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "y" = runif(200),
  "g" = factor(rep(1:5, each = 40))
)

# Square 'y'
df_sq <- square(df, y_col = "y")
df_sq

# Plot square
if (has_ggplot){
  df_sq %>%
    ggplot(aes(x = .square_x, y = y, color = .edge)) +
    geom_point() +
    theme_minimal()
}

#
# Grouped squaring
#

# Square 'y' for each group
# First cluster the groups a bit to move the
# squares away from each other
df_sq <- df %>%
  cluster_groups(
    cols = "y",
    group_cols = "g",
    suffix = "",
    overwrite = TRUE
  ) %>%
  dplyr::group_by(g) %>%
  square(
    y_col = "y",
    overwrite = TRUE
  )

# Plot squares
if (has_ggplot){
  df_sq %>%
    ggplot(aes(x = .square_x, y = y, color = g)) +
    geom_point() +
    theme_minimal()
}

#
# Specifying minimum value
```

```
#

# Specify minimum value manually
df_sq <- square(df, y_col = "y", .min = -2)
df_sq

# Plot square
if (has_ggplot){
  df_sq %>%
    ggplot(aes(x = .square_x, y = y, color = .edge)) +
    geom_point() +
    theme_minimal()
}

#
# Multiple squares by contraction
#

# Start by squaring 'y'
df_sq <- square(df, y_col = "y")

# Contract '.square_x' and 'y' towards the centroid
# To contract with multiple multipliers at once,
# we wrap the call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = c(1, 0.75, 0.5, 0.25, 0.125),
  .f = function(mult) {
    expand_distances(
      data = df_sq,
      cols = c(".square_x", "y"),
      multiplier = mult,
      origin_fn = centroid,
      overwrite = TRUE
    )
  }
)
df_expanded

if (has_ggplot){
  df_expanded %>%
    ggplot(aes(
      x = .square_x_expanded, y = y_expanded,
      color = .edge, alpha = .multiplier
    )) +
    geom_point() +
    theme_minimal()
}
```

swirl_2d *Swirl the values around an origin in 2 dimensions*

Description

[Experimental]

The values are swirled counterclockwise around a specified origin. The swirling is done by rotating around the origin with the degrees based on the distances to the origin as so:

$$degrees = scale_fn(distances)/(2 * radius) * 360$$

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and swirling around e.g. the centroid of each group.

Usage

```
swirl_2d(
  data,
  radius,
  x_col = NULL,
  y_col = NULL,
  suffix = "_swirled",
  origin = NULL,
  origin_fn = NULL,
  scale_fn = identity,
  keep_original = TRUE,
  degrees_col_name = ".degrees",
  radius_col_name = ".radius",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`radius`	Radius of the most-inner swirl on the x-axis in the *simplest* case. A negative number changes the direction to clockwise rotation. Can be a `vector` with multiple radiuses. Note: With a custom `` `scaling_fn` ``, this might not be the actual swirl radius anymore. Think of it more as a width setting where a larger number leads to fewer full rotations.
`x_col`	Name of x column in `` `data` ``. If `NULL` and `` `data` `` is a `vector`, the index of `` `data` `` is used. If `` `data` `` is a `data.frame`, it must be specified.
`y_col`	Name of y column in `` `data` ``. If `` `data` `` is a `data.frame`, it must be specified.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`origin`	Coordinates of the origin to swirl around. `Vector` with 2 elements (i.e. origin_x, origin_y). Ignored when `` `origin_fn` `` is not `NULL`.

`origin_fn`	Function for finding the origin coordinates. **Input**: Each column will be passed as a `vector` in the order of `` `cols` ``. **Output**: A `vector` with one scalar per dimension. Can be created with `create_origin_fn()` if you want to apply the same function to each dimension. E.g. `` `create_origin_fn(median)` `` would find the median of each column. **Built-in functions** are `centroid()`, `most_centered()`, and `midrange()`

`scale_fn`	Function for scaling the distances before calculating the degrees.
	Input: A `numeric vector` (the distances).
	Output: A `numeric vector` (the scaled distances) of the same length.
	E.g.:
	`function(d){`
	`  d ^ 1.5`
	`}`

`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`degrees_col_name`	Name of new column with the degrees. If `NULL`, no column is added.
`radius_col_name`	Name of new column with the radius. If `NULL`, no column is added.
`origin_col_name`	Name of new column with the origin coordinates. If `NULL`, no column is added.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with three new columns containing the swirled x- and y-values, the degrees, the radius, and the origin coordinates.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_3d()`

Other rotation functions: `rotate_2d()`, `rotate_3d()`, `swirl_3d()`

Other distance functions: `closest_to()`, `dim_values()`, `distance()`, `expand_distances()`, `expand_distances_each()`, `furthest_from()`, `swirl_3d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(4)

# Create a data frame
df <- data.frame(
  "x" = 1:50,
  "y" = 1,
  "r1" = runif(50),
  "r2" = runif(50) * 35,
  "g" = rep(1:5, each = 10)
)

# Swirl values around (0, 0)
swirl_2d(
  data = df,
  radius = 45,
  x_col = "x",
  y_col = "y",
  origin = c(0, 0)
)


# Swirl around the centroid
# with 6 different radius settings
# Scale the distances with custom function
df_swirled <- swirl_2d(
  data = df,
  radius = c(95, 96, 97, 98, 99, 100),
  x_col = "x",
  y_col = "y",
  origin_fn = centroid,
  scale_fn = function(d) {
    d^1.6
  }
)

df_swirled

# Plot swirls
if (has_ggplot){
  df_swirled %>%
    ggplot(aes(x = x_swirled, y = y_swirled, color = factor(.radius))) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = ".radius")
}
```

```
#
# Swirl random data
# The trick lies in finding the right radius
#

# Swirl the random columns
df_swirled <- swirl_2d(
  data = df,
  radius = 5,
  x_col = "r1",
  y_col = "r2",
  origin_fn = centroid
)

# Plot swirls
if (has_ggplot){
  df_swirled %>%
    ggplot(aes(x = r1_swirled, y = r2_swirled)) +
    geom_point() +
    theme_minimal() +
    labs(x = "r1", y = "r2")
}
```

swirl_3d — *Swirl the values around an origin in 3 dimensions*

Description

[Experimental]

The values are swirled counterclockwise around a specified origin. The swirling is done by rotating around the origin, basing the degrees for each rotation-axis on the distances to the origin as so:

$$x_degrees = scale_fn(distances)/(2 * x_radius) * 360$$

The origin can be supplied as coordinates or as a function that returns coordinates. The latter can be useful when supplying a grouped `data.frame` and swirling around e.g. the centroid of each group.

Usage

```
swirl_3d(
  data,
  x_col,
  y_col,
  z_col,
  x_radius = 0,
  y_radius = 0,
  z_radius = 0,
  suffix = "_swirled",
  origin = NULL,
  origin_fn = NULL,
  scale_fn = identity,
  keep_original = TRUE,
  degrees_col_name = ".degrees",
  radius_col_name = ".radius",
  origin_col_name = ".origin",
  overwrite = FALSE
)
```

Arguments

`data` `data.frame` or `vector`.

`x_col, y_col, z_col` Name of x/y/z column in `` `data` ``. All must be specified.

`x_radius, y_radius, z_radius` Radiuses of the most-inner swirls around each axis (in the *simplest* case). Can be `vectors` with multiple radiuses.

E.g. the `` `x_radius` `` specifies the radius when rotating *around* the x-axis, not the radius *on* the x-axis.

Note: With a custom `` `scaling_fn` ``, these might not be the actual swirl radiuses anymore. Think of them more as width settings where a larger number leads to fewer full rotations.

`suffix` Suffix to add to the names of the generated columns.

Use an empty string (i.e. `""`) to overwrite the original columns.

`origin` Coordinates of the origin to swirl around. `Vector` with 3 elements (i.e. origin_x, origin_y, origin_z). Ignored when `` `origin_fn` `` is not `NULL`.

`origin_fn` Function for finding the origin coordinates.

Input: Each column will be passed as a `vector` in the order of `` `cols` ``.

Output: A `vector` with one scalar per dimension.

Can be created with `create_origin_fn()` if you want to apply the same function to each dimension.

E.g. `` `create_origin_fn(median)` `` would find the median of each column.

Built-in functions are `centroid()`, `most_centered()`, and `midrange()`

`scale_fn` Function for scaling the distances before calculating the degrees.

Input: A `numeric vector` (the distances).

Output: A `numeric vector` (the scaled distances) of the same length.

E.g.:

```
function(d){
  d ^ 1.5
}
```

`keep_original` Whether to keep the original columns. (Logical)

Some columns may have been overwritten, in which case only the newest versions are returned.

`degrees_col_name`
: Name of new column with the degrees. If `NULL`, no column is added.

  Also adds a string version with the same name + `"_str"`, making it easier to group by the degrees when plotting multiple rotations.

`radius_col_name`
: Name of new column with the radiuses. If `NULL`, no column is added.

`origin_col_name`
: Name of new column with the origin coordinates. If `NULL`, no column is added.

`overwrite`
: Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with new columns containing the swirled x- and y-values, the degrees, the radiuses, and the origin coordinates.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other mutate functions: `apply_transformation_matrix()`, `cluster_groups()`, `dim_values()`, `expand_distances()`, `expand_distances_each()`, `flip_values()`, `roll_values()`, `rotate_2d()`, `rotate_3d()`, `shear_2d()`, `shear_3d()`, `swirl_2d()`

Other rotation functions: `rotate_2d()`, `rotate_3d()`, `swirl_2d()`

Other distance functions: `closest_to()`, `dim_values()`, `distance()`, `expand_distances()`, `expand_distances_each()`, `furthest_from()`, `swirl_2d()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(4)

# Create a data frame
df <- data.frame(
  "x" = 1:50,
  "y" = 1:50,
  "z" = 1:50,
  "r1" = runif(50),
  "r2" = runif(50) * 35,
  "o" = 1,
  "g" = rep(1:5, each = 10)
)

# Swirl values around (0, 0, 0)
```

```
swirl_3d(
  data = df,
  x_radius = 45,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  origin = c(0, 0, 0)
)

# Swirl around the centroid
df_swirled <- swirl_3d(
  data = df,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  x_radius = c(100, 0, 0),
  y_radius = c(0, 100, 0),
  z_radius = c(0, 0, 100),
  origin_fn = centroid
)

df_swirled

# Plot swirls
if (has_ggplot){
 ggplot(df_swirled, aes(x = x_swirled, y = y_swirled, color = .radius_str, alpha = z_swirled)) +
    geom_vline(xintercept = mean(df$x), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_hline(yintercept = mean(df$y), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_path(alpha = .4) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "radius", alpha = "z (opacity)")
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_swirled$x_swirled,
  y = df_swirled$y_swirled,
  z = df_swirled$z_swirled,
  type = "scatter3d",
  mode = "markers",
  color = df_swirled$.radius_str
)

## End(Not run)

# Swirl around the centroid
df_swirled <- swirl_3d(
  data = df,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  x_radius = c(50, 0, 0),
  y_radius = c(0, 50, 0),
  z_radius = c(0, 0, 50),
  origin_fn = centroid
)

df_swirled

# Plot swirls
if (has_ggplot){
 ggplot(df_swirled, aes(x = x_swirled, y = y_swirled, color = .radius_str, alpha = z_swirled)) +
    geom_vline(xintercept = mean(df$x), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_hline(yintercept = mean(df$y), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_path(alpha = .4) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "radius", alpha = "z (opacity)")
}

## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_swirled$x_swirled,
  y = df_swirled$y_swirled,
  z = df_swirled$z_swirled,
  type = "scatter3d",
  mode = "markers",
  color = df_swirled$.radius_str
)

## End(Not run)


df_swirled <- swirl_3d(
  data = df,
  x_col = "x",
  y_col = "y",
  z_col = "z",
  x_radius = c(25, 50, 25, 25),
  y_radius = c(50, 75, 100, 25),
  z_radius = c(75, 25, 25, 25),
  origin_fn = centroid,
  scale_fn = function(x) {
    x^0.81
  }
)

# Plot swirls
if (has_ggplot){
 ggplot(df_swirled, aes(x = x_swirled, y = y_swirled, color = .radius_str, alpha = z_swirled)) +
    geom_vline(xintercept = mean(df$x), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_hline(yintercept = mean(df$y), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_path(alpha = .4) +
    geom_point() +
    theme_minimal() +
    labs(x = "x", y = "y", color = "radius", alpha = "z (opacity)")
}


## Not run:
# Plot 3d with plotly
plotly::plot_ly(
  x = df_swirled$x_swirled,
  y = df_swirled$y_swirled,
  z = df_swirled$z_swirled,
  type = "scatter3d",
  mode = "markers",
  color = df_swirled$.radius_str
)

## End(Not run)


#
# Swirl random data
# The trick lies in finding the right radiuses
#

# Swirl the random columns
df_swirled <- swirl_3d(
  data = df,
  x_col = "r1",
  y_col = "r2",
  z_col = "o",
  x_radius = c(0, 0, 0, 0),
  y_radius = c(0, 30, 60, 90),
  z_radius = c(10, 10, 10, 10),
  origin_fn = centroid
)

# Not let's rotate it every 10 degrees
df_rotated <- df_swirled %>%
  rotate_3d(
    x_col = "r1_swirled",
    y_col = "r2_swirled",
    z_col = "o_swirled",
    x_deg = rep(0, 36),
    y_deg = rep(0, 36),
    z_deg = (1:36) * 10,
    suffix = "",
    origin = df_swirled$.origin[[1]],
    overwrite = TRUE
  )

# Plot rotated swirls
if (has_ggplot){
  ggplot(
    df_rotated,
    aes(
      x = r1_swirled,
      y = r2_swirled,
      color = .degrees_str,
      alpha = o_swirled
    )
  ) +
    geom_vline(xintercept = mean(df$r1), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_hline(yintercept = mean(df$r2), size = 0.2, alpha = .4, linetype = "dashed") +
    geom_point(show.legend = FALSE) +
    theme_minimal() +
    labs(x = "r1", y = "r2", color = "radius", alpha = "o (opacity)")
}
```

to_unit_length *Scale to unit length*

Description

[Experimental]

Scales the vectors to unit length *row-wise* or *column-wise*.

The `*_vec()` version take and return a `vector`.

Usage

```
to_unit_length(
  data,
  cols = NULL,
  by_row = is.data.frame(data),
  suffix = ifelse(isTRUE(by_row), "_row_unit", "_col_unit"),
  overwrite = FALSE
)

to_unit_length_vec(data)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Names of columns in `` `data` `` to scale.
`by_row`	Whether to scale row `vectors` instead of column `vectors`. (Logical) Note: Disable when `` `data` `` is a `vector`.
`suffix`	Suffix to add to the names of the generated columns. Use an empty string (i.e. `""`) to overwrite the original columns.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

Scaled `vector` or `data.frame` (`tibble`) with the scaled columns.

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other scaling functions: `min_max_scale()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)

# Scale row-wise
to_unit_length(df, cols = c("x", "y"), by_row = TRUE)

# Scale column-wise
to_unit_length(df, cols = c("x", "y"), by_row = FALSE)

# Overwrite columns
to_unit_length(df, cols = c("x", "y"), suffix = "", overwrite = TRUE)

# By groups in 'g'
df %>%
  dplyr::group_by(g) %>%
  to_unit_length(cols = c("x", "y"), by_row = FALSE)

# Scale a vector
to_unit_length_vec(c(1:10))
to_unit_length(c(1:10), suffix = "", overwrite = TRUE)
vector_length(to_unit_length_vec(c(1:10)))
```

`transfer_centroids` *Transfer centroids from one data frame to another*

Description

[Experimental]

Given two `data.frames` with the same columns (and groupings), transfer the centroids from one to the other.

This is commonly used to restore the centroids after transforming the columns.

Usage

```
transfer_centroids(to_data, from_data, cols, group_cols = NULL)
```

Arguments

`to_data`	`data.frame`. Existing `` `dplyr` `` groups are ignored. Specify in `` `group_cols` `` instead.
`from_data`	`data.frame` with the same columns (and groupings) as `` `to_data` ``. Existing `` `dplyr` `` groups are ignored. Specify in `` `group_cols` `` instead.
`cols`	Names of numeric columns to transfer centroids to. Must exist in both `` `to_data` `` and `` `from_data` ``.
`group_cols`	Names of grouping columns.

Value

The `` `to_data` `` `data.frame` (`tibble`) with the centroids from the `` `from_data` `` `data.frame`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other clustering functions: `cluster_groups()`, `generate_clusters()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)

# Create another data frame with different x and y values
df2 <- df
df2$x <- runif(20)
df2$y <- runif(20)

# Check centroids before transfer

df %>%
  dplyr::group_by(g) %>%
  dplyr::summarize_all(mean)

df2 %>%
  dplyr::group_by(g) %>%
  dplyr::summarize_all(mean)

# Now let's transfer the centroids from df to df2

df3 <- transfer_centroids(
  to_data = df2,
  from_data = df,
  cols = c("x", "y"),
  group_cols = "g"
)

# Check that the transfer gave us the same centroids as df
df3 %>%
  dplyr::group_by(g) %>%
  dplyr::summarize_all(mean)
```

`Transformation` *Transformation*

Description

[Experimental]

Container for the type of transformation used in `Pipeline`.

Note: For internal use.

Public fields

`name` Name of transformation.

`fn` Transformation function.

`args` `list` of arguments for `` `fn` ``.

`group_cols` Names of columns to group `data.frame` by before applying `` `fn` ``.
When `` `NULL` ``, the `data.frame` is not grouped.

Methods

Public methods:

- `Transformation$new()`
- `Transformation$apply()`
- `Transformation$print()`
- `Transformation$clone()`

Method `new()`**:** Initialize transformation.

Usage:

```
Transformation$new(fn, args, name = NULL, group_cols = NULL)
```

Arguments:

`fn` Transformation function.

`args` `list` of arguments for `` `fn` ``.

`name` Name of transformation.

`group_cols` Names of columns to group `data.frame` by before applying `` `fn` ``.
When `` `NULL` ``, the `data.frame` is not grouped.

Method `apply()`**:** Apply the transformation to a `data.frame`.

Usage:

```
Transformation$apply(data)
```

Arguments:

`data` `data.frame`.
A grouped `data.frame` will first be ungrouped. If `` `group_cols` `` is specified, it will then be grouped by those columns.

Returns: Transformed version of `` `data` ``.

Method `print()`**:** Print an overview of the transformation.

Usage:

```
Transformation$print(..., indent = 0, show_class = TRUE)
```

Arguments:

`...` further arguments passed to or from other methods.

`indent` How many spaces to indent when printing.

`show_class` Whether to print the transformation class name.

Returns: The pipeline. To allow chaining of methods.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Transformation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other transformation classes: `FixedGroupsTransformation`, `GeneratedTransformation`

`triangularize` *Create x-coordinates so the points form a triangle*

Description

[Experimental]

Create the x-coordinates for a `vector` of y-coordinates such that they form a triangle.

The data points are stochastically distributed based on edge lengths, why it might be preferable to set a random seed.

This will likely look most like a triangle when the y-coordinates are somewhat equally distributed, e.g. a uniform distribution.

Usage

```
triangularize(
  data,
  y_col = NULL,
  .min = NULL,
  .max = NULL,
  offset_x = 0,
  keep_original = TRUE,
  x_col_name = ".triangle_x",
  edge_col_name = ".edge",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`y_col`	Name of column in `` `data` `` with y-coordinates to create x-coordinates for.
`.min`	Minimum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`.max`	Maximum y-coordinate. If `NULL`, it is inferred by the given y-coordinates.
`offset_x`	Value to offset the x-coordinates by.
`keep_original`	Whether to keep the original columns. (Logical) Some columns may have been overwritten, in which case only the newest versions are returned.
`x_col_name`	Name of new column with the x-coordinates.

`edge_col_name`	Name of new column with the edge identifiers. If `NULL`, no column is added. Numbering is clockwise and starts at the upper-right edge.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

`data.frame` (`tibble`) with the added x-coordinates and an identifier for the edge the data point is a part of.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other forming functions: `circularize()`, `hexagonalize()`, `square()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)
library(purrr)
has_ggplot <- require(ggplot2)  # Attach if installed

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "y" = runif(200),
  "g" = factor(rep(1:5, each = 40))
)

# Triangularize 'y'
df_tri <- triangularize(df, y_col = "y")
df_tri

# Plot triangle
if (has_ggplot){
  df_tri %>%
    ggplot(aes(x = .triangle_x, y = y, color = .edge)) +
    geom_point() +
    theme_minimal()
}

#
# Grouped squaring
#

# Triangularize 'y' for each group
# First cluster the groups a bit to move the
# triangles away from each other
df_tri <- df %>%
  cluster_groups(
    cols = "y",
    group_cols = "g",
    suffix = "",
    overwrite = TRUE
  ) %>%
  dplyr::group_by(g) %>%
  triangularize(
    y_col = "y",
    overwrite = TRUE
  )

# Plot triangles
if (has_ggplot){
  df_tri %>%
    ggplot(aes(x = .triangle_x, y = y, color = g)) +
    geom_point() +
    theme_minimal()
}

#
# Specifying minimum value
#

# Specify minimum value manually
df_tri <- triangularize(df, y_col = "y", .min = -2)
df_tri

# Plot triangle
if (has_ggplot){
  df_tri %>%
    ggplot(aes(x = .triangle_x, y = y, color = .edge)) +
    geom_point() +
    theme_minimal()
}

#
# Multiple triangles by contraction
#


# Start by squaring 'y'
df_tri <- triangularize(df, y_col = "y")

# Contract '.triangle_x' and 'y' towards the centroid
# To contract with multiple multipliers at once,
# we wrap the call in purrr::map_dfr
df_expanded <- purrr::map_dfr(
  .x = 1:10 / 10,
  .f = function(mult) {
    expand_distances(
      data = df_tri,
      cols = c(".triangle_x", "y"),
      multiplier = mult,
      origin_fn = centroid,
      overwrite = TRUE
    )
  }
)
df_expanded

if (has_ggplot){
  df_expanded %>%
    ggplot(aes(
      x = .triangle_x_expanded, y = y_expanded,
      color = .edge, alpha = .multiplier
    )) +
    geom_point() +
    theme_minimal()
}
```

`triplet_extremes` *Makes triplets of extreme values and sort by them*

Description

[Experimental]

The values are grouped in three such that the first group is formed by the lowest and highest values and the value closest to the median, the second group is formed by the second lowest and second highest values and the value second closest to the median, and so on. The values are then sorted by these groups and their actual value.

When the number of rows/elements in `` `data` `` is not evenly divisible by three, the `` `unequal_method_1` `` (single excessive element) and `` `unequal_method_2` `` (two excessive elements) determines which element(s) should form a smaller group. This group will be the first group *in a given grouping* (see `` `num_groupings` ``) with the identifier `1`.

The `*_vec()` version takes and returns a `vector`.

Example:

The column values:

`c(1, 2, 3, 4, 5, 6)`

Are sorted in triplets as:

`c(1, 3, 6, 2, 4, 5)`

Usage

```
triplet_extremes(
  data,
  col = NULL,
  middle_is = "middle",
  unequal_method_1 = "middle",
  unequal_method_2 = c("middle", "middle"),
  num_groupings = 1,
  balance = "mean",
  order_by_aggregates = FALSE,
  shuffle_members = FALSE,
  shuffle_triplets = FALSE,
  factor_name = ifelse(num_groupings == 1, ".triplet", ".tripleting"),
  overwrite = FALSE
)

triplet_extremes_vec(
  data,
  middle_is = "middle",
  unequal_method_1 = "middle",
  unequal_method_2 = c("middle", "middle"),
  num_groupings = 1,
  balance = "mean",
  order_by_aggregates = FALSE,
  shuffle_members = FALSE,
  shuffle_triplets = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`col`	Column to create sorting factor by. When `` `NULL` `` and `` `data` `` is a `data.frame`, the row numbers are used.

`middle_is`	Whether the middle element in the triplet is the nth closest element to the median value or the nth+1 lowest/highest value.
	One of: `middle` (default), `min`, or `max`.
	Triplet grouping is performed greedily from the most extreme values to the least extreme values. E.g. `c(1, 6, 12)` is created before `c(2, 5, 11)` which is made before `c(3, 7, 10)`.
	Examples:
	When `` `middle_is` `` == `'middle'`, a `1:12` sequence is grouped into:
	`c( c(1, 6, 12), c(2, 7, 11), c(3, 5, 10), c(4, 8, 9) )`
	When `` `middle_is` `` == `'min'`, a `1:12` sequence is grouped into:
	`c( c(1, 2, 12), c(3, 4, 11), c(5, 6, 10), c(7, 8, 9) )`
	When `` `middle_is` `` == `'max'`, a `1:12` sequence is grouped into:
	`c( c(1, 11, 12), c(2, 9, 10), c(3, 7, 8), c(4, 5, 6) )`

`unequal_method_1`, `unequal_method_2`
: Method for dealing with either a single excessive element (`` `unequal_method_1` ``) or two excessive elements (`` `unequal_method_2` ``) when the number of rows/elements in `` `data` `` are not evenly divisible by three.

  `` `unequal_method_1` ``: One of: `min`, `middle` or `max`.

  `` `unequal_method_2` ``: Vector with two of: `min`, `middle` or `max`. Can be the same value twice.

  Note: The excessive element(s) are extracted before triplet grouping. These elements are put in their own group and given group identifier `1`.

  E.g. When `` `unequal_method_2` `` is `c("middle", "middle")` the two elements closest to the median are extracted.

`num_groupings`
: Number of times to group into triplets (recursively). At least `1`.

  Based on `` `balance` ``, the secondary groupings perform extreme triplet grouping on either the *sum*, *absolute difference*, *min*, or *max* of the triplet elements.

`balance`
: What to balance triplets for in a given *secondary* triplet grouping. Either `"mean"`, `"spread"`, `"min"`, or `"max"`. Can be a single string used for all secondary groupings or one for each secondary grouping (`` `num_groupings` - 1 ``).

  The first triplet grouping always groups the actual element values.

  mean: Triplets have similar means. The values in the triplets from the previous grouping are aggregated with `` `sum()` `` and extreme triplet grouped.

  spread: Triplets have similar spread (e.g. standard deviations). The values in the triplets from the previous triplet grouping are aggregated with `` `sum(abs(diff()))` `` and extreme triplet grouped.

  min / max: Triplets have similar minimum / maximum values. The values in the triplets from the previous triplet grouping are aggregated with `` `min()` `` / `` `max()` `` and extreme triplet grouped.

`order_by_aggregates`
: Whether to order the groups from initial groupings (first `` `num_groupings` - 1 ``) by their aggregate values instead of their group identifiers.

  N.B. Only used when `` `num_groupings` > 1 ``.

`shuffle_members`
: Whether to shuffle the order of the group members within the groups. (Logical)

`shuffle_triplets`
: Whether to shuffle the order of the triplets. Triplet members remain together. (Logical)

`factor_name`
: Name of new column with the sorting factor. If `` `NULL` ``, no column is added.

`overwrite`
: Whether to allow overwriting of existing columns. (Logical)

Value

The sorted `data.frame` (`tibble`) / `vector`. Optionally with the sorting factor added.

When `` `data` `` is a `vector` and `` `keep_factors` `` is `` `FALSE` ``, the output will be a `vector`. Otherwise, a `data.frame`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other rearrange functions: center_max(), center_min(), closest_to(), furthest_from(), pair_extremes(), position_max(), position_min(), rev_windows(), roll_elements(), shuffle_hierarchy()

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "index" = 1:12,
  "A" = sample(1:12),
  "B" = runif(12),
  "C" = LETTERS[1:12],
  "G" = c(
    1, 1, 1, 1, 2, 2,
    2, 2, 3, 3, 3, 3
  ),
  stringsAsFactors = FALSE
)

# Triplet group extreme indices (row numbers)
triplet_extremes(df)

# Triplet group extremes in each of the columns
triplet_extremes(df, col = "A")$A
triplet_extremes(df, col = "B")$B
triplet_extremes(df, col = "C")$C

# Shuffle the members triplet-wise
# The triplets maintain their order
# but the rows within each triplet are shuffled
triplet_extremes(df, col = "A", shuffle_members = TRUE)

# Shuffle the order of the triplets
# The triplets are shuffled but
# the rows within each triplet maintain their order
triplet_extremes(df, col = "A", shuffle_triplets = TRUE)

# Use recursive grouping
# Mostly meaningful with much larger datasets
# Order initial grouping by group identifiers
triplet_extremes(df, col = "A", num_groupings = 2)
```

```
# Order initial grouping by aggregate values
triplet_extremes(df, col = "A", num_groupings = 2, order_by_aggregates = TRUE)

# Grouped by G
# Each G group only has 4 elements
# so it only creates 1 triplet and a group
# with the single excessive element
# per G group
df %>%
  dplyr::select(G, A) %>% # For clarity
  dplyr::group_by(G) %>%
  triplet_extremes(col = "A")

# Plot the extreme triplets
plot(
  x = 1:12,
  y = triplet_extremes(df, col = "A")$A,
  col = as.character(rep(1:4, each = 3))
)
# With shuffled triplet members (run a few times)
plot(
  x = 1:12,
  y = triplet_extremes(df, col = "A", shuffle_members = TRUE)$A,
  col = as.character(rep(1:4, each = 3))
)
# With shuffled triplets (run a few times)
plot(
  x = rep(1:6, each = 2),
  y = triplet_extremes(df, col = "A", shuffle_triplets = TRUE)$A,
  col = as.character(rep(1:4, each = 3))
)
```

`vector_length` *Calculate vector length(s)*

Description

[Experimental]

Calculates vector lengths/magnitudes *row-* or *column-wise* with

$$sqrt(sum(x^2))$$

Where x is the `vector` to get the length/magnitude of.

Should not be confused with `length()`, which counts the elements.

Usage

```
vector_length(
  data,
  cols = NULL,
  by_row = is.data.frame(data),
  len_col_name = ".vec_len",
  overwrite = FALSE
)
```

Arguments

`data`	`data.frame` or `vector`.
`cols`	Names of columns in `` `data` `` to measure vector length of.
`by_row`	Whether to measure length of row vectors instead of column vectors. (Logical) Note: Disable when `` `data` `` is a `vector`.
`len_col_name`	Name of new column with the row vector lengths when `` `data` `` is a `data.frame` and `` `by_row` `` is `TRUE`.
`overwrite`	Whether to allow overwriting of existing columns. (Logical)

Value

Vector length(s).

When `` `data` `` is a `vector`: `scalar`

When `` `data` `` is a `data.frame` and `` `by_row` `` is `TRUE`: `` `data` `` with an extra column with row vector lengths.

When `` `data` `` is a `data.frame` and `` `by_row` `` is `FALSE`: A `data.frame` with the summarized column vector lengths.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other measuring functions: `angle()`, `distance()`

Examples

```
# Attach packages
library(rearrr)
library(dplyr)

# Set seed
set.seed(1)

# Create a data frame
df <- data.frame(
  "x" = runif(20),
  "y" = runif(20),
  "g" = rep(1:4, each = 5)
)
```

```
# Measure row-wise
vector_length(df, cols = c("x", "y"), by_row = TRUE)

# Measure column-wise
vector_length(df, cols = c("x", "y"), by_row = FALSE)

# By groups in 'g'
df %>%
  dplyr::group_by(g) %>%
  vector_length(cols = c("x", "y"), by_row = FALSE)

# Measure vector length of a vector
vector_length(c(1:10))
```

Index