Package 'distantia'

February 1, 2025

Type Package

Title Advanced Toolset for Efficient Time Series Dissimilarity Analysis

Version 2.0.2

URL https://blasbenito.github.io/distantia/

BugReports https://github.com/BlasBenito/distantia/issues

Description Fast C++ implementation of Dynamic Time Warping for time series dissimilarity analysis, with applications in environmental monitoring and sensor data analysis, climate science, signal processing and pattern recognition, and financial data analysis. Built upon the ideas presented in Benito and Birks (2020) <doi:10.1111/ecog.04895>, provides tools for analyzing time series of varying lengths and structures, including irregular multivariate time series. Key features include individual variable contribution analysis, restricted permutation tests for statistical significance, and imputation of missing data via GAMs. Additionally, the package provides an ample set of tools to prepare and manage time series data.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Depends R (>= 4.1), doFuture

Imports Rcpp, zoo, foreach, future.apply, lubridate, progressr

Suggests roxyglobals, spelling, sf, lwgeom, testthat (>= 3.0.0)

Config/roxyglobals/filename globals.R

Config/roxyglobals/unique FALSE

Config/testthat/edition 3

LinkingTo Rcpp

Language en-US

NeedsCompilation yes

Author Blas M. Benito [aut, cre, cph]
(<https://orcid.org/0000-0001-5105-7232>)

Maintainer Blas M. Benito <blasbenito@gmail.com>

Repository CRAN

Date/Publication 2025-02-01 19:50:02 UTC

Contents

`albatross` *Flight Path Time Series of Albatrosses in The Pacific*

Description

Daily mean flight path data of 4 individuals of Waved Albatross (Phoebastria irrorata) captured via GPS during the summer of 2008. Sf data frame with columns name, time, latitude, longitude, ground speed, heading, and (uncalibrated) temperature.

The full dataset at hourly resolution can be downloaded from https://github.com/BlasBenito/distantia/blob/main/data_full/albatross.rda (use the "Download raw file" button).

Usage

```
data(albatross)
```

Format

data frame

References

doi:10.5441/001/1.3hp3s250

See Also

Other example_data: cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

Examples

```
#load as tsl
#scale al variables
#aggregate to daily resolution
#align all time series to same temporal span
tsl <- tsl_initialize(
  x = albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_local
  ) |>
  tsl_aggregate(
    new_time = "days"
  )

if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 5
    )
}
```

---

auto_distance_cpp *(C++) Sum Distances Between Consecutive Samples in a Time Series*

Description

Computes the cumulative sum of distances between consecutive samples in a univariate or multivariate time series. NA values should be removed before using this function.

Usage

```
auto_distance_cpp(x, distance = "euclidean")
```

Arguments

x	(required, numeric matrix) univariate or multivariate time series.
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean"

Value

numeric

See Also

Other Rcpp_auto_sum: auto_sum_cpp(), auto_sum_full_cpp(), auto_sum_path_cpp(), subset_matrix_by_rows_cpp(

Examples

```
#simulate a time series
x <- zoo_simulate()

#compute auto distance
auto_distance_cpp(
  x = x,
  distance = "euclidean"
  )
```

---

auto_sum_cpp *(C++) Sum Distances Between Consecutive Samples in Two Time Series*

Description

Sum of auto-distances of two time series. This function switches between auto_sum_full_cpp() and auto_sum_path_cpp() depending on the value of the argument ignore_blocks.

Usage

```
auto_sum_cpp(x, y, path, distance = "euclidean", ignore_blocks = FALSE)
```

Arguments

x	(required, numeric matrix) of same number of columns as 'y'.
y	(required, numeric matrix) of same number of columns as 'x'.
path	(required, data frame) output of cost_path_orthogonal_cpp().
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean"
ignore_blocks	(optional, logical). If TRUE, blocks of consecutive path coordinates are trimmed to avoid inflating the psi distance. Default: FALSE.

Value

numeric

See Also

Other Rcpp_auto_sum: auto_distance_cpp(), auto_sum_full_cpp(), auto_sum_path_cpp(), subset_matrix_by_rows_cpp()

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_orthogonal_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

nrow(cost_path)

#remove blocks from least-cost path
cost_path_trimmed <- cost_path_trim_cpp(
  path = cost_path
)

nrow(cost_path_trimmed)

#auto sum
auto_sum_cpp(
  x = x,
  y = y,
  path = cost_path_trimmed,
  distance = "euclidean",
  ignore_blocks = FALSE
)
```

`auto_sum_full_cpp` *(C++) Sum Distances Between All Consecutive Samples in Two Time Series*

Description

Computes the cumulative auto sum of autodistances of two time series. The output value is used as normalization factor when computing dissimilarity scores.

Usage

```
auto_sum_full_cpp(x, y, distance = "euclidean")
```

Arguments

`x`	(required, numeric matrix) univariate or multivariate time series.
`y`	(required, numeric matrix) univariate or multivariate time series with the same number of columns as 'x'.
`distance`	(optional, character string) distance name from the "names" column of the dataset `distances` (see `distances$name`). Default: "euclidean"

Value

numeric

See Also

Other Rcpp_auto_sum: `auto_distance_cpp()`, `auto_sum_cpp()`, `auto_sum_path_cpp()`, `subset_matrix_by_rows_cpp(`

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#auto sum
auto_sum_full_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)
```

auto_sum_path_cpp *(C++) Sum Distances Between All Consecutive Samples in the Least Cost Path Between Two Time Series*

Description

Computes the cumulative auto sum of auto-distances of two time series for the coordinates of a trimmed least cost path. The output value is used as normalization factor when computing dissimilarity scores.

Usage

```
auto_sum_path_cpp(x, y, path, distance = "euclidean")
```

Arguments

x	(required, numeric matrix) univariate or multivariate time series.
y	(required, numeric matrix) univariate or multivariate time series with the same number of columns as 'x'.
path	(required, data frame) least-cost path produced by cost_path_orthogonal_cpp(). Default: NULL
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".

Value

numeric

See Also

Other Rcpp_auto_sum: auto_distance_cpp(), auto_sum_cpp(), auto_sum_full_cpp(), subset_matrix_by_rows_cpp(

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
```

```
)

#least cost path
cost_path <- cost_path_orthogonal_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

nrow(cost_path)

#remove blocks from least-cost path
cost_path_trimmed <- cost_path_trim_cpp(
  path = cost_path
)

nrow(cost_path_trimmed)

#auto sum
auto_sum_path_cpp(
  x = x,
  y = y,
  path = cost_path_trimmed,
  distance = "euclidean"
)
```

`cities_coordinates` *Coordinates of 100 Major Cities*

Description

City coordinates and several environmental variables for the dataset `cities_temperature`.

The full dataset with 100 cities can be downloaded from https://github.com/BlasBenito/distantia/blob/main/data_full/cities_coordinates.rda (use the "Download raw file" button).

Usage

```
data(cities_coordinates)
```

Format

sf data frame with 5 columns and 100 rows.

See Also

Other example_data: albatross, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

Other example_data: albatross, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

---

`cities_temperature` *Long Term Monthly Temperature in 20 Major Cities*

Description

Average temperatures between 1975 and 2010 of 20 major cities of the world. Source.

Site coordinates for this dataset are in cities_coordinates.

The full dataset with 100 cities can be downloaded from https://github.com/BlasBenito/distantia/blob/main/data_full/cities_temperature.rda (use the "Download raw file" button).

Usage

```
data(cities_temperature)
```

Format

data frame with 3 columns and 52100 rows.

See Also

Other example_data: albatross, cities_coordinates, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

Examples

```
data("cities_temperature")

#to time series list
cities <- tsl_initialize(
  x = cities_temperature,
  name_column = "name",
  time_column = "time"
)

#time series plot
if(interactive()){

 #only four cities are shown
 tsl_plot(
  tsl = tsl_subset(
    tsl = tsl,
    names = 1:4
```

```
    ),
  guide = FALSE
  )

}
```

`color_continuous` *Default Continuous Color Palette*

Description

Uses the function `grDevices::hcl.colors()` to generate a continuous color palette.

Usage

```
color_continuous(n = 5, palette = "Zissou 1", rev = FALSE)
```

Arguments

`n`	(required, integer) number of colors to generate. Default = NULL
`palette`	(required, character string) Argument `palette` of `grDevices::hcl.colors()`. Default: "Zissou 1"
`rev`	(optional, logical) If TRUE, the color palette is reversed. Default: FALSE

Value

color vector

See Also

Other internal_plotting: `color_discrete()`, `utils_color_breaks()`, `utils_line_color()`, `utils_line_guide()`, `utils_matrix_guide()`, `utils_matrix_plot()`

Examples

```
color_continuous(n = 20)
```

`color_discrete` *Default Discrete Color Palettes*

Description

Uses the function `grDevices::palette.colors()` to generate discrete color palettes using the following rules:

- `n <= 9`: "Okabe-Ito".
- `n == 10`: "Tableau 10"
- `n > 10 && n <= 12`: "Paired"
- `n > 12 && n <= 26`: "Alphabet"
- `n > 26 && n <= 36`: "Polychrome 36"

Usage

```
color_discrete(n = NULL, rev = FALSE)
```

Arguments

`n`	(required, integer) number of colors to generate. Default = NULL
`rev`	(optional, logical) If TRUE, the color palette is reversed. Default: FALSE

Value

color vector

See Also

Other internal_plotting: `color_continuous()`, `utils_color_breaks()`, `utils_line_color()`, `utils_line_guide()`, `utils_matrix_guide()`, `utils_matrix_plot()`

Examples

```
color_discrete(n = 9)
```

`cost_matrix_diagonal_cpp`

(C++) Compute Orthogonal and Diagonal Least Cost Matrix from a Distance Matrix

---

Description

Computes the least cost matrix from a distance matrix. Considers diagonals during computation of least-costs.

Usage

```
cost_matrix_diagonal_cpp(dist_matrix)
```

Arguments

`dist_matrix`	(required, distance matrix). Square distance matrix, output of `distance_matrix_cpp()`.

Value

Least cost matrix.

See Also

Other Rcpp_matrix: `cost_matrix_diagonal_weighted_cpp()`, `cost_matrix_orthogonal_cpp()`, `distance_ls_cpp()`, `distance_matrix_cpp()`

`cost_matrix_diagonal_weighted_cpp`

(C++) Compute Orthogonal and Weighted Diagonal Least Cost Matrix from a Distance Matrix

---

Description

Computes the least cost matrix from a distance matrix. Weights diagonals by a factor of 1.414214 (square root of 2) with respect to orthogonal paths.

Usage

```
cost_matrix_diagonal_weighted_cpp(dist_matrix)
```

Arguments

`dist_matrix`	(required, distance matrix). Distance matrix.

Value

Least cost matrix.

See Also

Other Rcpp_matrix: cost_matrix_diagonal_cpp(), cost_matrix_orthogonal_cpp(), distance_ls_cpp(), distance_matrix_cpp()

`cost_matrix_orthogonal_cpp` *(C++) Compute Orthogonal Least Cost Matrix from a Distance Matrix*

Description

Computes the least cost matrix from a distance matrix.

Usage

```
cost_matrix_orthogonal_cpp(dist_matrix)
```

Arguments

`dist_matrix`	(required, distance matrix). Output of `distance_matrix_cpp()`.

Value

Least cost matrix.

See Also

Other Rcpp_matrix: cost_matrix_diagonal_cpp(), cost_matrix_diagonal_weighted_cpp(), distance_ls_cpp(), distance_matrix_cpp()

`cost_path_cpp` *Least Cost Path*

Description

Least cost path between two time series `x` and `y`. NA values must be removed from `x` and `y` before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
cost_path_cpp(
  x,
  y,
  distance = "euclidean",
  diagonal = TRUE,
  weighted = TRUE,
  ignore_blocks = FALSE,
  bandwidth = 1
)
```

Arguments

x	(required, numeric matrix) multivariate time series.
y	(required, numeric matrix) multivariate time series with the same number of columns as 'x'.
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".
diagonal	(optional, logical). If TRUE, diagonals are included in the computation of the cost matrix. Default: TRUE.
weighted	(optional, logical). Only relevant when diagonal is TRUE. When TRUE, diagonal cost is weighted by y factor of 1.414214 (square root of 2). Default: TRUE.
ignore_blocks	(optional, logical). If TRUE, blocks of consecutive path coordinates are trimmed to avoid inflating the psi distance. Default: FALSE.
bandwidth	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default. Default: 1

Value

data frame

See Also

Other Rcpp_cost_path: cost_path_diagonal_bandwidth_cpp(), cost_path_diagonal_cpp(), cost_path_orthogonal_bandwidth_cpp(), cost_path_orthogonal_cpp(), cost_path_slotting_cpp(), cost_path_sum_cpp(), cost_path_trim_cpp()

---

cost_path_diagonal_bandwidth_cpp

(C++) Orthogonal and Diagonal Least Cost Path Restricted by Sakoe-Chiba band

---

Description

Computes the least cost matrix from a distance matrix. Considers diagonals during computation of least-costs. In case of ties, diagonals are favored.

Usage

```
cost_path_diagonal_bandwidth_cpp(dist_matrix, cost_matrix, bandwidth = 1)
```

Arguments

dist_matrix	(required, numeric matrix). Distance matrix between two time series.
cost_matrix	(required, numeric matrix). Cost matrix generated from dist_matrix.
bandwidth	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default. Default: 1

Value

data frame

See Also

Other Rcpp_cost_path: cost_path_cpp(), cost_path_diagonal_cpp(), cost_path_orthogonal_bandwidth_cpp(), cost_path_orthogonal_cpp(), cost_path_slotting_cpp(), cost_path_sum_cpp(), cost_path_trim_cpp()

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_diagonal_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

cost_path
```

cost_path_diagonal_cpp *(C++) Orthogonal and Diagonal Least Cost Path*

Description

Computes the least cost matrix from a distance matrix. Considers diagonals during computation of least-costs. In case of ties, diagonals are favored.

Usage

```
cost_path_diagonal_cpp(dist_matrix, cost_matrix)
```

Arguments

dist_matrix	(required, numeric matrix). Distance matrix between two time series.
cost_matrix	(required, numeric matrix). Cost matrix generated from dist_matrix.

Value

data frame

See Also

Other Rcpp_cost_path: cost_path_cpp(), cost_path_diagonal_bandwidth_cpp(), cost_path_orthogonal_bandwidth
cost_path_orthogonal_cpp(), cost_path_slotting_cpp(), cost_path_sum_cpp(), cost_path_trim_cpp()

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_diagonal_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
```

```
)

cost_path
```

`cost_path_orthogonal_bandwidth_cpp` *(C++) Orthogonal Least Cost Path*

Description

Computes an orthogonal least-cost path within a cost matrix. Each steps within the least-cost path either moves in the x or the y direction, but never diagonally.

Usage

```
cost_path_orthogonal_bandwidth_cpp(dist_matrix, cost_matrix, bandwidth = 1)
```

Arguments

`dist_matrix`	(required, numeric matrix). Distance matrix between two time series.
`cost_matrix`	(required, numeric matrix). Cost matrix generated from `dist_matrix`.
`bandwidth`	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default. Default: 1

Value

data frame

See Also

Other Rcpp_cost_path: `cost_path_cpp()`, `cost_path_diagonal_bandwidth_cpp()`, `cost_path_diagonal_cpp()`, `cost_path_orthogonal_cpp()`, `cost_path_slotting_cpp()`, `cost_path_sum_cpp()`, `cost_path_trim_cpp()`

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
```

```
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_orthogonal_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

cost_path
```

`cost_path_orthogonal_cpp` *(C++) Orthogonal Least Cost Path*

Description

Computes an orthogonal least-cost path within a cost matrix. Each steps within the least-cost path either moves in the x or the y direction, but never diagonally.

Usage

```
cost_path_orthogonal_cpp(dist_matrix, cost_matrix)
```

Arguments

`dist_matrix`	(required, numeric matrix). Distance matrix between two time series.
`cost_matrix`	(required, numeric matrix). Cost matrix generated from `dist_matrix`.

Value

data frame

See Also

Other Rcpp_cost_path: `cost_path_cpp()`, `cost_path_diagonal_bandwidth_cpp()`, `cost_path_diagonal_cpp()`, `cost_path_orthogonal_bandwidth_cpp()`, `cost_path_slotting_cpp()`, `cost_path_sum_cpp()`, `cost_path_trim_cpp()`

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
```

```
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_orthogonal_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

cost_path
```

`cost_path_slotting_cpp` *(C++) Least Cost Path for Sequence Slotting*

Description

Computes a least-cost matrix from a distance matrix. This version differs from `cost_path_orthogonal_cpp()` in the way it solves ties. In the case of a tie, `cost_path_orthogonal_cpp()` uses the first neighbor satisfying the minimum distance condition, while this function selects the neighbor that changes the axis of movement within the least-cost matrix. This function is not used anywhere within the package, but was left here for future reference.

Usage

```
cost_path_slotting_cpp(dist_matrix, cost_matrix)
```

Arguments

`dist_matrix`	(required, numeric matrix). Distance matrix between two time series.
`cost_matrix`	(required, numeric matrix). Least-cost matrix generated from `dist_matrix`.

Value

data frame

See Also

Other Rcpp_cost_path: `cost_path_cpp()`, `cost_path_diagonal_bandwidth_cpp()`, `cost_path_diagonal_cpp()`, `cost_path_orthogonal_bandwidth_cpp()`, `cost_path_orthogonal_cpp()`, `cost_path_sum_cpp()`, `cost_path_trim_cpp()`

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_slotting_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

cost_path
```

---

cost_path_sum_cpp *(C++) Sum Distances in a Least Cost Path*

Description

(C++) Sum Distances in a Least Cost Path

Usage

```
cost_path_sum_cpp(path)
```

Arguments

path	(required, data frame) least-cost path produced by cost_path_orthogonal_cpp().

Value

numeric

See Also

Other Rcpp_cost_path: cost_path_cpp(), cost_path_diagonal_bandwidth_cpp(), cost_path_diagonal_cpp(), cost_path_orthogonal_bandwidth_cpp(), cost_path_orthogonal_cpp(), cost_path_slotting_cpp(), cost_path_trim_cpp()

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_slotting_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

cost_path_sum_cpp(
  path = cost_path
  )
```

---

cost_path_trim_cpp *(C++) Remove Blocks from a Least Cost Path*

Description

(C++) Remove Blocks from a Least Cost Path

Usage

```
cost_path_trim_cpp(path)
```

Arguments

path	(required, data frame) least-cost path produced by cost_path_orthogonal_cpp().

Value

data frame

See Also

Other Rcpp_cost_path: cost_path_cpp(), cost_path_diagonal_bandwidth_cpp(), cost_path_diagonal_cpp(), cost_path_orthogonal_bandwidth_cpp(), cost_path_orthogonal_cpp(), cost_path_slotting_cpp(), cost_path_sum_cpp()

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

#least cost matrix
cost_matrix <- cost_matrix_orthogonal_cpp(
  dist_matrix = dist_matrix
)

#least cost path
cost_path <- cost_path_slotting_cpp(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

nrow(cost_path)

#remove blocks from least-cost path
cost_path_trimmed <- cost_path_trim_cpp(
  path = cost_path
)

nrow(cost_path_trimmed)
```

---

covid_counties *County Coordinates of the Covid Prevalence Dataset*

Description

County Coordinates of the Covid Prevalence Dataset

Usage

```
data(covid_counties)
```

Format

sf data frame with county polygons and census data.

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

---

covid_prevalence *Time Series of Covid Prevalence in California Counties*

Description

Dataset with Covid19 maximum weekly prevalence in California counties between 2020 and 2024, from healthdata.gov.

Usage

```
data(covid_prevalence)
```

Format

data frame with 3 columns and 51048 rows

Details

County polygons and additional data for this dataset are in covid_counties.

The full dataset at daily resolution can be downloaded from https://github.com/BlasBenito/distantia/blob/main/data_full/covid_prevalence.rda (use the "Download raw file" button).

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

Examples

```
#to time series list
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
)

#time series plot
```

```
if(interactive()){

 #subset to avoid margin errors
 tsl_plot(
  tsl = tsl_subset(
    tsl = tsl,
    names = 1:4
    ),
  guide = FALSE
  )

}
```

`distance` *Distance Between Two Numeric Vectors*

Description

Computes the distance between two numeric vectors with a distance metric included in the data frame `distantia::distances`.

Usage

```
distance(x = NULL, y = NULL, distance = "euclidean")
```

Arguments

`x`	(required, numeric vector).
`y`	(required, numeric vector) of same length as `x`.
`distance`	(optional, character string) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset `distances`. Default: "euclidean".

Value

numeric value

See Also

Other distances: `distance_matrix()`, `distances`

Examples

```
distance(
  x = runif(100),
  y = runif(100),
  distance = "euclidean"
)
```

distances *Distance Methods*

Description

Data frame with the names, abbreviations, and expressions of the distance metrics implemented in the package.

Usage

```
data(distances)
```

Format

data frame with 5 columns and 10 rows

See Also

Other distances: `distance()`, `distance_matrix()`

distance_bray_curtis_cpp *(C++) Bray-Curtis Distance Between Two Vectors*

Description

Computes the Bray-Curtis distance, suitable for species abundance data.

Usage

```
distance_bray_curtis_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as `x`.

Value

numeric

See Also

Other Rcpp_distance_methods: `distance_canberra_cpp()`, `distance_chebyshev_cpp()`, `distance_chi_cpp()`, `distance_cosine_cpp()`, `distance_euclidean_cpp()`, `distance_hamming_cpp()`, `distance_hellinger_cpp()`, `distance_jaccard_cpp()`, `distance_manhattan_cpp()`, `distance_russelrao_cpp()`, `distance_sorensen_cpp()`

Examples

```
distance_bray_curtis_cpp(x = runif(100), y = runif(100))
```

`distance_canberra_cpp` *(C++) Canberra Distance Between Two Binary Vectors*

Description

Computes the Canberra distance between two binary vectors.

Usage

```
distance_canberra_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_chebyshev_cpp(), distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(), distance_hellinger_cpp(), distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_canberra_cpp(c(0, 1, 0, 1), c(1, 1, 0, 0))
```

`distance_chebyshev_cpp`
(C++) Chebyshev Distance Between Two Vectors

Description

Computed as: `max(abs(x - y))`. Cannot handle NA values.

Usage

```
distance_chebyshev_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(), distance_hellinger_cpp(), distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_chebyshev_cpp(x = runif(100), y = runif(100))
```

---

distance_chi_cpp *(C++) Normalized Chi Distance Between Two Vectors*

Description

Computed as: `xy <- x + y y. <- y / sum(y) x. <- x / sum(x) sqrt(sum(((x. - y.)^2) / (xy / sum(xy))))`. Cannot handle NA values. When x and y have zeros in the same position, NaNs are produced. Please replace these zeros with pseudo-zeros (i.e. 0.0001) if you wish to use this distance metric.

Usage

```
distance_chi_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(), distance_hellinger_cpp(), distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_chi_cpp(x = runif(100), y = runif(100))
```

`distance_cosine_cpp` *(C++) Cosine Dissimilarity Between Two Vectors*

Description

Computes the cosine dissimilarity between two numeric vectors.

Usage

```
distance_cosine_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(), distance_hellinger_cpp(),
distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_cosine_cpp(c(0.2, 0.4, 0.5), c(0.1, 0.8, 0.2))
```

`distance_euclidean_cpp`
(C++) Euclidean Distance Between Two Vectors

Description

Computed as: `sqrt(sum((x - y)^2)`. Cannot handle NA values.

Usage

```
distance_euclidean_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_cosine_cpp(), distance_hamming_cpp(), distance_hellinger_cpp(),
distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_euclidean_cpp(x = runif(100), y = runif(100))
```

---

distance_hamming_cpp *(C++) Hamming Distance Between Two Binary Vectors*

Description

Computes the Hamming distance between two binary vectors.

Usage

```
distance_hamming_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hellinger_cpp(),
distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_hamming_cpp(c(0, 1, 0, 1), c(1, 1, 0, 0))
```

distance_hellinger_cpp *(C++) Hellinger Distance Between Two Vectors*

Description

Computed as: `sqrt(1/2 * sum((sqrt(x) - sqrt(y))^2))`. Cannot handle NA values.

Usage

```
distance_hellinger_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(),
distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_hellinger_cpp(x = runif(100), y = runif(100))
```

distance_jaccard_cpp *(C++) Jaccard Distance Between Two Binary Vectors*

Description

Computes the Jaccard distance between two binary vectors.

Usage

```
distance_jaccard_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(),
distance_hellinger_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp(), distance_sorensen_cpp()

Examples

```
distance_jaccard_cpp(x = c(0, 1, 0, 1), y = c(1, 1, 0, 0))
```

---

distance_ls_cpp *(C++) Sum of Pairwise Distances Between Cases in Two Aligned Time Series*

Description

Computes the lock-step sum of distances between two regular and aligned time series. NA values should be removed before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
distance_ls_cpp(x, y, distance = "euclidean")
```

Arguments

x	(required, numeric matrix) univariate or multivariate time series.
y	(required, numeric matrix) univariate or multivariate time series with the same number of columns and rows as 'x'.
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".

Value

numeric

See Also

Other Rcpp_matrix: cost_matrix_diagonal_cpp(), cost_matrix_diagonal_weighted_cpp(),
cost_matrix_orthogonal_cpp(), distance_matrix_cpp()

Examples

```
#simulate two regular time series
x <- zoo_simulate(
  seed = 1,
  irregular = FALSE
  )
y <- zoo_simulate(
  seed = 2,
  irregular = FALSE
  )

#distance matrix
dist_matrix <- distance_ls_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)
```

`distance_manhattan_cpp`

(C++) Manhattan Distance Between Two Vectors

Description

Computed as: `sum(abs(x - y))`. Cannot handle NA values.

Usage

```
distance_manhattan_cpp(x, y)
```

Arguments

`x`	(required, numeric vector).
`y`	(required, numeric vector) of same length as `x`.

Value

numeric

See Also

Other Rcpp_distance_methods: `distance_bray_curtis_cpp()`, `distance_canberra_cpp()`, `distance_chebyshev_cpp()`, `distance_chi_cpp()`, `distance_cosine_cpp()`, `distance_euclidean_cpp()`, `distance_hamming_cpp()`, `distance_hellinger_cpp()`, `distance_jaccard_cpp()`, `distance_russelrao_cpp()`, `distance_sorensen_cpp()`

Examples

```
distance_manhattan_cpp(x = runif(100), y = runif(100))
```

distance_matrix *Data Frame to Distance Matrix*

Description

Data Frame to Distance Matrix

Usage

```
distance_matrix(df = NULL, name_column = NULL, distance = "euclidean")
```

Arguments

df	(required, data frame) Data frame with numeric columns to transform into a distance matrix. Default: NULL
name_column	(optional, column name) Column naming individual time series. Numeric names are converted to character with the prefix "X". Default: NULL
distance	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

square matrix

See Also

Other distances: distance(), distances

Examples

```
#compute distance matrix
m <- distance_matrix(
  df = cities_coordinates,
  name_column = "name",
  distance = "euclidean"
)

#get data used to compute the matrix
attributes(m)$df

#check matrix
m
```

`distance_matrix_cpp` *(C++) Distance Matrix of Two Time Series*

Description

Computes the distance matrix between the rows of two matrices `y` and `x` representing regular or irregular time series with the same number of columns. NA values should be removed before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
distance_matrix_cpp(x, y, distance = "euclidean")
```

Arguments

`x`	(required, numeric matrix) univariate or multivariate time series.
`y`	(required, numeric matrix) univariate or multivariate time series with the same number of columns as 'x'.
`distance`	(optional, character string) distance name from the "names" column of the dataset `distances` (see `distances$name`). Default: "euclidean".

Value

numeric matrix

See Also

Other Rcpp_matrix: `cost_matrix_diagonal_cpp()`, `cost_matrix_diagonal_weighted_cpp()`, `cost_matrix_orthogonal_cpp()`, `distance_ls_cpp()`

Examples

```
#simulate two time series
x <- zoo_simulate(seed = 1)
y <- zoo_simulate(seed = 2)

#distance matrix
dist_matrix <- distance_matrix_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)
```

distance_russelrao_cpp *(C++) Russell-Rao Distance Between Two Binary Vectors*

Description

Computes the Russell-Rao distance between two binary vectors.

Usage

```
distance_russelrao_cpp(x, y)
```

Arguments

x	(required, numeric). Binary vector of 1s and 0s.
y	(required, numeric) Binary vector of 1s and 0s of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(),
distance_hellinger_cpp(), distance_jaccard_cpp(), distance_manhattan_cpp(), distance_sorensen_cpp()

Examples

```
distance_russelrao_cpp(c(0, 1, 0, 1), c(1, 1, 0, 0))
```

distance_sorensen_cpp *(C++) Sørensen Distance Between Two Binary Vectors*

Description

Computes the Sørensen distance, suitable for presence/absence data.

Usage

```
distance_sorensen_cpp(x, y)
```

Arguments

x	(required, numeric vector).
y	(required, numeric vector) of same length as x.

Value

numeric

See Also

Other Rcpp_distance_methods: distance_bray_curtis_cpp(), distance_canberra_cpp(), distance_chebyshev_cpp(
distance_chi_cpp(), distance_cosine_cpp(), distance_euclidean_cpp(), distance_hamming_cpp(),
distance_hellinger_cpp(), distance_jaccard_cpp(), distance_manhattan_cpp(), distance_russelrao_cpp()

Examples

```
distance_sorensen_cpp(x = c(0, 1, 1, 0), y = c(1, 1, 0, 0))
```

---

distantia *Dissimilarity Analysis of Time Series Lists*

Description

This function combines *dynamic time warping* or *lock-step comparison* with the *psi dissimilarity score* and *permutation methods* to assess dissimilarity between pairs time series or any other sort of data composed of events ordered across a relevant dimension.

Dynamic Time Warping (DTW) finds the optimal alignment between two time series by minimizing the cumulative distance between their samples. It applies dynamic programming to identify the least-cost path through a distance matrix between all pairs of samples. The resulting sum of distances along the least cost path is a metric of time series similarity. DTW disregards the exact timing of samples and focuses on their order and pattern similarity between time series, making it suitable for comparing both *regular and irregular time series of the same or different lengths*, such as phenological data from different latitudes or elevations, time series from various years or periods, and movement trajectories like migration paths. Additionally, `distantia()` implements constrained DTW via Sakoe-Chiba bands with the `bandwidth` argument, which defines a region around the distance matrix diagonal to restrict the spread of the least cost path.

Lock-step (LS) sums pairwise distances between samples in *regular or irregular time series of the same length*, preferably captured at the same times. This method is an alternative to dynamic time warping when the goal is to assess the synchronicity of two time series.

The **psi score** normalizes the cumulative sum of distances between two time series by the cumulative sum of distances between their consecutive samples to generate a comparable dissimilarity score. If for two time series x and y D_xy represents the cumulative sum of distances between them, either resulting from dynamic time warping or the lock-step method, and S_xy represents the cumulative sum of distances of their consecutive samples, then the psi score can be computed in two ways depending on the scenario:

Equation 1: $\psi = \frac{D_{xy} - S_{xy}}{S_{xy}}$

Equation 2: $\psi = \frac{D_{xy} - S_{xy}}{S_{xy}} + 1$

When D_xy is computed via dynamic time warping **ignoring the distance matrix diagonals** (`diagonal = FALSE`), then *Equation 1* is used. On the other hand, if D_xy results from the lock-step method (`lock_step = TRUE`), or from dynamic time warping considering diagonals (`diagonal = TRUE`), then *Equation 2* is used instead:

In both equations, a psi score of zero indicates maximum similarity.

Permutation methods are provided here to help assess the robustness of observed psi scores by direct comparison with a null distribution of psi scores resulting from randomized versions of the compared time series. The fraction of null scores smaller than the observed score is returned as a *p_value* in the function output and interpreted as "the probability of finding a higher similarity (lower psi score) by chance".

In essence, restricted permutation is useful to answer the question "how robust is the similarity between two time series?"

Four different permutation methods are available:

- **"restricted"**: Separates the data into blocks of contiguous rows, and re-shuffles data points randomly within these blocks, independently by row and column. Applied when the data is structured in blocks that should be preserved during permutations (e.g., "seasons", "years", "decades", etc) and the columns represent independent variables.
- **"restricted_by_row"**: Separates the data into blocks of contiguous rows, and re-shuffles complete rows within these blocks. This method is suitable for cases where the data is organized into blocks as described above, but columns represent interdependent data (e.g., rows represent percentages or proportions), and maintaining the relationships between data within each row is important.
- **"free"**: Randomly re-shuffles data points across the entire time series, independently by row and column. This method is useful for loosely structured time series where data independence is assumed. When the data exhibits a strong temporal structure, this approach may lead to an overestimation of the robustness of dissimilarity scores.
- **"free_by_row"**: Randomly re-shuffles complete rows across the entire time series. This method is useful for loosely structured time series where dependency between columns is assumed (e.g., rows represent percentages or proportions). This method has the same drawbacks as the "free" method, when the data exhibits a strong temporal structure.

This function allows computing dissimilarity between pairs of time series using different combinations of arguments at once. For example, when the argument `distance` is set to `c("euclidean", "manhattan")`, the output data frame will show two dissimilarity scores for each pair of time series, one based on euclidean distances, and another based on manhattan distances. The same happens for most other parameters.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr. However, due to the high performance of the C++ backend, parallelization might only result in efficiency gains when running permutation tests with large number of iterations, or working with very long time series.

Usage

```
distantia(
  tsl = NULL,
  distance = "euclidean",
```

```
  diagonal = TRUE,
  bandwidth = 1,
  lock_step = FALSE,
  permutation = "restricted_by_row",
  block_size = NULL,
  repetitions = 0,
  seed = 1
)
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".
`diagonal`	(optional, logical vector). If TRUE, diagonals are included in the dynamic time warping computation. Default: TRUE
`bandwidth`	(optional, numeric) Proportion of space at each side of the cost matrix diagonal (aka *Sakoe-Chiba band*) defining a valid region for dynamic time warping, used to control the flexibility of the warping path. This method prevents degenerate alignments due to differences in magnitude between time series when the data is not properly scaled. If `1` (default), DTW is unconstrained. If `0`, DTW is fully constrained and the warping path follows the matrix diagonal. Recommended values may vary depending on the nature of the data. Ignored if `lock_step = TRUE`. Default: 1.
`lock_step`	(optional, logical vector) If TRUE, time series captured at the same times are compared sample wise (with no dynamic time warping). Requires time series in argument `tsl` to be fully aligned, or it will return an error. Default: FALSE.
`permutation`	(optional, character vector) permutation method, only relevant when `repetitions` is higher than zero. Valid values are: "restricted_by_row", "restricted", "free_by_row", and "free". Default: "restricted_by_row".
`block_size`	(optional, integer) Size of the row blocks for the restricted permutation test. Only relevant when permutation methods are "restricted" or "restricted_by_row" and `repetitions` is higher than zero. A block of size `n` indicates that a row can only be permuted within a block of `n` adjacent rows. If NULL, defaults to the rounded one tenth of the shortest time series in `tsl`. Default: NULL.
`repetitions`	(optional, integer vector) number of permutations to compute the p-value. If 0, p-values are not computed. Otherwise, the minimum is 2. The resolution of the p-values and the overall computation time depends on the number of permutations. Default: 0
`seed`	(optional, integer) initial random seed to use for replicability when computing p-values. Default: 1

Value

data frame with columns:

- x: time series name.
- y: time series name.
- distance: name of the distance metric.
- diagonal: value of the argument diagonal.
- lock_step: value of the argument lock_step.
- repetitions (only if repetitions > 0): value of the argument repetitions.
- permutation (only if repetitions > 0): name of the permutation method used to compute p-values.
- seed (only if repetitions > 0): random seed used to in the permutations.
- psi: psi dissimilarity of the sequences x and y.
- null_mean (only if repetitions > 0): mean of the null distribution of psi scores.
- null_sd (only if repetitions > 0): standard deviation of the null distribution of psi values.
- p_value (only if repetitions > 0): proportion of scores smaller or equal than psi in the null distribution.

See Also

Other distantia: distantia_dtw(), distantia_dtw_plot(), distantia_ls()

Examples

```
#parallelization setup
#not worth it for this data size
# future::plan(
#   strategy = future::multisession,
#   workers = 2
# )

#progress bar (does not work in R examples)
# progressr::handlers(global = TRUE)


#load fagus_dynamics as tsl
#global centering and scaling
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )
```

```
}

#dynamic time warping dissimilarity analysis
#-----------------------------------------
#permutation restricted by row to preserve dependency of ndvi on temperature and rainfall
#block size is 3 months to permute within same season
df_dtw <- distantia(
  tsl = tsl,
  distance = "euclidean",
  permutation = "restricted_by_row",
  block_size = 3, #months
  repetitions = 10, #increase to 100 or more
  seed = 1
)

#focus on the important details
df_dtw[, c("x", "y", "psi", "p_value", "null_mean", "null_sd")]
#higher psi values indicate higher dissimilarity
#p-values indicate chance of finding a random permutation with a psi smaller than the observed

#visualize dynamic time warping
if(interactive()){

  distantia_dtw_plot(
    tsl = tsl[c("Spain", "Sweden")],
    distance = "euclidean"
  )

}

#recreating the null distribution
#direct call to C++ function
#use same args as distantia() call
psi_null <- psi_null_dtw_cpp(
  x = tsl[["Spain"]],
  y = tsl[["Sweden"]],
  distance = "euclidean",
  repetitions = 10, #increase to 100 or more

  permutation = "restricted_by_row",
  block_size = 3,
  seed = 1
)

#compare null mean with output of distantia()
mean(psi_null)
df_dtw$null_mean[3]
```

---

distantia_aggregate *Aggregate* distantia() *Data Frames Across Parameter Combinations*

Description

The function distantia() allows dissimilarity assessments based on several combinations of arguments at once. For example, when the argument distance is set to c("euclidean", "manhattan"), the output data frame will show two dissimilarity scores for each pair of compared time series, one based on euclidean distances, and another based on manhattan distances.

This function computes dissimilarity stats across combinations of parameters.

If psi scores smaller than zero occur in the aggregated output, then the the smaller psi value is added to the column psi to start dissimilarity scores at zero.

If there are no different combinations of arguments in the input data frame, no aggregation happens, but all parameter columns are removed.

Usage

```
distantia_aggregate(df = NULL, f = mean, ...)
```

Arguments

df	(required, data frame) Output of distantia(), distantia_ls(), distantia_dtw(), or distantia_time_delay(). Default: NULL
f	(optional, function) Function to summarize psi scores (for example, mean) when there are several combinations of parameters in df. Ignored when there is a single combination of arguments in the input. Default: mean
...	(optional, arguments of f) Further arguments to pass to the function f.

Value

data frame

See Also

Other distantia_support: distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_opti utils_cluster_silhouette()

Examples

```
#three time series
#climate and ndvi in Fagus sylvatica stands in Spain, Germany, and Sweden
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )
```

```
if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )
}

#distantia with multiple parameter combinations
#-----------------------------------
df <- distantia(
  tsl = tsl,
  distance = c("euclidean", "manhattan"),
  lock_step = TRUE
)

df[, c(
  "x",
  "y",
  "distance",
  "psi"
)]

#aggregation using means
df <- distantia_aggregate(
  df = df,
  f = mean
)

df
```

`distantia_boxplot` *Distantia Boxplot*

Description

Boxplot of a data frame returned by `distantia()` summarizing the stats of the psi scores of each time series against all others.

Usage

```
distantia_boxplot(df = NULL, fill_color = NULL, f = median, text_cex = 1)
```

Arguments

`df`	(required, data frame) Output of `distantia()`, `distantia_ls()`, `distantia_dtw()`, or `distantia_time_delay()`. Default: NULL
`fill_color`	(optional, character vector) boxplot fill color. Default: NULL
`f`	(optional, function) function used to aggregate the input data frame and arrange the boxes. One of `mean` or `median`. Default: `median`.
`text_cex`	(optional, numeric) Multiplier of the text size. Default: 1

Value

boxplot

See Also

Other distantia_support: distantia_aggregate(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_opti utils_cluster_silhouette()

Examples

```
tsl <- tsl_initialize(
  x = distantia::albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

df <- distantia(
  tsl = tsl,
  lock_step = TRUE
  )

distantia_boxplot(
  df = df,
  text_cex = 1.5
  )
```

---

distantia_cluster_hclust

Hierarchical Clustering of Dissimilarity Analysis Data Frames

---

Description

This function combines the dissimilarity scores computed by distantia(), the agglomerative clustering methods provided by stats::hclust(), and the clustering optimization method implemented in utils_cluster_hclust_optimizer() to help group together time series with similar features.

When clusters = NULL, the function utils_cluster_hclust_optimizer() is run underneath to perform a parallelized grid search to find the number of clusters maximizing the overall silhouette width of the clustering solution (see utils_cluster_silhouette()). When method = NULL as well, the optimization also includes all methods available in stats::hclust() in the grid search.

This function supports a parallelization setup via future::plan(), and progress bars provided by the package progressr.

Usage

```
distantia_cluster_hclust(df = NULL, clusters = NULL, method = "complete")
```

Arguments

df	(required, data frame) Output of distantia(), distantia_ls(), distantia_dtw(), or distantia_time_delay(). Default: NULL
clusters	(required, integer) Number of groups to generate. If NULL (default), utils_cluster_kmeans_optimize is used to find the number of clusters that maximizes the mean silhouette width of the clustering solution (see utils_cluster_silhouette()). Default: NULL
method	(optional, character string) Argument of stats::hclust() defining the agglomerative method. One of: "ward.D", "ward.D2", "single", "complete", "average" (= UPGMA), "mcquitty" (= WPGMA), "median" (= WPGMC) or "centroid" (= UPGMC). Unambiguous abbreviations are accepted as well. If NULL (default), utils_cluster_hclust_optimizer() finds the optimal method. Default: "complete".

Value

list:

- cluster_object: hclust object for further analyses and custom plotting.
- clusters: integer, number of clusters.
- silhouette_width: mean silhouette width of the clustering solution.
- df: data frame with time series names, their cluster label, and their individual silhouette width scores.
- d: psi distance matrix used for clustering.
- optimization: only if clusters = NULL, data frame with optimization results from utils_cluster_hclust_optimiz

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_opti utils_cluster_silhouette()

Examples

```
#weekly covid prevalence in California
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
)

#subset 10 elements to accelerate example execution
tsl <- tsl_subset(
  tsl = tsl,
```

```
  names = 1:10
)

if(interactive()){
  #plotting first three time series
  tsl_plot(
    tsl = tsl[1:3],
    guide_columns = 3
  )
}

#dissimilarity analysis
distantia_df <- distantia(
  tsl = tsl,
  lock_step = TRUE
)

#hierarchical clustering
#automated number of clusters
#automated method selection
distantia_clust <- distantia_cluster_hclust(
  df = distantia_df,
  clusters = NULL,
  method = NULL
)

#names of the output object
names(distantia_clust)

#cluster object
distantia_clust$cluster_object

#distance matrix used for clustering
distantia_clust$d

#number of clusters
distantia_clust$clusters

#clustering data frame
#group label in column "cluster"
#negatives in column "silhouette_width" higlight anomalous cluster assignation
distantia_clust$df

#mean silhouette width of the clustering solution
distantia_clust$silhouette_width

#plot
if(interactive()){

  dev.off()

  clust <- distantia_clust$cluster_object
  k <- distantia_clust$clusters
```

```
  #tree plot
  plot(
    x = clust,
    hang = -1
  )

  #highlight groups
  stats::rect.hclust(
    tree = clust,
    k = k,
    cluster = stats::cutree(
      tree = clust,
      k = k
    )
  )

}
```

`distantia_cluster_kmeans` *K-Means Clustering of Dissimilarity Analysis Data Frames*

Description

This function combines the dissimilarity scores computed by `distantia()`, the K-means clustering method implemented in `stats::kmeans()`, and the clustering optimization method implemented in `utils_cluster_hclust_optimizer()` to help group together time series with similar features.

When `clusters = NULL`, the function `utils_cluster_hclust_optimizer()` is run underneath to perform a parallelized grid search to find the number of clusters maximizing the overall silhouette width of the clustering solution (see `utils_cluster_silhouette()`).

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
distantia_cluster_kmeans(df = NULL, clusters = NULL, seed = 1)
```

Arguments

`df`	(required, data frame) Output of `distantia()`, `distantia_ls()`, `distantia_dtw()`, or `distantia_time_delay()`. Default: NULL
`clusters`	(required, integer) Number of groups to generate. If NULL (default), `utils_cluster_kmeans_optimize` is used to find the number of clusters that maximizes the mean silhouette width of the clustering solution (see `utils_cluster_silhouette()`). Default: NULL
`seed`	(optional, integer) Random seed to be used during the K-means computation. Default: 1

Value

list:

- `cluster_object`: kmeans object object for further analyses and custom plotting.
- `clusters`: integer, number of clusters.
- `silhouette_width`: mean silhouette width of the clustering solution.
- `df`: data frame with time series names, their cluster label, and their individual silhouette width scores.
- `d`: psi distance matrix used for clustering.
- `optimization`: only if `clusters` = `NULL`, data frame with optimization results from `utils_cluster_hclust_optimiz`

See Also

Other distantia_support: `distantia_aggregate()`, `distantia_boxplot()`, `distantia_cluster_hclust()`, `distantia_matrix()`, `distantia_model_frame()`, `distantia_spatial()`, `distantia_stats()`, `distantia_time_delay()`, `utils_block_size()`, `utils_cluster_hclust_optimizer()`, `utils_cluster_kmeans_opti` `utils_cluster_silhouette()`

Examples

```
#weekly covid prevalence in California
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
)

#subset 10 elements to accelerate example execution
tsl <- tsl_subset(
  tsl = tsl,
  names = 1:10
)

if(interactive()){
  #plotting first three time series
  tsl_plot(
    tsl = tsl[1:3],
    guide_columns = 3
  )
}

#dissimilarity analysis
distantia_df <- distantia(
  tsl = tsl,
  lock_step = TRUE
)

#hierarchical clustering
#automated number of clusters
distantia_kmeans <- distantia_cluster_kmeans(
```

```
  df = distantia_df,
  clusters = NULL
)

#names of the output object
names(distantia_kmeans)

#kmeans object
distantia_kmeans$cluster_object

#distance matrix used for clustering
distantia_kmeans$d

#number of clusters
distantia_kmeans$clusters

#clustering data frame
#group label in column "cluster"
distantia_kmeans$df

#mean silhouette width of the clustering solution
distantia_kmeans$silhouette_width

#kmeans plot
# factoextra::fviz_cluster(
#   object = distantia_kmeans$cluster_object,
#   data = distantia_kmeans$d,
#   repel = TRUE
# )
```

`distantia_dtw` *Dynamic Time Warping Dissimilarity Analysis of Time Series Lists*

Description

Minimalistic but slightly faster version of `distantia()` to compute dynamic time warping dissimilarity scores using diagonal least cost paths.

Usage

```
distantia_dtw(tsl = NULL, distance = "euclidean")
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

data frame with columns:

- x: time series name.
- y: time series name.
- distance: name of the distance metric.
- psi: psi dissimilarity of the sequences x and y.

See Also

Other distantia: distantia(), distantia_dtw_plot(), distantia_ls()

Examples

```
#load fagus_dynamics as tsl
#global centering and scaling
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )
}

#dynamic time warping dissimilarity analysis
df_dtw <- distantia_dtw(
  tsl = tsl,
  distance = "euclidean"
)

df_dtw[, c("x", "y", "psi")]

#visualize dynamic time warping
if(interactive()){

  distantia_dtw_plot(
    tsl = tsl[c("Spain", "Sweden")],
    distance = "euclidean"
  )

}
```

`distantia_dtw_plot` *Two-Way Dissimilarity Plots of Time Series Lists*

Description

Plots two sequences, their distance or cost matrix, their least cost path, and all relevant values used to compute dissimilarity.

Unlike `distantia()`, this function does not accept vectors as inputs for the arguments to compute dissimilarity (`distance`, `diagonal`, and `weighted`), and only plots a pair of sequences at once.

The argument `lock_step` is not available because this plot does not make sense in such a case.

Usage

```
distantia_dtw_plot(
  tsl = NULL,
  distance = "euclidean",
  diagonal = TRUE,
  bandwidth = 1,
  matrix_type = "cost",
  matrix_color = NULL,
  path_width = 1,
  path_color = "black",
  diagonal_width = 1,
  diagonal_color = "white",
  line_color = NULL,
  line_width = 1,
  text_cex = 1
)
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".
`diagonal`	(optional, logical vector). If TRUE, diagonals are included in the dynamic time warping computation. Default: TRUE
`bandwidth`	(optional, numeric) Proportion of space at each side of the cost matrix diagonal (aka *Sakoe-Chiba band*) defining a valid region for dynamic time warping, used to control the flexibility of the warping path. This method prevents degenerate alignments due to differences in magnitude between time series when the data is not properly scaled. If `1` (default), DTW is unconstrained. If `0`, DTW is fully constrained and the warping path follows the matrix diagonal. Recommended values may vary depending on the nature of the data. Ignored if `lock_step = TRUE`. Default: 1.

matrix_type	(optional, character string): one of "cost" or "distance" (the abbreviation "dist" is accepted as well). Default: "cost".
matrix_color	(optional, character vector) vector of colors for the distance or cost matrix. If NULL, uses the palette "Zissou 1" provided by the function grDevices::hcl.colors(). Default: NULL
path_width	(optional, numeric) width of the least cost path. Default: 1
path_color	(optional, character string) color of the least-cost path. Default: "black"
diagonal_width	(optional, numeric) width of the diagonal. Set to 0 to remove the diagonal line. Default: 0.5
diagonal_color	(optional, character string) color of the diagonal. Default: "white"
line_color	(optional, character vector) Vector of colors for the time series plot. If not provided, defaults to a subset of matrix_color.
line_width	(optional, numeric vector) Width of the time series plot. Default: 1
text_cex	(optional, numeric) Multiplier of the text size. Default: 1

Value

multipanel plot

See Also

Other distantia: distantia(), distantia_dtw(), distantia_ls()

Examples

```
#three time series
#climate and ndvi in Fagus sylvatica stands in Spain, Germany, and Sweden
#convert to time series list
#scale and center to neutralize effect of different scales in temperature, rainfall, and ndvi
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global #see help(f_scale_global)
  )

if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )
}

#visualize dynamic time warping
if(interactive()){

  #plot pair with cost matrix (default)
```

```
  distantia_dtw_plot(
    tsl = tsl[c("Spain", "Sweden")] #only two time series!
  )

  #plot pair with distance matrix
  distantia_dtw_plot(
    tsl = tsl[c("Spain", "Sweden")],
    matrix_type = "distance"
  )

  #plot pair with different distance
  distantia_dtw_plot(
    tsl = tsl[c("Spain", "Sweden")],
    distance = "manhattan", #sed data(distances)
    matrix_type = "distance"
  )


  #with different colors
  distantia_dtw_plot(
    tsl = tsl[c("Spain", "Sweden")],
    matrix_type = "distance",
    matrix_color = grDevices::hcl.colors(
      n = 100,
      palette = "Inferno"
    ),
    path_color = "white",
    path_width = 2,
    line_color = grDevices::hcl.colors(
      n = 3, #same as variables in tsl
      palette = "Inferno"
    )
  )

}
```

`distantia_ls` *Lock-Step Dissimilarity Analysis of Time Series Lists*

Description

Minimalistic but slightly faster version of `distantia()` to compute lock-step dissimilarity scores.

Usage

```
distantia_ls(tsl = NULL, distance = "euclidean")
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

data frame:

- `x`: time series name.
- `y`: time series name.
- `distance`: name of the distance metric.
- `psi`: psi dissimilarity of the sequences `x` and `y`.

See Also

Other distantia: `distantia()`, `distantia_dtw()`, `distantia_dtw_plot()`

Examples

```
#load fagus_dynamics as tsl
#global centering and scaling
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )
}

#lock-step dissimilarity analysis
df_ls <- distantia_ls(
  tsl = tsl,
  distance = "euclidean"
)

df_ls
```

distantia_matrix *Convert Dissimilarity Analysis Data Frame to Distance Matrix*

Description

Transforms a data frame resulting from `distantia()` into a dissimilarity matrix.

Usage

```
distantia_matrix(df = NULL)
```

Arguments

df	(required, data frame) Output of `distantia()`, `distantia_ls()`, `distantia_dtw()`, or `distantia_time_delay()`. Default: NULL

Value

numeric matrix

See Also

Other distantia_support: `distantia_aggregate()`, `distantia_boxplot()`, `distantia_cluster_hclust()`, `distantia_cluster_kmeans()`, `distantia_model_frame()`, `distantia_spatial()`, `distantia_stats()`, `distantia_time_delay()`, `utils_block_size()`, `utils_cluster_hclust_optimizer()`, `utils_cluster_kmeans_opti`
`utils_cluster_silhouette()`

Examples

```
#weekly covid prevalence in three California counties
#load as tsl
#subset 5 counties
#sum by month
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
) |>
  tsl_subset(
    names = 1:5
  ) |>
  tsl_aggregate(
    new_time = "months",
    method = sum
  )

if(interactive()){

  #plotting first three time series
```

```
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )

  dev.off()

}

#dissimilarity analysis
#two combinations of arguments
distantia_df <- distantia(
  tsl = tsl,
  lock_step = c(TRUE, FALSE)
)

#to dissimilarity matrix
distantia_matrix <- distantia_matrix(
  df = distantia_df
)

#returns a list of matrices
lapply(
  X = distantia_matrix,
  FUN = class
  )

#these matrices have attributes tracing how they were generated
lapply(
  X = distantia_matrix,
  FUN = \(x) attributes(x)$distantia_args
)

#plot matrix
if(interactive()){

  #plot first matrix (default behavior of utils_matrix_plot())
  utils_matrix_plot(
    m = distantia_matrix
  )

  #plot second matrix
  utils_matrix_plot(
    m = distantia_matrix[[2]]
  )

}
```

---

distantia_model_frame *Dissimilarity Model Frame*

Description

This function generates a model frame for statistical or machine learning analysis from these objects:

- : Dissimilarity data frame generated by `distantia()`, `distantia_ls()`, `distantia_dtw()`, or `distantia_time_delay()`. The output model frame will have as many rows as this data frame.
- : Data frame with static descriptors of the time series. These descriptors are converted to distances between pairs of time series via `distance_matrix()`.
- : List defining composite predictors. This feature allows grouping together predictors that have a common meaning. For example, `composite_predictors = list(temperature = c("temperature_mean", "temp` generates a new predictor named "temperature", which results from computing the multivariate distances between the vectors of temperature variables of each pair of time series. Predictors in one of such groups will be scaled before distance computation if their maximum standard deviations differ by a factor of 10 or more.

The resulting data frame contains the following columns:

- `x` and `y`: names of the pair of time series represented in the row.
- response columns in `response_df`.
- predictors columns: representing the distance between the values of the given static predictor between `x` and `y`.
- (optional) `geographic_distance`: If `predictors_df` is an sf `sf` data frame, then geographic distances are computed via `sf::st_distance()`.

This function supports a parallelization setup via `future::plan()`.

Usage

```
distantia_model_frame(
  response_df = NULL,
  predictors_df = NULL,
  composite_predictors = NULL,
  scale = TRUE,
  distance = "euclidean"
)
```

Arguments

`response_df`	(required, data frame) output of `distantia()`, `distantia_ls()`, `distantia_dtw()`, or `distantia_time_delay()`. Default: NULL
`predictors_df`	(required, data frame or sf data frame) data frame with numeric predictors for the the model frame. Must have a column with the time series names in `response_df$x` and `response_df$y`. If `sf` data frame, the column "geographic_distance" with distances between pairs of time series is added to the model frame. Default: NULL

composite_predictors
: (optional, list) list defining composite predictors. For example, `composite_predictors = list(a = c("b", "c"))` uses the columns "b" and "c" from predictors_df to generate the predictor a as the multivariate distance between "b" and "c" for each pair of time series in response_df. Default: NULL

scale
: (optional, logical) if TRUE, all predictors are scaled and centered with scale(). Default: TRUE

distance
: (optional, string) Method to compute the distance between predictor values for all pairs of time series in response_df. Default: "euclidean".

Value

data frame: with attributes "predictors", "response", and "formula".

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_opti utils_cluster_silhouette()

Examples

```
#covid prevalence in California counties
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
) |>
  #subset to shorten example runtime
  tsl_subset(
    names = 1:5
  )

#dissimilarity analysis
df <- distantia_ls(tsl = tsl)

#combine several predictors
#into a new one
composite_predictors <- list(
  economy = c(
    "poverty_percentage",
    "median_income",
    "domestic_product"
    )
)

#generate model frame
model_frame <- distantia_model_frame(
  response_df = df,
  predictors_df = covid_counties,
```

```
  composite_predictors = composite_predictors,
  scale = TRUE
)

head(model_frame)

#names of response and predictors
#and an additive formula
#are stored as attributes
attributes(model_frame)$predictors

#if response_df is output of distantia():
attributes(model_frame)$response
attributes(model_frame)$formula

#example of linear model
# model <- lm(
#   formula = attributes(model_frame)$formula,
#   data = model_frame
# )
#
# summary(model)
```

`distantia_spatial` *Spatial Representation of* `distantia()` *Data Frames*

Description

Given an sf data frame with geometry types POLYGON, MULTIPOLYGON, or POINT representing time series locations, this function transforms the output of `distantia()`, `distantia_ls()`, `distantia_dtw()` or `distantia_time_delay()` to an sf data frame.

If `network = TRUE`, the sf data frame is of type LINESTRING, with edges connecting time series locations. This output is helpful to build many-to-many dissimilarity maps (see examples).

If `network = FALSE`, the sf data frame contains the geometry in the input `sf` argument. This output helps build one-to-many dissimilarity maps.

Usage

```
distantia_spatial(df = NULL, sf = NULL, network = TRUE)
```

Arguments

`df`	(required, data frame) Output of `distantia()` or `distantia_time_delay()`. Default: NULL
`sf`	(required, sf data frame) Points or polygons representing the location of the time series in argument 'df'. It must have a column with all time series names in `df$x` and `df$y`. Default: NULL

network (optional, logical) If TRUE, the resulting sf data frame is of time LINESTRING and represent network edges. Default: TRUE

Value

sf data frame (LINESTRING geometry)

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_opti utils_cluster_silhouette()

Examples

```
tsl <- distantia::tsl_initialize(
  x = distantia::covid_prevalence,
  name_column = "name",
  time_column = "time"
) |>
distantia::tsl_subset(
  names = c(
    "Los_Angeles",
    "San_Francisco",
    "Fresno",
    "San_Joaquin"
    )
 )

df_psi <- distantia::distantia_ls(
  tsl = tsl
)

#network many to many
sf_psi <- distantia::distantia_spatial(
  df = df_psi,
  sf = distantia::covid_counties,
  network = TRUE
)

#network map
# mapview::mapview(
#   distantia::covid_counties,
#   col.regions = NA,
#   alpha.regions = 0,
#   color = "black",
#   label = "name",
#   legend = FALSE,
#   map.type = "OpenStreetMap"
# ) +
#   mapview::mapview(
```

```
#     sf_psi_subset,
#     layer.name = "Psi",
#     label = "edge_name",
#     zcol = "psi",
#     lwd = 3
#   ) |>
#   suppressWarnings()
```

`distantia_stats` *Stats of Dissimilarity Data Frame*

Description

Takes the output of `distantia()` to return a data frame with one row per time series with the stats of its dissimilarity scores with all other time series.

Usage

```
distantia_stats(df = NULL)
```

Arguments

`df`	(required, data frame) Output of `distantia()`, `distantia_ls()`, `distantia_dtw()`, or `distantia_time_delay()`. Default: NULL

Value

data frame

See Also

Other distantia_support: `distantia_aggregate()`, `distantia_boxplot()`, `distantia_cluster_hclust()`, `distantia_cluster_kmeans()`, `distantia_matrix()`, `distantia_model_frame()`, `distantia_spatial()`, `distantia_time_delay()`, `utils_block_size()`, `utils_cluster_hclust_optimizer()`, `utils_cluster_kmeans_opti`
`utils_cluster_silhouette()`

Examples

```
tsl <- tsl_simulate(
  n = 5,
  irregular = FALSE
  )

df <- distantia(
  tsl = tsl,
  lock_step = TRUE
  )
```

```
df_stats <- distantia_stats(df = df)

df_stats
```

`distantia_time_delay` *Time Shift Between Time Series*

Description

This function computes an approximation to the time-shift between pairs of time series as the absolute time difference between pairs of observations in the time series *x* and *y* connected by the dynamic time warping path.

If the time series are long enough, the extremes of the warping path are trimmed (5% of the total path length each) to avoid artifacts due to early misalignments.

It returns a data frame with the modal, mean, median, minimum, maximum, quantiles 0.25 and 0.75, and standard deviation. The modal and the median are the most generally accurate time-shift descriptors.

This function requires scaled and detrended time series. Still, it might yield non-sensical results in case of degenerate warping paths. Plotting dubious results with [distantia_dtw_plot())] is always a good approach to identify these cases.

[distantia_dtw_plot())]: R:distantia_dtw_plot())

Usage

```
distantia_time_delay(
  tsl = NULL,
  distance = "euclidean",
  bandwidth = 1,
  two_way = FALSE
)
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".
`bandwidth`	(optional, numeric) Proportion of space at each side of the cost matrix diagonal (aka *Sakoe-Chiba band*) defining a valid region for dynamic time warping, used to control the flexibility of the warping path. This method prevents degenerate alignments due to differences in magnitude between time series when the data is not properly scaled. If `1` (default), DTW is unconstrained. If `0`, DTW is fully constrained and the warping path follows the matrix diagonal. Recommended values may vary depending on the nature of the data. Ignored if `lock_step = TRUE`. Default: 1.
`two_way`	(optional, logical) If TRUE, the time shift between the time series pairs *y* and *x* is added to the results

Value

data frame

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_optimizer, utils_cluster_silhouette()

Examples

```
#load two long-term temperature time series
#local scaling to focus in shape rather than values
#polynomial detrending to make them stationary
tsl <- tsl_init(
  x = cities_temperature[
    cities_temperature$name %in% c("London", "Kinshasa"),
    ],
  name = "name",
  time = "time"
) |>
  tsl_transform(
    f = f_scale_local
  ) |>
  tsl_transform(
    f = f_detrend_poly,
    degree = 35 #data years
  )


if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide = FALSE
  )
}

#compute shifts
df_shift <- distantia_time_delay(
  tsl = tsl,
  two_way = TRUE
)

df_shift
#positive shift values indicate
#that the samples in Kinshasa
#are aligned with older samples in London.
```

eemian_coordinates *Site Coordinates of Nine Interglacial Sites in Central Europe*

Description

Site Coordinates of Nine Interglacial Sites in Central Europe

Usage

```
data(eemian_coordinates)
```

Format

sf data frame with 4 columns and 9 rows.

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

eemian_pollen *Pollen Counts of Nine Interglacial Sites in Central Europe*

Description

Pollen counts of nine interglacial sites in central Europe.

Site coordinates for this dataset are in eemian_coordinates.

Usage

```
data(eemian_pollen)
```

Format

data frame with 24 columns and 376 rows.

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, fagus_coordinates, fagus_dynamics, honeycomb_climate, honeycomb_polygons

Examples

```
data("eemian_pollen")

#to time series list
tsl <- tsl_initialize(
  x = eemian_pollen,
  name_column = "name",
  time_column = "time"
)

#time series plot
if(interactive()){

 tsl_plot(
  tsl = tsl_subset(
    tsl = tsl,
    names = 1:3
    ),
  columns = 2,
  guide_columns = 2
  )

}
```

---

fagus_coordinates *Site Coordinates of Fagus sylvatica Stands*

Description

Site Coordinates of Fagus sylvatica Stands

Usage

```
data(fagus_coordinates)
```

Format

sf data frame with 3 rows and 4 columns

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_dynamics, honeycomb_climate, honeycomb_polygons

fagus_dynamics *Time Series Data from Three Fagus sylvatica Stands*

Description

A data frame with 648 rows representing enhanced vegetation index, rainfall and temperature in three stands of Fagus sylvatica in Spain, Germany, and Sweden.

Usage

```
data(fagus_dynamics)
```

Format

data frame with 5 columns and 648 rows.

Details

Site coordinates for this dataset are in fagus_coordinates.

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, honeycomb_climate, honeycomb_polygons

Examples

```
data("fagus_dynamics")

#to time series list
fagus <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)

#time series plot
if(interactive()){

 tsl_plot(
  tsl = fagus
  )

}
```

f_binary *Transform Zoo Object to Binary*

Description

Converts a zoo object to binary (1 and 0) based on a given threshold.

Usage

```
f_binary(x = NULL, threshold = NULL)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
threshold	(required, numeric) Values greater than this number become 1, others become 0. Set to the mean of the time series by default. Default: NULL

Value

zoo object

See Also

Other tsl_transformation: f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(
  data_range = c(0, 1)
  )

y <- f_binary(
  x = x,
  threshold = 0.5
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

f_clr *Data Transformation: Rowwise Centered Log-Ratio*

Description

Centers log-transformed proportions by subtracting the geometric mean of the row.

Usage

```
f_clr(x = NULL, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(
  cols = 5,
  data_range = c(0, 500)
  )

y <- f_clr(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

f_detrend_difference *Data Transformation: Detrending and Differencing*

Description

Performs differencing to remove trends from a zoo time series, isolating short-term fluctuations by subtracting values at specified lags. The function preserves the original index and metadata, with an option to center the output around the mean of the original series. Suitable for preprocessing time series data to focus on random fluctuations unrelated to overall trends.

Usage

```
f_detrend_difference(x = NULL, lag = 1, center = TRUE, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
lag	(optional, integer)
center	(required, logical) If TRUE, the output is centered at zero. If FALSE, it is centered at the data mean. Default: TRUE
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(cols = 2)

y_lag1 <- f_detrend_difference(
  x = x,
  lag = 1
)

y_lag5 <- f_detrend_difference(
  x = x,
  lag = 5
)

if(interactive()){
  zoo_plot(x)
```

```
  zoo_plot(y_lag1)
  zoo_plot(y_lag5)
}
```

`f_detrend_linear` *Data Transformation: Linear Detrending of Zoo Time Series*

Description

Fits a linear model on each column of a zoo object using time as a predictor, predicts the outcome, and subtracts it from the original data to return a detrended time series. This method might not be suitable if the input data is not seasonal and has a clear trend, so please be mindful of the limitations of this function when applied blindly.

Usage

```
f_detrend_linear(x = NULL, center = TRUE, ...)
```

Arguments

`x`	(required, zoo object) Zoo time series object to transform.
`center`	(required, logical) If TRUE, the output is centered at zero. If FALSE, it is centered at the data mean. Default: TRUE
`...`	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: `f_binary()`, `f_clr()`, `f_detrend_difference()`, `f_detrend_poly()`, `f_hellinger()`, `f_list()`, `f_log()`, `f_percent()`, `f_proportion()`, `f_proportion_sqrt()`, `f_rescale_global()`, `f_rescale_local()`, `f_scale_global()`, `f_scale_local()`, `f_trend_linear()`, `f_trend_poly()`

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_detrend_linear(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

f_detrend_poly *Data Transformation: Polynomial Linear Detrending of Zoo Time Series*

Description

Fits a polynomial linear model on each column of a zoo object using time as a predictor, predicts the outcome, and subtracts it from the original data to return a detrended time series. This method is a useful alternative to f_detrend_linear when the overall trend of the time series does not follow a straight line.

Usage

```
f_detrend_poly(x = NULL, degree = 2, center = TRUE, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
degree	(optional, integer) Degree of the polynomial. Default: 2
center	(required, logical) If TRUE, the output is centered at zero. If FALSE, it is centered at the data mean. Default: TRUE
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_detrend_poly(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

f_hellinger *Data Transformation: Rowwise Hellinger Transformation*

Description

Transforms the input zoo object to proportions via f_proportion and then applies the Hellinger transformation.

Usage

```
f_hellinger(x = NULL, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(
  cols = 5,
  data_range = c(0, 500)
  )

y <- f_hellinger(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

f_list *Lists Available Transformation Functions*

Description

Lists Available Transformation Functions

Usage

```
f_list()
```

Value

character vector

See Also

Other tsl_transformation: `f_binary()`, `f_clr()`, `f_detrend_difference()`, `f_detrend_linear()`, `f_detrend_poly()`, `f_hellinger()`, `f_log()`, `f_percent()`, `f_proportion()`, `f_proportion_sqrt()`, `f_rescale_global()`, `f_rescale_local()`, `f_scale_global()`, `f_scale_local()`, `f_trend_linear()`, `f_trend_poly()`

Examples

```
f_list()
```

f_log *Data Transformation: Log*

Description

Applies logarithmic transformation to data to reduce skewness.

Usage

```
f_log(x = NULL, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(
  cols = 5,
  data_range = c(0, 500)
  )

y <- f_log(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

---

f_percent *Data Transformation: Rowwise Percentages*

Description

Data Transformation: Rowwise Percentages

Usage

```
f_percent(x = NULL, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_percent(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

---

f_proportion *Data Transformation: Rowwise Proportions*

Description

Data Transformation: Rowwise Proportions

Usage

```
f_proportion(x = NULL, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_proportion(
  x = x
)

if(interactive()){
```

```
  zoo_plot(x)
  zoo_plot(y)
}
```

`f_proportion_sqrt` *Data Transformation: Rowwise Square Root of Proportions*

Description

Data Transformation: Rowwise Square Root of Proportions

Usage

```
f_proportion_sqrt(x = NULL, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: `f_binary()`, `f_clr()`, `f_detrend_difference()`, `f_detrend_linear()`, `f_detrend_poly()`, `f_hellinger()`, `f_list()`, `f_log()`, `f_percent()`, `f_proportion()`, `f_rescale_global()`, `f_rescale_local()`, `f_scale_global()`, `f_scale_local()`, `f_trend_linear()`, `f_trend_poly()`

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_proportion_sqrt(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

f_rescale_global *Data Transformation: Global Rescaling of to a New Range*

Description

Data Transformation: Global Rescaling of to a New Range

Usage

```
f_rescale_global(
  x = NULL,
  new_min = 0,
  new_max = 1,
  old_min = NULL,
  old_max = NULL,
  .global,
  ...
)
```

Arguments

x	(required, zoo object) Time Series. Default: NULL
new_min	(optional, numeric) New minimum value. Default: 0
new_max	(optional_numeric) New maximum value. Default: 1
old_min	(optional, numeric) Old minimum value. Default: NULL
old_max	(optional_numeric) Old maximum value. Default: NULL
.global	(optional, logical) Used to trigger global scaling within tsl_transform().
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_rescale_global(
  x = x,
  new_min = 0,
```

```
  new_max = 100
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

`f_rescale_local` *Data Transformation: Local Rescaling of to a New Range*

Description

Data Transformation: Local Rescaling of to a New Range

Usage

```
f_rescale_local(
  x = NULL,
  new_min = 0,
  new_max = 1,
  old_min = NULL,
  old_max = NULL,
  ...
)
```

Arguments

`x`	(required, zoo object) Time Series. Default: `NULL`
`new_min`	(optional, numeric) New minimum value. Default: `0`
`new_max`	(optional_numeric) New maximum value. Default: `1`
`old_min`	(optional, numeric) Old minimum value. Default: `NULL`
`old_max`	(optional_numeric) Old maximum value. Default: `NULL`
`...`	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: `f_binary()`, `f_clr()`, `f_detrend_difference()`, `f_detrend_linear()`, `f_detrend_poly()`, `f_hellinger()`, `f_list()`, `f_log()`, `f_percent()`, `f_proportion()`, `f_proportion_sqrt()`, `f_rescale_global()`, `f_scale_global()`, `f_scale_local()`, `f_trend_linear()`, `f_trend_poly()`

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_rescale_global(
  x = x,
  new_min = 0,
  new_max = 100
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

---

f_scale_global *Data Transformation: Global Centering and Scaling*

Description

Scaling and/or centering by variable using the mean and standard deviation computed across all time series. Global scaling helps dynamic time warping take variable offsets between time series into account.

Usage

```
f_scale_global(x = NULL, center = TRUE, scale = TRUE, .global, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
center	(optional, logical or numeric vector) Triggers centering if TRUE. Default: TRUE
scale	(optional, logical or numeric vector) Triggers scaling if TRUE. Default: TRUE
.global	(optional, logical) Used to trigger global scaling within tsl_transform().
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_local(), f_trend_linear(), f_trend_poly()

Examples

```
x <- zoo_simulate()

y <- f_scale_global(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

`f_scale_local` *Data Transformation: Local Centering and Scaling*

Description

Scaling and/or centering by variable and time series. Local scaling helps dynamic time warping focus entirely on shape comparisons.

Usage

```
f_scale_local(x = NULL, center = TRUE, scale = TRUE, ...)
```

Arguments

`x`	(required, zoo object) Zoo time series object to transform.
`center`	(optional, logical or numeric vector) Triggers centering if TRUE. Default: TRUE
`scale`	(optional, logical or numeric vector) Triggers scaling if TRUE. Default: TRUE
`...`	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: `f_binary()`, `f_clr()`, `f_detrend_difference()`, `f_detrend_linear()`, `f_detrend_poly()`, `f_hellinger()`, `f_list()`, `f_log()`, `f_percent()`, `f_proportion()`, `f_proportion_sqrt()`, `f_rescale_global()`, `f_rescale_local()`, `f_scale_global()`, `f_trend_linear()`, `f_trend_poly()`

Examples

```
x <- zoo_simulate()

y <- f_scale_global(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

`f_trend_linear` *Data Transformation: Linear Trend of Zoo Time Series*

Description

Fits a linear model on each column of a zoo object using time as a predictor, and predicts the outcome.

Usage

```
f_trend_linear(x = NULL, center = TRUE, ...)
```

Arguments

`x`	(required, zoo object) Zoo time series object to transform.
`center`	(required, logical) If TRUE, the output is centered at zero. If FALSE, it is centered at the data mean. Default: TRUE
`...`	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: `f_binary()`, `f_clr()`, `f_detrend_difference()`, `f_detrend_linear()`, `f_detrend_poly()`, `f_hellinger()`, `f_list()`, `f_log()`, `f_percent()`, `f_proportion()`, `f_proportion_sqrt()`, `f_rescale_global()`, `f_rescale_local()`, `f_scale_global()`, `f_scale_local()`, `f_trend_poly()`

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_trend_linear(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

---

f_trend_poly *Data Transformation: Polynomial Linear Trend of Zoo Time Series*

Description

Fits a polynomial linear model on each column of a zoo object using time as a predictor, and predicts the outcome to return the polynomial trend of the time series. This method is a useful alternative to f_trend_linear when the overall. trend of the time series does not follow a straight line.

Usage

```
f_trend_poly(x = NULL, degree = 2, center = TRUE, ...)
```

Arguments

x	(required, zoo object) Zoo time series object to transform.
degree	(optional, integer) Degree of the polynomial. Default: 2
center	(required, logical) If TRUE, the output is centered at zero. If FALSE, it is centered at the data mean. Default: TRUE
...	(optional, additional arguments) Ignored in this function.

Value

zoo object

See Also

Other tsl_transformation: f_binary(), f_clr(), f_detrend_difference(), f_detrend_linear(), f_detrend_poly(), f_hellinger(), f_list(), f_log(), f_percent(), f_proportion(), f_proportion_sqrt(), f_rescale_global(), f_rescale_local(), f_scale_global(), f_scale_local(), f_trend_linear()

Examples

```
x <- zoo_simulate(cols = 2)

y <- f_trend_poly(
  x = x
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(y)
}
```

---

honeycomb_climate *Rainfall and Temperature in The Americas*

Description

Monthly temperature and rainfall between 2009 and 2019 in 72 hexagonal cells covering The Americas.

Usage

```
data(honeycomb_climate)
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 9432 rows and 4 columns.

See Also

Other example_data: albatross, cities_coordinates, cities_temperature, covid_counties, covid_prevalence, eemian_coordinates, eemian_pollen, fagus_coordinates, fagus_dynamics, honeycomb_polygons

---

honeycomb_polygons *Hexagonal Grid*

Description

Sf data frame with hexagonal grid of the dataset honeycomb_climate.

Usage

```
data(honeycomb_polygons)
```

Format

An object of class `sf` (inherits from `data.frame`) with 72 rows and 2 columns.

See Also

Other example_data: `albatross`, `cities_coordinates`, `cities_temperature`, `covid_counties`, `covid_prevalence`, `eemian_coordinates`, `eemian_pollen`, `fagus_coordinates`, `fagus_dynamics`, `honeycomb_climate`

`importance_dtw_cpp` *(C++) Contribution of Individual Variables to the Dissimilarity Between Two Time Series (Robust Version)*

Description

Computes the contribution of individual variables to the similarity/dissimilarity between two irregular multivariate time series. In opposition to the legacy version, importance computation is performed taking the least-cost path of the whole sequence as reference. This operation makes the importance scores of individual variables fully comparable. This function generates a data frame with the following columns:

- variable: name of the individual variable for which the importance is being computed, from the column names of the arguments `x` and `y`.
- psi: global dissimilarity score `psi` of the two time series.
- psi_only_with: dissimilarity between `x` and `y` computed from the given variable alone.
- psi_without: dissimilarity between `x` and `y` computed from all other variables.
- psi_difference: difference between `psi_only_with` and `psi_without`.
- importance: contribution of the variable to the similarity/dissimilarity between `x` and `y`, computed as `(psi_difference * 100) / psi_all`. Positive scores represent contribution to dissimilarity, while negative scores represent contribution to similarity.

Usage

```
importance_dtw_cpp(
  x,
  y,
  distance = "euclidean",
  diagonal = TRUE,
  weighted = TRUE,
  ignore_blocks = FALSE,
  bandwidth = 1
)
```

Arguments

x	(required, numeric matrix) multivariate time series.
y	(required, numeric matrix) multivariate time series with the same number of columns as 'x'.
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".
diagonal	(optional, logical). If TRUE, diagonals are included in the computation of the cost matrix. Default: TRUE.
weighted	(optional, logical). Only relevant when diagonal is TRUE. When TRUE, diagonal cost is weighted by y factor of 1.414214 (square root of 2). Default: TRUE.
ignore_blocks	(optional, logical). If TRUE, blocks of consecutive path coordinates are trimmed to avoid inflating the psi distance. Default: FALSE.
bandwidth	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default. Default: 1

Value

data frame

See Also

Other Rcpp_importance: importance_dtw_legacy_cpp(), importance_ls_cpp()

Examples

```
#simulate two regular time series
x <- zoo_simulate(
  seed = 1,
  rows = 100
  )

y <- zoo_simulate(
  seed = 2,
  rows = 150
  )

#different number of rows
#this is not a requirement though!
nrow(x) == nrow(y)

#compute importance
df <- importance_dtw_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

df
```

`importance_dtw_legacy_cpp` *(C++) Contribution of Individual Variables to the Dissimilarity Between Two Time Series (Legacy Version)*

Description

Computes the contribution of individual variables to the similarity/dissimilarity between two irregular multivariate time series. In opposition to the robust version, least-cost paths for each combination of variables are computed independently, which makes the results of individual variables harder to compare. This function should only be used when the objective is replicating importance scores generated with previous versions of the package `distantia`. This function generates a data frame with the following columns:

- variable: name of the individual variable for which the importance is being computed, from the column names of the arguments `x` and `y`.
- psi: global dissimilarity score `psi` of the two time series.
- psi_only_with: dissimilarity between `x` and `y` computed from the given variable alone.
- psi_without: dissimilarity between `x` and `y` computed from all other variables.
- psi_difference: difference between `psi_only_with` and `psi_without`.
- importance: contribution of the variable to the similarity/dissimilarity between `x` and `y`, computed as `((psi_all - psi_without) * 100) / psi_all`. Positive scores represent contribution to dissimilarity, while negative scores represent contribution to similarity.

Usage

```
importance_dtw_legacy_cpp(
  y,
  x,
  distance = "euclidean",
  diagonal = FALSE,
  weighted = TRUE,
  ignore_blocks = FALSE,
  bandwidth = 1
)
```

Arguments

`y`	(required, numeric matrix) multivariate time series with the same number of columns as 'x'.
`x`	(required, numeric matrix) multivariate time series.
`distance`	(optional, character string) distance name from the "names" column of the dataset `distances` (see `distances$name`). Default: "euclidean".
`diagonal`	(optional, logical). If TRUE, diagonals are included in the computation of the cost matrix. Default: TRUE.

weighted	(optional, logical). Only relevant when diagonal is TRUE. When TRUE, diagonal cost is weighted by y factor of 1.414214 (square root of 2). Default: TRUE.
ignore_blocks	(optional, logical). If TRUE, blocks of consecutive path coordinates are trimmed to avoid inflating the psi distance. Default: FALSE.
bandwidth	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default. Default: 1

Value

data frame

See Also

Other Rcpp_importance: `importance_dtw_cpp()`, `importance_ls_cpp()`

Examples

```
#simulate two regular time series
x <- zoo_simulate(
  seed = 1,
  rows = 100
  )

y <- zoo_simulate(
  seed = 2,
  rows = 150
  )

#different number of rows
#this is not a requirement though!
nrow(x) == nrow(y)

#compute importance
df <- importance_dtw_legacy_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

df
```

`importance_ls_cpp` *(C++) Contribution of Individual Variables to the Dissimilarity Between Two Aligned Time Series*

Description

Computes the contribution of individual variables to the similarity/dissimilarity between two aligned multivariate time series. This function generates a data frame with the following columns:

- variable: name of the individual variable for which the importance is being computed, from the column names of the arguments `x` and `y`.
- psi: global dissimilarity score `psi` of the two time series.
- psi_only_with: dissimilarity between `x` and `y` computed from the given variable alone.
- psi_without: dissimilarity between `x` and `y` computed from all other variables.
- psi_difference: difference between `psi_only_with` and `psi_without`.
- importance: contribution of the variable to the similarity/dissimilarity between `x` and `y`, computed as `(psi_difference * 100) / psi_all`. Positive scores represent contribution to dissimilarity, while negative scores represent contribution to similarity.

Usage

```
importance_ls_cpp(x, y, distance = "euclidean")
```

Arguments

`x`	(required, numeric matrix) multivariate time series.
`y`	(required, numeric matrix) multivariate time series with the same number of columns and rows as 'x'.
`distance`	(optional, character string) distance name from the "names" column of the dataset `distances` (see `distances$name`). Default: "euclidean".

Value

data frame

See Also

Other Rcpp_importance: `importance_dtw_cpp()`, `importance_dtw_legacy_cpp()`

Examples

```
#simulate two regular time series
x <- zoo_simulate(
  seed = 1,
  irregular = FALSE
  )

y <- zoo_simulate(
  seed = 2,
  irregular = FALSE
  )

#same number of rows
```

```
nrow(x) == nrow(y)

#compute importance
df <- importance_ls_cpp(
  x = x,
  y = y,
  distance = "euclidean"
)

df
```

---

momentum *Contribution of Individual Variables to Time Series Dissimilarity*

Description

This function measures the contribution of individual variables to the dissimilarity between pairs of time series to help answer the question *what makes two time series more or less similar?*

Three key values are required to assess individual variable contributions:

- **psi**: dissimilarity when all variables are considered.
- **psi_only_with**: dissimilarity when using only the target variable.
- **psi_without**: dissimilarity when removing the target variable.

The values `psi_only_with` and `psi_without` can be computed in two different ways defined by the argument `robust`.

- `robust = FALSE`: This method replicates the importance algorithm released with the first version of the package, and it is only recommended when the goal to compare new results with previous studies. It normalizes `psi_only_with` and `psi_without` using the least cost path obtained from the individual variable. As different variables may have different least cost paths for the same time series, normalization values may change from variable to variable, making individual importance scores harder to compare.
- `robust = TRUE` (default, recommended): This a novel version of the importance algorithm that yields more stable and comparable solutions. It uses the least cost path of the complete time series to normalize `psi_only_with` and `psi_without`, making importance scores of separate variables fully comparable.

The individual importance score of each variable (column "importance" in the output data frame) is based on different expressions depending on the `robust` argument, even when `lock_step = TRUE`:

- `robust = FALSE`: Importance is computed as `((psi - psi_without) * 100)/psi` and interpreted as "change in similarity when a variable is removed".
- `robust = TRUE`: Importance is computed as `((psi_only_with - psi_without) * 100)/psi` and interpreted as "relative dissimilarity induced by the variable expressed as a percentage".

In either case, positive values indicate that the variable contributes to dissimilarity, while negative values indicate a net contribution to similarity.

This function allows computing dissimilarity between pairs of time series using different combinations of arguments at once. For example, when the argument `distance` is set to `c("euclidean", "manhattan")`, the output data frame will show two dissimilarity scores for each pair of time series, one based on euclidean distances, and another based on manhattan distances. The same happens for most other parameters.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
momentum(
  tsl = NULL,
  distance = "euclidean",
  diagonal = TRUE,
  bandwidth = 1,
  lock_step = FALSE,
  robust = TRUE
)
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".
`diagonal`	(optional, logical vector). If TRUE, diagonals are included in the dynamic time warping computation. Default: TRUE
`bandwidth`	(optional, numeric) Proportion of space at each side of the cost matrix diagonal (aka *Sakoe-Chiba band*) defining a valid region for dynamic time warping, used to control the flexibility of the warping path. This method prevents degenerate alignments due to differences in magnitude between time series when the data is not properly scaled. If `1` (default), DTW is unconstrained. If `0`, DTW is fully constrained and the warping path follows the matrix diagonal. Recommended values may vary depending on the nature of the data. Ignored if `lock_step = TRUE`. Default: 1.
`lock_step`	(optional, logical vector) If TRUE, time series captured at the same times are compared sample wise (with no dynamic time warping). Requires time series in argument `tsl` to be fully aligned, or it will return an error. Default: FALSE.
`robust`	(required, logical). If TRUE (default), importance scores are computed using the least cost path of the complete time series as reference. Setting it to FALSE allows to replicate importance scores of the previous versions of this package. This option is irrelevant when `lock_step = TRUE`. Default: TRUE

Value

data frame:

- x: name of the time series x.
- y: name of the time series y.
- psi: psi score of x and y.
- variable: name of the individual variable.
- importance: importance score of the variable.
- effect: interpretation of the "importance" column, with the values "increases similarity" and "decreases similarity".
- psi_only_with: psi score of the variable.
- psi_without: psi score without the variable.
- psi_difference: difference between psi_only_with and psi_without.
- distance: name of the distance metric.
- diagonal: value of the argument diagonal.
- lock_step: value of the argument lock_step.
- robust: value of the argument robust.

See Also

Other momentum: momentum_dtw(), momentum_ls()

Examples

```
#progress bar
# progressr::handlers(global = TRUE)

tsl <- tsl_initialize(
  x = distantia::albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

df <- momentum(
  tsl = tsl,
  lock_step = TRUE #to speed-up example
  )

#focus on important columns
df[, c(
  "x",
  "y",
  "variable",
  "importance",
```

```
  "effect"
  )]
```

`momentum_aggregate` *Aggregate* `momentum()` *Data Frames Across Parameter Combinations*

Description

The function `momentum()` allows variable importance assessments based on several combinations of arguments at once. For example, when the argument `distance` is set to `c("euclidean", "manhattan")`, the output data frame will show two importance scores for each pair of compared time series and variable, one based on euclidean distances, and another based on manhattan distances.

This function computes importance stats across combinations of parameters.

If there are no different combinations of arguments in the input data frame, no aggregation happens, but all parameter columns are removed.

Usage

```
momentum_aggregate(df = NULL, f = mean, ...)
```

Arguments

`df`	(required, data frame) Output of `momentum()`, `momentum_ls()`, or `momentum_dtw()`. Default: NULL
`f`	(optional, function) Function to summarize psi scores (for example, `mean`) when there are several combinations of parameters in `df`. Ignored when there is a single combination of arguments in the input. Default: `mean`
`...`	(optional, arguments of `f`) Further arguments to pass to the function `f`.

Value

data frame

See Also

Other momentum_support: `momentum_boxplot()`, `momentum_model_frame()`, `momentum_spatial()`, `momentum_stats()`, `momentum_to_wide()`

Examples

```
#three time series
#climate and ndvi in Fagus sylvatica stands in Spain, Germany, and Sweden
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

if(interactive()){
  tsl_plot(
    tsl = tsl,
    guide_columns = 3
    )
}

#momentum with multiple parameter combinations
#-------------------------------------
df <- momentum(
  tsl = tsl,
  distance = c("euclidean", "manhattan"),
  lock_step = TRUE
)

df[, c(
  "x",
  "y",
  "distance",
  "importance"
)]

#aggregation using means
df <- momentum_aggregate(
  df = df,
  f = mean
)

df
```

`momentum_boxplot` *Momentum Boxplot*

Description

Boxplot of a data frame returned by `momentum()` summarizing the contribution to similarity (negative) and/or dissimilarity (positive) of individual variables across all time series.

Usage

```
momentum_boxplot(df = NULL, fill_color = NULL, f = median, text_cex = 1)
```

Arguments

df	(required, data frame) Output of momentum(), momentum_ls(), or momentum_dtw(). Default: NULL
fill_color	(optional, character vector) boxplot fill color. Default: NULL
f	(optional, function) Function to summarize psi scores (for example, mean) when there are several combinations of parameters in df. Ignored when there is a single combination of arguments in the input. Default: mean
text_cex	(optional, numeric) Multiplier of the text size. Default: 1

Value

boxplot

See Also

Other momentum_support: momentum_aggregate(), momentum_model_frame(), momentum_spatial(), momentum_stats(), momentum_to_wide()

Examples

```
tsl <- tsl_initialize(
  x = distantia::albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

df <- momentum(
  tsl = tsl,
  lock_step = TRUE
  )

momentum_boxplot(
  df = df
  )
```

momentum_dtw *Dynamic Time Warping Variable Importance Analysis of Multivariate Time Series Lists*

Description

Minimalistic but slightly faster version of momentum() to compute dynamic time warping importance analysis with the "robust" setup in multivariate time series lists.

Usage

```
momentum_dtw(tsl = NULL, distance = "euclidean")
```

Arguments

tsl	(required, time series list) list of zoo time series. Default: NULL
distance	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

data frame:

- x: name of the time series x.
- y: name of the time series y.
- psi: psi score of x and y.
- variable: name of the individual variable.
- importance: importance score of the variable.
- effect: interpretation of the "importance" column, with the values "increases similarity" and "decreases similarity".

See Also

Other momentum: momentum(), momentum_ls()

Examples

```
tsl <- tsl_initialize(
  x = distantia::albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )
```

```
df <- momentum_dtw(
  tsl = tsl,
  distance = "euclidean"
  )

#focus on important columns
df[, c(
  "x",
  "y",
  "variable",
  "importance",
  "effect"
  )]
```

`momentum_ls` *Lock-Step Variable Importance Analysis of Multivariate Time Series Lists*

Description

Minimalistic but slightly faster version of `momentum()` to compute lock-step importance analysis in multivariate time series lists.

Usage

```
momentum_ls(tsl = NULL, distance = "euclidean")
```

Arguments

`tsl`	(required, time series list) list of zoo time series. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

data frame:

- `x`: name of the time series `x`.
- `y`: name of the time series `y`.
- `psi`: psi score of `x` and `y`.
- `variable`: name of the individual variable.
- `importance`: importance score of the variable.
- `effect`: interpretation of the "importance" column, with the values "increases similarity" and "decreases similarity".

See Also

Other momentum: momentum(), momentum_dtw()

Examples

```
tsl <- tsl_initialize(
  x = distantia::albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
  )

df <- momentum_ls(
  tsl = tsl,
  distance = "euclidean"
  )

#focus on important columns
df[, c(
  "x",
  "y",
  "variable",
  "importance",
  "effect"
  )]
```

---

momentum_model_frame *Dissimilarity Model Frame*

Description

This function generates a model frame for statistical or machine learning analysis from these objects:

- : Dissimilarity data frame generated by momentum(), momentum_ls(), or momentum_dtw(). The output model frame will have as many rows as this data frame.
- : Data frame with static descriptors of the time series. These descriptors are converted to distances between pairs of time series via distance_matrix().
- : List defining composite predictors. This feature allows grouping together predictors that have a common meaning. For example, composite_predictors = list(temperature = c("temperature_mean", "temp generates a new predictor named "temperature", which results from computing the multivariate distances between the vectors of temperature variables of each pair of time series. Predictors in one of such groups will be scaled before distance computation if their maximum standard deviations differ by a factor of 10 or more.

The resulting data frame contains the following columns:

- x and y: names of the pair of time series represented in the row.
- response columns.
- predictors columns: representing the distance between the values of the given static predictor between x and y.
- (optional) geographic_distance: If predictors_df is an sf data frame, then geographic distances are computed via sf::st_distance().

This function supports a parallelization setup via future::plan().

Usage

```
momentum_model_frame(
  response_df = NULL,
  predictors_df = NULL,
  composite_predictors = NULL,
  scale = TRUE,
  distance = "euclidean"
)
```

Arguments

response_df	(required, data frame) output of momentum(), momentum_ls(), or momentum_dtw(). Default: NULL
predictors_df	(required, data frame or sf data frame) data frame with numeric predictors for the the model frame. Must have a column with the time series names in response_df$x and response_df$y. If sf data frame, the column "geographic_distance" with distances between pairs of time series is added to the model frame. Default: NULL
composite_predictors	(optional, list) list defining composite predictors. For example, composite_predictors = list(a = c("b", "c")) uses the columns "b" and "c" from predictors_df to generate the predictor a as the multivariate distance between "b" and "c" for each pair of time series in response_df. Default: NULL
scale	(optional, logical) if TRUE, all predictors are scaled and centered with scale(). Default: TRUE
distance	(optional, string) Method to compute the distance between predictor values for all pairs of time series in response_df. Default: "euclidean".

Value

data frame: with the attribute "predictors".

See Also

Other momentum_support: momentum_aggregate(), momentum_boxplot(), momentum_spatial(), momentum_stats(), momentum_to_wide()

Examples

```
#Fagus sylvatica dynamics in Europe
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)

#dissimilarity analysis
df <- momentum_ls(tsl = tsl)

#generate model frame
model_frame <- momentum_model_frame(
  response_df = df,
  predictors_df = fagus_coordinates,
  scale = TRUE
)

head(model_frame)

#names of response and predictors
#and an additive formula
#are stored as attributes
attributes(model_frame)$predictors
```

---

momentum_spatial *Spatial Representation of* momentum() *Data Frames*

Description

Given an sf data frame with geometry types POLYGON, MULTIPOLYGON, or POINT representing time series locations, this function transforms the output of momentum(), momentum_ls(), momentum_dtw() to an sf data frame.

If network = TRUE, the sf data frame is of type LINESTRING, with edges connecting time series locations. This output is helpful to build many-to-many dissimilarity maps (see examples).

If network = FALSE, the sf data frame contains the geometry in the input sf argument. This output helps build one-to-many dissimilarity maps.

Usage

```
momentum_spatial(df = NULL, sf = NULL, network = TRUE)
```

Arguments

df	(required, data frame) Output of momentum(), momentum_ls(), or momentum_dtw(). Default: NULL

sf	(required, sf data frame) Points or polygons representing the location of the time series in argument 'df'. It must have a column with all time series names in df$x and df$y. Default: NULL
network	(optional, logical) If TRUE, the resulting sf data frame is of time LINESTRING and represent network edges. Default: TRUE

Value

sf data frame (LINESTRING geometry)

See Also

Other momentum_support: momentum_aggregate(), momentum_boxplot(), momentum_model_frame(), momentum_stats(), momentum_to_wide()

Examples

```
tsl <- distantia::tsl_initialize(
  x = distantia::eemian_pollen,
  name_column = "name",
  time_column = "time"
) |>
#reduce size to speed-up example runtime
distantia::tsl_subset(
  names = 1:3
  )

df_momentum <- distantia::momentum(
  tsl = tsl
)

#network many to many
sf_momentum <- distantia::momentum_spatial(
  df = df_momentum,
  sf = distantia::eemian_coordinates,
  network = TRUE
)

#network map
# mapview::mapview(
#   sf_momentum,
#   layer.name = "Importance - Abies",
#   label = "edge_name",
#   zcol = "importance__Abies",
#   lwd = 3
# ) |>
#   suppressWarnings()
```

momentum_stats *Stats of Dissimilarity Data Frame*

Description

Takes the output of `distantia()` to return a data frame with one row per time series with the stats of its dissimilarity scores with all other time series.

Usage

```
momentum_stats(df = NULL)
```

Arguments

df	(required, data frame) Output of `momentum()`, `momentum_ls()`, or `momentum_dtw()`. Default: NULL

Value

data frame

See Also

Other momentum_support: `momentum_aggregate()`, `momentum_boxplot()`, `momentum_model_frame()`, `momentum_spatial()`, `momentum_to_wide()`

Examples

```
tsl <- tsl_simulate(
  n = 5,
  irregular = FALSE
  )

df <- distantia(
  tsl = tsl,
  lock_step = TRUE
  )

df_stats <- distantia_stats(df = df)

df_stats
```

momentum_to_wide *Momentum Data Frame to Wide Format*

Description

Transforms a data frame returned by momentum() to wide format with the following columns:

- most_similar: name of the variable with the highest contribution to similarity (most negative value in the importance column) for each pair of time series.
- most_dissimilar: name of the variable with the highest contribution to dissimilarity (most positive value in the importance column) for each pair of time series.
- importance__variable_name: contribution to similarity (negative values) or dissimilarity (positive values) of the given variable.
- psi_only_with__variable_name: dissimilarity of the two time series when only using the given variable.
- psi_without__variable_name: dissimilarity of the two time series when removing the given variable.

Usage

```
momentum_to_wide(df = NULL, sep = "__")
```

Arguments

df	(required, data frame) Output of momentum(), momentum_ls(), or momentum_dtw(). Default: NULL
sep	(required, character string) Separator between the name of the importance metric and the time series variable. Default: "__".

Value

data frame

See Also

Other momentum_support: momentum_aggregate(), momentum_boxplot(), momentum_model_frame(), momentum_spatial(), momentum_stats()

Examples

```
tsl <- tsl_initialize(
  x = distantia::albatross,
  name_column = "name",
  time_column = "time"
) |>
  tsl_transform(
    f = f_scale_global
```

```
  )

#importance data frame
df <- momentum(
  tsl = tsl
)

df

#to wide format
df_wide <- momentum_to_wide(
  df = df
)

df_wide
```

`permute_free_by_row_cpp`

(C++) Unrestricted Permutation of Complete Rows

Description

Unrestricted shuffling of rows within the whole sequence.

Usage

```
permute_free_by_row_cpp(x, block_size, seed = 1L)
```

Arguments

`x`	(required, numeric matrix). Numeric matrix to permute.
`block_size`	(optional, integer) this function ignores this argument and sets it to x.nrow().
`seed`	(optional, integer) random seed to use.

Value

numeric matrix

See Also

Other Rcpp_permutation: `permute_free_cpp()`, `permute_restricted_by_row_cpp()`, `permute_restricted_cpp()`

`permute_free_cpp` *(C++) Unrestricted Permutation of Cases*

Description

Unrestricted shuffling of cases within the whole sequence.

Usage

```
permute_free_cpp(x, block_size, seed = 1L)
```

Arguments

`x`	(required, numeric matrix). Numeric matrix to permute.
`block_size`	(optional, integer) this function ignores this argument and sets it to x.nrow().
`seed`	(optional, integer) random seed to use.

Value

numeric matrix

See Also

Other Rcpp_permutation: `permute_free_by_row_cpp()`, `permute_restricted_by_row_cpp()`, `permute_restricted_cpp()`

`permute_restricted_by_row_cpp`
(C++) Restricted Permutation of Complete Rows Within Blocks

Description

Divides a sequence in blocks of a given size and permutes rows within these blocks. Larger block sizes increasingly disrupt the data structure over time.

Usage

```
permute_restricted_by_row_cpp(x, block_size, seed = 1L)
```

Arguments

`x`	(required, numeric matrix). Numeric matrix to permute.
`block_size`	(optional, integer) block size in number of rows. Minimum value is 2, and maximum value is nrow(x).
`seed`	(optional, integer) random seed to use.

Value

numeric matrix

See Also

Other Rcpp_permutation: `permute_free_by_row_cpp()`, `permute_free_cpp()`, `permute_restricted_cpp()`

`permute_restricted_cpp`

(C++) Restricted Permutation of Cases Within Blocks

Description

Divides a sequence or time series in blocks and permutes cases within these blocks. This function does not preserve rows, and should not be used if the sequence has dependent columns. Larger block sizes increasingly disrupt the data structure over time.

Usage

```
permute_restricted_cpp(x, block_size, seed = 1L)
```

Arguments

`x`	(required, numeric matrix). Numeric matrix to permute.
`block_size`	(optional, integer) block size in number of rows. Minimum value is 2, and maximum value is nrow(x).
`seed`	(optional, integer) random seed to use.

Value

numeric matrix

See Also

Other Rcpp_permutation: `permute_free_by_row_cpp()`, `permute_free_cpp()`, `permute_restricted_by_row_cpp()`

psi_auto_distance *Cumulative Sum of Distances Between Consecutive Cases in a Time Series*

Description

Demonstration function to compute the sum of distances between consecutive cases in a time series.

Usage

```
psi_auto_distance(x = NULL, distance = "euclidean")
```

Arguments

x	(required, zoo object or matrix) univariate or multivariate time series with no NAs. Default: NULL
distance	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

numeric value

See Also

Other psi_demo: psi_auto_sum(), psi_cost_matrix(), psi_cost_path(), psi_cost_path_sum(), psi_distance_lock_step(), psi_distance_matrix(), psi_equation()

Examples

```
#distance metric
d <- "euclidean"

#simulate zoo time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

#sum distance between consecutive samples
psi_auto_distance(
  x = x,
  distance = d
)
```

psi_auto_sum *Auto Sum*

Description

Demonstration function to computes the sum of distances between consecutive samples in two time series.

Usage

```
psi_auto_sum(x = NULL, y = NULL, distance = "euclidean")
```

Arguments

x	(required, zoo object or numeric matrix) univariate or multivariate time series with no NAs. Default: NULL.
y	(required, zoo object or numeric matrix) a time series with the same number of columns as x and no NAs. Default: NULL.
distance	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

numeric vector

See Also

Other psi_demo: psi_auto_distance(), psi_cost_matrix(), psi_cost_path(), psi_cost_path_sum(), psi_distance_lock_step(), psi_distance_matrix(), psi_equation()

Examples

```
#distance metric
d <- "euclidean"

#simulate two irregular time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

y <- zoo_simulate(
  name = "y",
  rows = 80,
  seasons = 2,
```

```
  seed = 2
)

if(interactive()){
  zoo_plot(x = x)
  zoo_plot(x = y)
}

#auto sum of distances
psi_auto_sum(
  x = x,
  y = y,
  distance = d
)

#same as:
x_sum <- psi_auto_distance(
  x = x,
  distance = d
)

y_sum <- psi_auto_distance(
  x = y,
  distance = d
)

x_sum + y_sum
```

`psi_cost_matrix` *Cost Matrix*

Description

Demonstration function to compute a cost matrix from a distance matrix.

Usage

```
psi_cost_matrix(dist_matrix = NULL, diagonal = TRUE)
```

Arguments

`dist_matrix`	(required, numeric matrix). Distance matrix generated by `psi_distance_matrix()`. Default: NULL
`diagonal`	(optional, logical vector). If TRUE, diagonals are included in the dynamic time warping computation. Default: TRUE

Value

numeric matrix

See Also

Other psi_demo: psi_auto_distance(), psi_auto_sum(), psi_cost_path(), psi_cost_path_sum(), psi_distance_lock_step(), psi_distance_matrix(), psi_equation()

Examples

```
#distance metric
d <- "euclidean"

#use diagonals in least cost computations
diagonal <- TRUE

#simulate two irregular time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

y <- zoo_simulate(
  name = "y",
  rows = 80,
  seasons = 2,
  seed = 2
)

if(interactive()){
  zoo_plot(x = x)
  zoo_plot(x = y)
}

#distance matrix
dist_matrix <- psi_distance_matrix(
  x = x,
  y = y,
  distance = d
)

#cost matrix
cost_matrix <- psi_cost_matrix(
  dist_matrix = dist_matrix,
  diagonal = diagonal
)

if(interactive()){
  utils_matrix_plot(
    m = cost_matrix
    )
}
```

psi_cost_path *Least Cost Path*

Description

Demonstration function to compute the least cost path within a least cost matrix.

Usage

```
psi_cost_path(
  dist_matrix = NULL,
  cost_matrix = NULL,
  diagonal = TRUE,
  bandwidth = 1
)
```

Arguments

dist_matrix	(required, numeric matrix) Distance matrix generated by psi_distance_matrix(). Default: NULL
cost_matrix	(required, numeric matrix) Cost matrix generated from the distance matrix with psi_cost_matrix(). Default: NULL
diagonal	(optional, logical vector). If TRUE, diagonals are included in the dynamic time warping computation. Default: TRUE
bandwidth	(optional, numeric) Proportion of space at each side of the cost matrix diagonal (aka *Sakoe-Chiba band*) defining a valid region for dynamic time warping, used to control the flexibility of the warping path. This method prevents degenerate alignments due to differences in magnitude between time series when the data is not properly scaled. If 1 (default), DTW is unconstrained. If 0, DTW is fully constrained and the warping path follows the matrix diagonal. Recommended values may vary depending on the nature of the data. Ignored if lock_step = TRUE. Default: 1.

Value

data frame

See Also

Other psi_demo: psi_auto_distance(), psi_auto_sum(), psi_cost_matrix(), psi_cost_path_sum(), psi_distance_lock_step(), psi_distance_matrix(), psi_equation()

Examples

```
#distance metric
d <- "euclidean"

#simulate two irregular time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

y <- zoo_simulate(
  name = "y",
  rows = 80,
  seasons = 2,
  seed = 2
)

if(interactive()){
  zoo_plot(x = x)
  zoo_plot(x = y)
}

#distance matrix
dist_matrix <- psi_distance_matrix(
  x = x,
  y = y,
  distance = d
)

#diagonal least cost path
#------------------------

cost_matrix <- psi_cost_matrix(
  dist_matrix = dist_matrix,
  diagonal = TRUE
)

cost_path <- psi_cost_path(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix,
  diagonal = TRUE
)

if(interactive()){
  utils_matrix_plot(
    m = cost_matrix,
    path = cost_path
    )
}
```

```
#orthogonal least cost path
#-------------------------
cost_matrix <- psi_cost_matrix(
  dist_matrix = dist_matrix,
  diagonal = FALSE
)

cost_path <- psi_cost_path(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix,
  diagonal = FALSE
)

if(interactive()){
  utils_matrix_plot(
    m = cost_matrix,
    path = cost_path
  )
}
```

---

psi_cost_path_sum *Sum of Distances in Least Cost Path*

Description

Demonstration function to sum the distances of a least cost path.

Usage

```
psi_cost_path_sum(path = NULL)
```

Arguments

path	(required, data frame) least cost path produced by psi_cost_path(). Default: NULL

Value

numeric value

See Also

Other psi_demo: psi_auto_distance(), psi_auto_sum(), psi_cost_matrix(), psi_cost_path(), psi_distance_lock_step(), psi_distance_matrix(), psi_equation()

Examples

```
#distance metric
d <- "euclidean"

#simulate two irregular time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

y <- zoo_simulate(
  name = "y",
  rows = 80,
  seasons = 2,
  seed = 2
)

if(interactive()){
  zoo_plot(x = x)
  zoo_plot(x = y)
}

#distance matrix
dist_matrix <- psi_distance_matrix(
  x = x,
  y = y,
  distance = d
)

#orthogonal least cost matrix
cost_matrix <- psi_cost_matrix(
  dist_matrix = dist_matrix
)

#orthogonal least cost path
cost_path <- psi_cost_path(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix
)

#sum of distances in cost path
psi_cost_path_sum(
  path = cost_path
)
```

`psi_distance_lock_step` *Lock-Step Distance*

Description

Demonstration function to compute the lock-step distance between two univariate or multivariate time series.

This function does not accept NA data in the matrices x and y.

Usage

```
psi_distance_lock_step(x = NULL, y = NULL, distance = "euclidean")
```

Arguments

x	(required, zoo object or numeric matrix) a time series with no NAs. Default: NULL
y	(zoo object or numeric matrix) a time series with the same columns as x and no NAs. Default: NULL
distance	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

numeric

See Also

Other psi_demo: psi_auto_distance(), psi_auto_sum(), psi_cost_matrix(), psi_cost_path(), psi_cost_path_sum(), psi_distance_matrix(), psi_equation()

Examples

```
#distance metric
d <- "euclidean"

#simulate two time series
#of the same length
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

y <- zoo_simulate(
  name = "y",
  rows = 100,
  seasons = 2,
  seed = 2
)

if(interactive()){
```

```
  zoo_plot(x = x)
  zoo_plot(x = y)
}

#sum of distances
#between pairs of samples
psi_distance_lock_step(
  x = x,
  y = y,
  distance = d
)
```

`psi_distance_matrix` *Distance Matrix*

Description

Demonstration function to compute the distance matrix between two univariate or multivariate time series.

This function does not accept NA data in the matrices `x` and `y`.

Usage

```
psi_distance_matrix(x = NULL, y = NULL, distance = "euclidean")
```

Arguments

`x`	(required, zoo object or numeric matrix) a time series with no NAs. Default: NULL
`y`	(zoo object or numeric matrix) a time series with the same columns as `x` and no NAs. Default: NULL
`distance`	(optional, character vector) name or abbreviation of the distance method. Valid values are in the columns "names" and "abbreviation" of the dataset distances. Default: "euclidean".

Value

numeric matrix

See Also

Other psi_demo: `psi_auto_distance()`, `psi_auto_sum()`, `psi_cost_matrix()`, `psi_cost_path()`, `psi_cost_path_sum()`, `psi_distance_lock_step()`, `psi_equation()`

Examples

```
#distance metric
d <- "euclidean"

#simulate two irregular time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
)

y <- zoo_simulate(
  name = "y",
  rows = 80,
  seasons = 2,
  seed = 2
)

if(interactive()){
  zoo_plot(x = x)
  zoo_plot(x = y)
}

#distance matrix
dist_matrix <- psi_distance_matrix(
  x = x,
  y = y,
  distance = d
)

if(interactive()){
  utils_matrix_plot(
    m = dist_matrix
    )
}
```

`psi_dtw_cpp` *(C++) Psi Dissimilarity Score of Two Time-Series*

Description

Computes the psi score of two time series `y` and `x` with the same number of columns. NA values should be removed before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
psi_dtw_cpp(
```

```
  x,
  y,
  distance = "euclidean",
  diagonal = TRUE,
  weighted = TRUE,
  ignore_blocks = FALSE,
  bandwidth = 1
)
```

Arguments

x	(required, numeric matrix) of same number of columns as 'y'.
y	(required, numeric matrix) time series.
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".
diagonal	(optional, logical). If TRUE, diagonals are included in the computation of the cost matrix. Default: TRUE.
weighted	(optional, logical). Only relevant when diagonal is TRUE. When TRUE, diagonal cost is weighted by y factor of 1.414214 (square root of 2). Default: TRUE.
ignore_blocks	(optional, logical). If TRUE, blocks of consecutive path coordinates are trimmed to avoid inflating the psi distance. Default: FALSE.
bandwidth	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default.

Value

numeric

See Also

Other Rcpp_dissimilarity_analysis: psi_equation_cpp(), psi_ls_cpp(), psi_null_dtw_cpp(), psi_null_ls_cpp()

---

psi_equation *Normalized Dissimilarity Score*

Description

Demonstration function to computes the psi dissimilarity score (Birks and Gordon 1985). Psi is computed as $\psi = (2a/b) - 1$, where a is the sum of distances between the relevant samples of two time series, and b is the cumulative sum of distances between consecutive samples in the two time series.

If a is computed with dynamic time warping, and diagonals are used in the computation of the least cost path, then one is added to the result of the equation above.

Usage

```
psi_equation(a = NULL, b = NULL, diagonal = TRUE)
```

Arguments

a	(required, numeric) Result of psi_cost_path_sum(), the sum of distances of the least cost path between two time series. Default: NULL
b	(required, numeric) Result of psi_auto_sum(), the cumulative sum of the consecutive cases of two time series. Default: NULL
diagonal	(optional, logical) Used to correct psi when diagonals are used during the computation of the least cost path. If the cost matrix and least cost path were computed using diagonal = TRUE, this argument should be TRUE as well. Default: TRUE

Value

numeric value

See Also

Other psi_demo: psi_auto_distance(), psi_auto_sum(), psi_cost_matrix(), psi_cost_path(), psi_cost_path_sum(), psi_distance_lock_step(), psi_distance_matrix()

Examples

```
#distance metric
d <- "euclidean"

#use diagonals in least cost computations
diagonal <- TRUE

#simulate two irregular time series
x <- zoo_simulate(
  name = "x",
  rows = 100,
  seasons = 2,
  seed = 1
  )

y <- zoo_simulate(
  name = "y",
  rows = 80,
  seasons = 2,
  seed = 2
  )

if(interactive()){
  zoo_plot(x = x)
  zoo_plot(x = y)
}
```

```
#dynamic time warping

#distance matrix
dist_matrix <- psi_distance_matrix(
  x = x,
  y = y,
  distance = d
)

#cost matrix
cost_matrix <- psi_cost_matrix(
  dist_matrix = dist_matrix,
  diagonal = diagonal
)

#least cost path
cost_path <- psi_cost_path(
  dist_matrix = dist_matrix,
  cost_matrix = cost_matrix,
  diagonal = diagonal
)

if(interactive()){
  utils_matrix_plot(
    m = cost_matrix,
    path = cost_path
    )
}


#computation of psi score

#sum of distances in least cost path
a <- psi_cost_path_sum(
  path = cost_path
  )

#auto sum of both time series
b <- psi_auto_sum(
  x = x,
  y = y,
  distance = d
)

#dissimilarity score
psi_equation(
  a = a,
  b = b,
  diagonal = diagonal
)

#full computation with distantia()
```

```
tsl <- list(
  x = x,
  y = y
)

distantia(
  tsl = tsl,
  distance = d,
  diagonal = diagonal
)$psi

if(interactive()){
  distantia_dtw_plot(
    tsl = tsl,
    distance = d,
    diagonal = diagonal
  )
}
```

`psi_equation_cpp` *(C++) Equation of the Psi Dissimilarity Score*

Description

Equation to compute the `psi` dissimilarity score (Birks and Gordon 1985). Psi is computed as $\psi = (2a/b) - 1$, where a is the sum of distances between the relevant samples of two time series, and b is the cumulative sum of distances between consecutive samples in the two time series. If `a` is computed with dynamic time warping, and diagonals are used in the computation of the least cost path, then one is added to the result of the equation above.

Usage

```
psi_equation_cpp(a, b, diagonal = TRUE)
```

Arguments

`a`	(required, numeric) output of `cost_path_sum_cpp()` on a least cost path.
`b`	(required, numeric) auto sum of both sequences, result of `auto_sum_cpp()`.
`diagonal`	(optional, logical). Must be TRUE when diagonals are used in dynamic time warping and for lock-step distances. Default: FALSE.

Value

numeric

See Also

Other Rcpp_dissimilarity_analysis: `psi_dtw_cpp()`, `psi_ls_cpp()`, `psi_null_dtw_cpp()`, `psi_null_ls_cpp()`

psi_ls_cpp *(C++) Psi Dissimilarity Score of Two Aligned Time Series*

Description

Computes the psi dissimilarity score between two time series observed at the same times. Time series `y` and `x` with the same number of columns and rows. NA values should be removed before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
psi_ls_cpp(x, y, distance = "euclidean")
```

Arguments

`x`	(required, numeric matrix) of same number of columns as 'y'.
`y`	(required, numeric matrix) of same number of columns as 'x'.
`distance`	(optional, character string) distance name from the "names" column of the dataset `distances` (see `distances$name`). Default: "euclidean".

Value

numeric

See Also

Other Rcpp_dissimilarity_analysis: `psi_dtw_cpp()`, `psi_equation_cpp()`, `psi_null_dtw_cpp()`, `psi_null_ls_cpp()`

psi_null_dtw_cpp *(C++) Null Distribution of Dissimilarity Scores of Two Time Series*

Description

Applies permutation methods to compute null distributions for the psi scores of two time series. NA values should be removed before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
psi_null_dtw_cpp(
  x,
  y,
  distance = "euclidean",
  diagonal = TRUE,
  weighted = TRUE,
  ignore_blocks = FALSE,
  bandwidth = 1,
  repetitions = 100L,
  permutation = "restricted_by_row",
  block_size = 3L,
  seed = 1L
)
```

Arguments

x	(required, numeric matrix) of same number of columns as 'y'.
y	(required, numeric matrix).
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".
diagonal	(optional, logical). If TRUE, diagonals are included in the computation of the cost matrix. Default: FALSE.
weighted	(optional, logical). If TRUE, diagonal is set to TRUE, and diagonal cost is weighted by a factor of 1.414214 (square root of 2). Default: FALSE.
ignore_blocks	(optional, logical). If TRUE, blocks of consecutive path coordinates are trimmed to avoid inflating the psi distance. This argument has nothing to do with block_size!. Default: FALSE.
bandwidth	(required, numeric) Size of the Sakoe-Chiba band at both sides of the diagonal used to constrain the least cost path. Expressed as a fraction of the number of matrix rows and columns. Unrestricted by default. Default: 1
repetitions	(optional, integer) number of null psi values to generate. Default: 100
permutation	(optional, character) permutation method. Valid values are listed below from higher to lower randomness: • "free": unrestricted shuffling of rows and columns. Ignores block_size. • "free_by_row": unrestricted shuffling of complete rows. Ignores block size. • "restricted": restricted shuffling of rows and columns within blocks. • "restricted_by_row": restricted shuffling of rows within blocks.
block_size	(optional, integer) block size in rows for restricted permutation. A block size of 3 indicates that a row can only be permuted within a block of 3 adjacent rows. Minimum value is 2. Default: 3.
seed	(optional, integer) initial random seed to use for replicability. Default: 1

Value

numeric vector

See Also

Other Rcpp_dissimilarity_analysis: psi_dtw_cpp(), psi_equation_cpp(), psi_ls_cpp(), psi_null_ls_cpp()

---

psi_null_ls_cpp *(C++) Null Distribution of the Dissimilarity Scores of Two Aligned Time Series*

Description

Applies permutation methods to compute null distributions for the psi scores of two time series observed at the same times. NA values should be removed before using this function. If the selected distance function is "chi" or "cosine", pairs of zeros should be either removed or replaced with pseudo-zeros (i.e. 0.00001).

Usage

```
psi_null_ls_cpp(
  x,
  y,
  distance = "euclidean",
  repetitions = 100L,
  permutation = "restricted_by_row",
  block_size = 3L,
  seed = 1L
)
```

Arguments

x	(required, numeric matrix) of same number of columns as 'y'.
y	(required, numeric matrix) of same number of columns as 'x'.
distance	(optional, character string) distance name from the "names" column of the dataset distances (see distances$name). Default: "euclidean".
repetitions	(optional, integer) number of null psi values to generate. Default: 100
permutation	(optional, character) permutation method. Valid values are listed below from higher to lower randomness: • "free": unrestricted shuffling of rows and columns. Ignores block_size. • "free_by_row": unrestricted shuffling of complete rows. Ignores block size. • "restricted": restricted shuffling of rows and columns within blocks. • "restricted_by_row": restricted shuffling of rows within blocks.
block_size	(optional, integer) block size in rows for restricted permutation. A block size of 3 indicates that a row can only be permuted within a block of 3 adjacent rows. Minimum value is 2. Default: 3.
seed	(optional, integer) initial random seed to use for replicability. Default: 1

Value

numeric vector

See Also

Other Rcpp_dissimilarity_analysis: psi_dtw_cpp(), psi_equation_cpp(), psi_ls_cpp(), psi_null_dtw_cpp()

---

subset_matrix_by_rows_cpp

(C++) Subset Matrix by Rows

Description

Subsets a time series matrix to the coordinates of a trimmed least-cost path when blocks are ignored during a dissimilarity analysis.

Usage

```
subset_matrix_by_rows_cpp(m, rows)
```

Arguments

m	(required, numeric matrix) a univariate or multivariate time series.
rows	(required, integer vector) vector of rows to subset from a least-cost path data frame.

Value

numeric matrix

See Also

Other Rcpp_auto_sum: auto_distance_cpp(), auto_sum_cpp(), auto_sum_full_cpp(), auto_sum_path_cpp()

Examples

```
#simulate a time series
m <- zoo_simulate(seed = 1)

#sample some rows
rows <- sample(
  x = nrow(m),
  size = 10
  ) |>
  sort()

#subset by rows
m_subset <- subset_matrix_by_rows_cpp(
```

```
  m = m,
  rows = rows
  )

#compare with original
m[rows, ]
```

`tsl_aggregate` *Aggregate Time Series List Over Time Periods*

Description

Time series aggregation involves grouping observations and summarizing group values with a statistical function. This operation is useful to:

- Decrease (downsampling) the temporal resolution of a time series.
- Highlight particular states of a time series over time. For example, a daily temperature series can be aggregated by month using `max` to represent the highest temperatures each month.
- Transform irregular time series into regular.

This function aggregates time series lists **with overlapping times**. Please check such overlap by assessing the columns "begin" and "end " of the data frame resulting from `df <- tsl_time(tsl = tsl)`. Aggregation will be limited by the shortest time series in your time series list. To aggregate non-overlapping time series, please subset the individual components of `tsl` one by one either using `tsl_subset()` or the syntax `tsl = my_tsl[[i]]`.

Methods

Any function returning a single number from a numeric vector can be used to aggregate a time series list. Quoted and unquoted function names can be used. Additional arguments to these functions can be passed via the argument `...`. Typical examples are:

- `mean` or `"mean"`: see `mean()`.
- `median` or `"median"`: see `stats::median()`.
- `quantile` or "quantile": see `stats::quantile()`.
- `min` or `"min"`: see `min()`.
- `max` or `"max"`: see `max()`.
- `sd` or `"sd"`: to compute standard deviation, see `stats::sd()`.
- `var` or `"var"`: to compute the group variance, see `stats::var()`.
- `length` or `"length"`: to compute group length.
- `sum` or `"sum"`: see `sum()`.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
tsl_aggregate(tsl = NULL, new_time = NULL, f = mean, ...)
```

Arguments

tsl	(required, list) Time series list. Default: NULL
new_time	(required, numeric, numeric vector, Date vector, POSIXct vector, or keyword) Definition of the aggregation pattern. The available options are: • numeric vector: only for the "numeric" time class, defines the breakpoints for time series aggregation. • "Date" or "POSIXct" vector: as above, but for the time classes "Date" and "POSIXct." In any case, the input vector is coerced to the time class of the tsl argument. • numeric: defines fixed time intervals in the units of tsl for time series aggregation. Used as is when the time class is "numeric", and coerced to integer and interpreted as days for the time classes "Date" and "POSIXct". • keyword (see utils_time_units()): the common options for the time classes "Date" and "POSIXct" are: "millennia", "centuries", "decades", "years", "quarters", "months", and "weeks". Exclusive keywords for the "POSIXct" time class are: "days", "hours", "minutes", and "seconds". The time class "numeric" accepts keywords coded as scientific numbers, from "1e8" to "1e-8".
f	(required, function name) Name of function taking a vector as input and returning a single value as output. Typical examples are mean, max,min, median, and quantile. Default: mean.
...	(optional) further arguments for f.

Value

time series list

See Also

zoo_aggregate()

Other tsl_processing: tsl_resample(), tsl_smooth(), tsl_stats(), tsl_transform()

Examples

```
# yearly aggregation
#----------------------------------
#long-term monthly temperature of 20 cities
tsl <- tsl_initialize(
  x = cities_temperature,
  name_column = "name",
  time_column = "time"
)

#plot time series
```

```
if(interactive()){
  tsl_plot(
    tsl = tsl[1:4],
    guide_columns = 4
  )
}

#check time features
tsl_time(tsl)[, c("name", "resolution", "units")]

#aggregation: mean yearly values
tsl_year <- tsl_aggregate(
  tsl = tsl,
  new_time = "year",
  f = mean
)

#' #check time features
tsl_time(tsl_year)[, c("name", "resolution", "units")]

if(interactive()){
  tsl_plot(
    tsl = tsl_year[1:4],
    guide_columns = 4
  )
}


# other supported keywords
#----------------------------------

#simulate full range of calendar dates
tsl <- tsl_simulate(
  n = 2,
  rows = 1000,
  time_range = c(
    "0000-01-01",
    as.character(Sys.Date())
  )
)

#mean value by millennia (extreme case!!!)
tsl_millennia <- tsl_aggregate(
  tsl = tsl,
  new_time = "millennia",
  f = mean
)

if(interactive()){
  tsl_plot(tsl_millennia)
}

#max value by centuries
```

```
tsl_century <- tsl_aggregate(
  tsl = tsl,
  new_time = "century",
  f = max
)

if(interactive()){
  tsl_plot(tsl_century)
}

#quantile 0.75 value by centuries
tsl_centuries <- tsl_aggregate(
  tsl = tsl,
  new_time = "centuries",
  f = stats::quantile,
  probs = 0.75 #argument of stats::quantile()
)
```

`tsl_burst` *Multivariate TSL to Univariate TSL*

Description

Takes a time series list with multivariate zoo objects to generate a new one with one univariate zoo objects per variable. A time series list with the the zoo objects "A" and "B", each with the columns "a", "b", and "c", becomes a time series list with the zoo objects "A__a", "A__b", "A__c", "B__a", "B__b", and "B__c". The only column of each new zoo object is named "x".

Usage

```
tsl_burst(tsl = NULL, sep = "__")
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`sep`	(required, character string) separator between the time series name and the column name. Default: "__"

Value

time series list: list of univariate zoo objects with a column named "x".

See Also

Other tsl_management: `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
tsl <- tsl_simulate(
  n = 2,
  time_range = c(
    "2010-01-01",
    "2024-12-31"
  ),
  cols = 3
)

tsl_names_get(tsl)
tsl_colnames_get(tsl)

if(interactive()){
  tsl_plot(tsl)
}

tsl <- tsl_burst(tsl)

tsl_names_get(tsl)
tsl_colnames_get(tsl)

if(interactive()){
  tsl_plot(tsl)
}
```

---

tsl_colnames_clean *Clean Column Names in Time Series Lists*

Description

Uses the function `utils_clean_names()` to simplify and normalize messy column names in a time series list.

The cleanup operations are applied in the following order:

- Remove leading and trailing whitespaces.
- Generates syntactically valid names with `base::make.names()`.
- Replaces dots and spaces with the `separator`.
- Coerces names to lowercase.
- If `capitalize_first = TRUE`, the first letter is capitalized.
- If `capitalize_all = TRUE`, all letters are capitalized.
- If argument `length` is provided, `base::abbreviate()` is used to abbreviate the new column names.
- If `suffix` is provided, it is added at the end of the column name using the separator.
- If `prefix` is provided, it is added at the beginning of the column name using the separator.

Usage

```
tsl_colnames_clean(
  tsl = NULL,
  lowercase = FALSE,
  separator = "_",
  capitalize_first = FALSE,
  capitalize_all = FALSE,
  length = NULL,
  suffix = NULL,
  prefix = NULL
)
```

Arguments

tsl	(required, list) Time series list. Default: NULL
lowercase	(optional, logical) If TRUE, all names are coerced to lowercase. Default: FALSE
separator	(optional, character string) Separator when replacing spaces and dots. Also used to separate suffix and prefix from the main word. Default: "_".
capitalize_first	(optional, logical) Indicates whether to capitalize the first letter of each name Default: FALSE.
capitalize_all	(optional, logical) Indicates whether to capitalize all letters of each name Default: FALSE.
length	(optional, integer) Minimum length of abbreviated names. Names are abbreviated via abbreviate(). Default: NULL.
suffix	(optional, character string) String to append to the column names. Default: NULL.
prefix	(optional, character string) String to prepend to the column names. Default: NULL.

Value

time series list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#generate example data
tsl <- tsl_simulate(cols = 3)

#list all column names
```

```
tsl_colnames_get(
  tsl = tsl
)

#rename columns
tsl <- tsl_colnames_set(
  tsl = tsl,
  names = c(
  "New name 1",
  "new Name 2",
  "NEW NAME 3"
  )
)

#check new names
tsl_colnames_get(
  tsl = tsl,
  names = "all"
)

#clean names
tsl <- tsl_colnames_clean(
  tsl = tsl
)

tsl_colnames_get(
  tsl = tsl
)

#abbreviated
tsl <- tsl_colnames_clean(
  tsl = tsl,
  capitalize_first = TRUE,
  length = 6,
  suffix = "clean"
)

tsl_colnames_get(
  tsl = tsl
)
```

`tsl_colnames_get` *Get Column Names from a Time Series Lists*

Description

Get Column Names from a Time Series Lists

Usage

```
tsl_colnames_get(tsl = NULL, names = c("all", "shared", "exclusive"))
```

Arguments

tsl	(required, list) Time series list. Default: NULL
names	(optional, character string) Three different sets of column names can be requested: • "all" (default): list with the column names in each zoo object in tsl. Unnamed columns are tagged with the string "unnamed". • "shared": character vector with the shared column names in at least two zoo objects in tsl. • "exclusive": list with names of exclusive columns (if any) in each zoo object in tsl.

Value

list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#generate example data
tsl <- tsl_simulate()

#list all column names
tsl_colnames_get(
  tsl = tsl,
  names = "all"
)

#change one column name
names(tsl[[1]])[1] <- "new_column"

#all names again
tsl_colnames_get(
  tsl = tsl,
  names = "all"
)

#shared column names
tsl_colnames_get(
  tsl = tsl,
  names = "shared"
)

#exclusive column names
tsl_colnames_get(
```

```
  tsl = tsl,
  names = "exclusive"
)
```

`tsl_colnames_prefix` *Append Prefix to Column Names of Time Series List*

Description

Append Prefix to Column Names of Time Series List

Usage

```
tsl_colnames_prefix(tsl = NULL, prefix = NULL)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`prefix`	(optional, character string) String to prepend to the column names. Default: NULL.

Value

time series list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
tsl <- tsl_simulate()

tsl_colnames_get(tsl = tsl)

tsl <- tsl_colnames_prefix(
  tsl = tsl,
  prefix = "my_prefix_"
)

tsl_colnames_get(tsl = tsl)
```

tsl_colnames_set *Set Column Names in Time Series Lists*

Description

Set Column Names in Time Series Lists

Usage

```
tsl_colnames_set(tsl = NULL, names = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL
names	(required, list or character vector): • list: with same names as 'tsl', containing a vector of new column names for each time series in 'tsl'. • character vector: vector of new column names assigned by position.

Value

time series list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
tsl <- tsl_simulate(
  cols = 3
  )

tsl_colnames_get(
  tsl = tsl
  )

#using a vector
#extra names are ignored
tsl <- tsl_colnames_set(
  tsl = tsl,
  names = c("x", "y", "z", "zz")
)

tsl_colnames_get(
```

```
  tsl = tsl
)

#using a list
#extra names are ignored too
tsl <- tsl_colnames_set(
  tsl = tsl,
  names = list(
    A = c("A", "B", "C"),
    B = c("X", "Y", "Z", "ZZ")
  )
)

tsl_colnames_get(
  tsl = tsl
)
```

`tsl_colnames_suffix` *Append Suffix to Column Names of Time Series List*

Description

Append Suffix to Column Names of Time Series List

Usage

```
tsl_colnames_suffix(tsl = NULL, suffix = NULL)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`suffix`	(optional, character string) String to append to the column names. Default: NULL.

Value

time series list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
tsl <- tsl_simulate()

tsl_colnames_get(tsl = tsl)

tsl <- tsl_colnames_suffix(
  tsl = tsl,
  suffix = "_my_suffix"
)

tsl_colnames_get(tsl = tsl)
```

---

tsl_count_NA *Count NA Cases in Time Series Lists*

Description

Converts Inf, -Inf, and NaN to NA (via tsl_Inf_to_NA() and tsl_NaN_to_NA()), and counts the total number of NA cases in each time series.

Usage

```
tsl_count_NA(tsl = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL

Value

list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#tsl with no NA cases
tsl <- tsl_simulate()

tsl_count_NA(tsl = tsl)

#tsl with NA cases
tsl <- tsl_simulate(
  na_fraction = 0.3
```

```
)

tsl_count_NA(tsl = tsl)

#tsl with variety of empty cases
tsl <- tsl_simulate()
tsl[[1]][1, 1] <- Inf
tsl[[1]][2, 1] <- -Inf
tsl[[1]][3, 1] <- NaN
tsl[[1]][4, 1] <- NaN

tsl_count_NA(tsl = tsl)
```

`tsl_diagnose` *Diagnose Issues in Time Series Lists*

Description

A Time Series List (`tsl` for short) is a named list of zoo time series. This type of object, not defined as a class, is used throughout the `distantia` package to contain time series data ready for processing and analysis.

The structure and values of a `tsl` must fulfill several general conditions:

Structure:

- List names match the attributes "name" of the zoo time series.
- Zoo time series must have at least one shared column name.
- The index (as extracted by `zoo::index()`) of all zoo objects must be of the same class (either "Date", "POSIXct", "numeric", or "integer").
- The "core data" (as extracted by `zoo::coredata()`) of univariate zoo time series must be of class "matrix".

Values (optional, when `full = TRUE`):

- All time series have at least one shared numeric column.
- There are no NA, Inf, or NaN values in the time series.

This function analyzes a `tsl` without modifying it to returns messages describing what conditions are not met, and provides hints on how to fix most issues.

Usage

```
tsl_diagnose(tsl = NULL, full = TRUE)
```

Arguments

`tsl`	(required, list of zoo time series) Time series list to diagnose. Default: NULL
`full`	(optional, logical) If TRUE, a full diagnostic is triggered. Otherwise, only the data structure is tested. Default: TRUE

Value

invisible

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#creating three zoo time series

#one with NA values
x <- zoo_simulate(
  name = "x",
  cols = 1,
  na_fraction = 0.1
  )

#with different number of columns
#wit repeated name
y <- zoo_simulate(
  name = "x",
  cols = 2
  )

#with different time class
z <- zoo_simulate(
  name = "z",
  cols = 1,
  time_range = c(1, 100)
  )

#adding a few structural issues

#changing the column name of x
colnames(x) <- c("b")

#converting z to vector
z <- zoo::zoo(
  x = runif(nrow(z)),
  order.by = zoo::index(z)
)

#storing zoo objects in a list
#with mismatched names
tsl <- list(
  a = x,
  b = y,
  c = z
```

```
)

#running full diagnose
tsl_diagnose(
  tsl = tsl,
  full = TRUE
  )
```

`tsl_handle_NA` *Handle NA Cases in Time Series Lists*

Description

Removes or imputes NA cases in time series lists. Imputation is done via interpolation against time via `zoo::na.approx()`, and if there are still leading or trailing NA cases after NA interpolation, then `zoo::na.spline()` is applied as well to fill these gaps. Interpolated values are forced to fall within the observed data range.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
tsl_handle_NA(tsl = NULL, na_action = c("impute", "omit"))

tsl_Inf_to_NA(tsl = NULL)

tsl_NaN_to_NA(tsl = NULL)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`na_action`	(required, character) NA handling action. Available options are: • "impute" (default): NA cases are interpolated from neighbors as a function of time (see `zoo::na.approx()` and `zoo::na.spline()`). • "omit": rows with NA cases are removed.

Value

time series list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
#tsl with NA cases
tsl <- tsl_simulate(
  na_fraction = 0.25
)

tsl_count_NA(tsl = tsl)

if(interactive()){
  #issues warning
  tsl_plot(tsl = tsl)
}

#omit NA (default)
#--------------------------------------

#original row count
tsl_nrow(tsl = tsl)

#remove rows with NA
tsl_no_na <- tsl_handle_NA(
  tsl = tsl,
  na_action = "omit"
)

#count rows again
#large data loss in this case!
tsl_nrow(tsl = tsl_no_na)

#count NA again
tsl_count_NA(tsl = tsl_no_na)

if(interactive()){
  tsl_plot(tsl = tsl_no_na)
}


#impute NA with zoo::na.approx
#--------------------------------------

#impute NA cases
tsl_no_na <- tsl_handle_NA(
  tsl = tsl,
  na_action = "impute"
)

#count rows again
#large data loss in this case!
tsl_nrow(tsl = tsl_no_na)

if(interactive()){
  tsl_plot(tsl = tsl_no_na)
```

```
}
```

`tsl_initialize` *Transform Raw Time Series Data to Time Series List*

Description

Most functions in this package take a **time series list** (or **tsl** for short) as main input. A `tsl` is a list of zoo time series objects (see `zoo::zoo()`). There is not a formal class for `tsl` objects, but there are requirements these objects must follow to ensure the stability of the package functionalities (see `tsl_diagnose()`). These requirements are:

- There are no NA, Inf, -Inf, or NaN cases in the zoo objects (see `tsl_count_NA()` and `tsl_handle_NA()`).
- All zoo objects must have at least one common column name to allow time series comparison (see `tsl_colnames_get()`).
- All zoo objects have a character attribute "name" identifying the object. This attribute is not part of the zoo class, but the package ensures that this attribute is not lost during data manipulations.
- Each element of the time series list is named after the zoo object it contains (see `tsl_names_get()`, `tsl_names_set()` and `tsl_names_clean()`).
- The time series list contains two zoo objects or more.

The function `tsl_initialize()` (and its alias `tsl_init()`) is designed to convert the following data structures to a time series list:

- Long data frame: with an ID column to separate time series, and a time column that can be of the classes "Date", "POSIXct", "integer", or "numeric". The resulting zoo objects and list elements are named after the values in the ID column.
- Wide data frame: each column is a time series representing the same variable observed at the same time in different places. Each column is converted to a separate zoo object and renamed.
- List of vectors: an object like `list(a = runif(10), b = runif(10))` is converted to a time series list with as many zoo objects as vectors are defined in the original list.
- List of matrices: a list containing matrices, such as `list(a = matrix(runif(30), 10, 3), b = matrix(runif(36), 12, 3))`.
- List of zoo objects: a list with zoo objects, such as `list(a = zoo_simulate(), b = zoo_simulate())`

Usage

```
tsl_initialize(
  x = NULL,
  name_column = NULL,
  time_column = NULL,
  lock_step = FALSE
)

tsl_init(x = NULL, name_column = NULL, time_column = NULL, lock_step = FALSE)
```

Arguments

x	(required, list or data frame) Matrix or data frame in long format, list of vectors, list of matrices, or list of zoo objects. Default: NULL.
name_column	(optional, column name) Column naming individual time series. Numeric names are converted to character with the prefix "X". Default: NULL
time_column	(optional if lock_step = FALSE, and required otherwise, character string) Name of the column representing time, if any. Default: NULL.
lock_step	(optional, logical) If TRUE, all input sequences are subsetted to their common times according to the values in time_column.

Value

list of matrices

Examples

```
#long data frame
#---------------------
data("fagus_dynamics")

#name_column is name
#time column is time
str(fagus_dynamics)

#to tsl
#each group in name_column is a different time series
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)

#check validity (no messages or errors if valid)
tsl_diagnose(tsl)

#class of contained objects
lapply(X = tsl, FUN = class)

#get list and zoo names (between double quotes)
tsl_names_get(
  tsl = tsl,
  zoo = TRUE
  )

#plot tsl
if(interactive()){
  tsl_plot(tsl)
}

#list of zoo objects
```

```
#--------------------
x <- zoo_simulate()
y <- zoo_simulate()

tsl <- tsl_initialize(
  x = list(
    x = x,
    y = y
  )
)

#plot
if(interactive()){
  tsl_plot(tsl)
}


#wide data frame
#--------------------
#wide data frame
#each column is same variable in different places
df <- stats::reshape(
  data = fagus_dynamics[, c(
    "name",
    "time",
    "evi"
  )],
  timevar = "name",
  idvar = "time",
  direction = "wide",
  sep = "_"
)

str(df)

#to tsl
#key assumptions:
#all columns but "time" represent
#the same variable in different places
#all time series are of the same length
tsl <- tsl_initialize(
  x = df,
  time_column = "time"
  )

#colnames are forced to be the same...
tsl_colnames_get(tsl)

#...but can be changed
tsl <- tsl_colnames_set(
  tsl = tsl,
  names = "evi"
)
```

```
tsl_colnames_get(tsl)

#plot
if(interactive()){
  tsl_plot(tsl)
}


#list of vectors
#---------------------
#create list of vectors
vector_list <- list(
  a = cumsum(stats::rnorm(n = 50)),
  b = cumsum(stats::rnorm(n = 70)),
  c = cumsum(stats::rnorm(n = 20))
)

#to tsl
#key assumptions:
#all vectors represent the same variable
#in different places
#time series can be of different lengths
#no time column, integer indices are used as time
tsl <- tsl_initialize(
  x = vector_list
)

#plot
if(interactive()){
  tsl_plot(tsl)
}

#list of matrices
#------------------------
#create list of matrices
matrix_list <- list(
  a = matrix(runif(30), nrow = 10, ncol = 3),
  b = matrix(runif(80), nrow = 20, ncol = 4)
)

#to tsl
#key assumptions:
#each matrix represents a multivariate time series
#in a different place
#all multivariate time series have the same columns
#no time column, integer indices are used as time
tsl <- tsl_initialize(
  x = matrix_list
)

#check column names
tsl_colnames_get(tsl = tsl)
```

```
#remove exclusive column
tsl <- tsl_subset(
  tsl = tsl,
  shared_cols = TRUE
  )
tsl_colnames_get(tsl = tsl)

#plot
if(interactive()){
  tsl_plot(tsl)
}

#list of zoo objects
#------------------------
zoo_list <- list(
  a = zoo_simulate(),
  b = zoo_simulate()
)

#looks like a time series list! But...
tsl_diagnose(tsl = zoo_list)

#let's set the names
zoo_list <- tsl_names_set(tsl = zoo_list)

#check again: it's now a valid time series list
tsl_diagnose(tsl = zoo_list)

#to do all this in one go:
tsl <- tsl_initialize(
  x = list(
    a = zoo_simulate(),
    b = zoo_simulate()
  )
)

#plot
if(interactive()){
  tsl_plot(tsl)
}
```

`tsl_join` *Join Time Series Lists*

Description

Joins an arbitrary of time series lists by name and time. Pairs of zoo objects are joined with `zoo::merge.zoo()`. Names that are not shared across all input TSLs are ignored, and observations with no matching time are filled with NA and then managed via `tsl_handle_NA()` depending on the value of the argument `na_action`.

Usage

```
tsl_join(..., na_action = "impute")
```

Arguments

...	(required, time series lists) names of the time series lists to merge.
na_action	(required, character) NA handling action. Available options are: • "impute" (default): NA cases are interpolated from neighbors as a function of time (see zoo::na.approx() and zoo::na.spline()). • "omit": rows with NA cases are removed.

Value

time series list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#generate two time series list to join
tsl_a <- tsl_simulate(
  n = 2,
  cols = 2,
  irregular = TRUE,
  seed = 1
)

#needs renaming
tsl_b <- tsl_simulate(
  n = 3,
  cols = 2,
  irregular = TRUE,
  seed = 2
) |>
  tsl_colnames_set(
    names = c("c", "d")
  )

#join
tsl <- tsl_join(
  tsl_a,
  tsl_b
)

#plot result
```

```
if(interactive()){
  tsl_plot(
    tsl = tsl
  )
}
```

`tsl_names_clean` *Clean Time Series Names in a Time Series List*

Description

Combines `utils_clean_names()` and `tsl_names_set()` to help clean, abbreviate, capitalize, and add a suffix or a prefix to time series list names.

Usage

```
tsl_names_clean(
  tsl = NULL,
  lowercase = FALSE,
  separator = "_",
  capitalize_first = FALSE,
  capitalize_all = FALSE,
  length = NULL,
  suffix = NULL,
  prefix = NULL
)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`lowercase`	(optional, logical) If TRUE, all names are coerced to lowercase. Default: FALSE
`separator`	(optional, character string) Separator when replacing spaces and dots. Also used to separate `suffix` and `prefix` from the main word. Default: "_".
`capitalize_first`	(optional, logical) Indicates whether to capitalize the first letter of each name Default: FALSE.
`capitalize_all`	(optional, logical) Indicates whether to capitalize all letters of each name Default: FALSE.
`length`	(optional, integer) Minimum length of abbreviated names. Names are abbreviated via `abbreviate()`. Default: NULL.
`suffix`	(optional, character string) Suffix for the clean names. Default: NULL.
`prefix`	(optional, character string) Prefix for the clean names. Default: NULL.

Value

time series list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#initialize time series list
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)

#original names
tsl_names_get(
  tsl = tsl
)

#abbreviate names
#--------------------------
tsl_clean <- tsl_names_clean(
  tsl = tsl,
  capitalize_first = TRUE,
  length = 4 #abbreviate to 4 characters
)

#new names
tsl_names_get(
  tsl = tsl_clean
)

#suffix and prefix
#--------------------------
tsl_clean <- tsl_names_clean(
  tsl = tsl,
  capitalize_all = TRUE,
  separator = "_",
  suffix = "fagus",
  prefix = "country"
)

#new names
tsl_names_get(
  tsl = tsl_clean
)
```

tsl_names_get *Get Time Series Names from a Time Series Lists*

Description

A time series list has two sets of names: the names of the list items (as returned by `names(tsl)`), and the names of the contained zoo objects, as stored in their attribute "name". These names should ideally be the same, for the sake of data consistency. This function extracts either set of names,

Usage

```
tsl_names_get(tsl = NULL, zoo = TRUE)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`zoo`	(optional, logical) If TRUE, the attributes "name" of the zoo objects are returned. Default: TRUE

Value

list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
#initialize a time series list
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)


#get names of zoo objects
tsl_names_get(
  tsl = tsl,
  zoo = TRUE
)

#get list names only
tsl_names_get(
  tsl = tsl,
```

```
  zoo = FALSE
  )

#same as
names(tsl)
```

`tsl_names_set` *Set Time Series Names in a Time Series List*

Description

Sets the names of a time series list and the internal names of the zoo objects inside, stored in their attribute "name".

Usage

```
tsl_names_set(tsl = NULL, names = NULL)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`names`	(optional, character vector) names to set. Must be of the same length of `x`. If NULL, and the list `x` has names, then the names of the zoo objects inside of the list are taken from the names of the list elements.

Value

time series list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
#simulate time series list
tsl <- tsl_simulate(n = 3)

#assess validity
tsl_diagnose(
  tsl = tsl
)

#list and zoo names (default)
tsl_names_get(
  tsl = tsl
```

```
)

#list names
tsl_names_get(
  tsl = tsl,
  zoo = FALSE
)

#renaming list items and zoo objects
#-------------------------------------
tsl <- tsl_names_set(
  tsl = tsl,
  names = c("X", "Y", "Z")
)

# check new names
tsl_names_get(
  tsl = tsl
)

#fixing naming issues
#-------------------------------------

#creating a invalid time series list
names(tsl)[2] <- "B"

# check names
tsl_names_get(
  tsl = tsl
)

#validate tsl
#returns NOT VALID
#recommends a solution
tsl_diagnose(
  tsl = tsl
)

#fix issue with tsl_names_set()
#uses names of zoo objects for the list items
tsl <- tsl_names_set(
  tsl = tsl
)

#validate again
tsl_diagnose(
  tsl = tsl
)

#list names
tsl_names_get(
  tsl = tsl
)
```

tsl_names_test *Tests Naming Issues in Time Series Lists*

Description

Tests Naming Issues in Time Series Lists

Usage

```
tsl_names_test(tsl = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL

Value

logical

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#creating three zoo time series

#one with NA values
x <- zoo_simulate(
  name = "x",
  cols = 1,
  na_fraction = 0.1
  )

#with different number of columns
#wit repeated name
y <- zoo_simulate(
  name = "x",
  cols = 2
  )

#with different time class
z <- zoo_simulate(
  name = "z",
  cols = 1,
  time_range = c(1, 100)
```

```
  )

#adding a few structural issues

#changing the column name of x
colnames(x) <- c("b")

#converting z to vector
z <- zoo::zoo(
  x = runif(nrow(z)),
  order.by = zoo::index(z)
)

#storing zoo objects in a list
#with mismatched names
tsl <- list(
  a = x,
  b = y,
  c = z
)

#running full diagnose
tsl_names_test(
  tsl = tsl
  )
```

`tsl_ncol` *Get Number of Columns in Time Series Lists*

Description

Get Number of Columns in Time Series Lists

Usage

```
tsl_ncol(tsl = NULL)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL

Value

list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_nrow()`, `tsl_repair()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
#initialize time series list
tsl <- tsl_simulate(
  n = 2,
  cols = 6
)

#number of columns per zoo object
tsl_ncol(tsl = tsl)
```

---

tsl_nrow *Get Number of Rows in Time Series Lists*

Description

Get Number of Rows in Time Series Lists

Usage

```
tsl_nrow(tsl = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL

Value

list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_repair(), tsl_subset(), tsl_time(), tsl_to_df()

Examples

```
#simulate zoo time series
tsl <- tsl_simulate(
  rows = 150
  )

#count rows
tsl_nrow(
  tsl = tsl
)
```

`tsl_plot` *Plot Time Series List*

Description

Plot Time Series List

Usage

```
tsl_plot(
  tsl = NULL,
  columns = 1,
  xlim = NULL,
  ylim = "absolute",
  line_color = NULL,
  line_width = 1,
  text_cex = 1,
  guide = TRUE,
  guide_columns = 1,
  guide_cex = 0.8
)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`columns`	(optional, integer) Number of columns of the multipanel plot. Default: 1
`xlim`	(optional, numeric vector) Numeric vector with the limits of the x axis. Applies to all sequences. Default: NULL
`ylim`	(optional, numeric vector or character string) Numeric vector of length two with the limits of the vertical axis or a keyword. Accepted keywords are: • "absolute" (default): all time series are plotted using the overall data range. When this option is used, horizontal lines indicating the overall mean, minimum, and maximum are shown as reference. • "relative": each time series is plotted using its own range. Equivalent result can be achieved using `ylim = NULL`.
`line_color`	(optional, character vector) vector of colors for the distance or cost matrix. If NULL, uses an appropriate palette generated with `grDevices::palette.colors()`. Default: NULL
`line_width`	(optional, numeric vector) Width of the time series plot. Default: 1
`text_cex`	(optional, numeric) Multiplicator of the text size. Default: 1
`guide`	(optional, logical) If TRUE, plots a legend. Default: TRUE
`guide_columns`	(optional, integer) Columns of the line guide. Default: 1.
`guide_cex`	(optional, numeric) Size of the guide's text and separation between the guide's rows. Default: 0.7.

Value

plot

Examples

```
#simulate zoo time series
tsl <- tsl_simulate(
  cols = 3
  )

if(interactive()){

  #default plot
  tsl_plot(
    tsl = tsl
    )

  #relative vertical limits
  tsl_plot(
    tsl = tsl,
    ylim = "relative"
  )

  #changing layout
  tsl_plot(
    tsl = tsl,
    columns = 2,
    guide_columns = 2
  )

  #no legend
  tsl_plot(
    tsl = tsl,
    guide = FALSE
  )

  #changing color
  tsl_plot(
    tsl = tsl,
    line_color = c("red", "green", "blue"))

}
```

---

tsl_repair *Repair Issues in Time Series Lists*

Description

A Time Series List (`tsl` for short) is a list of zoo time series. This type of object, not defined as a class, is used throughout the `distantia` package to contain time series data ready for processing and analysis.

The structure and values of a `tsl` must fulfill several general conditions:

Structure:

- The list names match the attributes "name" of the zoo time series
- All zoo time series must have at least one shared column name.
- Data in univariate zoo time series (as extracted by `zoo::coredata(x)`) must be of the class "matrix". Univariate zoo time series are often represented as vectors, but this breaks several subsetting and transformation operations implemented in this package.

Values (optional, when `full = TRUE`):

- All time series have at least one shared numeric column.
- There are no NA, Inf, or NaN values in the time series.

This function analyzes a `tsl`, and tries to fix all possible issues to make it comply with the conditions listed above without any user input. Use with care, as it might defile your data.

Usage

```
tsl_repair(tsl = NULL, full = TRUE)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`full`	(optional, logical) If TRUE, a full repair (structure and values) is triggered. Otherwise, only the data structure is repaired Default: TRUE

Value

time series list

See Also

Other tsl_management: `tsl_burst()`, `tsl_colnames_clean()`, `tsl_colnames_get()`, `tsl_colnames_prefix()`, `tsl_colnames_set()`, `tsl_colnames_suffix()`, `tsl_count_NA()`, `tsl_diagnose()`, `tsl_handle_NA()`, `tsl_join()`, `tsl_names_clean()`, `tsl_names_get()`, `tsl_names_set()`, `tsl_names_test()`, `tsl_ncol()`, `tsl_nrow()`, `tsl_subset()`, `tsl_time()`, `tsl_to_df()`

Examples

```
#creating three zoo time series

#one with NA values
x <- zoo_simulate(
  name = "x",
  cols = 1,
  na_fraction = 0.1
  )

#with different number of columns
```

```
#wit repeated name
y <- zoo_simulate(
  name = "x",
  cols = 2
  )

#with different time class
z <- zoo_simulate(
  name = "z",
  cols = 1,
  time_range = c(1, 100)
  )

#adding a few structural issues

#changing the column name of x
colnames(x) <- c("b")

#converting z to vector
z <- zoo::zoo(
  x = runif(nrow(z)),
  order.by = zoo::index(z)
)

#storing zoo objects in a list
#with mismatched names
tsl <- list(
  a = x,
  b = y,
  c = z
)

#running full diagnose
tsl_diagnose(
  tsl = tsl,
  full = TRUE
  )

tsl <- tsl_repair(tsl)
```

`tsl_resample` *Resample Time Series Lists to a New Time*

Description

Objective

Time series resampling interpolates new values for time steps not available in the original time series. This operation is useful to:

- Transform irregular time series into regular.

- Align time series with different temporal resolutions.
- Increase (upsampling) or decrease (downsampling) the temporal resolution of a time series.

Time series resampling **should not be used** to extrapolate new values outside of the original time range of the time series, or to increase the resolution of a time series by a factor of two or more. These operations are known to produce non-sensical results.

Warning: This function resamples time series lists **with overlapping times**. Please check such overlap by assessing the columns "begin" and "end " of the data frame resulting from `df <- tsl_time(tsl = tsl)`. Resampling will be limited by the shortest time series in your time series list. To resample non-overlapping time series, please subset the individual components of `tsl` one by one either using `tsl_subset()` or the syntax `tsl = my_tsl[[i]]`.

Methods

This function offers three methods for time series interpolation:

- "linear" (default): interpolation via piecewise linear regression as implemented in `zoo::na.approx()`.
- "spline": cubic smoothing spline regression as implemented in `stats::smooth.spline()`.
- "loess": local polynomial regression fitting as implemented in `stats::loess()`.

These methods are used to fit models `y ~ x` where `y` represents the values of a univariate time series and `x` represents a numeric version of its time.

The functions `utils_optimize_spline()` and `utils_optimize_loess()` are used under the hood to optimize the complexity of the methods "spline" and "loess" by finding the configuration that minimizes the root mean squared error (RMSE) between observed and predicted `y`. However, when the argument `max_complexity = TRUE`, the complexity optimization is ignored, and a maximum complexity model is used instead.

New time

The argument `new_time` offers several alternatives to help define the new time of the resulting time series:

- `NULL`: the target time series (`x`) is resampled to a regular time within its original time range and number of observations.
- `zoo object`: a zoo object to be used as template for resampling. Useful when the objective is equalizing the frequency of two separate zoo objects.
- `time series list`: a time series list to be used as template. The range of overlapping dates and the average resolution are used to generate the new resampling time. This method cannot be used to align two time series lists, unless the template is resampled beforehand.
- `time vector`: a time vector of a class compatible with the time in `x`.
- `keyword`: character string defining a resampling keyword, obtained via `zoo_time(x, keywords = "resample")$keywords`..
- `numeric`: a single number representing the desired interval between consecutive samples in the units of `x` (relevant units can be obtained via `zoo_time(x)$units`).

Step by Step

The steps to resample a time series list are:

1. The time interpolation range is computed from the intersection of all times in `tsl`. This step ensures that no extrapolation occurs during resampling, but it also makes resampling of non-overlapping time series impossible.
2. If `new_time` is provided, any values of `new_time` outside of the minimum and maximum interpolation times are removed to avoid extrapolation. If `new_time` is not provided, a regular time within the interpolation time range with the length of the shortest time series in `tsl` is generated.
3. For each univariate time time series, a model `y ~ x`, where `y` is the time series and `x` is its own time coerced to numeric is fitted.
   - If `max_complexity == FALSE`, the model with the complexity that minimizes the root mean squared error between the observed and predicted `y` is returned.
   - If `max_complexity == TRUE` and `method = "spline"` or `method = "loess"`, the first valid model closest to a maximum complexity is returned.
4. The fitted model is predicted over `new_time` to generate the resampled time series.

Other Details

Please use this operation with care, as there are limits to the amount of resampling that can be done without distorting the data. The safest option is to keep the distance between new time points within the same magnitude of the distance between the old time points.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
tsl_resample(
  tsl = NULL,
  new_time = NULL,
  method = "linear",
  max_complexity = FALSE
)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`new_time`	(required, zoo object, time series list, character string, time vector, numeric) New time to resample to. If a time vector is provided, it must be of a class compatible with the time of `tsl`. If a zoo object or time series list is provided, its time is used as a template to resample `tsl`. Valid resampling keywords (see `tsl_time()`) are allowed. Numeric values are interpreted as interval widths in the time units of the time series. If NULL, irregular time series are predicted into a regular version of their own time. Default: NULL
`method`	(optional, character string) Name of the method to resample the time series. One of "linear", "spline" or "loess". Default: "linear".
`max_complexity`	(required, logical). Only relevant for methods "spline" and "loess". If TRUE, model optimization is ignored, and the a model of maximum complexity (an overfitted model) is used for resampling. Default: FALSE

Value

time series list

See Also

zoo_resample()

Other tsl_processing: tsl_aggregate(), tsl_smooth(), tsl_stats(), tsl_transform()

Examples

```
#generate irregular time series
tsl <- tsl_simulate(
  n = 2,
  rows = 100,
  irregular = TRUE
)

if(interactive()){
  tsl_plot(tsl)
}


#range of times between samples
tsl_time_summary(tsl)[
  c(
    "units",
    "resolution_min",
    "resolution_max"
    )
  ]

#resample to regular using linear interpolation
tsl_regular <- tsl_resample(
  tsl = tsl
)

if(interactive()){
  tsl_plot(tsl_regular)
}

#check new resolution
tsl_time_summary(tsl_regular)[
  c(
    "units",
    "resolution_min",
    "resolution_max"
  )
]

#resample using keywords

#valid resampling keywords
```

```
tsl_time_summary(
  tsl = tsl,
  keywords = "resample"
)$keywords

#by month
tsl_months <- tsl_resample(
  tsl = tsl,
  new_time = "months"
)

if(interactive()){
  tsl_plot(tsl_months)
}

#by week
tsl_weeks <- tsl_resample(
  tsl = tsl,
  new_time = "weeks"
)

if(interactive()){
  tsl_plot(tsl_weeks)
}

#resample using time interval

#get relevant units
tsl_time(tsl)$units

#resampling to 15 days intervals
tsl_15_days <- tsl_resample(
  tsl = tsl,
  new_time = 15 #days
)

tsl_time_summary(tsl_15_days)[
  c(
    "units",
    "resolution_min",
    "resolution_max"
  )
]

if(interactive()){
  tsl_plot(tsl_15_days)
}

#aligning two time series listsç

#two time series lists with different time ranges
tsl1 <- tsl_simulate(
  n = 2,
```

```
  rows = 80,
  time_range = c("2010-01-01", "2020-01-01"),
  irregular = TRUE
)

tsl2 <- tsl_simulate(
  n = 2,
  rows = 120,
  time_range = c("2005-01-01", "2024-01-01"),
  irregular = TRUE
)

#check time features
tsl_time_summary(tsl1)[
  c(
    "begin",
    "end",
    "resolution_min",
    "resolution_max"
  )
]

tsl_time_summary(tsl2)[
  c(
    "begin",
    "end",
    "resolution_min",
    "resolution_max"
  )
]

#tsl1 to regular
tsl1_regular <- tsl_resample(
  tsl = tsl1
)

#tsl2 resampled to time of tsl1_regular
tsl2_regular <- tsl_resample(
  tsl = tsl2,
  new_time = tsl1_regular
)

#check alignment
tsl_time_summary(tsl1_regular)[
  c(
    "begin",
    "end",
    "resolution_min",
    "resolution_max"
  )
]

tsl_time_summary(tsl2_regular)[
```

```
  c(
    "begin",
    "end",
    "resolution_min",
    "resolution_max"
  )
]
```

`tsl_simulate` *Simulate a Time Series List*

Description

Generates simulated time series lists for testing and learning.

This function supports progress bars generated by the `progressr` package, and accepts a parallelization setup via `future::plan()` (see examples).

Usage

```
tsl_simulate(
  n = 2,
  cols = 5,
  rows = 100,
  time_range = c("2010-01-01", "2020-01-01"),
  data_range = c(0, 1),
  seasons = 0,
  na_fraction = 0,
  independent = FALSE,
  irregular = TRUE,
  seed = NULL
)
```

Arguments

`n`	(optional, integer) Number of time series to simulate. Default: 2.
`cols`	(optional, integer) Number of columns of each time series. Default: 5
`rows`	(optional, integer) Length of each time series. Minimum is 10, but maximum is not limited. Very large numbers might crash the R session. Default: 100
`time_range`	(optional character or numeric vector) Time interval of the time series. Either a character vector with dates in format YYYY-MM-DD or or a numeric vector. If there is a mismatch between `time_range` and `rows` (for example, the number of days in `time_range` is smaller than `rows`), the upper value in `time_range` is adapted to `rows`. Default: c("2010-01-01", "2020-01-01")
`data_range`	(optional, numeric vector of length 2) Extremes of the time series values. Default: c(0, 1)

seasons	(optional, integer) Number of seasons in the resulting time series. The maximum number of seasons is computed as `floor(rows/3)`. Default: 0
na_fraction	(optional, numeric) Value between 0 and 0.5 indicating the approximate fraction of NA data in the simulated time series. Default: 0.
independent	(optional, logical) If TRUE, each new column in a simulated time series is averaged with the previous column to generate dependency across columns, and each new simulated time series is weighted-averaged with a time series template to generate dependency across time series. Irrelevant when `cols < 2` or `n < 2`, and hard to perceive in the output when `seasons > 0`. Default: FALSE
irregular	(optional, logical) If TRUE, the time intervals between consecutive samples and the number of rows are irregular. Default: TRUE
seed	(optional, integer) Random seed used to simulate the zoo object. If NULL (default), a seed is selected at random. Default: NULL

Value

time series list

See Also

Other simulate_time_series: `zoo_simulate()`

Examples

```
# generates a different time series list on each iteration when seed = NULL
tsl <- tsl_simulate(
  n = 2,
  seasons = 4
)

if(interactive()){
  tsl_plot(
    tsl = tsl
  )
}

# generate 3 independent time series
tsl_independent <- tsl_simulate(
  n = 3,
  cols = 3,
  independent = TRUE
)

if(interactive()){
  tsl_plot(
    tsl = tsl_independent
  )
}

# generate 3 independent time series
```

```
tsl_dependent <- tsl_simulate(
  n = 3,
  cols = 3,
  independent = FALSE
)

if(interactive()){
  tsl_plot(
    tsl = tsl_dependent
  )
}

# with seasons
tsl_seasons <- tsl_simulate(
  n = 3,
  cols = 3,
  seasons = 4,
  independent = FALSE
)

if(interactive()){
  tsl_plot(
    tsl = tsl_seasons
  )
}
```

`tsl_smooth` *Smoothing of Time Series Lists*

Description

Rolling-window and exponential smoothing of Time Series Lists.

Rolling-window smoothing This computes a statistic over a fixed-width window of consecutive cases and replaces each central value with the computed statistic. It is commonly used to mitigate noise in high-frequency time series.

Exponential smoothing computes each value as the weighted average of the current value and past smoothed values. This method is useful for reducing noise in time series data while preserving the overall trend.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
tsl_smooth(tsl = NULL, window = 3, f = mean, alpha = NULL, ...)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`window`	(optional, integer) Smoothing window width, in number of cases. Default: 3
`f`	(optional, quoted or unquoted function name) Name of a standard or custom function to aggregate numeric vectors. Typical examples are `mean`, `max`,`min`, `median`, and `quantile`. Default: `mean`.
`alpha`	(required, numeric) Exponential smoothing factor in the range (0, 1]. Determines the weight of the current value relative to past values. If not NULL, the arguments `window` and `f` are ignored, and exponential smoothing is performed instead. Default: NULL
`...`	(optional, additional arguments) additional arguments to `f`.

Value

time series list

See Also

Other tsl_processing: `tsl_aggregate()`, `tsl_resample()`, `tsl_stats()`, `tsl_transform()`

Examples

```
tsl <- tsl_simulate(n = 2)

#rolling window smoothing
tsl_smooth <- tsl_smooth(
  tsl = tsl,
  window = 5,
  f = mean
)

if(interactive()){
  tsl_plot(tsl)
  tsl_plot(tsl_smooth)
}

#exponential smoothing
tsl_smooth <- tsl_smooth(
  tsl = tsl,
  alpha = 0.2
)

if(interactive()){
  tsl_plot(tsl)
  tsl_plot(tsl_smooth)
}
```

`tsl_stats` *Summary Statistics of Time Series Lists*

Description

This function computes a variety of summary statistics for each time series and numeric column within a time series list. The statistics include common metrics such as minimum, maximum, quartiles, mean, standard deviation, range, interquartile range, skewness, kurtosis, and autocorrelation for specified lags.

For irregular time series, autocorrelation computation is performed after regularizing the time series via interpolation with `zoo_resample()`. This regularization does not affect the computation of all other stats.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
tsl_stats(tsl = NULL, lags = 1L)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`lags`	(optional, integer) An integer specifying the number of autocorrelation lags to compute. If NULL, autocorrelation computation is disabled. Default: 1.

Value

data frame:

- name: name of the zoo object.
- rows: rows of the zoo object.
- columns: columns of the zoo object.
- time_units: time units of the zoo time series (see `zoo_time()`).
- time_begin: beginning time of the time series.
- time_end: end time of the time series.
- time_length: total length of the time series, expressed in time units.
- time_resolution: average distance between consecutive observations
- variable: name of the variable, a column of the zoo object.
- min: minimum value of the zoo column.
- q1: first quartile (25th percentile).
- median: 50th percentile.
- q3: third quartile (75th percentile).

- max: maximum value.
- mean: average value.
- sd: standard deviation.
- range: range of the variable, computed as max - min.
- iq_range: interquartile range of the variable, computed as q3 - q1.
- skewness: asymmetry of the variable distribution.
- kurtosis:"tailedness" of the variable distribution.
- ac_lag_1, ac_lag_2, ...: autocorrelation values for the specified lags.

See Also

Other tsl_processing: `tsl_aggregate()`, `tsl_resample()`, `tsl_smooth()`, `tsl_transform()`

Examples

```
#three time series
#climate and ndvi in Fagus sylvatica stands in Spain, Germany, and Sweden
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)


#stats computation
df <- tsl_stats(
  tsl = tsl,
  lags = 3
  )

df
```

---

tsl_subset *Subset Time Series Lists by Time Series Names, Time, and/or Column Names*

Description

Subset Time Series Lists by Time Series Names, Time, and/or Column Names

Usage

```
tsl_subset(
  tsl = NULL,
  names = NULL,
  colnames = NULL,
  time = NULL,
  numeric_cols = TRUE,
  shared_cols = TRUE
)
```

Arguments

tsl	(required, list) Time series list. Default: NULL
names	(optional, character or numeric vector) Character vector of names or numeric vector with list indices. If NULL, all time series are kept. Default: NULL
colnames	(optional, character vector) Column names of the zoo objects in tsl. If NULL, all columns are returned. Default: NULL
time	(optional, numeric vector) time vector of length two used to subset rows by time. If NULL, all rows in tsl are preserved. Default: NULL
numeric_cols	(optional, logical) If TRUE, only the numeric columns of the zoo objects are returned. Default: TRUE
shared_cols	(optional, logical) If TRUE, only columns shared across all zoo objects are returned. Default: TRUE

Value

time series list

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_time(), tsl_to_df()

Examples

```
#initialize time series list
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)

#checking available dimensions

#names
tsl_names_get(
```

```
  tsl = tsl
)

#colnames
tsl_colnames_get(
  tsl = tsl
)

#time
tsl_time(
  tsl = tsl
)[, c("name", "begin", "end")]

#subset
tsl_new <- tsl_subset(
  tsl = tsl,
  names = c("Sweden", "Germany"),
  colnames = c("rainfall", "temperature"),
  time = c("2010-01-01", "2015-01-01")
)

#check new dimensions

#names
tsl_names_get(
  tsl = tsl_new
)

#colnames
tsl_colnames_get(
  tsl = tsl_new
)

#time
tsl_time(
  tsl = tsl_new
)[, c("name", "begin", "end")]
```

`tsl_time` *Time Features of Time Series Lists*

Description

The functions `tsl_time()` and `tsl_time_summary()` summarize the time features of a time series list.

- `tsl_time()` returns a data frame with one row per time series in the argument 'tsl'
- `tsl_time_summary()` returns a list with the features captured by `tsl_time()`, but aggregated across time series.

Both functions return keywords useful for the functions `tsl_aggregate()` and `tsl_resample()`, depending on the value of the argument `keywords`.

Usage

```
tsl_time(tsl = NULL, keywords = c("resample", "aggregate"))

tsl_time_summary(tsl = NULL, keywords = c("resample", "aggregate"))
```

Arguments

tsl	(required, list) Time series list. Default: NULL
keywords	(optional, character string or vector) Defines what keywords are returned. If "aggregate", returns valid keywords for zoo_aggregate(). If "resample", returns valid keywords for zoo_resample(). If both, returns all valid keywords. Default: c("aggregate", "resample").

Value

- tsl_time(): data frame with the following columns:
  - name (string): time series name.
  - rows (integer): number of observations.
  - class (string): time class, one of "Date", "POSIXct", or "numeric."
  - units (string): units of the time series.
  - length (numeric): total length of the time series expressed in units.
  - resolution (numeric): average interval between observations expressed in units.
  - begin (date or numeric): begin time of the time series.
  - end (date or numeric): end time of the time series.
  - keywords (character vector): valid keywords for tsl_aggregate() or tsl_resample(), depending on the value of the argument keywords.
- tsl_time_summary(): list with the following objects:
  - class (string): time class, one of "Date", "POSIXct", or "numeric."
  - units (string): units of the time series.
  - begin (date or numeric): begin time of the time series.
  - end (date or numeric): end time of the time series.
  - resolution_max (numeric): longer time interval between consecutive samples expressed in units.
  - resolution_min (numeric): shorter time interval between consecutive samples expressed in units.
  - keywords (character vector): valid keywords for tsl_aggregate() or tsl_resample(), depending on the value of the argument keywords.
  - units_df (data frame) data frame for internal use within tsl_aggregate() and tsl_resample().

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_to_df()

Examples

```
#simulate a time series list
tsl <- tsl_simulate(
  n = 3,
  rows = 150,
  time_range = c(
    Sys.Date() - 365,
    Sys.Date()
  ),
  irregular = TRUE
)

#time data frame
tsl_time(
  tsl = tsl
)

#time summary
tsl_time_summary(
  tsl = tsl
)
```

---

tsl_to_df *Transform Time Series List to Data Frame*

Description

Transform Time Series List to Data Frame

Usage

```
tsl_to_df(tsl = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL

Value

data frame

See Also

Other tsl_management: tsl_burst(), tsl_colnames_clean(), tsl_colnames_get(), tsl_colnames_prefix(), tsl_colnames_set(), tsl_colnames_suffix(), tsl_count_NA(), tsl_diagnose(), tsl_handle_NA(), tsl_join(), tsl_names_clean(), tsl_names_get(), tsl_names_set(), tsl_names_test(), tsl_ncol(), tsl_nrow(), tsl_repair(), tsl_subset(), tsl_time()

Examples

```
tsl <- tsl_simulate(
  n = 3,
  rows = 10,
  time_range = c(
    "2010-01-01",
    "2020-01-01"
  ),
  irregular = FALSE
)

df <- tsl_to_df(
  tsl = tsl
)

names(df)
nrow(df)
head(df)
```

---

tsl_transform *Transform Values in Time Series Lists*

Description

Function for time series transformations without changes in data dimensions. Generally, functions introduced via the argument `f` should not change the dimensions of the output time series list. See `tsl_resample()` and `tsl_aggregate()` for transformations requiring changes in time series dimensions.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
tsl_transform(tsl = NULL, f = NULL, ...)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`f`	(required, transformation function) name of a function taking a matrix as input. Currently, the following options are implemented, but any other function taking a matrix as input (for example, `scale()`) should work as well: • f_proportion: proportion computed by row. • f_percent: percentage computed by row. • f_hellinger: Hellinger transformation computed by row • f_scale_local: Local centering and/or scaling based on the variable mean and standard deviation in each time series within `tsl`.

- f_scale_global: Global centering and/or scaling based on the variable mean and standard deviation across all time series within `tsl`.
- f_smooth: Time series smoothing with a user defined rolling window.
- f_detrend_difference: Differencing detrending of time series via `diff()`.
- f_detrend_linear: Detrending of seasonal time series via linear modeling.
- f_detrend_gam: Detrending of seasonal time series via Generalized Additive Models.

`...` (optional, additional arguments of `f`) Optional arguments for the transformation function.

Value

time series list

See Also

Other tsl_processing: `tsl_aggregate()`, `tsl_resample()`, `tsl_smooth()`, `tsl_stats()`

Examples

```
#two time series
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
) |>
  tsl_subset(
    names = c("Spain", "Sweden"),
    colnames = c("rainfall", "temperature")
  )

if(interactive()){
  tsl_plot(
    tsl = tsl
  )
}

#centering and scaling
#---------------------------------------
#same mean and standard deviation are used to scale each variable across all time series
tsl_scale <- tsl_transform(
  tsl = tsl,
  f = f_scale_local
)

if(interactive()){
  tsl_plot(
    tsl = tsl_scale,
    guide_columns = 3
  )
}
```

```
#rescaling to a new range
#----------------------------------------

#rescale between -100 and 100
tsl_rescaled <- tsl_transform(
  tsl = tsl,
  f = f_rescale_local,
  new_min = -100,
  new_max = 100
)

#old range
sapply(X = tsl, FUN = range)

#new range
sapply(X = tsl_rescaled, FUN = range)



#numeric transformations
#----------------------------------------
#eemian pollen counts
tsl <- tsl_initialize(
  x = distantia::eemian_pollen,
  name_column = "name",
  time_column = "time"
)

if(interactive()){
  tsl_plot(
    tsl = tsl
  )
}

#percentages
tsl_percentage <- tsl_transform(
  tsl = tsl,
  f = f_percent
)

if(interactive()){
  tsl_plot(
    tsl = tsl_percentage
  )
}

#hellinger transformation
tsl_hellinger <- tsl_transform(
  tsl = tsl,
  f = f_hellinger
)
```

```
if(interactive()){
  tsl_plot(
    tsl = tsl_hellinger
  )
}
```

`utils_as_time` *Ensures Correct Class for Time Arguments*

Description

This function guesses the class of a vector based on its elements. It can handle numeric vectors, character vectors that can be coerced to either "Date" or "POSIXct" classes, and vectors already in "Date" or "POSIXct" classes.

Usage

```
utils_as_time(x = NULL, to_class = NULL)
```

Arguments

`x`	(required, vector) Vectors of the classes 'numeric', 'Date', and 'POSIXct' are valid and returned without any transformation. Character vectors are analyzed to determine their more probable type, and are coerced to 'Date' or 'POSIXct' depending on their number of elements. Generally, any character vector representing an ISO 8601 standard, like "YYYY-MM-DD" or "YYYY-MM-DD HH:MM:SS" will be converted to a valid class. If a character vector cannot be coerced to date, it is returned as is. Default: NULL
`to_class`	(optional, class) Options are: NULL, "numeric", "Date", and "POSIXct". If NULL, 'x' is returned as the most appropriate time class. Otherwise, 'x' is coerced to the given class. Default: NULL

Value

time vector

See Also

Other internal_time_handling: `utils_coerce_time_class()`, `utils_is_time()`, `utils_new_time()`, `utils_time_keywords()`, `utils_time_keywords_dictionary()`, `utils_time_keywords_translate()`, `utils_time_units()`

Examples

```
# numeric
utils_as_time(
  x = c(-123120, 1200)
  )

# character string to Date
utils_as_time(
  x = c("2022-03-17", "2024-02-05")
  )

# incomplete character strings to Date
utils_as_time(
  x = c("2022", "2024")
  )

utils_as_time(
  x = c("2022-02", "2024-03")
  )

# character string to POSIXct
utils_as_time(
  x = c("2022-03-17 12:30:45", "2024-02-05 11:15:45")
  )

# Date vector (returns the input)
utils_as_time(
  x = as.Date(c("2022-03-17", "2024-02-05"))
  )

# POSIXct vector (returns the input)
utils_as_time(
  x = as.POSIXct(c("2022-03-17 12:30:45", "2024-02-05 11:15:45"))
  )
```

---

utils_block_size *Default Block Size for Restricted Permutation in Dissimilarity Analyses*

Description

Default Block Size for Restricted Permutation in Dissimilarity Analyses

Usage

```
utils_block_size(tsl = NULL, block_size = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL
block_size	(optional, integer vector) Row block sizes for restricted permutation tests. Only relevant when permutation methods are "restricted" or "restricted_by_row". A block of size n indicates that a row can only be permuted within a block of n adjacent rows. If NULL, defaults to the 20% rows of the shortest time series in tsl. Minimum value is 2, and maximum value is 50% rows of the shortest time series in tsl. Default: NULL.

Value

integer

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_optir utils_cluster_silhouette()

---

utils_cluster_hclust_optimizer

Optimize the Silhouette Width of Hierarchical Clustering Solutions

---

Description

Performs a parallelized grid search to find the number of clusters maximizing the overall silhouette width of the clustering solution (see utils_cluster_silhouette()). When method = NULL, the optimization also includes all methods available in stats::hclust() in the grid search. This function supports parallelization via future::plan() and a progress bar generated by the progressr package (see Examples).

Usage

```
utils_cluster_hclust_optimizer(d = NULL, method = NULL)
```

Arguments

d	(required, matrix) distance matrix typically resulting from distantia_matrix(), but any other square matrix should work. Default: NULL
method	(optional, character string) Argument of stats::hclust() defining the agglomerative method. One of: "ward.D", "ward.D2", "single", "complete", "average" (= UPGMA), "mcquitty" (= WPGMA), "median" (= WPGMC) or "centroid" (= UPGMC). Unambiguous abbreviations are accepted as well. This function supports a parallelization setup via future::plan(), and progress bars provided by the package progressr.

Value

data frame

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_kmeans_optimizer(), utils_cluster_silhouette()

Examples

```
#weekly covid prevalence
#in 10 California counties
#aggregated by month
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
) |>
  tsl_subset(
    names = 1:10
  ) |>
  tsl_aggregate(
    new_time = "months",
    fun = max
  )

if(interactive()){
  #plotting first three time series
  tsl_plot(
    tsl = tsl_subset(
      tsl = tsl,
      names = 1:3
    ),
    guide_columns = 3
  )
}

#compute dissimilarity matrix
psi_matrix <- distantia(
  tsl = tsl,
  lock_step = TRUE
) |>
  distantia_matrix()

#optimize hierarchical clustering
hclust_optimization <- utils_cluster_hclust_optimizer(
  d = psi_matrix
)

#best solution in first row
```

```
head(hclust_optimization)
```

`utils_cluster_kmeans_optimizer` *Optimize the Silhouette Width of K-Means Clustering Solutions*

Description

Generates k-means solutions from 2 to `nrow(d) - 1` number of clusters and returns the number of clusters with a higher silhouette width median. See `utils_cluster_silhouette()` for more details.

This function supports a parallelization setup via `future::plan()`, and progress bars provided by the package progressr.

Usage

```
utils_cluster_kmeans_optimizer(d = NULL, seed = 1)
```

Arguments

d	(required, matrix) distance matrix typically resulting from `distantia_matrix()`, but any other square matrix should work. Default: NULL
seed	(optional, integer) Random seed to be used during the K-means computation. Default: 1

Value

data frame

See Also

Other distantia_support: `distantia_aggregate()`, `distantia_boxplot()`, `distantia_cluster_hclust()`, `distantia_cluster_kmeans()`, `distantia_matrix()`, `distantia_model_frame()`, `distantia_spatial()`, `distantia_stats()`, `distantia_time_delay()`, `utils_block_size()`, `utils_cluster_hclust_optimizer()`, `utils_cluster_silhouette()`

Examples

```
#weekly covid prevalence
#in 10 California counties
#aggregated by month
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
) |>
  tsl_subset(
```

```
    names = 1:10
  ) |>
  tsl_aggregate(
    new_time = "months",
    fun = max
  )

if(interactive()){
  #plotting first three time series
  tsl_plot(
    tsl = tsl_subset(
      tsl = tsl,
      names = 1:3
    ),
    guide_columns = 3
  )
}

#compute dissimilarity matrix
psi_matrix <- distantia(
  tsl = tsl,
  lock_step = TRUE
) |>
  distantia_matrix()

#optimize hierarchical clustering
kmeans_optimization <- utils_cluster_kmeans_optimizer(
  d = psi_matrix
)

#best solution in first row
head(kmeans_optimization)
```

`utils_cluster_silhouette`

Compute Silhouette Width of a Clustering Solution

Description

The silhouette width is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation).

There are some general guidelines to interpret the individual silhouette widths of the clustered objects (as returned by this function when `mean = FALSE`):

- Close to 1: object is well matched to its own cluster and poorly matched to neighboring clusters.
- Close to 0: the object is between two neighboring clusters.
- Close to -1: the object is likely to be assigned to the wrong cluster

When mean = TRUE, the overall mean of the silhouette widths of all objects is returned. These values should be interpreted as follows:

- Higher than 0.7: robust clustering .
- Higher than 0.5: reasonable clustering.
- Higher than 0.25: weak clustering.

This metric may not perform well when the clusters have irregular shapes or sizes.

This code was adapted from https://svn.r-project.org/R-packages/trunk/cluster/R/silhouette.R.

Usage

```
utils_cluster_silhouette(labels = NULL, d = NULL, mean = FALSE)
```

Arguments

labels	(required, integer vector) Labels resulting from a clustering algorithm applied to d. Must have the same length as columns and rows in d. Default: NULL
d	(required, matrix) distance matrix typically resulting from distantia_matrix(), but any other square matrix should work. Default: NULL
mean	(optional, logical) If TRUE, the mean of the silhouette widths is returned. Default: FALSE

Value

data frame

See Also

Other distantia_support: distantia_aggregate(), distantia_boxplot(), distantia_cluster_hclust(), distantia_cluster_kmeans(), distantia_matrix(), distantia_model_frame(), distantia_spatial(), distantia_stats(), distantia_time_delay(), utils_block_size(), utils_cluster_hclust_optimizer(), utils_cluster_kmeans_optimizer()

Examples

```
#weekly covid prevalence in three California counties
#load as tsl
#subset first 10 time series
#sum by month
tsl <- tsl_initialize(
  x = covid_prevalence,
  name_column = "name",
  time_column = "time"
) |>
  tsl_subset(
    names = 1:10
  ) |>
```

```
  tsl_aggregate(
    new_time = "months",
    method = max
  )

#compute dissimilarity
distantia_df <- distantia(
  tsl = tsl,
  lock_step = TRUE
)

#generate dissimilarity matrix
psi_matrix <- distantia_matrix(
  df = distantia_df
)

#example with kmeans clustering
#-------------------------------------

#kmeans with 3 groups
psi_kmeans <- stats::kmeans(
  x = as.dist(psi_matrix[[1]]),
  centers = 3
)

#case-wise silhouette width
utils_cluster_silhouette(
  labels = psi_kmeans$cluster,
  d = psi_matrix
)

#overall silhouette width
utils_cluster_silhouette(
  labels = psi_kmeans$cluster,
  d = psi_matrix,
  mean = TRUE
)


#example with hierarchical clustering
#-------------------------------------

#hierarchical clustering
psi_hclust <- stats::hclust(
  d = as.dist(psi_matrix[[1]])
)

#generate labels for three groups
psi_hclust_labels <- stats::cutree(
  tree = psi_hclust,
  k = 3,
)
```

```
#case-wise silhouette width
utils_cluster_silhouette(
  labels = psi_hclust_labels,
  d = psi_matrix
)

#overall silhouette width
utils_cluster_silhouette(
  labels = psi_hclust_labels,
  d = psi_matrix,
  mean = TRUE
)
```

`utils_coerce_time_class`

Coerces Vector to a Given Time Class

Description

Coerces Vector to a Given Time Class

Usage

```
utils_coerce_time_class(x = NULL, to = "POSIXct")
```

Arguments

`x`	(required, vector of class Date or POSIXct) time vector to convert. Default: NULL
`to`	(required, class name) class to coerce x to. Either "Date", "POSIXct", "integer" or "numeric". Default: "POSIXct"

Value

time vector

See Also

Other internal_time_handling: `utils_as_time()`, `utils_is_time()`, `utils_new_time()`, `utils_time_keywords()`, `utils_time_keywords_dictionary()`, `utils_time_keywords_translate()`, `utils_time_units()`

Examples

```
x <- utils_coerce_time_class(
  x = c("2024-01-01", "2024-02-01"),
  to = "Date"
)

x
```

```
class(x)

x <- utils_coerce_time_class(
  x = c("2024-01-01", "2024-02-01"),
  to = "POSIXct"
)

x
class(x)

x <- utils_coerce_time_class(
  x = c("2024-01-01", "2024-02-01"),
  to = "numeric"
)

x
class(x)
```

`utils_color_breaks` *Auto Breaks for Matrix Plotting Functions*

Description

Auto Breaks for Matrix Plotting Functions

Usage

```
utils_color_breaks(m = NULL, n = 100)
```

Arguments

`m`	(required, numeric matrix) distance or cost matrix generated by `psi_distance_matrix()` or `psi_cost_matrix()`, but any numeric matrix will work. Default: NULL
`n`	(required, integer) number of colors to compute the breaks for. Default: 100

Value

numeric vector

See Also

Other internal_plotting: `color_continuous()`, `color_discrete()`, `utils_line_color()`, `utils_line_guide()`, `utils_matrix_guide()`, `utils_matrix_plot()`

utils_drop_geometry *Removes Geometry Column from SF Data Frames*

Description

Replicates the functionality of `sf::st_drop_geometry()` without depending on the `sf` package.

Usage

```
utils_drop_geometry(df = NULL)
```

Arguments

`df`	(required, data frame) Input data frame. Default: NULL.

Value

data frame

See Also

Other tsl_processing_internal: `utils_global_scaling_params()`, `utils_optimize_loess()`, `utils_optimize_spline()`, `utils_rescale_vector()`

utils_global_scaling_params *Global Centering and Scaling Parameters of Time Series Lists*

Description

Internal function to compute global scaling parameters (mean and standard deviation) for time series lists. Used within `tsl_transform()` when the scaling function `f_scale_global()` is used as input for the argument `f`.

Warning: this function removes exclusive columns from the data. See function `tsl_subset()`.

Usage

```
utils_global_scaling_params(tsl = NULL, f = NULL, ...)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`f`	(required, function) function `f_scale_global()`. Default: NULL
`...`	(optional, arguments of `f`) Optional arguments for the transformation function.

Value

list

See Also

Other tsl_processing_internal: utils_drop_geometry(), utils_optimize_loess(), utils_optimize_spline(), utils_rescale_vector()

`utils_is_time` *Title*

Description

Title

Usage

```
utils_is_time(x = NULL)
```

Arguments

x	(required, vector) Vector to test. If the class of the vector elements is 'numeric', 'POSIXct', or 'Date', the function returns TRUE. Default: NULL.

Value

logical

See Also

Other internal_time_handling: utils_as_time(), utils_coerce_time_class(), utils_new_time(), utils_time_keywords(), utils_time_keywords_dictionary(), utils_time_keywords_translate(), utils_time_units()

Examples

```
utils_is_time(
  x = c("2024-01-01", "2024-02-01")
)

utils_is_time(
  x = utils_as_time(
    x = c("2024-01-01", "2024-02-01")
    )
)
```

utils_line_color *Handles Line Colors for Sequence Plots*

Description

This is an internal function, but can be used to better understand how line colors are handled within other plotting functions.

Usage

```
utils_line_color(x = NULL, line_color = NULL)
```

Arguments

x	(required, sequence) zoo object or time series list. Default: NULL
line_color	(optional, character vector) vector of colors for the time series columns. Selected palette depends on the number of columns to plot. Default: NULL

Value

color vector

See Also

Other internal_plotting: color_continuous(), color_discrete(), utils_color_breaks(), utils_line_guide(), utils_matrix_guide(), utils_matrix_plot()

utils_line_guide *Guide for Time Series Plots*

Description

Guide for Time Series Plots

Usage

```
utils_line_guide(
  x = NULL,
  position = "topright",
  line_color = NULL,
  line_width = 1,
  length = 1,
  text_cex = 0.7,
  guide_columns = 1,
  subpanel = FALSE
)
```

Arguments

x	(required, sequence) a zoo time series or a time series list. Default: NULL
position	(optional, vector of xy coordinates or character string). This is a condensed version of the x and y arguments of the graphics::legend() function. Coordinates (in the range 0 1) or keyword to position the legend. Accepted keywords are: "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center". Default: "topright".
line_color	(optional, character vector) vector of colors for the time series columns. If NULL, uses the palette "Zissou 1" provided by the function grDevices::hcl.colors(). Default: NULL
line_width	(optional, numeric vector) Widths of the time series lines. Default: 1
length	(optional, numeric) maps to the argument seg.len of graphics::legend(). Length of the lines drawn in the legend. Default: 1
text_cex	(optional, numeric) Multiplier of the text size. Default: 0.7
guide_columns	(optional, integer) Number of columns in which to set the legend items. Default: 1.
subpanel	(optional, logical) internal argument used when generating the multipanel plot produced by distantia_dtw_plot().

Value

plot

See Also

Other internal_plotting: color_continuous(), color_discrete(), utils_color_breaks(), utils_line_color(), utils_matrix_guide(), utils_matrix_plot()

Examples

```
x <- zoo_simulate()

if(interactive()){

  zoo_plot(x, guide = FALSE)

  utils_line_guide(
    x = x,
    position = "right"
  )

}
```

utils_matrix_guide *Color Guide for Matrix Plot*

Description

Plots a color legend for a distance or cost matrix for multi-panel plots or external image editors.

Usage

```
utils_matrix_guide(
  m = NULL,
  matrix_color = NULL,
  breaks = NULL,
  title = NULL,
  text_cex = 1
)
```

Arguments

m	(required, numeric matrix) distance or cost matrix generated by psi_distance_matrix() or psi_cost_matrix(), but any numeric matrix will work. Default: NULL
matrix_color	(optional, character vector) vector of colors. Default: NULL
breaks	(optional, numeric vector) vector of breaks for the color guide. Default: NULL
title	(optional, character string) guide title. Default: NULL
text_cex	(optional, numeric) multiplier for the text size. Default: 1

Value

Plot

See Also

Other internal_plotting: color_continuous(), color_discrete(), utils_color_breaks(), utils_line_color(), utils_line_guide(), utils_matrix_plot()

Examples

```
#prepare time series list
tsl <- tsl_simulate(
  n = 2,
  independent = TRUE
)

#distance matrix between time series
dm <- psi_distance_matrix(
  x = tsl[[1]],
  y = tsl[[2]]
```

```
)

if(interactive()){
  utils_matrix_guide(m = dm)
}
```

`utils_matrix_plot` *Plot Distance or Cost Matrix and Least Cost Path*

Description

This function is a simplified version of `fields::imagePlot()`, by Douglas Nychka. The original version is recommended in case more customization than the provided here is needed.

Usage

```
utils_matrix_plot(
  m = NULL,
  matrix_color = NULL,
  title = NULL,
  subtitle = NULL,
  xlab = NULL,
  ylab = NULL,
  text_cex = 1,
  path = NULL,
  path_width = 1,
  path_color = "black",
  diagonal_width = 1,
  diagonal_color = "white",
  guide = TRUE,
  subpanel = FALSE
)
```

Arguments

`m`	(required, numeric matrix) distance or cost matrix generated by `psi_distance_matrix()` or `psi_cost_matrix()`, but any numeric matrix will work. Default: NULL
`matrix_color`	(optional, character vector) vector of colors. Uses the palette "Zissou 1" by default. Default: NULL
`title`	(optional, character string) plot title. By default, names of the sequences used to compute the matrix `m`. Default: NULL
`subtitle`	(optional, character string) plot subtitle. Default: NULL
`xlab`	(optional, character string) title of the x axis (matrix columns). By default, the name of one of the sequences used to compute the matrix `m`. Default: NULL
`ylab`	(optional, character string) title of the y axis (matrix rows). By default, the name of one of the sequences used to compute the matrix `m`. Default: NULL

text_cex	(optional, numeric) multiplicator of the text size for the plot labels and titles. Default: 1
path	(optional, data frame) least cost path generated with psi_cost_path(). This data frame must have the attribute type == "cost_path, and must have been computed from the same sequences used to compute the matrix m. Default: NULL.
path_width	(optional, numeric) width of the least cost path. Default: 1
path_color	(optional, character string) color of the least-cost path. Default: "black"
diagonal_width	(optional, numeric) width of the diagonal. Set to 0 to remove the diagonal line. Default: 0.5
diagonal_color	(optional, character string) color of the diagonal. Default: "white"
guide	(optional, logical) if TRUE, a color guide for the matrix m is added by utils_matrix_guide().
subpanel	(optional, logical) internal argument used when generating the multi-panel plot produced by distantia_dtw_plot().

Value

plot

See Also

Other internal_plotting: color_continuous(), color_discrete(), utils_color_breaks(), utils_line_color(), utils_line_guide(), utils_matrix_guide()

Examples

```
#prepare time series list
tsl <- tsl_simulate(
  n = 2,
  independent = TRUE
)

#distance matrix between time series
dm <- psi_distance_matrix(
  x = tsl[[1]],
  y = tsl[[2]]
)

#cost matrix
cm <- psi_cost_matrix(
  dist_matrix = dm
  )

#least cost path
cp <- psi_cost_path(
  dist_matrix = dm,
  cost_matrix = cm
)
```

```
#plot cost matrix and least cost path
if(interactive()){
  utils_matrix_plot(
    m = cm,
    path = cp,
    guide = TRUE
  )
}
```

`utils_new_time` *New Time for Time Series Aggregation*

Description

Internal function called by `tsl_aggregate()` and `tsl_resample()` to help transform the input argument `new_time` into the proper format for time series aggregation or resampling.

Usage

```
utils_new_time(tsl = NULL, new_time = NULL, keywords = "aggregate")

utils_new_time_type(
  tsl = NULL,
  new_time = NULL,
  keywords = c("resample", "aggregate")
)
```

Arguments

`tsl`	(required, list) Time series list. Default: NULL
`new_time`	(required, zoo object, numeric, numeric vector, Date vector, POSIXct vector, or keyword) breakpoints defining aggregation groups. Options are: • numeric vector: only for the "numeric" time class, defines the breakpoints for time series aggregation. • "Date" or "POSIXct" vector: as above, but for the time classes "Date" and "POSIXct." In any case, the input vector is coerced to the time class of the `tsl` argument. • numeric: defines fixed with time intervals for time series aggregation. Used as is when the time class is "numeric", and coerced to integer and interpreted as days for the time classes "Date" and "POSIXct". • keyword (see `utils_time_units()` and `tsl_time_summary()`): the common options for the time classes "Date" and "POSIXct" are: "millennia", "centuries", "decades", "years", "quarters", "months", and "weeks". Exclusive keywords for the "POSIXct" time class are: "days", "hours", "minutes", and "seconds". The time class "numeric" accepts keywords coded as scientific numbers, from "1e8" to "1e-8".

keywords (optional, character string or vector) Defines what keywords are returned. If "aggregate", returns valid keywords for zoo_aggregate(). If "resample", returns valid keywords for zoo_resample(). Default: "aggregate".

Value

Vector of class numeric, Date, or POSIXct

See Also

Other internal_time_handling: utils_as_time(), utils_coerce_time_class(), utils_is_time(), utils_time_keywords(), utils_time_keywords_dictionary(), utils_time_keywords_translate(), utils_time_units()

Examples

```
#three time series
#climate and ndvi in Fagus sylvatica stands in Spain, Germany, and Sweden
tsl <- tsl_initialize(
  x = fagus_dynamics,
  name_column = "name",
  time_column = "time"
)

# new time for aggregation using keywords
#-----------------------------------

#get valid keywords for aggregation
tsl_time_summary(
  tsl = tsl,
  keywords = "aggregate"
)$keywords

#if no keyword is used, for aggregation the highest resolution keyword is selected automatically
new_time <- utils_new_time(
  tsl = tsl,
  new_time = NULL,
  keywords = "aggregate"
)

new_time

#if no keyword is used
#for resampling a regular version
#of the original time based on the
#average resolution is used instead
new_time <- utils_new_time(
  tsl = tsl,
  new_time = NULL,
  keywords = "resample"
)
```

```
new_time

#aggregation time vector form keyword "years"
new_time <- utils_new_time(
  tsl = tsl,
  new_time = "years",
  keywords = "aggregate"
)

new_time

#same from shortened keyword
#see utils_time_keywords_dictionary()
utils_new_time(
  tsl = tsl,
  new_time = "year",
  keywords = "aggregate"
)

#same for abbreviated keyword
utils_new_time(
  tsl = tsl,
  new_time = "y",
  keywords = "aggregate"
)

#from a integer defining a time interval in days
utils_new_time(
  tsl = tsl,
  new_time = 365,
  keywords = "aggregate"
)

#using this vector as input for aggregation
tsl_aggregated <- tsl_aggregate(
  tsl = tsl,
  new_time = new_time
)
```

`utils_optimize_loess` *Optimize Loess Models for Time Series Resampling*

Description

Internal function used in `zoo_resample()`. It finds the span parameter of a univariate Loess (Locally Estimated Scatterplot Smoothing.) model `y ~ x` fitted with `stats::loess()` that minimizes the root mean squared error (rmse) between observations and predictions, and returns a model fitted with such span.

Usage

```
utils_optimize_loess(x = NULL, y = NULL, max_complexity = FALSE)
```

Arguments

x	(required, numeric vector) predictor, a time vector coerced to numeric. Default: NULL
y	(required, numeric vector) response, a column of a zoo object. Default: NULL
max_complexity	(required, logical). If TRUE, RMSE optimization is ignored, and the model of maximum complexity is returned. Default: FALSE

Value

Loess model.

See Also

Other tsl_processing_internal: utils_drop_geometry(), utils_global_scaling_params(), utils_optimize_spline(), utils_rescale_vector()

Examples

```
#zoo time series
xy <- zoo_simulate(
  cols = 1,
  rows = 30
)

#optimize loess model
m <- utils_optimize_loess(
  x = as.numeric(zoo::index(xy)), #predictor
  y = xy[, 1] #response
)

print(m)

#plot observation
plot(
  x = zoo::index(xy),
  y = xy[, 1],
  col = "forestgreen",
  type = "l",
  lwd = 2
  )

#plot prediction
points(
  x = zoo::index(xy),
  y = stats::predict(
    object = m,
    newdata = as.numeric(zoo::index(xy))
```

```
  ),
col = "red4"
)
```

`utils_optimize_spline` *Optimize Spline Models for Time Series Resampling*

Description

Internal function used in `zoo_resample()`. It finds optimal `df` parameter of a smoothing spline model `y ~ x` fitted with `stats::smooth.spline()` that minimizes the root mean squared error (rmse) between observations and predictions, and returns a model fitted with such `df`.

Usage

```
utils_optimize_spline(x = NULL, y = NULL, max_complexity = FALSE)
```

Arguments

`x`	(required, numeric vector) predictor, a time vector coerced to numeric. Default: NULL
`y`	(required, numeric vector) response, a column of a zoo object. Default: NULL
`max_complexity`	(required, logical). If TRUE, RMSE optimization is ignored, and the model of maximum complexity is returned. Default: FALSE

Value

Object of class "smooth.spline".

See Also

Other tsl_processing_internal: `utils_drop_geometry()`, `utils_global_scaling_params()`, `utils_optimize_loess()`, `utils_rescale_vector()`

Examples

```
#zoo time series
xy <- zoo_simulate(
  cols = 1,
  rows = 30
)

#optimize splines model
m <- utils_optimize_spline(
  x = as.numeric(zoo::index(xy)), #predictor
  y = xy[, 1] #response
)
```

```
print(m)

#plot observation
plot(
  x = zoo::index(xy),
  y = xy[, 1],
  col = "forestgreen",
  type = "l",
  lwd = 2
  )

#plot prediction
points(
  x = zoo::index(xy),
  y = stats::predict(
    object = m,
    x = as.numeric(zoo::index(xy))
  )$y,
  col = "red"
)
```

`utils_rescale_vector` *Rescale Numeric Vector to a New Data Range*

Description

Rescale Numeric Vector to a New Data Range

Usage

```
utils_rescale_vector(
  x = NULL,
  new_min = 0,
  new_max = 1,
  old_min = NULL,
  old_max = NULL
)
```

Arguments

`x`	(required, numeric vector) Numeric vector. Default: `NULL`
`new_min`	(optional, numeric) New minimum value. Default: `0`
`new_max`	(optional_numeric) New maximum value. Default: `1`
`old_min`	(optional, numeric) Old minimum value. Default: `NULL`
`old_max`	(optional_numeric) Old maximum value. Default: `NULL`

Value

numeric vector

See Also

Other tsl_processing_internal: utils_drop_geometry(), utils_global_scaling_params(), utils_optimize_loess(), utils_optimize_spline()

Examples

```
out <- utils_rescale_vector(
  x = stats::rnorm(100),
  new_min = 0,
  new_max = 100,
  )

out
```

---

utils_time_keywords *Valid Aggregation Keywords*

Description

Internal function to obtain valid aggregation keywords from a zoo object or a time series list.

Usage

```
utils_time_keywords(tsl = NULL)
```

Arguments

tsl	(required, list) Time series list. Default: NULL

Value

Character string, aggregation keyword, or "none".

See Also

Other internal_time_handling: utils_as_time(), utils_coerce_time_class(), utils_is_time(), utils_new_time(), utils_time_keywords_dictionary(), utils_time_keywords_translate(), utils_time_units()

Examples

```
#one minute time series
#----------------------------------
tsl <- tsl_simulate(
  time_range = c(
    Sys.time() - 60,
    Sys.time()
  )
)

#valid keywords for aggregation and/or resampling
utils_time_keywords(
  tsl = tsl
)

#10 minutes time series
#----------------------------------
tsl <- tsl_simulate(
  time_range = c(
    Sys.time() - 600,
    Sys.time()
  )
)

utils_time_keywords(
  tsl = tsl
)

#10 hours time series
#----------------------------------
tsl <- tsl_simulate(
  time_range = c(
    Sys.time() - 6000,
    Sys.time()
  )
)

utils_time_keywords(
  tsl = tsl
)

#10 days time series
#----------------------------------
tsl <- tsl_simulate(
  time_range = c(
    Sys.Date() - 10,
    Sys.Date()
  )
)

utils_time_keywords(
  tsl = tsl
```

```
)

#10 years time series
#----------------------------------
tsl <- tsl_simulate(
  time_range = c(
    Sys.Date() - 3650,
    Sys.Date()
  )
)

utils_time_keywords(
  tsl = tsl
)
```

`utils_time_keywords_dictionary` *Dictionary of Time Keywords*

Description

Called by `utils_time_keywords_translate()` to generate a data frame that helps translate misnamed or abbreviated time keywords, like "day", "daily", or "d", into correct ones such as "days".

Usage

```
utils_time_keywords_dictionary()
```

Value

data frame

See Also

Other internal_time_handling: `utils_as_time()`, `utils_coerce_time_class()`, `utils_is_time()`, `utils_new_time()`, `utils_time_keywords()`, `utils_time_keywords_translate()`, `utils_time_units()`

Examples

```
df <- utils_time_keywords_dictionary()
```

`utils_time_keywords_translate` *Translates The User's Time Keywords Into Valid Ones*

Description

Internal function to translate misnamed or abbreviated keywords into valid ones. Uses `utils_time_keywords_dictionary(` as reference dictionary.

Usage

```
utils_time_keywords_translate(keyword = NULL)
```

Arguments

`keyword`	(optional, character string) A time keyword such as "day". Default: NULL

Value

Time keyword.

See Also

Other internal_time_handling: `utils_as_time()`, `utils_coerce_time_class()`, `utils_is_time()`, `utils_new_time()`, `utils_time_keywords()`, `utils_time_keywords_dictionary()`, `utils_time_units()`

Examples

```
#millennia
utils_time_keywords_translate(
  keyword = "1000 years"
)

utils_time_keywords_translate(
  keyword = "1000 y"
)

utils_time_keywords_translate(
  keyword = "thousands"
)

#years
utils_time_keywords_translate(
  keyword = "year"
)

utils_time_keywords_translate(
  keyword = "y"
)
```

```
#days
utils_time_keywords_translate(
  keyword = "d"
)

utils_time_keywords_translate(
  keyword = "day"
)

#seconds
utils_time_keywords_translate(
  keyword = "s"
)

utils_time_keywords_translate(
  keyword = "sec"
)
```

`utils_time_units` *Data Frame with Supported Time Units*

Description

Returns a data frame with the names of the supported time units, the classes that can handle each time unit, and a the threshold used to identify what time units can be used when aggregating a time series.

Usage

```
utils_time_units(all_columns = FALSE, class = NULL)
```

Arguments

`all_columns`	(optional, logical) If TRUE, all columns are returned. Default: FALSE
`class`	(optional, class name). Used to filter rows and columns. Accepted values are "numeric", "Date", and "POSIXct". Default: NULL

Value

data frame

See Also

Other internal_time_handling: `utils_as_time()`, `utils_coerce_time_class()`, `utils_is_time()`, `utils_new_time()`, `utils_time_keywords()`, `utils_time_keywords_dictionary()`, `utils_time_keywords_translat`

Examples

```
df <- utils_time_units()
head(df)
```

`zoo_aggregate` *Aggregate Cases in Zoo Time Series*

Description

Aggregate Cases in Zoo Time Series

Usage

```
zoo_aggregate(x = NULL, new_time = NULL, f = mean, ...)
```

Arguments

`x`	(required, zoo object) Time series to aggregate. Default: NULL
`new_time`	(optional, zoo object, keyword, or time vector) New time to aggregate `x` to. The available options are: • NULL: the highest resolution keyword returned by `zoo_time(x)$keywords` is used to generate a new time vector to aggregate `x`. • zoo object: the index of the given zoo object is used as template to aggregate `x`. • time vector: a vector with new times to resample `x` to. If time in `x` is of class "numeric", this vector must be numeric as well. Otherwise, vectors of classes "Date" and "POSIXct" can be used indistinctly. • keyword: a valid keyword returned by `zoo_time(x)$keywords`, used to generate a time vector with the relevant units. • numeric of length 1: interpreted as new time interval, in the highest resolution units returned by `zoo_time(x)$units`.
`f`	(optional, quoted or unquoted function name) Name of a standard or custom function to aggregate numeric vectors. Typical examples are `mean`, `max`,`min`, `median`, and `quantile`. Default: `mean`.
`...`	(optional, additional arguments) additional arguments to `f`.

Value

zoo object

See Also

Other zoo_functions: `zoo_name_clean()`, `zoo_name_get()`, `zoo_name_set()`, `zoo_permute()`, `zoo_plot()`, `zoo_resample()`, `zoo_smooth_exponential()`, `zoo_smooth_window()`, `zoo_time()`, `zoo_to_tsl()`, `zoo_vector_to_matrix()`

Examples

```
#full range of calendar dates
x <- zoo_simulate(
  rows = 1000,
  time_range = c(
    "0000-01-01",
    as.character(Sys.Date())
    )
)

#plot time series
if(interactive()){
  zoo_plot(x)
}


#find valid aggregation keywords
x_time <- zoo_time(x)
x_time$keywords

#mean value by millennia (extreme case!!!)
x_millennia <- zoo_aggregate(
  x = x,
  new_time = "millennia",
  f = mean
)

if(interactive()){
  zoo_plot(x_millennia)
}

#max value by centuries
x_centuries <- zoo_aggregate(
  x = x,
  new_time = "centuries",
  f = max
)

if(interactive()){
  zoo_plot(x_centuries)
}

#quantile 0.75 value by centuries
x_centuries <- zoo_aggregate(
  x = x,
  new_time = "centuries",
  f = stats::quantile,
  probs = 0.75 #argument of stats::quantile()
)

if(interactive()){
  zoo_plot(x_centuries)
```

```
}
```

`zoo_name_clean` *Clean Name of a Zoo Time Series*

Description

Combines `utils_clean_names()` and `zoo_name_set()` to help clean, abbreviate, capitalize, and add a suffix or a prefix to the name of a zoo object.

Usage

```
zoo_name_clean(
  x = NULL,
  lowercase = FALSE,
  separator = "_",
  capitalize_first = FALSE,
  capitalize_all = FALSE,
  length = NULL,
  suffix = NULL,
  prefix = NULL
)
```

Arguments

`x`	(required, zoo object) Zoo time series to analyze. Default: NULL.
`lowercase`	(optional, logical) If TRUE, all names are coerced to lowercase. Default: FALSE
`separator`	(optional, character string) Separator when replacing spaces and dots. Also used to separate `suffix` and `prefix` from the main word. Default: "_".
`capitalize_first`	(optional, logical) Indicates whether to capitalize the first letter of each name Default: FALSE.
`capitalize_all`	(optional, logical) Indicates whether to capitalize all letters of each name Default: FALSE.
`length`	(optional, integer) Minimum length of abbreviated names. Names are abbreviated via `abbreviate()`. Default: NULL.
`suffix`	(optional, character string) Suffix for the clean names. Default: NULL.
`prefix`	(optional, character string) Prefix for the clean names. Default: NULL.

Value

zoo time series

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_get(), zoo_name_set(), zoo_permute(), zoo_plot(), zoo_resample(), zoo_smooth_exponential(), zoo_smooth_window(), zoo_time(), zoo_to_tsl(), zoo_vector_to_matrix()

Examples

```
#simulate zoo time series
x <- zoo_simulate()

#get current name
zoo_name_get(x = x)

#change name
x <- zoo_name_set(
  x = x,
  name = "My.New.name"
)

zoo_name_get(x = x)

#clean name
x <- zoo_name_clean(
  x = x,
  lowercase = TRUE
)

zoo_name_get(x = x)
```

---

zoo_name_get *Get Name of a Zoo Time Series*

Description

Just a convenient wrapper of `attributes(x)$name`.

Usage

```
zoo_name_get(x = NULL)
```

Arguments

x	(required, zoo object) Zoo time series to analyze. Default: NULL.

Value

character string

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_clean(), zoo_name_set(), zoo_permute(), zoo_plot(), zoo_resample(), zoo_smooth_exponential(), zoo_smooth_window(), zoo_time(), zoo_to_tsl(), zoo_vector_to_matrix()

Examples

```
#simulate zoo time series
x <- zoo_simulate()

#get current name
zoo_name_get(x = x)

#change name
x <- zoo_name_set(
  x = x,
  name = "My.New.name"
)

zoo_name_get(x = x)

#clean name
x <- zoo_name_clean(
  x = x,
  lowercase = TRUE
)

zoo_name_get(x = x)
```

---

zoo_name_set *Set Name of a Zoo Time Series*

Description

Zoo time series do not have an attribute 'name'. However, within distantia, to keep data consistency in several plotting and analysis operations, an attribute 'name' is used for these objects. This function is a convenient wrapper of attr(x = x, which = "name") <- "xxx".

Usage

```
zoo_name_set(x = NULL, name = NULL)
```

Arguments

x	(required, zoo object) Zoo time series to analyze. Default: NULL.
name	(required, character string) name or new name of the zoo object. If NULL, x is returned as is. Default: NULL

Value

zoo time series

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_clean(), zoo_name_get(), zoo_permute(), zoo_plot(), zoo_resample(), zoo_smooth_exponential(), zoo_smooth_window(), zoo_time(), zoo_to_tsl(), zoo_vector_to_matrix()

Examples

```
#simulate zoo time series
x <- zoo_simulate()

#get current name
zoo_name_get(x = x)

#change name
x <- zoo_name_set(
  x = x,
  name = "My.New.name"
)

zoo_name_get(x = x)

#clean name
x <- zoo_name_clean(
  x = x,
  lowercase = TRUE
)

zoo_name_get(x = x)
```

---

zoo_permute *Random or Restricted Permutation of Zoo Time Series*

Description

Fast permutation of zoo time series for null model testing using a fast and efficient C++ implementations of different restricted and free permutation methods.

The available permutation methods are:

- "free" (see permute_free_cpp()): Unrestricted and independent re-shuffling of individual cases across rows and columns. Individual values are relocated to a new row and column within the dimensions of the original matrix.
- "free_by_row" (see permute_free_by_row_cpp()): Unrestricted re-shuffling of complete rows. Each individual row is given a new random row number, and the data matrix is re-ordered accordingly.

- "restricted" (see permute_restricted_cpp()): Data re-shuffling across rows and columns is restricted to blocks of contiguous rows. The algorithm divides the data matrix into a set of blocks of contiguous rows, and individual cases are then assigned to a new row and column within their original block.
- "restricted_by_row" (see permute_restricted_by_row_cpp()): Re-shuffling of complete rows is restricted to blocks of contiguous rows. The algorithm divides the data matrix into a set of blocks of contiguous rows, each individual row is given a new random row number within its original block, and the block is reordered accordingly to generate the permuted output.

This function supports a parallelization setup via future::plan(), and progress bars provided by the package progressr.

Usage

```
zoo_permute(
  x = NULL,
  repetitions = 1L,
  permutation = "restricted_by_row",
  block_size = NULL,
  seed = 1L
)
```

Arguments

x	(required, zoo object) zoo time series. Default: NULL
repetitions	(optional, integer) number of permutations to compute. Large numbers may compromise your R session. Default: 1
permutation	(optional, character string) permutation method. Valid values are listed below from higher to lower induced randomness: • "free": unrestricted re-shuffling of individual cases across rows and columns. Ignores block_size. • "free_by_row": unrestricted re-shuffling of complete rows. Ignores block size. • "restricted": restricted shuffling across rows and columns within blocks of rows. • "restricted_by_row": restricted re-shuffling of rows within blocks.
block_size	(optional, integer) Block size in number of rows for restricted permutations. Only relevant when permutation methods are "restricted" or "restricted_by_row". A block of size n indicates that the original data is pre-divided into blocks of such size, and a given row can only be permuted within their original block. If NULL, defaults to the rounded one tenth of the number of rows in x. Default: NULL.
seed	(optional, integer) initial random seed to use during permutations. Default: 1

Value

Time Series List

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_clean(), zoo_name_get(), zoo_name_set(), zoo_plot(), zoo_resample(), zoo_smooth_exponential(), zoo_smooth_window(), zoo_time(), zoo_to_tsl(), zoo_vector_to_matrix()

Examples

```
#simulate zoo time series
x <- zoo_simulate(cols = 2)

if(interactive()){
  zoo_plot(x)
}

#free
x_free <- zoo_permute(
  x = x,
  permutation = "free",
  repetitions = 2
)

if(interactive()){
  tsl_plot(
    tsl = x_free,
    guide = FALSE
    )
}

#free by row
x_free_by_row <- zoo_permute(
  x = x,
  permutation = "free_by_row",
  repetitions = 2
)

if(interactive()){
  tsl_plot(
    tsl = x_free_by_row,
    guide = FALSE
  )
}

#restricted
x_restricted <- zoo_permute(
  x = x,
  permutation = "restricted",
  repetitions = 2
)

if(interactive()){
  tsl_plot(
    tsl = x_restricted,
```

```
    guide = FALSE
  )
}

#restricted by row
x_restricted_by_row <- zoo_permute(
  x = x,
  permutation = "restricted_by_row",
  repetitions = 2
)

if(interactive()){
  tsl_plot(
    tsl = x_restricted_by_row,
    guide = FALSE
  )
}
```

---

zoo_plot *Plot Zoo Time Series*

Description

Plot Zoo Time Series

Usage

```
zoo_plot(
  x = NULL,
  line_color = NULL,
  line_width = 1,
  xlim = NULL,
  ylim = NULL,
  title = NULL,
  xlab = NULL,
  ylab = NULL,
  text_cex = 1,
  guide = TRUE,
  guide_position = "topright",
  guide_cex = 0.8,
  vertical = FALSE,
  subpanel = FALSE
)
```

Arguments

x	(required, zoo object) zoo time series. Default: NULL

line_color	(optional, character vector) vector of colors for the distance or cost matrix. If NULL, uses an appropriate palette generated with grDevices::palette.colors(). Default: NULL
line_width	(optional, numeric vector) Width of the time series lines. Default: 1
xlim	(optional, numeric vector) Numeric vector with the limits of the x axis. Default: NULL
ylim	(optional, numeric vector) Numeric vector with the limits of the x axis. Default: NULL
title	(optional, character string) Main title of the plot. If NULL, it's set to the name of the time series. Default: NULL
xlab	(optional, character string) Title of the x axis. Disabled if subpanel or vertical are TRUE. If NULL, the word "Time" is used. Default: NULL
ylab	(optional, character string) Title of the x axis. Disabled if subpanel or vertical are TRUE. If NULL, it is left empty. Default: NULL
text_cex	(optional, numeric) Multiplicator of the text size. Default: 1
guide	(optional, logical) If TRUE, plots a legend. Default: TRUE
guide_position	(optional, vector of xy coordinates or character string). This is a condensed version of the x and y arguments of the graphics::legend() function. Coordinates (in the range 0 1) or keyword to position the legend. Accepted keywords are: "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center". Default: "topright".
guide_cex	(optional, numeric) Size of the guide's text and separation between the guide's rows. Default: 0.7.
vertical	(optional, logical) For internal use within the package in multipanel plots. Switches the plot axes. Disabled if subpanel = FALSE. Default: FALSE
subpanel	(optional, logical) For internal use within the package in multipanel plots. Strips down the plot for a sub-panel. Default: FALSE

Value

A plot.

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_clean(), zoo_name_get(), zoo_name_set(), zoo_permute(), zoo_resample(), zoo_smooth_exponential(), zoo_smooth_window(), zoo_time(), zoo_to_tsl(), zoo_vector_to_matrix()

Examples

```
#simulate zoo time series
x <- zoo_simulate()

if(interactive()){

  zoo_plot(
```

```
    x = x,
    xlab = "Date",
    ylab = "Value",
    title = "My time series"
  )

}
```

`zoo_resample` *Resample Zoo Objects to a New Time*

Description

Objective

Time series resampling involves interpolating new values for time steps not available in the original time series. This operation is useful to:

- Transform irregular time series into regular.
- Align time series with different temporal resolutions.
- Increase (upsampling) or decrease (downsampling) the temporal resolution of a time series.

On the other hand, time series resampling **should not be used** to extrapolate new values outside of the original time range of the time series, or to increase the resolution of a time series by a factor of two or more. These operations are known to produce non-sensical results.

Methods This function offers three methods for time series interpolation:

- "linear" (default): interpolation via piecewise linear regression as implemented in `zoo::na.approx()`.
- "spline": cubic smoothing spline regression as implemented in `stats::smooth.spline()`.
- "loess": local polynomial regression fitting as implemented in `stats::loess()`.

These methods are used to fit models `y ~ x` where `y` represents the values of a univariate time series and `x` represents a numeric version of its time.

The functions `utils_optimize_spline()` and `utils_optimize_loess()` are used under the hood to optimize the complexity of the methods "spline" and "loess" by finding the configuration that minimizes the root mean squared error (RMSE) between observed and predicted `y`. However, when the argument `max_complexity = TRUE`, the complexity optimization is ignored, and a maximum complexity model is used instead.

New time

The argument `new_time` offers several alternatives to help define the new time of the resulting time series:

- `NULL`: the target time series (`x`) is resampled to a regular time within its original time range and number of observations.
- `zoo object`: a zoo object to be used as template for resampling. Useful when the objective is equalizing the frequency of two separate zoo objects.

- `time vector`: a time vector of a class compatible with the time in `x`.
- `keyword`: character string defining a resampling keyword, obtained via `zoo_time(x, keywords = "resample")$keywords`..
- `numeric`: a single number representing the desired interval between consecutive samples in the units of `x` (relevant units can be obtained via `zoo_time(x)$units`).

Step by Step

The steps to resample a time series list are:

1. The time interpolation range taken from the index of the zoo object. This step ensures that no extrapolation occurs during resampling.
2. If `new_time` is provided, any values of `new_time` outside of the minimum and maximum interpolation times are removed to avoid extrapolation. If `new_time` is not provided, a regular time within the interpolation time range of the zoo object is generated.
3. For each univariate time time series, a model `y ~ x`, where `y` is the time series and `x` is its own time coerced to numeric is fitted.
   - If `max_complexity == FALSE` and `method = "spline"` or `method = "loess"`, the model with the complexity that minimizes the root mean squared error between the observed and predicted `y` is returned.
   - If `max_complexity == TRUE` and `method = "spline"` or `method = "loess"`, the first valid model closest to a maximum complexity is returned.
4. The fitted model is predicted over `new_time` to generate the resampled time series.

Other Details

Please use this operation with care, as there are limits to the amount of resampling that can be done without distorting the data. The safest option is to keep the distance between new time points within the same magnitude of the distance between the old time points.

Usage

```
zoo_resample(
  x = NULL,
  new_time = NULL,
  method = "linear",
  max_complexity = FALSE
)
```

Arguments

`x`	(required, zoo object) Time series to resample. Default: NULL
`new_time`	(optional, zoo object, keyword, or time vector) New time to resample `x` to. The available options are: • NULL: a regular version of the time in `x` is generated and used for resampling. • zoo object: the index of the given zoo object is used as template to resample `x`.

- time vector: a vector with new times to resample `x` to. If time in `x` is of class "numeric", this vector must be numeric as well. Otherwise, vectors of classes "Date" and "POSIXct" can be used indistinctly.
- keyword: a valid keyword returned by `zoo_time(x)$keywords`, used to generate a time vector with the relevant units.
- numeric of length 1: interpreted as new time interval, in the highest resolution units returned by `zoo_time(x)$units`.

`method` (optional, character string) Name of the method to resample the time series. One of "linear", "spline" or "loess". Default: "linear".

`max_complexity` (required, logical). Only relevant for methods "spline" and "loess". If TRUE, model optimization is ignored, and the a model of maximum complexity (an overfitted model) is used for resampling. Default: FALSE

Value

zoo object

See Also

Other zoo_functions: `zoo_aggregate()`, `zoo_name_clean()`, `zoo_name_get()`, `zoo_name_set()`, `zoo_permute()`, `zoo_plot()`, `zoo_smooth_exponential()`, `zoo_smooth_window()`, `zoo_time()`, `zoo_to_tsl()`, `zoo_vector_to_matrix()`

Examples

```
#simulate irregular time series
x <- zoo_simulate(
  cols = 2,
  rows = 50,
  time_range = c("2010-01-01", "2020-01-01"),
  irregular = TRUE
  )

#plot time series
if(interactive()){
  zoo_plot(x)
}

#intervals between samples
x_intervals <- diff(zoo::index(x))
x_intervals

#create regular time from the minimum of the observed intervals
new_time <- seq.Date(
  from = min(zoo::index(x)),
  to = max(zoo::index(x)),
  by = floor(min(x_intervals))
)

new_time
```

```
diff(new_time)

#resample using piecewise linear regression
x_linear <- zoo_resample(
  x = x,
  new_time = new_time,
  method = "linear"
)

#resample using max complexity splines
x_spline <- zoo_resample(
  x = x,
  new_time = new_time,
  method = "spline",
  max_complexity = TRUE
)

#resample using max complexity loess
x_loess <- zoo_resample(
  x = x,
  new_time = new_time,
  method = "loess",
  max_complexity = TRUE
)


#intervals between new samples
diff(zoo::index(x_linear))
diff(zoo::index(x_spline))
diff(zoo::index(x_loess))

#plotting results
if(interactive()){

  par(mfrow = c(4, 1), mar = c(3,3,2,2))

  zoo_plot(
    x,
    guide = FALSE,
    title = "Original"
    )

  zoo_plot(
    x_linear,
    guide = FALSE,
    title = "Method: linear"
  )

  zoo_plot(
    x_spline,
    guide = FALSE,
    title = "Method: spline"
    )
```

```
  zoo_plot(
    x_loess,
    guide = FALSE,
    title = "Method: loess"
  )

}
```

`zoo_simulate` *Simulate a Zoo Time Series*

Description

Generates simulated zoo time series.

Usage

```
zoo_simulate(
  name = "A",
  cols = 5,
  rows = 100,
  time_range = c("2010-01-01", "2020-01-01"),
  data_range = c(0, 1),
  seasons = 0,
  na_fraction = 0,
  independent = FALSE,
  irregular = TRUE,
  seed = NULL
)
```

Arguments

`name`	(optional, character string) Name of the zoo object, to be stored in the attribute "name". Default: "A"
`cols`	(optional, integer) Number of time series. Default: 5
`rows`	(optional, integer) Length of the time series. Minimum is 10, but maximum is not limited. Very large numbers might crash the R session. Default: 100
`time_range`	(optional character or numeric vector) Interval of the time series. Either a character vector with dates in format YYYY-MM-DD or or a numeric vector. If there is a mismatch between `time_range` and `rows` (for example, the number of days in `time_range` is smaller than `rows`), the upper value in `time_range` is adapted to `rows`. Default: c("2010-01-01", "2020-01-01")
`data_range`	(optional, numeric vector of length 2) Extremes of the simulated time series values. The simulated time series are independently adjusted to random values within the provided range. Default: c(0, 1)

seasons	(optional, integer) Number of seasons in the resulting time series. The maximum number of seasons is computed as `floor(rows/3)`. Default: 0
na_fraction	(optional, numeric) Value between 0 and 0.5 indicating the approximate fraction of NA data in the simulated time series. Default: 0.
independent	(optional, logical) If TRUE, each new column in a simulated time series is averaged with the previous column. Irrelevant when `cols <= 2`, and hard to perceive in the output when `seasons > 0`. Default: FALSE
irregular	(optional, logical) If TRUE, the time series is created with 20 percent more rows, and a random 20 percent of rows are removed at random. Default: TRUE
seed	(optional, integer) Random seed used to simulate the zoo object. Default: NULL

Value

zoo object

See Also

Other simulate_time_series: `tsl_simulate()`

Examples

```
#generates a different time series on each execution when 'seed = NULL'
x <- zoo_simulate()

#returns a zoo object
class(x)

#time series names are uppercase letters
#this attribute is not defined in the zoo class and might be lost during data transformations
attributes(x)$name

#column names are lowercase letters
names(x)

#plotting methods
if(interactive()){

  #plot time series with default zoo method
  plot(x)

  #plot time series with distantia
  zoo_plot(
    x = x,
    xlab = "Date",
    ylab = "Value",
    title = "My time series"
  )

}
```

`zoo_smooth_exponential` *Exponential Smoothing of Zoo Time Series*

Description

Applies exponential smoothing to a zoo time series object, where each value is a weighted average of the current value and past smoothed values. This method is useful for reducing noise in time series data while preserving the general trend.

Usage

```
zoo_smooth_exponential(x = NULL, alpha = 0.2)
```

Arguments

`x`	(required, zoo object) time series to smooth Default: NULL
`alpha`	(required, numeric) Smoothing factor in the range (0, 1]. Determines the weight of the current value relative to past values. A higher value gives more weight to recent observations, while a lower value gives more weight to past observations. Default: 0.2

Value

zoo object

See Also

Other zoo_functions: `zoo_aggregate()`, `zoo_name_clean()`, `zoo_name_get()`, `zoo_name_set()`, `zoo_permute()`, `zoo_plot()`, `zoo_resample()`, `zoo_smooth_window()`, `zoo_time()`, `zoo_to_tsl()`, `zoo_vector_to_matrix()`

Examples

```
x <- zoo_simulate()

x_smooth <- zoo_smooth_exponential(
  x = x,
  alpha = 0.2
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(x_smooth)
}
```

zoo_smooth_window *Rolling Window Smoothing of Zoo Time Series*

Description

Just a fancy wrapper for zoo::rollapply().

Usage

```
zoo_smooth_window(x = NULL, window = 3, f = mean, ...)
```

Arguments

x	(required, zoo object) Time series to smooth Default: NULL
window	(optional, integer) Smoothing window width, in number of cases. Default: 3
f	(optional, quoted or unquoted function name) Name of a standard or custom function to aggregate numeric vectors. Typical examples are mean, max,min, median, and quantile. Default: mean.
...	(optional, additional arguments) additional arguments to f.

Value

zoo object

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_clean(), zoo_name_get(), zoo_name_set(), zoo_permute(), zoo_plot(), zoo_resample(), zoo_smooth_exponential(), zoo_time(), zoo_to_tsl(), zoo_vector_to_matrix()

Examples

```
x <- zoo_simulate()

x_smooth <- zoo_smooth_window(
  x = x,
  window = 5,
  f = mean
)

if(interactive()){
  zoo_plot(x)
  zoo_plot(x_smooth)
}
```

`zoo_time` *Get Time Features from Zoo Objects*

Description

This function generates a data frame summarizing the time features (class, length, resolution, and others) of zoo time series.

Usage

```
zoo_time(x = NULL, keywords = c("resample", "aggregate"))
```

Arguments

`x`	(required, zoo object) Zoo time series to analyze. Default: NULL.
`keywords`	(optional, character string or vector) Defines what keywords are returned. If "aggregate", returns valid keywords for `zoo_aggregate()`. If "resample", returns valid keywords for `zoo_resample()`. If both, returns all valid keywords. Default: c("aggregate", "resample").

Value

Data frame with the following columns:

- `name` (string): time series name.
- `rows` (integer): number of observations.
- `class` (string): time class, one of "Date", "POSIXct", or "numeric."
- `units` (string): units of the time series.
- `length` (numeric): total length of the time series expressed in `units`.
- `resolution` (numeric): average interval between observations expressed in `units`.
- `begin` (date or numeric): begin time of the time series.
- `end` (date or numeric): end time of the time series.
- `keywords` (character vector): valid keywords for `tsl_aggregate()` or `tsl_resample()`, depending on the value of the argument `keywords`.

See Also

Other zoo_functions: `zoo_aggregate()`, `zoo_name_clean()`, `zoo_name_get()`, `zoo_name_set()`, `zoo_permute()`, `zoo_plot()`, `zoo_resample()`, `zoo_smooth_exponential()`, `zoo_smooth_window()`, `zoo_to_tsl()`, `zoo_vector_to_matrix()`

Examples

```
#simulate a zoo time series
x <- zoo_simulate(
  rows = 150,
  time_range = c(
    Sys.Date() - 365,
    Sys.Date()
  ),
  irregular = TRUE
)

#time data frame
zoo_time(
  x = x
)
```

---

zoo_to_tsl *Convert Individual Zoo Objects to Time Series List*

Description

Internal function to wrap a zoo object into a time series list.

Usage

```
zoo_to_tsl(x = NULL)
```

Arguments

x	(required, zoo object) Time series. Default: NULL

Value

time series list of length one.

See Also

Other zoo_functions: zoo_aggregate(), zoo_name_clean(), zoo_name_get(), zoo_name_set(), zoo_permute(), zoo_plot(), zoo_resample(), zoo_smooth_exponential(), zoo_smooth_window(), zoo_time(), zoo_vector_to_matrix()

Examples

```
#create zoo object
x <- zoo_simulate()
class(x)

#to time series list
tsl <- zoo_to_tsl(
```

```
  x = x
)

class(tsl)
class(tsl[[1]])
names(tsl)
attributes(tsl[[1]])$name
```

`zoo_vector_to_matrix` *Coerce Coredata of Univariate Zoo Time Series to Matrix*

Description

Transforms vector coredata of univariate zoo time series to class matrix. If the input zoo time series has the attribute "name", the output inherits the value of such attribute.

Multivariate zoo objects are returned without changes.

Usage

```
zoo_vector_to_matrix(x = NULL, name = NULL)
```

Arguments

`x`	(required, zoo object) zoo time series. Default: NULL
`name`	(required, character string) name of the matrix column. Default: NULL

Value

zoo time series

See Also

Other zoo_functions: `zoo_aggregate()`, `zoo_name_clean()`, `zoo_name_get()`, `zoo_name_set()`, `zoo_permute()`, `zoo_plot()`, `zoo_resample()`, `zoo_smooth_exponential()`, `zoo_smooth_window()`, `zoo_time()`, `zoo_to_tsl()`

Examples

```
#create zoo object from vector
x <- zoo::zoo(
  x = runif(100)
)

#coredata is not a matrix
is.matrix(zoo::coredata(x))

#convert to matrix
y <- zoo_vector_to_matrix(
  x = x
```

```
)

#coredata is now a matrix
is.matrix(zoo::coredata(y))
```

Index