Package ‘gflow’

September 9, 2025

Title Gradient Flow Data Analysis Framework

Version 0.1.0

Description The gflow package implements geometric methods for analyzing high-dimensional data by exploiting the fact that most real-world high-dimensional datasets are highly structured and sparse, displaying an intrinsic dimension often orders of magnitude smaller than the number of features (the dimension of the ambient space in which the data is embedded). The package provides tools for modeling the intrinsic geometric structures of data and estimating conditional expectations of response variables over these geometric objects. Inferential analysis is performed using the Morse-Smale regression approach, which decomposes the domain (the geometric object associated with the data) into gradient flow cells. Within each cell, the regression model exhibits an essentially monotonic structure that can be analyzed using classical statistical methods.

Copyright file inst/COPYRIGHTS

License GPL (>= 3)

URL https://github.com/pgajer/gflow

BugReports https://github.com/pgajer/gflow/issues

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

SystemRequirements C++17, GNU make, OpenMP

Depends R (>= 3.5.0)

Imports stats,
graphics,
grDevices,
rgl,
FNN,
doParallel,
foreach,
parallel,
igraph,
segmented,
transport

Suggests randomForest,
np,
mgcv,
kernlab,
glmnet,
e1071,
KernSmooth,
CVST,
dbscan,
clValid,
mclust,
RColorBrewer,
viridis,
Matrix,
utils,
testthat (>= 3.0.0),
rstan,
HDInterval,
infotheo,
rootSolve,
lme4,
clue

Config/testthat/edition 3

Contents

`gflow-package` *gflow: Geometric Data Analysis Through Gradient Flow*

Description

The gflow package implements geometric methods for analyzing high-dimensional data by exploiting the fact that most real-world high-dimensional datasets are highly structured and sparse, displaying an intrinsic dimension often orders of magnitude smaller than the number of features (the dimension of the ambient space in which the data is embedded). The package provides tools for modeling the intrinsic geometric structures of data and estimating conditional expectations of response variables over these geometric objects. Inferential analysis is performed using the Morse-Smale regression approach, which decomposes the domain (the geometric object associated with the data) into gradient flow cells. Within each cell, the regression model exhibits an essentially monotonic structure that can be analyzed using classical statistical methods.

Details

Modern datasets, particularly in biological sciences, often contain thousands of features with complex non-linear relationships and multi-way interactions that cannot be captured by examining only pairwise associations. Traditional regression models struggle with this complexity, while machine learning approaches, though powerful for prediction, sacrifice interpretability.

The gflow package addresses this challenge by exploiting a fundamental observation: despite existing in high-dimensional spaces, real-world data typically lives on much lower-dimensional geometric structures. Like a twisted ribbon in 3D space that is fundamentally 2-dimensional, most real-world systems generate highly constrained data that traces out specific geometric shapes within the ambient space. The geometry of these shapes implicitly encodes complex associations between features.

Rather than working in a fixed coordinate system, gflow adopts a coordinate-free framework that models data as a weighted graph (or more generally, a Riemannian simplicial complex) capturing the underlying geometric structure. This transforms the inference problem from $E[Y|X]$ to

$E[Y|G(X)]$, where $G(X)$ represents the geometric object constructed from your data matrix X. This reformulation provides a more flexible representation that adapts to the inherent complexity of the data without imposing predetermined functional forms.

A key strength of gflow is its ability to provide interpretable results from complex data. The gradient-flow decomposition identifies natural regions where relationships between predictors and outcomes are locally simple and monotonic. This allows one to "peer inside" black-box models and understand how predictions change across different parts of the data space. By respecting the intrinsic geometry of the data, gflow bridges the gap between the flexibility of modern machine learning and the interpretability of classical statistical methods.

Key Features

1. Geometric Data Representation

- Pruned intersection k-nearest neighbor (ikNN) graphs that capture data geometry
- Simplicial complex construction for modeling higher-order relationships

2. Robust Signal Recovery and Smoothing

- Kernel graph Laplacian smoothing for noise reduction
- Spectral filtering and diffusion-based approaches

3. Gradient Flow Domain Decomposition

- Automatic identification of critical points (local minima, maxima, and saddles)
- Morse-Smale complex construction for function analysis
- Natural partitioning into regions of monotonic behavior

4. Interpretable Statistical Inference

- Within-region regression and classification
- Bootstrap-Wasserstein testing for non-linear associations
- Feature importance assessment across different data regions
- Visualization tools for understanding model behavior

Main Function Categories

Graph and Simplicial Complex Construction:

- `create.iknn.graphs` - Builds intersection k-nearest neighbor graphs
- `create.single.iknn.graph` - Creates mutual k-nearest neighbor graphs
- `create.cmst.graph` - Constructs a minimal spanning tree completion graph
- `create.nerve.complex` - Creates a nerve complex for k-nearest neighbor covering

Conditional Expectation Estimation Methods:

Model-Averaged Local Regression (1D):

- `amagelo` - Adaptive MAGELO with automatic bandwidth selection, extrema detection, and robust fitting for continuous responses
- `magelo` - Disk-based neighborhoods with grid-centered models for smooth transitions in varying density data

- `mabilo` - Symmetric k-hop neighborhoods with model averaging and Bayesian bootstrap for uncertainty quantification
- `amagelogit` - Grid-based logistic regression with model averaging for binary outcomes
- `maelog` - Data-centered logistic regression with adaptive disk neighborhoods and k-NN fall-back

Graph-Based Local Regression Methods:

- `deg0.lowess.graph.smoothing` - Locally weighted averaging using graph distances with adaptive bandwidth selection
- `graph.spectral.lowess` - Spectral embedding transforms graph distances to Euclidean space for local linear regression
- `spectral.lowess.graph.smoothing` - Includes Cleveland's robustness iterations and multiple kernel options

Diffusion and Kernel Methods:

- `graph.kernel.smoother` - Spatially-aware cross-validation with buffer zones to prevent autocorrelation bias
- `graph.diffusion.smoother` - Iterative diffusion for denoising and interpolating missing values
- `harmonic.smoother` - Solves discrete Laplace equation for smooth interpolation with fixed boundary values
- `graph.spectral.filter` - Frequency-based smoothing using Laplacian eigendecomposition

Path and Geodesic Methods:

- `pgmalo` - Piecewise segmentation with automatic change point detection for data with structural breaks
- `agemalo` - Adaptive geodesic regression using hierarchies of local geodesics through maximal packing vertices
- `adaptive.uggmalo` - Uniform grid approach with path-based local models and cross-validated bandwidth selection

Specialized Methods:

- `magelog`, `mabilog`, `uggmalog` - Log-space variants for positive-valued responses
- `ulogit` - Uniform grid logistic regression for binary outcomes on graphs
- `nerve.cx.spectral.filter` - Spectral filtering over simplicial complexes
- `meanshift.data.smoother` - Data denoising using mean-shift algorithm with adaptive step sizes

Morse-Smale Complex Analysis:

- `create.basin.cx` - Constructs basin complex from gradient flow
- `compute.graph.gradient.flow` - Computes gradient flow trajectories
- `find.critical.points` - Identifies local minima, maxima, and saddles

Statistical Inference:

- `fassoc.test` - Tests for non-linear functional associations
- `compute.bayesian.effects` - Bayesian effect size estimation
- `wasserstein.distance` - Computes optimal transport distances

Author(s)

Pawel Gajer <pgajer@gmail.com>

References

Gajer, P. and Ravel, J. (2025). The Geometry of Machine Learning Models. *arXiv preprint* arXiv:2501.01234. https://arxiv.org/abs/2501.01234

See Also

Useful links:

- https://github.com/pgajer/gflow
- Report bugs at https://github.com/pgajer/gflow/issues

`adaptive.uggmalo` *Adaptive Uniform Grid Graph Model-Averaged Local Linear Regression*

Description

Performs model-averaged local linear regression on graph-structured data using an adaptive uniform grid approach. This method combines graph structure preservation with local linear modeling to capture both global and local patterns in the data.

Usage

```
adaptive.uggmalo(
  adj.list,
  weight.list,
  y,
  min.path.size,
  n.grid.vertices,
  n.bws,
  min.bw.factor = 0.05,
  max.bw.factor = 0.6,
  max.iterations = 20,
  precision = 1e-04,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  tolerance = 1e-06,
  n.bb = 0L,
  cri.probability = 0.95,
  n.perms = 0L,
  blending.coef = 0.1,
  verbose = FALSE
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`y`	Numeric vector of response values at each vertex.
`min.path.size`	Integer. Minimum number of vertices required in valid paths.
`n.grid.vertices`	Integer. Number of vertices in uniform grid representation.
`n.bws`	Integer. Number of candidate bandwidths to evaluate.
`min.bw.factor`	Numeric. Factor multiplied by graph diameter for minimum bandwidth.
`max.bw.factor`	Numeric. Factor multiplied by graph diameter for maximum bandwidth.
`max.iterations`	Integer. Maximum number of iterations for convergence.
`precision`	Numeric. Precision threshold for numerical computations.
`dist.normalization.factor`	Numeric. Factor for normalizing graph distances.
`kernel.type`	Integer. Type of kernel function (1: Gaussian, 2: Triangular).
`tolerance`	Numeric. Convergence tolerance for model fitting.
`n.bb`	Integer. Number of bootstrap iterations (0 for no bootstrap).
`cri.probability`	Numeric. Confidence level for bootstrap intervals (0-1).
`n.perms`	Integer. Number of permutation test iterations (0 for no testing).
`blending.coef`	Numeric. Blending coefficient for model averaging.
`verbose`	Logical. Whether to print progress information.

Details

The adaptive UGGMALO algorithm proceeds through several steps:

1. Creates a uniform grid representation of the input graph
2. Computes optimal bandwidths using cross-validation
3. Fits local linear models along paths through the graph
4. Combines predictions using model averaging

The function supports optional bootstrap confidence intervals and permutation testing for statistical inference.

Value

A list containing:

graph.diameter Numeric. Computed diameter of input graph.

grid.opt.bw Numeric vector. Optimal bandwidth for each grid vertex.

predictions Numeric vector. Model-averaged predictions for original vertices.

grid.predictions Numeric vector. Model-averaged predictions for grid vertices.

bb.predictions Matrix. Bootstrap predictions (if n.bb > 0).

cri.lower Numeric vector. Lower confidence bounds (if n.bb > 0).

cri.upper Numeric vector. Upper confidence bounds (if n.bb > 0).

null.predictions Matrix. Permutation test predictions (if n.perms > 0).

p.values Numeric vector. Vertex-wise p-values (if n.perms > 0).

effect.sizes Numeric vector. Effect sizes (if n.perms > 0).

significant.vertices Logical vector. Significance indicators (if n.perms > 0).

Examples

```
## Not run:
# Create a simple graph with 3 vertices
adj.list <- list(c(2), c(1, 3), c(2))
weight.list <- list(c(1), c(1, 1), c(1))
y <- c(1, 2, 3)

# Run basic analysis
result <- adaptive.uggmalo(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  min.path.size = 2,
  n.grid.vertices = 5,
  n.bws = 10
)

# Run with bootstrap confidence intervals
result.boot <- adaptive.uggmalo(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  min.path.size = 2,
  n.grid.vertices = 5,
  n.bws = 10,
  n.bb = 100
)

## End(Not run)
```

`add.grad.ED.arrows` *Add Gradient Arrows to 3D Plot*

Description

Adds gradient vectors as 3D arrows to an existing plot

Usage

```
add.grad.ED.arrows(ids, S, grad.ED, C = 2.5)
```

Arguments

ids	Vector of row names to display gradients for.
S	Matrix of 3D positions.
grad.ED	Matrix of gradient vectors (same dimensions as S).
C	Scaling constant for gradient visualization.

Details

This function adds arrows showing gradient directions at specified points. The arrows are scaled by factor C for better visualization.

Value

Invisibly returns NULL.

Examples

```
## Not run:
S <- matrix(rnorm(30), ncol = 3)
rownames(S) <- paste0("Point", 1:10)
grad.ED <- matrix(rnorm(30) * 0.1, ncol = 3)
rownames(grad.ED) <- rownames(S)

plot3D.plain(S)
add.grad.ED.arrows(c("Point1", "Point5"), S, grad.ED, C = 2.5)

## End(Not run)
```

`agemalo` *Adaptive Graph Geodesic Model-Averaged Local Linear Regression*

Description

Performs geodesic model-averaged local linear regression on graph-structured data using an adaptive uniform grid approach.

Usage

```
agemalo(
  adj.list,
  weight.list,
  y,
  min.path.size = 6,
  n.packing.vertices = length(y),
  max.packing.iterations = 20,
  packing.precision = 1e-04,
  n.bws = 50,
  log.grid = TRUE,
  min.bw.factor = 0.025,
  max.bw.factor = 0.99,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  model.tolerance = 1e-06,
  model.blending.coef = 0.1,
  n.bb = 0L,
  cri.probability = 0.95,
  n.perms = 0L,
  verbose = FALSE
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`y`	Numeric vector of response values at each vertex.
`min.path.size`	Integer. Minimum number of vertices required in valid paths.
`n.packing.vertices`	Integer. Number of vertices to use in the maximal packing. Defaults to the number of vertices (length of y).
`max.packing.iterations`	Integer. Maximum number of iterations for the packing algorithm.
`packing.precision`	Numeric. Precision threshold for packing convergence.
`n.bws`	Integer. Number of candidate bandwidths to evaluate.
`log.grid`	Logical. Whether to use logarithmic spacing for bandwidth candidates.
`min.bw.factor`	Numeric. Factor multiplied by graph diameter for minimum bandwidth.
`max.bw.factor`	Numeric. Factor multiplied by graph diameter for maximum bandwidth.
`dist.normalization.factor`	Numeric. Factor for normalizing graph distances.
`kernel.type`	Integer. Type of kernel function (1-7, with 7 as default).
`model.tolerance`	Numeric. Convergence tolerance for model fitting.
`model.blending.coef`	Numeric. Blending coefficient for model averaging.
`n.bb`	Integer. Number of bootstrap iterations (0 for no bootstrap).
`cri.probability`	Numeric. Confidence level for bootstrap intervals (0-1).
`n.perms`	Integer. Number of permutation test iterations (0 for no testing).
`verbose`	Logical. Whether to print progress information.

Details

The AGEMALO algorithm proceeds through several steps:

1. Creates a maximal packing of the input graph
2. For each packing vertex it

- creates a hierarchy of local geodesics passing through that vertex
- computes minimal and maximal bandwidths and bandwidth candidates
- fits local weighted linear models along geodesic paths

1. Combines predictions using model averaging

The function supports optional bootstrap confidence intervals and permutation testing for statistical inference.

Value

A list containing:

`graph.diameter`	Numeric. Computed diameter of input graph.
`grid.opt.bw`	Numeric vector. Optimal bandwidth for each grid vertex.
`predictions`	Numeric vector. Model-averaged predictions for original vertices.
`grid.predictions`	Numeric vector. Model-averaged predictions for grid vertices.
`bb.predictions`	Matrix. Bootstrap predictions (if n.bb > 0).
`cri.lower`	Numeric vector. Lower confidence bounds (if n.bb > 0).
`cri.upper`	Numeric vector. Upper confidence bounds (if n.bb > 0).
`null.predictions`	Matrix. Permutation test predictions (if n.perms > 0).
`p.values`	Numeric vector. Vertex-wise p-values (if n.perms > 0).
`effect.sizes`	Numeric vector. Effect sizes (if n.perms > 0).
`significant.vertices`	Logical vector. Significance indicators (if n.perms > 0).

Examples

```
## Not run:
# Create a simple graph with 3 vertices
adj.list <- list(c(2), c(1, 3), c(2))
weight.list <- list(c(1), c(1, 1), c(1))
y <- c(1, 2, 3)

# Run basic analysis
result <- agemalo(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  min.path.size = 2,
  n.packing.vertices = 5,
  n.bws = 10
)

# Run with bootstrap confidence intervals
result.boot <- agemalo(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  min.path.size = 2,
  n.packing.vertices = 5,
  n.bws = 10,
  n.bb = 100
)

## End(Not run)
```

`AIC.graph.spectral.lowess` *AIC for Graph Spectral LOWESS*

Description

Computes the Akaike Information Criterion

Usage

```
## S3 method for class 'graph.spectral.lowess'
AIC(object, ..., k = 2)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`...`	Additional arguments (currently unused)
`k`	Penalty parameter (default: 2)

Value

AIC value

`AIC.graph.spectral.ma.lowess` *AIC for Graph Spectral MA LOWESS*

Description

Computes the Akaike Information Criterion accounting for model averaging

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
AIC(object, ..., k = 2)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)
`k`	Penalty parameter (default: 2)

Value

AIC value

`AIC.mabilog` *Compute AIC for Mabilog Model*

Description

Computes Akaike Information Criterion for model selection

Usage

```
## S3 method for class 'mabilog'
AIC(object, ..., k = 2)
```

Arguments

`object`	A ’mabilog’ object
`...`	Additional arguments passed to logLik
`k`	Penalty parameter (default: 2)

Value

AIC value

`AIC.mabilo_plus` *Compute AIC for Mabilo Plus Model*

Description

Computes Akaike Information Criterion for model selection

Usage

```
## S3 method for class 'mabilo_plus'
AIC(object, type = c("ma", "sm"), k = 2, ...)
```

Arguments

`object`	A ’mabilo_plus’ object
`type`	Character string specifying which predictions to use: "ma" (default) or "sm"
`k`	Penalty parameter (default: 2)
`...`	Additional arguments passed to logLik

Value

AIC value

`amagelo` *Adaptive Model Averaged GEodesic LOcal linear smoothing*

Description

Performs nonparametric smoothing of 1D data using adaptive model averaged local linear regression. AMAGELO implements a grid-based approach with model averaging and automatic bandwidth selection to produce smooth predictions while adapting to local data characteristics. The method also identifies local extrema and provides measures of their significance.

Usage

```
amagelo(
  x,
  y,
  grid.size,
  min.bw.factor,
  max.bw.factor,
  n.bws,
  use.global.bw.grid = TRUE,
  with.bw.predictions = FALSE,
  log.grid = FALSE,
  domain.min.size = 4,
  kernel.type = 7L,
  dist.normalization.factor = 1.1,
  n.cleveland.iterations = 1,
  blending.coef = 0,
  use.linear.blending = TRUE,
  precision = 1e-06,
  small.depth.threshold = 0.05,
  depth.similarity.tol = 1e-04,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of predictor values
`y`	Numeric vector of response values
`grid.size`	Integer specifying the number of grid points (default: 100)
`min.bw.factor`	Numeric minimum bandwidth factor (default: 0.01)
`max.bw.factor`	Numeric maximum bandwidth factor (default: 0.5)
`n.bws`	Integer number of bandwidths to evaluate (default: 30)
`use.global.bw.grid`	Logical: use same bandwidth grid for all points? (default: TRUE)
`with.bw.predictions`	Logical: return predictions for all bandwidths? (default: FALSE)
`log.grid`	Logical: use logarithmic bandwidth spacing? (default: TRUE)
`domain.min.size`	Integer minimum data points per local model (default: 10)

`kernel.type`	Integer code for kernel function (default: 1 = Gaussian)
`dist.normalization.factor`	Numeric scale factor for distances (default: 1)
`n.cleveland.iterations`	Integer robustness iterations (default: 3)
`blending.coef`	Numeric model blending coefficient (default: 0.5)
`use.linear.blending`	Logical: use linear blending? (default: FALSE)
`precision`	Numeric precision threshold for optimization (default: 1e-6)
`small.depth.threshold`	Numeric threshold for wiggle detection (default: 0.05)
`depth.similarity.tol`	Numeric tolerance for depth similarity (default: 0.001)
`verbose`	Logical: print progress information? (default: FALSE)

Details

AMAGELO constructs a uniform grid over the data domain and fits local linear models at each grid point using a range of bandwidths. Models are then averaged with weights that depend on both spatial proximity and model quality. The optimal bandwidth is selected by cross-validation.

The method identifies local extrema (maxima and minima) and calculates their prominence using both absolute and relative depth measures. It also provides measures of overall monotonicity through the TVMI and Simpson index.

Triplet harmonic smoothing is applied to remove small wiggles while preserving significant features of the data.

Value

A list containing:

`x_sorted`	Sorted predictor values
`y_sorted`	Response values ordered by sorted x
`order`	Original indices of sorted data (1-based)
`grid_coords`	x-coordinates of grid points
`predictions`	Fitted values at optimal bandwidth
`bw_predictions`	Matrix of predictions by bandwidth (if requested)
`grid_predictions`	Predictions at grid points
`harmonic_predictions`	Predictions after triplet harmonic smoothing
`local_extrema`	Matrix of detected extrema with columns: idx (Index in sorted data, 1-based), x (x-coordinate of extremum), y (y-value at extremum), is_max (1 if maximum, 0 if minimum), depth (Vertical prominence of extremum), depth_idx (Index where min/max descent terminates, 1-based), rel_depth (Depth relative to total depth of all extrema), range_rel_depth (Depth relative to range of predictions)
`monotonic_interval_proportions`	Relative lengths of monotonic intervals

`change_scaled_monotonicity_index`	Weighted signed average of directional changes, quantifying monotonicity strength and directionality; values close to $+1$ or -1 indicate strong global monotonic trends.
`bw_errors`	Cross-validation errors for each bandwidth
`opt_bw_idx`	Index of optimal bandwidth (1-based)
`min_bw`	Minimum bandwidth value
`max_bw`	Maximum bandwidth value
`bws`	Vector of evaluated bandwidths

Examples

```
## Not run:
# Simulate data with smooth trend and noise
x <- seq(0, 10, length.out = 200)
y <- sin(x) + 0.2 * rnorm(length(x))

# Apply AMAGELO smoothing
result <- amagelo(x, y, grid_size = 100)

# Plot results
plot(x, y, pch = 16, col = "gray")
lines(x[result$order], result$predictions, col = "red", lwd = 2)

# Examine local extrema
extrema <- result$local_extrema
points(extrema[,"x"], extrema[,"y"],
       pch = ifelse(extrema[,"is_max"] == 1, 24, 25),
       bg = ifelse(extrema[,"is_max"] == 1, "red", "blue"),
       cex = 2 * extrema[,"range_rel_depth"])

## End(Not run)
```

`amagelogit` *Grid-based Model Averaged Bandwidth Logistic Regression*

Description

Performs model-averaged bandwidth logistic regression using local polynomial fitting. The function implements a flexible approach to binary regression by fitting local models at points of a uniform grid spanning the range of predictor values. Multiple local models are combined to produce robust predictions. It supports both linear and quadratic local models, automatic bandwidth selection, and various kernel types for weight calculation.

Usage

```
amagelogit(
  x,
  y,
  grid.size = 200,
  fit.quadratic = FALSE,
  pilot.bandwidth = -1,
  kernel = 7L,
  min.points = NULL,
  cv.folds = 5L,
  n.bws = 50L,
  min.bw.factor = 0.05,
  max.bw.factor = 0.9,
  max.iterations = 100L,
  ridge.lambda = 1e-06,
  tolerance = 1e-08,
  with.bw.predictions = TRUE
)
```

Arguments

`x`	Numeric vector of predictor variables
`y`	Binary vector (0 or 1) of response variables
`grid.size`	Integer; number of points in the uniform grid where local models are centered. Default is 200
`fit.quadratic`	Logical; whether to include quadratic terms in the local models. Default is FALSE
`pilot.bandwidth`	Numeric; bandwidth for local fitting. If <= 0, bandwidth is automatically selected. Default is -1
`kernel`	Integer; kernel type for weight calculation: • 1: Epanechnikov • 2: Triangular • 4: Laplace • 5: Normal • 6: Biweight • 7: Tricube (default)
`min.points`	Integer; minimum number of points required for local fitting. Default is automatically set based on fit.quadratic (4 for quadratic, 3 for linear)
`cv.folds`	Integer; number of cross-validation folds for bandwidth selection. If 0, LOOCV approximation is used. Default is 0
`n.bws`	Integer; number of bandwidths to try in automatic selection. Default is 50
`min.bw.factor`	Numeric; minimum bandwidth factor relative to data range. Default is 0.05
`max.bw.factor`	Numeric; maximum bandwidth factor relative to data range. Default is 0.9
`max.iterations`	Integer; maximum number of iterations for local fitting. Default is 100
`ridge.lambda`	Numeric; ridge parameter for local fitting. Default is 1e-6
`tolerance`	Numeric; convergence tolerance for local fitting. Default is 1e-8
`with.bw.predictions`	Logical; whether to return predictions for all bandwidths. Default is TRUE

Details

The function fits local logistic regression models centered at points of a uniform grid spanning the range of x values. Local models are fit using kernel-weighted maximum likelihood. The bandwidth determines the size of the local neighborhood. When pilot.bandwidth <= 0, the function automatically selects a bandwidth using cross-validation.

The local models can be either linear or quadratic (controlled by fit.quadratic). For numerical stability, the minimum number of points in each local fit is automatically set to 3 for linear and 4 for quadratic models, but can be overridden with min.points.

Predictions at the original x points are obtained by linear interpolation from the grid-based predictions. The grid approach provides computational efficiency and smooth prediction curves.

Value

A list containing:

- x.grid: Uniform grid points where local models are centered
- predictions: Predicted probabilities at original x points
- bw.grid.predictions: Matrix of predictions at grid points for each bandwidth
- mean.brier.errors: Cross-validation errors for each bandwidth
- opt.brier.bw.idx: Index of optimal bandwidth
- bws: Vector of tried bandwidths
- fit.info: List of fitting parameters used
- x: Original predictor values
- y: Original response values

Examples

```
## Not run:
x <- seq(0, 1, length.out = 100)
p <- 1/(1 + exp(-(x - 0.5)*10))
y <- rbinom(100, 1, p)
fit <- amagelogit(x, y, grid.size = 200, fit.quadratic = TRUE, cv.folds = 5)
plot(x, y)
lines(fit$x.grid, fit$bw.grid.predictions[,fit$opt.brier.bw.idx], col = "red")

## End(Not run)
```

---

analyze.categorical.proportions

Analyze Categorical Proportions with Mixed Effects Support

Description

Performs comprehensive analysis of proportions across categorical groups, with optional support for repeated measures using mixed effects models. Calculates confidence intervals, relative proportions, and performs appropriate statistical tests.

Usage

```
analyze.categorical.proportions(
  x,
  y,
  subj.ids = NULL,
  pos.label = "sPTB",
  neg.label = "TB",
  digits = 2
)
```

Arguments

`x`	factor or coercible to factor, the categorical grouping variable (e.g., community state types)
`y`	binary variable (factor, logical, or numeric 0/1) representing the outcome of interest
`subj.ids`	optional factor or coercible to factor, subject identifiers for repeated measures analysis. If provided, mixed effects models will be fitted
`pos.label`	character, label for the positive/case level of the binary outcome (default: "sPTB")
`neg.label`	character, label for the negative/control level of the binary outcome (default: "TB")
`digits`	integer, number of decimal places for numerical outputs (default: 2)

Details

The function performs different analyses based on whether subject IDs are provided:

Without subject IDs:

- Calculates proportions for each group
- Performs one-sample proportion tests against the overall proportion
- Computes 95\
- Calculates relative proportions compared to overall rate

With subject IDs (mixed effects):

- Fits a generalized linear mixed model with random intercepts for subjects
- Handles convergence issues by trying multiple optimizers
- Computes model-based confidence intervals and p-values
- Falls back to simple proportions if model fitting fails

Value

A list containing:

prop.test.mat matrix with columns for counts, proportions, relative proportions, confidence intervals, and p-values. If `subj.ids` is provided, includes mixed effects estimates

contingency contingency table of x vs y

overall.prop overall proportion of positive outcomes

latex.caption formatted LaTeX caption describing the analysis

model if mixed effects analysis was performed, the fitted glmer model object; NULL otherwise

Note

For mixed effects models, the function requires the **lme4** package. If convergence issues occur, the function will attempt multiple optimizers before falling back to simple proportion calculations.

See Also

`glmer` for mixed effects model details

Examples

```
## Not run:
# Simple analysis without repeated measures
set.seed(123)
cst <- factor(sample(c("I", "II", "III", "IV"), 200, replace = TRUE))
outcome <- rbinom(200, 1, c(0.1, 0.3, 0.2, 0.4)[as.numeric(cst)])

result <- analyze.categorical.proportions(
  x = cst,
  y = outcome,
  pos.label = "Case",
  neg.label = "Control"
)

# Analysis with repeated measures
subjects <- factor(rep(1:50, each = 4))
cst_repeated <- factor(sample(c("I", "II", "III", "IV"), 200, replace = TRUE))
outcome_repeated <- rbinom(200, 1, 0.3)

result_mixed <- analyze.categorical.proportions(
  x = cst_repeated,
  y = outcome_repeated,
  subj.ids = subjects
)

## End(Not run)
```

`analyze.function.aware.weights`
Analyze Function-Aware Weight Modifications

Description

Analyzes how different weighting schemes would modify edge weights based on function values.

Usage

```
analyze.function.aware.weights(
  adj.list,
  weight.list,
  function.values,
  weight.types = 0:5,
  epsilon = 1e-06,
  lambda = 1,
  alpha = 1,
  beta = 5,
  tau = 0.1,
  p = 2,
  q = 2,
  r = 2
)
```

Arguments

`adj.list`	A list where each element contains the indices of vertices adjacent to vertex i. The list should be 1-indexed (as is standard in R).
`weight.list`	A list of the same structure as adj.list, where each element contains the weights of edges connecting vertex i to its adjacent vertices.
`function.values`	A numeric vector containing function values at each vertex of the graph.
`weight.types`	A vector of weight.type values to analyze (see construct.function.aware.graph for details on weight types)
`epsilon`	Small constant to avoid division by zero (for weight.type 0)
`lambda`	Decay rate parameter (for weight.type 2)
`alpha`	Power law exponent (for weight.type 3) or scaling factor (for weight.type 5)
`beta`	Sigmoid steepness parameter (for weight.type 4)
`tau`	Sigmoid threshold parameter (for weight.type 4)
`p`	Power for feature distance term (for weight.type 5)
`q`	Power for function difference term (for weight.type 5)
`r`	Power for the overall normalization (for weight.type 5)

Value

A list where each element corresponds to a weight.type and contains a vector of modified weights for all edges in the graph

---

`analyze.harmonic.extensions`
Analyze Harmonic Extensions for Local Extrema

Description

Complete workflow for analyzing how different harmonic extension methods perform on local extrema in a graph function. This function extracts subgraphs around each extremum and applies various harmonic extension methods for comparison.

Usage

```
analyze.harmonic.extensions(
  adj_list,
  weight_list,
  y,
  predictions,
  extrema_df,
  hop_offset = 2,
  min_hop_idx = 0,
  max_hop_idx = Inf,
  methods = c("weighted_mean", "harmonic_iterative", "harmonic_eigen",
    "biharmonic_harmonic", "boundary_smoothed"),
  max_iterations = 100,
  tolerance = 1e-06,
  sigma = 1,
  visualize = TRUE,
  save_results = FALSE,
  output_dir = "harmonic_extension_analysis",
  verbose = TRUE
)
```

Arguments

`adj_list`	Original graph adjacency list
`weight_list`	Original graph weight list
`y`	Original function values on vertices
`predictions`	Smoothed predictions (e.g., from spectral filtering)
`extrema_df`	Data frame containing extrema information
`hop_offset`	Additional hop distance beyond extremum's hop radius (default: 2)
`min_hop_idx`	Minimum hop index for extrema to process (default: 0)
`max_hop_idx`	Maximum hop index for extrema to process (default: Inf)
`methods`	Vector of harmonic extension methods to test
`max_iterations`	Maximum iterations for iterative methods (default: 100)
`tolerance`	Convergence tolerance (default: 1e-6)
`sigma`	Parameter for weighted mean method (default: 1.0)
`visualize`	Whether to visualize results (default: TRUE)
`save_results`	Whether to save detailed results (default: FALSE)
`output_dir`	Directory for saving results (default: "harmonic_extension_analysis")
`verbose`	Whether to print progress messages (default: TRUE)

Value

A list containing:

subgraphs List of extracted subgraphs for each extremum

results List of results from applying harmonic extension methods

metrics Data frame with detailed metrics for all extrema and methods

summary Data frame with summary statistics across methods

extrema_info Data frame with information about processed extrema

Examples

```
## Not run:
result <- analyze.harmonic.extensions(
  graph$adj_list,
  graph$weight_list,
  y,
  gsf_res$predictions,
  gsf_b_cx$extrema_df,
  hop_offset = 2,
  min_hop_idx = 1,
  max_hop_idx = 5
)

## End(Not run)
```

`analyze.hierarchical.differences`
Perform Hierarchical Bayesian Analysis of Method Differences

Description

Fits a hierarchical Bayesian model to compare multiple methods, estimating method-specific means and variances while accounting for between-method variability. The model provides posterior distributions for all pairwise method differences and probabilities of superiority.

Usage

```
analyze.hierarchical.differences(bb.integrals)
```

Arguments

`bb.integrals`	A named list of Bayesian bootstrap integral values, where each element contains numeric samples representing the performance metric for a method. All elements must have the same length.

Details

The hierarchical model structure:

- Method means: `mu[k]` ~ `Normal(global_mu, tau)`
- Observations: `y[n,k]` ~ `Normal(mu[k], sigma[k])`
- Priors: `global_mu` ~ `Normal(0, 10)`, `tau` ~ `Cauchy(0, 2.5)`, `sigma` ~ `Cauchy(0, 2.5)`
- The model pools information across methods through the hierarchical structure

Value

A stanfit object containing:

- `mu`: Posterior samples of method-specific means
- `sigma`: Posterior samples of method-specific standard deviations

- `tau`: Posterior samples of between-method variability
- `global_mu`: Posterior samples of the overall mean across methods
- `diff`: Matrix of pairwise differences (mu[i] - mu[j])
- `prob_diff`: Matrix of probabilities that method i is worse than method j

Note

- Requires the 'rstan' package to be installed
- Uses 4 chains with 2000 iterations (1000 warmup, 1000 sampling)
- `prob_diff\[i,j\]` represents P(mu[i] < mu[j]), so values < 0.5 indicate method i is better
- Check convergence using `rstan::check_hmc_diagnostics()` and `summary()`

References

Gelman, A., & Hill, J. (2007). Data analysis using regression and multilevel/hierarchical models. Cambridge University Press.

See Also

`stan` for model fitting, `extract` for extracting posterior samples, `check_hmc_diagnostics` for convergence diagnostics

Examples

```
## Not run:
# Example: Compare three methods with hierarchical model
set.seed(123)
bb.integrals <- list(
  method_A = rnorm(100, mean = 0.5, sd = 0.1),
  method_B = rnorm(100, mean = 0.52, sd = 0.12),
  method_C = rnorm(100, mean = 0.48, sd = 0.08)
)

# Fit hierarchical model
fit <- analyze.hierarchical.differences(bb.integrals)

# Check convergence
rstan::check_hmc_diagnostics(fit)

# Extract posterior summaries
print(fit, pars = c("mu", "sigma", "tau"))

# Get pairwise probabilities
prob_matrix <- rstan::extract(fit, "prob_diff")$prob_diff
apply(prob_matrix, c(2,3), mean)  # Mean probabilities

## End(Not run)
```

`analyze.weighted.cst` *Time-Weighted CST Analysis for Binary Outcomes*

Description

Performs time-weighted analysis of Community State Types (CSTs) in longitudinal studies where sampling intervals may be irregular. Weights each CST observation by its duration to accurately represent the time spent in different microbial states.

Usage

```
analyze.weighted.cst(
  x,
  y,
  subj.ids,
  time.points = NULL,
  pos.label = "sPTB",
  neg.label = "TB"
)
```

Arguments

`x`	vector of CST measurements (factor or character), must be ordered chronologically within each subject
`y`	binary outcome vector, must be constant within each subject
`subj.ids`	vector of subject identifiers, same length as `x`
`time.points`	optional numeric vector of time points for each measurement. If NULL, assumes equal intervals. Must be chronologically ordered within each subject
`pos.label`	character, label for positive outcome (default: "sPTB")
`neg.label`	character, label for negative outcome (default: "TB")

Details

The time-weighting algorithm:

1. For each subject, calculates the duration of each CST period
2. When `time.points` is provided:
   - First observation: duration = `(time[2] - time[1]) / 2`
   - Middle observations: duration = `(time[i+1] - time[i-1]) / 2`
   - Last observation: duration = `(time[n] - time[n-1]) / 2`
3. When `time.points` is NULL, each observation gets equal weight
4. Sums time by CST and divides by total time to get proportions
5. Compares proportions between outcome groups using Wilcoxon tests

This approach is particularly useful when:

- Sampling is irregular (e.g., clinical visits at varying intervals)
- Some CST states are transient and might be missed with regular sampling
- The duration of CST states is biologically meaningful

Value

A list containing:

weighted_proportions matrix where rows are subjects and columns are CSTs. Values represent the proportion of time spent in each CST

summary data frame with columns:

- CST: Community State Type identifier
- median_case: Median time proportion in CST for positive outcome group
- median_control: Median time proportion in CST for negative outcome group
- p_value: P-value from Wilcoxon rank-sum test

subject_data data frame containing subject IDs and outcomes

Note

- Time points must be in the same units throughout the dataset
- The function assumes measurements are ordered chronologically within subjects
- For regular sampling, the weighted and unweighted approaches will give similar results

References

Gajer P, et al. (2012). Temporal dynamics of the human vaginal microbiota. Science Translational Medicine, 4(132), 132ra52.

See Also

`summarize.and.test.cst` for unweighted analysis approaches

Examples

```
## Not run:
# Example with gestational age time points
set.seed(123)

# Generate data for 20 subjects
subjects <- rep(1:20, each = 5)
weeks <- rep(c(12, 16, 20, 28, 32), 20)  # Gestational weeks

# Simulate CSTs with some subjects more stable than others
csts <- character(100)
for (i in 1:20) {
  if (i <= 10) {  # Stable subjects
    csts[(i-1)*5 + 1:5] <- sample(c("I", "III"), 5, replace = TRUE, prob = c(0.8, 0.2))
  } else {  # Variable subjects
    csts[(i-1)*5 + 1:5] <- sample(c("I", "III", "IV"), 5, replace = TRUE)
  }
}

# Binary outcome
outcomes <- rep(c(rep(0, 10), rep(1, 10)), each = 5)

# Analyze with time weighting
result <- analyze.weighted.cst(
  x = csts,
  y = outcomes,
  subj.ids = subjects,
  time.points = weeks
)

# View results
print(result$summary)

# Visualize distributions
library(ggplot2)
prop_df <- as.data.frame(result$weighted_proportions)
prop_df$outcome <- result$subject_data$outcome
prop_df$subject <- rownames(prop_df)

# Plot CST I proportions by outcome
ggplot(prop_df, aes(x = outcome, y = I)) +
  geom_boxplot() +
  geom_point(position = position_jitter(width = 0.1)) +
  labs(y = "Proportion of time in CST I",
       title = "Time-weighted CST proportions by outcome")

## End(Not run)
```

angle.3D *Computes the angle between two 3D vectors in radians.*

Description

Computes the angle between two 3D vectors in radians.

Usage

```
angle.3D(v, w, method = "arcsine")
```

Arguments

v	A 3D vector.
w	A 3D vector.
method	The method to be used for the angle estimate. Possible choices are: "arcsine" and "arctan".

Value

The angle between two 3D vectors in radians.

Examples

```
## Not run:
v <- c(0,1,3)
w <- c(2,3,4)
angle <- angle.3D(v, w)

## End(Not run)
```

`angle.alignment` *Align Estimated Angles with True Angles*

Description

Aligns a set of estimated angles with reference angles by finding the optimal rotation and handling circular boundaries.

Usage

```
angle.alignment(true.angles, est.angles)
```

Arguments

`true.angles`	A numeric vector of reference angles (in radians)
`est.angles`	A numeric vector of estimated angles (in radians) to be aligned

Details

This function seeks to align angles that represent positions on a circle by:

1. Finding the optimal shift that maximizes correlation with true angles
2. Handling boundary wrapping issues (when angles cross the $0/2pi$ boundary)
3. Optionally scaling the angles to match the range of true angles

The function returns either a simple shifted version or a scaled version, depending on which achieves higher correlation with the true angles.

Value

A list containing:

`angles`	The aligned angles after shifting and potential scaling
`correlation`	Pearson correlation between true angles and aligned angles
`type`	Character string indicating if the result is "shifted" or "scaled"

Examples

```
# Generate some true angles around a circle
true.angles <- seq(0, 2*pi - 0.1, length.out = 20)

# Create estimated angles with an arbitrary shift and some noise
est.angles <- (true.angles + 1.5) %% (2*pi) + rnorm(20, 0, 0.1)

# Align the estimated angles
aligned <- angle.alignment(true.angles, est.angles)

# Plot results
plot(true.angles, ylim=c(-0.5, 2*pi+0.5), pch=19, col="blue",
     main="Angle Alignment")
points(est.angles, pch=19, col="red")
points(aligned$angles, pch=19, col="green")
```

```
legend("topright", legend=c("True", "Estimated", "Aligned"),
       col=c("blue", "red", "green"), pch=19)
```

---

angular.wasserstein.index *Compute Angular Wasserstein Index between two point sets*

Description

This function calculates the Angular Wasserstein Index I_W(X, Y) between two point sets X and Y. It computes the angular distribution of k-nearest neighbors for each point and then calculates the Wasserstein distance between these angular distributions.

Usage

```
angular.wasserstein.index(X, Y, k)
```

Arguments

X	A numeric matrix where each row represents a point in n-dimensional space.
Y	A numeric matrix with the same dimensions as X, representing the second point set.
k	An integer specifying the number of nearest neighbors to consider.

Value

A numeric value representing the Angular Wasserstein Index.

---

apply.harmonic.extension *Apply Harmonic Extension to Local Subgraph*

Description

Applies a specified harmonic extension method to a local subgraph around an extremum. This function interfaces with C++ implementations of various harmonic extension algorithms.

Usage

```
apply.harmonic.extension(
  subgraph,
  method = c("weighted_mean", "harmonic_iterative", "harmonic_eigen",
    "biharmonic_harmonic", "boundary_smoothed"),
  max_iterations = 100,
  tolerance = 1e-06,
  sigma = 1,
  record_iterations = TRUE,
  verbose = FALSE
)
```

Arguments

`subgraph`	A subgraph object created by `extract.extrema.subgraphs`
`method`	Character string specifying the smoothing method: **"weighted_mean"** Weighted mean hop disk extension **"harmonic_iterative"** Iterative harmonic extension **"harmonic_eigen"** Eigen-based harmonic extension **"biharmonic_harmonic"** Hybrid biharmonic-harmonic extension **"boundary_smoothed"** Boundary smoothed harmonic extension
`max_iterations`	Integer maximum number of iterations for iterative methods (default: 100)
`tolerance`	Numeric convergence tolerance (default: 1e-6)
`sigma`	Numeric parameter for weighted mean method (default: 1.0)
`record_iterations`	Logical whether to record intermediate states (default: TRUE)
`verbose`	Logical whether to print progress information (default: FALSE)

Value

A list containing:

original Original function values

smoothed Smoothed function values after harmonic extension

iterations List of function values at each iteration (if `record_iterations = TRUE`)

method Method used for smoothing

extremum_info Information about the extremum

convergence_info List with convergence information:

- `iterations_performed`: Number of iterations
- `final_change`: Maximum change in final iteration
- `converged`: Whether convergence was achieved

Examples

```
## Not run:
# Assumes subgraph was created by extract.extrema.subgraphs
result <- apply.harmonic.extension(subgraph, method = "harmonic_eigen")

## End(Not run)
```

basin.cx.merge *Merge Two Basins in a Basin Complex*

Description

Merges one basin into another within a `basin_cx` object by updating vertices, labels, and the corresponding cell complex.

Usage

```
basin.cx.merge(basin_cx, absorbing.label, absorbed.label)
```

Arguments

`basin_cx`	An object of class `basin_cx`, as returned by `create.basin.cx()`.
`absorbing.label`	Label of the basin that will absorb another (e.g., "m1", "M2").
`absorbed.label`	Label of the basin to be absorbed (must be same type as `absorbing.label`).

Value

A new `basin_cx` object with updated basins and cell complex.

basins.merge *Merge Two Basins in a Gradient Flow Complex*

Description

Merges two basins in a gradient flow complex by having one basin absorb another. The function updates the basin vertices, recomputes the cells based on the new basin configuration, and updates the local extrema table.

Usage

```
basins.merge(flow, absorbing.label, absorbed.label)
```

Arguments

`flow`	An object of class `"ggflow"` returned by `construct.graph.gradient.flow`.
`absorbing.label`	Character string specifying the label of the basin that will absorb the other basin.
`absorbed.label`	Character string specifying the label of the basin that will be absorbed.

Details

The function identifies the type of basins (ascending or descending) from the label prefixes: "m" for minima (ascending basins) and "M" for maxima (descending basins). The absorbing basin incorporates all vertices from the absorbed basin, and the gradient flow structure is updated accordingly.

Value

A modified gradient flow object with merged basins and updated cells.

See Also

construct.graph.gradient.flow, summary.ggflow

Examples

```
## Not run:
# Merge descending basins M2 and M3, with M2 absorbing M3
merged_flow <- basins.merge(flow, "M2", "M3")

# Check the updated structure
summary(merged_flow)

## End(Not run)
```

---

basins.union *Union of Gradient Flow Basins*

Description

Get the union of vertices from specified basins in a basin complex.

Usage

```
basins.union(x, ids)
```

Arguments

x	An object of class "basin_cx" returned by create.basin.cx().
ids	Character vector of basin labels (e.g., c("M1", "m2", "M3")).

Value

An integer vector of vertex indices contained in the union of the specified basins.

`bayes.factors.symbols` *Format Matrix of Bayes Factors into Simplified Notation*

Description

Converts a matrix of Bayes factors into a more readable format using simplified notation for extreme values and comparison symbols. This is particularly useful for creating publication-ready tables with LaTeX math symbols.

Usage

```
bayes.factors.symbols(
  x,
  thresholds = c(strong = 1000, moderate = 10, weak = 3),
  digits = 2,
  ...
)
```

Arguments

`x`	A numeric matrix or data frame of Bayes factors, typically from pairwise method comparisons where `x[i,j]` represents evidence for method i vs method j
`thresholds`	Named numeric vector of thresholds for different symbols (default: `c(strong = 1000, moderate = 10, weak = 3)`). Values are used for both directions (e.g., BF > 10 and BF < 1/10)
`digits`	Number of decimal places for non-extreme values (default: 2)
`...`	Additional arguments (currently unused)

Details

The function maps Bayes factors to LaTeX symbols based on evidence strength:

- Strong evidence (BF $\geq$ 1000 or BF $\leq$ 0.001) uses double symbols
- Moderate evidence (BF $\geq$ 10 or BF $\leq$ 0.1) also uses double symbols
- Weak evidence (BF $\geq$ 3 or BF $\leq$ 0.333) uses single symbols
- Values between thresholds are displayed numerically

Value

A character matrix with the same dimensions and names as the input, where numeric values are replaced with symbolic notation:

- `"$\gg$"`: BF $\geq$ strong threshold (very strong evidence)
- `"$>$"`: weak $\leq$ BF < moderate (weak to moderate evidence)
- `"$<$"`: 1/moderate < BF $\leq$ 1/weak (weak to moderate evidence against)
- `"$\ll$"`: BF $\leq$ 1/strong (very strong evidence against)
- Numeric value: 1/weak < BF < weak (inconclusive)
- `"1"`: Diagonal elements (method compared to itself)

Note

- The output contains LaTeX math symbols suitable for knitr/RMarkdown documents
- The current implementation uses the same symbol ($\gg$) for both strong and moderate evidence, which may be confusing
- For interpretation: symbols point toward the better method

See Also

`compute.pairwise.bayes.factors` for generating Bayes factor matrices

Examples

```
# Create example Bayes factor matrix
bf.matrix <- matrix(c(1, 15.2, 0.05, 1200,
                      0.066, 1, 0.002, 8.5,
                      20.1, 500, 1, 2.1,
                      0.0008, 0.118, 0.476, 1),
                    nrow = 4, byrow = TRUE)
rownames(bf.matrix) <- colnames(bf.matrix) <- c("A", "B", "C", "D")

# Format with default thresholds
bayes.factors.symbols(bf.matrix)

# Format with custom thresholds
bayes.factors.symbols(bf.matrix,
                     thresholds = c(strong = 100, moderate = 10, weak = 3),
                     digits = 1)

# Use in knitr/RMarkdown for LaTeX output
# knitr::kable(bayes.factors.symbols(bf.matrix), escape = FALSE)
```

---

bbmwd.over.hHN.graphs *Calculate Bayesian Bootstrap Mean Wasserstein Distance over k-Hop Neighbor Graphs*

Description

Computes the Bayesian Bootstrap Mean Wasserstein Distance (BBMWD) over k-Hop Neighbor (hHN) graphs for various combinations of nearest neighbors (k) and hop values (h).

Usage

```
bbmwd.over.hHN.graphs(
  y,
  IkNN.graphs,
  k.values = 10:30,
  hop.values = 2:15,
  n.BB = 100,
  verbose = TRUE
)
```

Arguments

`y`	A numeric vector of binary outcomes (0 or 1) associated with each data point. Length must match the number of vertices in the graphs.
`IkNN.graphs`	A list of pre-computed intersection k-Nearest Neighbor graphs. Each element must be a list containing: **pruned_adj_list** Adjacency list of the pruned graph; **pruned_dist_list** Distance list of the pruned graph
`k.values`	An integer vector specifying the k values to consider. Default is `10:30`.
`hop.values`	An integer vector specifying the hop values to consider. Default is `2:15`.
`n.BB`	Integer; the number of bootstrap iterations for Bayesian Bootstrap. Default is 100.
`verbose`	Logical; if `TRUE`, progress messages are printed. Default is `TRUE`.

Details

This function iterates over all combinations of k and hop values, creating hHN graphs and computing BBMWD for each. Progress is displayed if `verbose = TRUE`.

Value

A nested list structure where:

Outer list Indexed by k values from `k.values`

Inner list Contains results for each hop value, with elements:

- **h** The hop value
- **khn.graph** The k-Hop Neighbor graph (list with adj_list and dist_list)
- **bbmwd** The calculated Bayesian Bootstrap Mean Wasserstein Distance

See Also

`create.hHN.graph`,

Examples

```
## Not run:
# Assuming IkNN.graphs is pre-computed
set.seed(123)
n <- 100
y <- sample(0:1, n, replace = TRUE)

# Run BBMWD analysis over a subset of k and hop values
results <- bbmwd.over.hHN.graphs(
  y = y,
  IkNN.graphs = IkNN.graphs,
  k.values = 10:15,
  hop.values = 2:5,
  n.BB = 50
)

# Access specific result
bbmwd_k10_h3 <- results[[10]][[3]]$bbmwd
```

```
## End(Not run)
```

bbox.hcube.tiling *Creates a hypercube tiling of a bounding box*

Description

This function constructs a hypercube tiling of a bounding box using hypercubes of width 'w'.

Usage

```
bbox.hcube.tiling(w, L, R)
```

Arguments

w	A numeric value representing the width of each sub-box, i.e., the edge length.
L	A numeric vector representing the left vertices of the bounding box.
R	A numeric vector representing the right vertices of the bounding box.

Details

The function first ensures that 'L' and 'R' have the same length and then calculates the number of grid elements along each coordinate. It then creates corresponding intervals and constructs the sub-boxes within the bounding box.

Value

A list of sub-boxes, where each sub-box is represented as a list with components 'L' and 'R', containing the left and right vertices of the corresponding sub-box.

See Also

extract.xy for extracting x and y values from a character vector.

Examples

```
## Not run:
w <- 1
L <- c(0, 0)
R <- c(2, 3)
sub_boxes <- bbox.hcube.tiling(w, L, R)

## End(Not run)
```

bi.gaussian — *Bi-Gaussian defined as Gaussian with sigma.left for x <= mu and a Gaussian with sigma.right for x > mu*

Description

Bi-Gaussian defined as Gaussian with sigma.left for x <= mu and a Gaussian with sigma.right for x > mu

Usage

```
bi.gaussian(x, mu, sigma.left, sigma.right)
```

Arguments

`x`	A vector of x-values.
`mu`	The location of the global maximum of the bi-gaussian function.
`sigma.left`	The standard deviation of the left part of the bi-gaussian function.
`sigma.right`	The standard deviation of the right part of the bi-gaussian function.

bi.gaussian.mixture — *Creates a Synthetic Function with Specified Number of Local Maxima using bi-Gaussians.*

Description

Generates a synthetic function with specified number of local maxima using bi-Gaussian functions and a partition of unity. This function is useful for creating complex, non-linear synthetic data for analysis and testing.

Usage

```
bi.gaussian.mixture(
  n.lmax,
  x.lmax = NULL,
  y.lmax = NULL,
  x.min = 0,
  x.max = 10,
  y.min = 1,
  y.max = 5,
  min.dist = NULL,
  C.min.dist = 0.5,
  max.itr = 1000,
  n.grid = 400,
  C = 1.5,
  q = 3,
  compact.support = FALSE
)
```

Arguments

`n.lmax`	Number of local maxima.
`x.lmax`	Locations of local maxima.
`y.lmax`	Values of local maxima.
`x.min`	Minimum x value for the grid (defaults to 0).
`x.max`	Maximum x value for the grid (defaults to 10).
`y.min`	Minimum y value for local maxima (defaults to 1).
`y.max`	Maximum y value for local maxima (defaults to 5).
`min.dist`	A positive numeric value specifying the minimum allowable distance between any two consecutive elements of x.
`C.min.dist`	A real number between 0 and 1 used to specify min.dist if it is NULL. We use the formula min.dist = C.min.dist * (x.max - x.min) / n.lmax.
`max.itr`	The maximal number of iterations for finding x.lmin so that the distance between consecutive elements is no less than min.dist.
`n.grid`	Number of grid points (defaults to 400).
`C`	Scaling factor for the partition of unity (defaults to 2).
`q`	A power parameter of q-Gaussian.
`compact.support`	Set to TRUE to use partition of unity with the components with compact support and unbound support otherwise.

Details

The function generates a set of local maxima points and their corresponding y-values. It then constructs a partition of unity and a matrix of bi-Gaussian functions centered at these maxima. The bi-Gaussian function is defined as a Gaussian with sigma.left for x <= mu and a Gaussian with sigma.right for x > mu. The final synthetic function is the sum of these bi-Gaussians weighted by the partition of unity.

Value

A list containing the following components:

- `x`: Vector of x locations for the grid points.
- `y`: Vector of function values at each grid point.
- `x.lmax`: Vector of x locations of the local maxima.
- `y.lmax`: Vector of y values of the local maxima.

`BIC.graph.spectral.lowess` *BIC for Graph Spectral LOWESS*

Description

Computes the Bayesian Information Criterion

Usage

```
## S3 method for class 'graph.spectral.lowess'
BIC(object, ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`...`	Additional arguments (currently unused)

Value

BIC value

`BIC.graph.spectral.ma.lowess` *BIC for Graph Spectral MA LOWESS*

Description

Computes the Bayesian Information Criterion accounting for model averaging

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
BIC(object, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

BIC value

BIC.mabilog *Compute BIC for Mabilog Model*

Description

Computes Bayesian Information Criterion for model selection

Usage

```
## S3 method for class 'mabilog'
BIC(object, ...)
```

Arguments

`object`	A 'mabilog' object
`...`	Additional arguments passed to logLik

Value

BIC value

BIC.mabilo_plus *Compute BIC for Mabilo Plus Model*

Description

Computes Bayesian Information Criterion for model selection

Usage

```
## S3 method for class 'mabilo_plus'
BIC(object, type = c("ma", "sm"), ...)
```

Arguments

`object`	A 'mabilo_plus' object
`type`	Character string specifying which predictions to use: "ma" (default) or "sm"
`...`	Additional arguments passed to logLik

Value

BIC value

bin.segments3d *Add 3D Line Segments for Binary Variable*

Description

Adds vertical line segments to a 3D plot at positions where a binary variable equals 1

Usage

```
bin.segments3d(
  X,
  y,
  offset,
  with.labels = TRUE,
  lab.tbl = NULL,
  lab.adj = c(0, 0),
  lab.cex = 1,
  C = 1,
  ...
)
```

Arguments

`X`	A matrix or data.frame with 3 columns representing 3D coordinates.
`y`	A named binary (0/1) vector.
`offset`	Numeric vector of length 3 specifying the offset for line segments.
`with.labels`	Logical. Whether to show labels for y=1 positions.
`lab.tbl`	Named vector mapping sample IDs to labels.
`lab.adj`	Adjustment parameter for label positioning.
`lab.cex`	Character expansion factor for labels.
`C`	Scaling factor for label position relative to stick center.
`...`	Additional arguments passed to `segments3d`.

Details

This function adds vertical line segments (sticks) to an existing 3D plot at positions where the binary variable y equals 1. Optionally, labels can be added above the sticks.

Value

Invisibly returns NULL.

Examples

```
## Not run:
X <- matrix(rnorm(300), ncol = 3)
y <- sample(0:1, 100, replace = TRUE)
names(y) <- rownames(X) <- paste0("Sample", 1:100)

plot3D.plain(X)
```

```
bin.segments3d(X, y, offset = c(0, 0, 0.1))

## End(Not run)
```

---

boa.overlap *Calculate Overlap Coefficients Between Basins of Attraction*

Description

Computes a matrix of overlap coefficients between basins of attraction (BoA) identified by local maxima indices in two variables. Only includes BoAs that meet the minimum size requirement.

Usage

```
boa.overlap(x, y, min.BoA.size, x.labels = NULL, y.labels = NULL)
```

Arguments

x	Named integer vector where values are indices of local maxima and names are IDs of points in the basin for variable X
y	Named integer vector where values are indices of local maxima and names are IDs of points in the basin for variable Y
min.BoA.size	Minimum number of elements required for a basin to be included
x.labels	Named vector of labels for basins in x. Names must match unique values in x.
y.labels	Named vector of labels for basins in y. Names must match unique values in y.

Value

A matrix where rows correspond to BoAs from x and columns to BoAs from y, containing their pairwise overlap coefficients. Only includes BoAs meeting the size threshold. Row and column names are taken from x.labels and y.labels if provided, otherwise from the local maxima indices.

Examples

```
ids <- paste0("id", 1:500)
x <- c(rep(1, 200), rep(2, 300))
names(x) <- ids
y <- c(rep(1, 150), rep(2, 350))
names(y) <- ids
x.labels <- c("1" = "Peak A", "2" = "Peak B")
y.labels <- c("1" = "Max 1", "2" = "Max 2")
boa.overlap(x, y, min.BoA.size = 100, x.labels = x.labels, y.labels = y.labels)
```

box.tiling — *Creates a box tiling of the bounding box of some set.*

Description

This function creates a box tiling of the bounding box of some set by selecting only boxes of the box tiling containing elements of that set. Box tiling is an arrangement of boxes without gaps or overlaps, meaning that two boxes can intersect only along their faces.

Usage

```
box.tiling(
  X,
  n.segments.per.axis,
  eps,
  n.itrs = 1,
  p.exp = 0.05,
  plot.it = FALSE,
  verbose = FALSE
)
```

Arguments

`X`	A matrix of points in some Euclidean space.
`n.segments.per.axis`	An integer number specifying the number of segments (subintervals) of each axis that will be used to construct box tiling of the given box. The segmenst are of equal length within each axis, but potentially of different lengths between different axes)
`eps`	A distance threshold (non-negative real number) such that each box is expanded by eps and then tested for the presence of the elements of X.
`n.itrs`	The number of iterations of box sub-divisions. Default 1.
`p.exp`	An expansion factor for the bounding box. Each edge of the box is scaled by the factor (1 + p.exp), where p.exp is a non-negative real number.
`plot.it`	Set to TRUE to see in 2D the progress of the tiling construct.
`verbose`	Set to TRUE for progress messages.

Value

A list with two components:

- `X.boxes`: A list of boxes covering X, where each box is represented by its left and right vertices.
- `X.vol.ratio`: A numerical vector containing estimates of X's volume ratio over the iterations within this function.

Examples

```
## Not run:
  X <- matrix(runif(100), ncol = 2)
  boxes <- box.tiling(X, n = 50, eps = 0.05, p.exp = 0.05)

## End(Not run)
```

---

box.tiling.of.box *Creates box tiling of a box.*

Description

This function constructs a box tiling of a box specified by vectors 'L' and 'R'. The boxes forming the tiling are created by subdividing each axis into the same number of equal length intervals. A box is (also called hyperrectangle or orthotope) is defined as the Cartesian product of k intervals, where k is the dimension of the orthotope. A box is defined by two opposite vertices L and R.

Usage

```
box.tiling.of.box(n, box)
```

Arguments

`n`	An integer number specifying the number of subintervals (of equal length within each axis, but of potentially of different lengths between different axes) of each axis that will be used to construct box tiling of the given box.
`box`	A box with L, R, i components.

Value

A list of sub-boxes, where each sub-box is represented as a list with components 'L' and 'R', containing the left and right vertices of the corresponding sub-box.

See Also

`extract.xy` for extracting x and y values from a character vector.

Examples

```
## Not run:
w <- 1
box <- list()
box$L <- c(0, 0)
box$R <- c(2, 3)
box$i <- 1
sub_boxes <- box.tiling.of.box(w, box)

## End(Not run)
```

`boxcox.mle` *Box-Cox MLE for the power parameter λ*

Description

Estimates the Box-Cox power parameter λ by profiling the Gaussian log-likelihood over a grid with optional local refinement.

Usage

```
boxcox.mle(formula, data, lambdas = seq(-2, 2, by = 0.1), refine = TRUE)
```

Arguments

`formula`	A model formula of the form `y ~ x1 + x2 + ...`, or an `lm` object. The response `y` must be strictly positive.
`data`	Optional data frame for `formula`. Ignored if `formula` is an `lm` object.
`lambdas`	Numeric vector of candidate λ values used to build the coarse profile (default `seq(-2, 2, by = 0.1)`).
`refine`	Logical; if `TRUE` (default) performs a 1D `optimize` search near the best grid point to refine λ.

Details

For each λ in `lambdas`, the response is transformed via `boxcox.transform` and a least-squares fit is computed with `lm.fit`. The profiled log-likelihood $\ell(\lambda)$ is

$$\ell(\lambda) = -\frac{n}{2}\log\left(\frac{\mathrm{RSS}(\lambda)}{n}\right) + (\lambda - 1)\sum_{i=1}^{n}\log y_i,$$

up to an additive constant. A 95\ likelihood-ratio cutoff $2\{\ell(\hat\lambda) - \ell(\lambda)\} \le \chi^2_{1,0.95}$.

The implementation covers the standard unweighted Gaussian linear model. (Weighted fits can be added using `lm.wfit` and a weighted $\sum \log y$ term.)

Value

A list with components:

`lambda`	The MLE $\hat\lambda$.
`loglik`	A data frame with columns `lambda` and `loglik` giving the profile evaluated on the coarse and dense grids.
`ci95`	Approximate 95\ from the LR cutoff, returned as a numeric vector of length 2.

References

Box, G. E. P. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B*, **26**(2), 211-252.

Examples

```
set.seed(1)
n <- 60
x <- runif(n)
y <- exp(1 + 2 * x + rnorm(n, sd = 0.2))  # positive response
d <- data.frame(y = y, x = x)

fit <- boxcox.mle(y ~ x, data = d)
fit$lambda
head(fit$loglik)
fit$ci95

# Transform y with the estimated lambda
y.bc <- boxcox.transform(d$y, fit$lambda)
```

`boxcox.transform` *Box-Cox power transform*

Description

Applies the Box-Cox transform to a positive numeric vector y. Uses $\log(y)$ when $\lambda \approx 0$, and $(y^\lambda - 1)/\lambda$ otherwise.

Usage

```
boxcox.transform(y, lambda)
```

Arguments

`y`	A numeric vector of strictly positive and finite values.
`lambda`	A numeric scalar giving the power parameter λ.

Details

The transform is defined as

$$T_\lambda(y) = \begin{cases} \dfrac{y^\lambda - 1}{\lambda}, & \lambda \neq 0, \\ \log(y), & \lambda = 0. \end{cases}$$

This implementation switches to the log form when $|\lambda| < 10^{-8}$.

Value

A numeric vector of the same length as `y`, containing the transformed values.

See Also

`boxcox.mle`

Examples

```
y <- rexp(50, rate = 2)          # positive data
boxcox.transform(y, lambda = 0)  # log-transform
boxcox.transform(y, lambda = 0.5)
```

`bump.fn` *Bump Function*

Description

This function creates a bump function, which is a smooth function with compact support. The bump function is commonly used in mathematical analysis, particularly in the construction of partitions of unity. The function reaches its peak at the `offset` and smoothly decreases to zero at the boundary of its support.

Usage

```
bump.fn(x, offset = 0, h = 1, q = 4)
```

Arguments

`x`	A numeric vector or a single numeric value where the bump function is evaluated.
`offset`	The center (offset) of the bump function (defaults to 0). The function reaches its maximum value at this point.
`h`	The radius of the support of the bump function (defaults to 1). The function is zero outside the interval $[offset - h, offset + h]$.
`q`	The power to which the Gaussian function is raised (defaults to 4).

Details

The bump function is defined as exp(-1 / (h - (x - offset)^2)) for |x - offset| < h, and 0 otherwise. It is infinitely differentiable and is used in situations where a smooth function with compact support is needed.

Value

A numeric vector or a single numeric value representing the value(s) of the bump function at the input `x`. The values are in the range $[0, 1]$, with the function smoothly approaching zero at the boundaries of its support.

Examples

```
## Not run:
x <- seq(-2, 2, length.out = 100)
plot(x, bump.fn(x), type = "l")

## End(Not run)
```

`calculate.edit.distances`
Calculate Edit Distances Between Sequential Graphs

Description

Calculates edit distances between pairs of graphs separated by a specified offset in the sequence of k values.

Usage

```
calculate.edit.distances(
  k.values,
  graph.data,
  offset = 1,
  edge.cost = 1,
  weight.cost.factor = 0.1,
  verbose = FALSE
)
```

Arguments

`k.values`	Numeric vector of k values corresponding to the loaded graphs.
`graph.data`	List containing graph data as produced by `load.graph.data`.
`offset`	Positive integer specifying the offset between compared graphs (default: 1). An offset of 1 compares consecutive graphs.
`edge.cost`	Cost parameter passed to `graph.edit.distance` (default: 1).
`weight.cost.factor`	Weight cost factor passed to `graph.edit.distance` (default: 0.1).
`verbose`	Logical indicating whether to show progress messages (default: FALSE).

Details

For a sequence of k values and an offset of 1, this function compares:

- Graph at `k[1]` with graph at `k[2]`
- Graph at `k[2]` with graph at `k[3]`
- And so on...

Value

A list containing:

`indices`	Integer vector of indices in the k.values sequence
`k.values`	Numeric vector of k values corresponding to the first graph in each comparison
`distances`	Numeric vector of calculated edit distances

Examples

```
## Not run:
# Assuming graph.data has been loaded
k.vals <- c(5, 10, 15, 20)
distances <- calculate.edit.distances(k.vals, graph.data, offset = 1)
plot(distances$k.values, distances$distances, type = "b",
     xlab = "k", ylab = "Edit Distance")

## End(Not run)
```

`centroid.shift` *Subtracts the centroid from each point of the given dataset.*

Description

This function centers a dataset by subtracting a centroid from each point (row). If no centroid is provided, it computes the centroid as the column means of X.

Usage

```
centroid.shift(X, centroid = NULL)
```

Arguments

`X`	A matrix or data.frame specifying a set of points, where each row represents a point and each column represents a dimension.
`centroid`	A numeric vector specifying the centroid to subtract from each point. If NULL (default), the centroid is computed as the column means of X. Must have length equal to ncol(X).

Value

A numeric matrix of the same dimensions as X, with the centroid subtracted from each row. The returned matrix represents the centered dataset.

Examples

```
# Example 1: Auto-compute centroid
X <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
centroid.shift(X)

# Example 2: Provide custom centroid
X <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
custom_centroid <- c(2, 5)
centroid.shift(X, centroid = custom_centroid)
```

`chain.with.path` *Create a chain.with.path object*

Description

Constructor function for creating a chain.with.path object that represents a graph structure with adjacency and weight information, typically used in path-based algorithms or network analysis.

Usage

```
chain.with.path(adj.list, weight.list, gpd.obj)
```

Arguments

`adj.list`	List representing the adjacency structure of the graph
`weight.list`	List of weights corresponding to edges in the adjacency list
`gpd.obj`	Object containing additional graph path data or metadata

Value

An object of class "chain.with.path" containing the graph structure and associated data

`circle.plot` *Create Circle Plot of Matrix Data*

Description

Generates a circle plot where matrix columns are mapped to points on a unit circle

Usage

```
circle.plot(
  X,
  n.circle.pts = 100,
  delta = 0.11,
  pch = ".",
  cex = 2.5,
  col = "blue",
  comp.pch = 19,
  comp.col = "red",
  comp.cex = 1.2,
  lab.cex = 1.2,
  edge.col = "gray",
  adj.df = NULL
)
```

Arguments

`X`	A data matrix where rows are observations and columns are variables.
`n.circle.pts`	Number of points used to draw the circle.
`delta`	Margin offset for accommodating labels.
`pch`	Plotting character for data points.
`cex`	Character expansion factor for data points.
`col`	Color of data points.
`comp.pch`	Plotting character for component points on circle.
`comp.col`	Color of component points.
`comp.cex`	Character expansion factor for component points.
`lab.cex`	Character expansion factor for labels.
`edge.col`	Color of edges from center to circle points.
`adj.df`	Matrix of text adjustment values (n.cols x 2) for label positioning.

Details

This function creates a visualization where the first variable is placed at the center of a unit circle, and remaining variables are uniformly distributed around the circle. Data points are then projected into this circular space.

Value

Invisibly returns the component points matrix.

Examples

```
## Not run:
X <- matrix(rnorm(50), ncol = 5)
colnames(X) <- paste0("Var", 1:5)
circle.plot(X)

## End(Not run)
```

`circular.synthetic.mixture.of.gaussians` *Create a Synthetic Mixture of Circular Gaussians*

Description

This function generates a mixture of periodic Gaussian distributions on a circle. It ensures that the resulting function is smooth and continuous at all points, including the 0/2pi connection point.

Usage

```
circular.synthetic.mixture.of.gaussians(x, x.knot, y.knot, sd.knot)
```

Arguments

`x`	A numeric vector of angles (in radians) at which to evaluate the mixture function.
`x.knot`	A numeric vector specifying the centers of the Gaussian distributions on the circle, in radians.
`y.knot`	A numeric vector specifying the heights of the Gaussian distributions. Must have the same length as x.knot.
`sd.knot`	A positive numeric value specifying the standard deviation for all Gaussian distributions in the mixture.

Details

The function uses a periodic Gaussian distribution to ensure smoothness and continuity around the entire circle. It calculates the shortest distance between points on the circle in both clockwise and counterclockwise directions, ensuring proper wrapping at the 0/2pi point.

Value

A numeric vector of the same length as x, containing the values of the Gaussian mixture function evaluated at the input angles.

Examples

```
# Generate a simple circular Gaussian mixture
x <- seq(0, 2*pi, length.out = 100)
y <- circular.synthetic.mixture.of.gaussians(
  x,
  x.knot = c(0, pi, 1.5*pi),
  y.knot = c(5, 8, 2.5),
  sd.knot = 0.3
)

# Plot the result
plot(x, y, type = "l", xlab = "Angle (radians)", ylab = "Value")
```

`clamp` *Restricts the range of value of a numeric vector to* `[xmin, xmax]`.

Description

Restricts the range of value of a numeric vector to `[xmin, xmax]`.

Usage

```
clamp(x, xmin, xmax)
```

Arguments

`x`	A numeric vector.
`xmin`	The minimum value of the clamped x.
`xmax`	The maximum value of the clamped x.

Value

A vector with all values below xmin set to xmin and all value above xmax set to xmax.

`classifier.based.divergence`
Estimate Divergence Using a Classifier-based Approach

Description

This function estimates the divergence between two datasets using a classifier-based approach. The intuition behind this method is that if two datasets are very different, it should be easy to train a classifier to distinguish between them, resulting in a low classification error.

The algorithm works by:

1. Combining X and Y into a single dataset with labels.
2. Training a logistic regression model with elastic net regularization to classify the points.
3. Using the negative log-likelihood of the model as a proxy for divergence.

This approach has several advantages:

1. It can handle high-dimensional data well, especially with the elastic net regularization.
2. It provides a flexible framework that can be adapted to different types of data by choosing appropriate classification algorithms.
3. The regularization parameter alpha allows for tuning between L1 and L2 regularization, providing control over feature selection and multicollinearity handling.

While not a direct measure of relative entropy, this method provides a robust way to quantify the dissimilarity between datasets, especially in high-dimensional spaces.

Usage

```
classifier.based.divergence(X, Y, alpha = 0.5)
```

Arguments

`X`	A matrix or data frame representing the first dataset.
`Y`	A matrix or data frame representing the second dataset.
`alpha`	The elastic net mixing parameter, with 0 <= alpha <= 1. alpha=1 is the lasso penalty, and alpha=0 is the ridge penalty.

Value

A numeric value representing the estimated divergence.

References

Friedman, J., Hastie, T., & Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. Journal of statistical software, 33(1), 1.

Examples

```
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- classifier.based.divergence(X, Y)
print(result)
```

`clusters.reorder` *Reorder Cluster IDs by Size*

Description

Reorders cluster IDs based on cluster sizes, with the largest cluster receiving ID 1, the second largest ID 2, and so on.

Usage

```
clusters.reorder(cltr, decreasing = TRUE)
```

Arguments

`cltr`	A vector of cluster IDs. Can be numeric, character, or factor.
`decreasing`	Logical. If `TRUE` (default), clusters are ordered from largest to smallest. If `FALSE`, from smallest to largest.

Value

A vector of the same length and type as `cltr` with reordered cluster IDs. The output preserves the names attribute if present in the input.

Examples

```
# Numeric clusters
cltr <- c(1, 2, 2, 3, 3, 3)
clusters.reorder(cltr)
# Returns: [1] 3 2 2 1 1 1

# Character clusters
cltr_char <- c("A", "B", "B", "C", "C", "C")
clusters.reorder(cltr_char)
# Returns: [1] "C" "B" "B" "A" "A" "A"

# With names preserved
named_cltr <- c(a = 1, b = 2, c = 2, d = 3, e = 3, f = 3)
clusters.reorder(named_cltr)
```

`coef.assoc1` *Coef Method for assoc1 Objects*

Description

Extracts key coefficients from an assoc1 object.

Usage

```
## S3 method for class 'assoc1'
coef(object, ...)
```

Arguments

`object`	An object of class "assoc1".
`...`	Additional arguments (currently ignored).

Value

A named numeric vector containing delta1, Delta1, and p-value.

`coef.graph.spectral.lowess` *Extract Coefficients from Graph Spectral LOWESS*

Description

Returns the smoothed values (predictions) as coefficients. Note: Unlike linear models, LOWESS doesn't have traditional coefficients.

Usage

```
## S3 method for class 'graph.spectral.lowess'
coef(object, ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`...`	Additional arguments (currently unused)

Value

Named numeric vector with smoothed values `coef.graph.spectral.ma.lowess` *Extract Coefficients from Graph Spectral MA LOWESS*

Description

Returns the model-averaged smoothed values (predictions) as coefficients. Note: Unlike linear models, LOWESS doesn't have traditional coefficients.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
coef(object, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

Named numeric vector with smoothed values

`coef.mabilog` *Extract Model Coefficients from Mabilog Model*

Description

Extracts the optimal k value and model information from a mabilog object

Usage

```
## S3 method for class 'mabilog'
coef(object, ...)
```

Arguments

`object`	A 'mabilog' object
`...`	Additional arguments (currently unused)

Value

Named vector containing optimal k value and other parameters

`coef.mabilo_plus` *Extract Model Coefficients from Mabilo Plus Model*

Description

Extracts the optimal k values from a mabilo_plus object

Usage

```
## S3 method for class 'mabilo_plus'
coef(object, ...)
```

Arguments

`object`	A 'mabilo_plus' object
`...`	Additional arguments (currently unused)

Value

Named vector containing optimal k values

`compare.adj.lists` *Compare Two Adjacency Lists*

Description

This function compares two adjacency lists to check if they are equivalent, meaning each vertex's list of neighbors in the first adjacency list contains exactly the same set of neighbors as in the corresponding list in the second adjacency list, regardless of order.

Usage

```
compare.adj.lists(adj.list1, adj.list2)
```

Arguments

`adj.list1`	A list of integer vectors, where each vector represents the adjacency list for a vertex in the first graph.
`adj.list2`	A list of integer vectors, where each vector represents the adjacency list for a vertex in the second graph.

Details

The function performs a length check on the adjacency lists and then uses `setequal` for each corresponding pair of sub-lists to verify equivalence of the adjacency sets. This function is useful for verifying the equality of graphs in terms of connectivity, ignoring the order of nodes in the adjacency lists and potential multiple edges between nodes.

Value

A logical value; TRUE if the two adjacency lists are equivalent for all corresponding vertices, otherwise FALSE.

Examples

```
adj.list1 <- list(c(2, 3), c(1, 3), c(1, 2))
adj.list2 <- list(c(3, 2), c(3, 1), c(2, 1))
compare.adj.lists(adj.list1, adj.list2) # returns TRUE

adj.list1 <- list(c(2, 3, 4), c(1, 3), c(1, 2), c(1))
adj.list2 <- list(c(3, 2), c(3, 1), c(2, 1), c(1))
compare.adj.lists(adj.list1, adj.list2) # returns FALSE
```

---

compare.graph.vs.nerve.cx.filtering

Compare Graph vs. Nerve Complex Spectral Filtering

Description

Compares the performance of graph-based spectral filtering (using only the 1-skeleton) with nerve complex spectral filtering (incorporating higher-order simplicial information). This function quantifies the improvement gained by utilizing the full simplicial structure.

Usage

```
compare.graph.vs.nerve.cx.filtering(
  complex,
  y,
  y.true = NULL,
  laplacian.type = "STANDARD",
  filter.type = "HEAT",
  laplacian.power = 1,
  dim.weights.complex = NULL,
  graph.only.weights = NULL,
  kernel.params = list(tau.factor = 0.01, radius.factor = 3),
  n.evectors = 100,
  n.candidates = 100,
  verbose = FALSE
)
```

Arguments

`complex`	A nerve complex object created by `create.nerve.complex`.
`y`	Numeric vector of observed function values at vertices.
`y.true`	Optional numeric vector of true function values for MSE calculation. If provided, must have the same length as `y`.
`laplacian.type`	Character string specifying the Laplacian type. See `nerve.cx.spectral.filter` for options.

`filter.type`	Character string specifying the filter type. See `nerve.cx.spectral.filter` for options.
`laplacian.power`	Positive integer for the Laplacian power.
`dim.weights.complex`	Numeric vector of dimension weights for complex filtering. If `NULL`, uses exponentially decreasing weights.
`graph.only.weights`	Numeric vector of dimension weights for graph filtering. If `NULL`, uses weight 1 for dimension 0 and 0 for others.
`kernel.params`	List of kernel parameters. See `nerve.cx.spectral.filter`.
`n.evectors`	Number of eigenvectors to compute.
`n.candidates`	Number of candidate parameter values to test.
`verbose`	Logical indicating whether to print progress information.

Details

This comparison function runs two separate filtering operations:

1. Graph-only filtering: Uses only the 1-skeleton (edges) of the complex
2. Full complex filtering: Incorporates all simplex dimensions

The improvement metrics help quantify the benefit of using higher-order simplicial information. Positive improvement percentages indicate that the complex filtering performs better than graph-only filtering.

Value

A list of class `nerve_cx_comparison` containing:

`graph_predictions`	Smoothed values using graph-only filtering
`complex_predictions`	Smoothed values using full complex filtering
`graph_gcv`	GCV score for graph-only filtering
`complex_gcv`	GCV score for complex filtering
`gcv_improvement_pct`	Percentage improvement in GCV score
`mse_graph`	MSE for graph filtering (if `y.true` provided)
`mse_complex`	MSE for complex filtering (if `y.true` provided)
`mse_improvement_pct`	Percentage improvement in MSE (if `y.true` provided)
`graph_result`	Full result object from graph filtering
`complex_result`	Full result object from complex filtering

Examples

```
## Not run:
# Generate test data
set.seed(123)
coords <- matrix(runif(200), ncol = 2)
f_true <- function(x) sin(2*pi*x[1]) * cos(2*pi*x[2])
y_true <- apply(coords, 1, f_true)
y_noisy <- y_true + rnorm(length(y_true), 0, 0.2)

# Create nerve complex
complex <- create.nerve.complex(coords, k = 8, max.dim = 2)
complex <- set.complex.function.values(complex, y_noisy)

# Compare filtering approaches
comparison <- compare.graph.vs.nerve.cx.filtering(
  complex, y_noisy, y_true,
  dim.weights.complex = c(1.0, 0.5, 0.25),
  verbose = TRUE
)

# Print improvement
cat("GCV improvement:", comparison$gcv_improvement_pct, "%\n")
cat("MSE improvement:", comparison$mse_improvement_pct, "%\n")

## End(Not run)
```

---

compare.harmonic.methods

Compare Multiple Harmonic Extension Methods

Description

Applies multiple harmonic extension methods to a subgraph around an extremum and compares their results using various metrics.

Usage

```
compare.harmonic.methods(
  subgraph,
  methods = c("weighted_mean", "harmonic_iterative", "harmonic_eigen",
    "biharmonic_harmonic", "boundary_smoothed"),
  max_iterations = 100,
  tolerance = 1e-06,
  sigma = 1,
  plot_results = TRUE,
  plot_type = c("2d", "3d")
)
```

Arguments

subgraph	A subgraph object created by `extract.extrema.subgraphs`

`methods`	Character vector of method names to test (default: all available methods)
`max_iterations`	Maximum number of iterations for iterative methods (default: 100)
`tolerance`	Convergence tolerance (default: 1e-6)
`sigma`	Parameter for weighted mean method (default: 1.0)
`plot_results`	Logical whether to visualize the results (default: TRUE)
`plot_type`	Character string specifying visualization type if `plot_results = TRUE`: "2d" or "3d" (default: "3d")

Value

A list containing:

results Named list of results from each method

metrics Data frame with comparison metrics for each method:

- `method`: Method name
- `iterations`: Number of iterations performed
- `final_change`: Final iteration change
- `converged`: Whether method converged
- `total_change`: Sum of absolute changes from original
- `extremum_removal`: Change at the extremum vertex

subgraph The input subgraph object

Examples

```
## Not run:
# Assumes subgraph was created by extract.extrema.subgraphs
comparison <- compare.harmonic.methods(
  subgraph,
  methods = c("harmonic_iterative", "harmonic_eigen")
)

## End(Not run)
```

`compare.paths` *Compare Paths Across Different Hop Limits*

Description

Analyzes how the shortest path between two vertices changes as the hop limit varies.

Usage

```
compare.paths(x, from, to)
```

Arguments

`x`	A path.graph.series object
`from`	Source vertex index (1-based)
`to`	Target vertex index (1-based)

Value

A data.frame with columns:

h Hop limit

path_exists Logical indicating if a path exists

path_length Total length of the shortest path

n_hops Number of hops in the path

path List column containing the path vertices

Examples

```
## Not run:
pgs <- create.path.graph.series(graph, edge.lengths, h.values = c(1, 2, 3))
compare.paths(pgs, from = 1, to = 3)

## End(Not run)
```

`complex.laplacian.solve`
Solve Full Laplacian Problem on Nerve Complex

Description

This function solves the full Laplacian problem on the nerve complex to extend the function values.

Usage

```
complex.laplacian.solve(
  complex,
  lambda = 1,
  dim.weights = rep(1, complex$max_dimension + 1)
)
```

Arguments

`complex`	A nerve complex object
`lambda`	Regularization parameter
`dim.weights`	Weights for each dimension's contribution

Value

Vector of extended function values

`compute.bayesian.effects` *Compute Standardized Effect Sizes with Bayesian Inference*

Description

Computes standardized effect sizes and associated Bayesian statistics for comparing multiple methods against a reference method. The standardization is performed using the standard deviation of the reference method. The function provides point estimates (mean and median effects), interval estimates (credible intervals and highest density intervals), and posterior probabilities for different hypotheses.

Usage

```
compute.bayesian.effects(bb.integrals, reference_idx = 1)
```

Arguments

`bb.integrals`	List of Bayesian bootstrap integral values, where each element contains bootstrap samples for a method. The list must be named with method identifiers.
`reference_idx`	Index of the reference method in the list (default: 1)

Details

The function computes several Bayesian statistics:

- Effect sizes are standardized by dividing differences by the reference method's SD
- 95% credible intervals show the central 95% of the effect size distribution
- 95% HDI (Highest Density Interval) shows the most probable 95% of effect sizes
- Posterior probabilities are computed for:
  - Method having smaller effect than reference (prob_smaller)
  - Method having larger effect than reference (prob_larger)
  - Effect being practically equivalent to reference (within ±0.1 SD)

Value

A data frame where each row represents a method (named by method identifiers) and columns contain:

- mean_effect: Mean standardized effect size
- median_effect: Median standardized effect size
- ci_lower: Lower bound of 95% credible interval
- ci_upper: Upper bound of 95% credible interval
- hdi_lower: Lower bound of 95% highest density interval
- hdi_upper: Upper bound of 95% highest density interval
- prob_smaller: Probability of smaller effect than reference
- prob_larger: Probability of larger effect than reference
- prob_practical_equiv: Probability of practical equivalence

Examples

```
# Create example data
bb.integrals <- list(
  method1 = rnorm(100, mean = 0, sd = 1),
  method2 = rnorm(100, mean = 0.5, sd = 1),
  method3 = rnorm(100, mean = -0.3, sd = 1.2)
)

results <- compute.bayesian.effects(bb.integrals)

# View results for specific method
results["method2", ]

# Compare probabilities
results[, c("prob_smaller", "prob_larger", "prob_practical_equiv")]
```

---

compute.diff.profiles *Create Higher-Order Difference Profiles*

Description

Computes and concatenates difference profiles up to order k from conditional mean estimates.

Usage

```
compute.diff.profiles(Eyg, k)
```

Arguments

Eyg	Numeric vector of conditional mean estimates E_x(y) over a uniform grid.
k	Integer specifying the highest order of difference profiles to compute. Must be positive.

Details

This function creates a comprehensive profile by concatenating the original values with successive differences. This can be useful for visualizing the behavior of a function and its derivatives simultaneously.

Value

A numeric vector containing the concatenated profiles: Eyg, diff(Eyg), diff(diff(Eyg)), ..., up to order k.

Examples

```
## Not run:
# Generate example curve
x <- seq(0, 1, length.out = 50)
Eyg <- sin(2*pi*x)

# Get profiles up to 3rd order
```

```
profiles <- compute.diff.profiles(Eyg, k = 3)

# The result contains:
# - Original values (50 elements)
# - First differences (49 elements)
# - Second differences (48 elements)
# - Third differences (47 elements)
# Total: 194 elements

## End(Not run)
```

---

compute.geodesic.stats

Compute Geodesic Statistics for Grid Vertices

Description

Analyzes how the number of geodesics scales with radius for each grid vertex. This function helps determine if collapsing/clustering geodesics is necessary and provides insights into the graph structure around grid vertices.

Usage

```
compute.geodesic.stats(
  adj.list,
  weight.list,
  min.radius = 0.2,
  max.radius = 0.5,
  n.steps = 5,
  n.packing.vertices = length(adj.list),
  max.packing.iterations = 20,
  packing.precision = 1e-04,
  verbose = FALSE
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`min.radius`	Numeric. Minimum radius as a fraction of graph diameter.
`max.radius`	Numeric. Maximum radius as a fraction of graph diameter.
`n.steps`	Integer. Number of radius steps to test.
`n.packing.vertices`	Integer. Number of vertices in the grid/packing.
`max.packing.iterations`	Integer. Maximum iterations for packing algorithm.
`packing.precision`	Numeric. Precision parameter for packing algorithm.
`verbose`	Logical. Whether to print progress information.

Value

A list of class "geodesic_stats" containing:

radii Numeric vector. Actual radii used in the analysis.

geodesic_rays Matrix. Number of geodesic rays for each vertex at each radius.

composite_geodesics Matrix. Number of composite geodesics for each vertex at each radius.

path_overlap List of matrices. Overlap statistics for each vertex at each radius.

grid_vertices Integer vector. The vertices selected as grid vertices.

summary Data frame. Summary statistics for each radius.

---

compute.gradient.trajectory

Compute Gradient Trajectory

Description

Computes both ascending and descending gradient trajectories from a grid point.

Usage

```
compute.gradient.trajectory(i, j, f.grid)
```

Arguments

`i`	Row index of starting point
`j`	Column index of starting point
`f.grid`	Matrix of function values

Value

List containing:

`ascending.path`	Matrix of (i,j) coordinates along ascending path
`descending.path`	Matrix of (i,j) coordinates along descending path
`local.max`	Coordinates of destination local maximum
`local.min`	Coordinates of destination local minimum

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) -((x-0.5)^2 + (y-0.5)^2)
f.grid <- evaluate.function.on.grid(f, grid)
traj <- compute.gradient.trajectory(15, 15, f.grid)
str(traj)

## End(Not run)
```

`compute.graph.diameter`
Compute the Diameter of a Weighted Undirected Graph

Description

This function calculates the diameter of a weighted undirected graph represented as adjacency and weight lists. The diameter is the length of the longest shortest path between any two vertices in the graph.

Usage

```
compute.graph.diameter(adj.list, weight.list)
```

Arguments

`adj.list`	A list where each element i contains the indices of vertices adjacent to vertex i
`weight.list`	A list where each element i contains the weights of edges connecting vertex i to its adjacent vertices in adj.list

Details

The function handles several special cases:

- If the graph has no edges, returns NA as the diameter
- If the graph is disconnected, returns Inf as the diameter
- For undirected graphs, edges are only added once to avoid duplication

Value

A list containing:

`diameter`	The diameter of the graph (numeric value, or Inf if disconnected)
`message`	A descriptive message about the diameter
`farthest_vertices`	The pair of vertices that are farthest apart
`diameter_path`	The shortest path between the farthest vertices

Note

Requires the ’igraph’ package to be installed

Examples

```
# Example with a simple graph
adj.list <- list(c(2,3), c(1,3), c(1,2))
weight.list <- list(c(1,2), c(1,3), c(2,3))
result <- compute.graph.diameter(adj.list, weight.list)
print(result$message)
```

compute.graph.distance

Calculate Shortest Path Distance Between Two Vertices in a Graph

Description

Computes the shortest path distance between two vertices in any weighted graph, using Dijkstra's algorithm.

Usage

```
compute.graph.distance(
  star.obj = NULL,
  i,
  j,
  adj.list = NULL,
  edge.lengths = NULL
)
```

Arguments

`star.obj`	Output from generate.star.dataset(). If NULL, adj.list and edge.lengths must be provided.
`i`	Index of source vertex
`j`	Index of target vertex
`adj.list`	List where each element i contains indices of vertices adjacent to vertex i
`edge.lengths`	List where element i contains lengths of edges from vertex i to its adjacent vertices

Value

Numeric value representing shortest path distance between vertices i and j

Examples

```
# Using star_obj
star.obj <- generate.star.dataset(n.points = 10, n.arms = 3)
dist <- compute.graph.distance(star.obj = star.obj, i = 1, j = 5)

## Not run:
# Using adjacency list and edge lengths directly
# Assuming adj_list and edge_lengths are defined
dist <- compute.graph.distance(i = 1, j = 5,
                              adj.list = adj_list,
                              edge.lengths = edge_lengths)

## End(Not run)
```

`compute.graph.gradient.flow`
Compute Gradient Flow Information for Vertices in a Graph

Description

Analyzes the gradient flow structure of a weighted graph by computing basins of attraction for local maxima and minima, determining flow directions at each vertex, and identifying ambiguous vertices that belong to multiple basins.

Usage

```
compute.graph.gradient.flow(
  adj.list,
  weight.list = NULL,
  y,
  lmax.list,
  lmin.list,
  verbose = FALSE
)
```

Arguments

`adj.list`	List where `adj.list[[i]]` contains indices of vertices adjacent to vertex i. Must be a valid adjacency list representation.
`weight.list`	List where `weight.list[[i]]` contains positive weights of edges from vertex i to corresponding vertices in `adj.list[[i]]`. If `NULL`, uniform weights of 1 are used.
`y`	Numeric vector of values at each vertex. Must have the same length as `adj.list`.
`lmax.list`	List of local maxima information. See `lmax.basins` for format details.
`lmin.list`	List of local minima information. See `lmin.basins` for format details.
`verbose`	Logical; if `TRUE`, prints progress messages during computation. Default is `FALSE`.

Details

The function performs the following steps:

- Computes basins of attraction for all local maxima using `lmax.basins`
- Computes basins of attraction for all local minima using `lmin.basins`
- Assigns each vertex to appropriate basins, identifying conflicts
- Determines gradient flow directions based on neighboring values
- For unassigned vertices, attempts to infer basin membership from neighbors

Gradient flow directions indicate the steepest ascent and descent paths from each vertex. When edge weights are provided, they influence the determination of these paths by scaling the effective gradients.

Value

A list with the following components:

flow_directions List where each element describes the gradient flow at a vertex.

lmax_basin_assignment Integer vector indicating which maximum basin each vertex belongs to, with NA for unassigned vertices

lmin_basin_assignment Integer vector indicating which minimum basin each vertex belongs to, with NA for unassigned vertices

ambiguous_vertices Integer vector of vertices that belong to multiple basins of the same type

basin_stats List containing statistics about the basins: - n_max_basins: Number of maximum basins - n_min_basins: Number of minimum basins - coverage: Proportion of vertices assigned to at least one basin - max_basin_sizes: Named vector of maximum basin sizes - min_basin_sizes: Named vector of minimum basin sizes

References

Morse, M. (1934). The Calculus of Variations in the Large. American Mathematical Society.

Forman, R. (1998). Morse theory for cell complexes. Advances in Mathematics, 134(1), 90-145.

See Also

`lmax.basins`, `lmin.basins`, `set.boundary`

Examples

```
## Not run:
# Create example graph
adj.list <- list(c(2), c(1,3), c(2,4,5), c(3), c(3))
y <- c(0.2, 0.5, 0.8, 1.0, 0.1)

# Define extrema
lmax.list <- list(list(lmax=4, vertices=c(4), label="peak"))
lmin.list <- list(list(lmin=5, vertices=c(5), label="valley"))

# Compute gradient flow
flow_info <- compute.graph.gradient.flow(adj.list, NULL, y,
                                         lmax.list, lmin.list)

# Examine results
print(flow_info$basin_stats)

## End(Not run)
```

`compute.local.distance.fidelity`

Assess Fidelity of Graph-Based Geodesic Distances to Euclidean Geometry

---

Description

This function compares local neighborhood structures defined by graph-based distances to those defined in the original Euclidean space. It evaluates how well a given graph preserves the local geometry of a dataset by computing two metrics for each data point:

Usage

```
compute.local.distance.fidelity(X, adj.list, weight.list, taus, max.k = 50)
```

Arguments

`X`	A numeric matrix of shape `[n, d]`, where each row represents a data point in d-dimensional space.
`adj.list`	A list of integer vectors of length `n`, giving the graph adjacency list. Each entry contains the 1-based indices of neighbors for that vertex.
`weight.list`	A list of numeric vectors of same length as `adj.list`, where each element contains edge weights for the corresponding neighbors.
`taus`	A numeric vector of radius values τ used to define local neighborhoods around each point.
`max.k`	Integer; the number of nearest neighbors to compute in the Euclidean space (used to approximate local neighborhoods).

Details

- **Jaccard Index:** Measures the similarity between the sets of neighbors within a radius τ in both the Euclidean and graph-based spaces.
- **Mean Absolute Deviation (MAD):** Measures the distortion in local distances over the intersection of the two neighborhood sets.

The graph is assumed to be provided as an adjacency list and a corresponding weight list. Any duplicate edges (i.e., undirected edges appearing in both directions) are automatically deduplicated.

Value

A named list of length equal to `length(taus)`. Each element is a list with:

`mean.jaccard` Mean Jaccard index over all data points for a given τ.

`mean.mad` Mean absolute deviation in distances between Euclidean and graph metrics over intersecting neighbors.

`jaccard.vals` A numeric vector of Jaccard index values per vertex.

`mad.vals` A numeric vector of MAD values per vertex.

Examples

```
## Not run:
set.seed(1)
X <- matrix(rnorm(300 * 2), ncol = 2)
g <- build_graph(X)  # your function to generate adj.list and weight.list
taus <- seq(0.05, 0.2, by = 0.01)
fidelity <- compute.local.distance.fidelity(X, g$adj.list, g$weight.list, taus)

## End(Not run)
```

`compute.morse.smale.cells`
Compute Morse-Smale Cells

Description

Assigns each grid point to a Morse-Smale cell based on gradient flow.

Usage

```
compute.morse.smale.cells(f.grid)
```

Arguments

`f.grid`	Matrix of function values

Value

Matrix of cell labels

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
cells <- compute.morse.smale.cells(f.grid)
image(cells)

## End(Not run)
```

`compute.pairwise.bayes.factors`
Compute Pairwise Bayes Factors for Method Comparisons

Description

Computes pairwise Bayes factors comparing all methods using Savage-Dickey density ratios. Each Bayes factor represents evidence for one method versus another.

Usage

```
compute.pairwise.bayes.factors(bb.integrals)
```

Arguments

`bb.integrals`	A named list of Bayesian bootstrap integral values, where each element contains numeric samples representing the performance metric for a method. Names of the list elements will be used as row and column names in the output matrix.

Details

The function uses the Savage-Dickey density ratio to compute Bayes factors:

- For each pair of methods, it computes the difference in their performance samples
- Estimates the posterior density of these differences using kernel density estimation
- Computes the prior density at 0 assuming a normal distribution
- The Bayes factor is the ratio: posterior density at 0 / prior density at 0
- When density estimation fails, uses extreme values (1000 or 0.001) based on mean difference

Value

A square matrix of Bayes factors where `BF[i,j]` represents the evidence for method i being better than method j. Values > 1 indicate evidence for method i, values < 1 indicate evidence for method j. The diagonal contains 1s (method compared to itself).

Note

- Uses Sheather-Jones bandwidth selection for robust density estimation
- Interpretation: BF > 10 (strong evidence), BF > 3 (moderate evidence), BF > 1 (weak evidence)
- The function assumes that higher values in bb.integrals indicate better performance

References

Wagenmakers, E. J., Lodewyckx, T., Kuriyal, H., & Grasman, R. (2010). Bayesian hypothesis testing for psychologists: A tutorial on the Savage-Dickey method. Cognitive Psychology, 60(3), 158-189.

Examples

```
# Example: Compare three methods
set.seed(123)
bb.integrals <- list(
  method_A = rnorm(1000, mean = 0.5, sd = 0.1),
  method_B = rnorm(1000, mean = 0.52, sd = 0.1),
  method_C = rnorm(1000, mean = 0.48, sd = 0.1)
)

bf_matrix <- compute.pairwise.bayes.factors(bb.integrals)
print(bf_matrix)

# Interpret results
# bf_matrix[1,2] > 1 suggests evidence for method_A over method_B
# bf_matrix[2,3] > 1 suggests evidence for method_B over method_C
```

`compute.persistence` *Compute Persistent Homology for Graphs*

Description

Computes 0-dimensional persistent homology for a graph based on vertex function values. This tracks the birth and death of connected components during a filtration process.

Usage

```
compute.persistence(adj.list, y, n.top.basins = 4, alpha = 0.5)
```

Arguments

`adj.list`	A list where each element contains the indices of adjacent vertices.
`y`	A numeric vector of function values at vertices.
`n.top.basins`	Integer; number of top persistence basins to return (default: 4).
`alpha`	Numeric between 0 and 1; weight parameter for weighted persistence (default: 0.5). Only used when `weighted = TRUE`.

Details

For unweighted graphs, the algorithm processes vertices in decreasing order of function values, tracking when connected components merge. For weighted graphs, the death values incorporate both vertex values and edge weights.

The persistence of a component quantifies its significance, with higher values indicating more prominent topological features.

Value

A list containing:

persistence Data frame with persistence information for each component

basins List of vertex sets for the top persistence basins

Examples

```
# Simple chain graph
adj.list <- list(c(2), c(1,3), c(2,4), c(3,5), c(4))
y <- c(10, 7, 9, 2, 5)

# Compute persistence
result <- compute.persistence(adj.list, y)
print(result$persistence)
```

compute.vertex.geodesic.stats *Compute Geodesic Statistics for a Single Grid Vertex*

Description

Analyzes how the number of geodesics scales with radius for a specific grid vertex.

Usage

```
compute.vertex.geodesic.stats(
  adj.list,
  weight.list,
  grid.vertex,
  min.radius = 0.2,
  max.radius = 0.5,
  n.steps = 5,
  n.packing.vertices = length(adj.list)
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`grid.vertex`	Integer. The grid vertex to analyze.
`min.radius`	Numeric. Minimum radius as a fraction of graph diameter.
`max.radius`	Numeric. Maximum radius as a fraction of graph diameter.
`n.steps`	Integer. Number of radius steps to test.
`n.packing.vertices`	Integer. Number of vertices in the grid/packing.

Value

A data frame of class "vertex_geodesic_stats" with columns:

radius Actual radius used.

rays Number of geodesic rays within radius.

composite_geodesics Number of composite geodesics.

overlap_min Minimum overlap ratio.

overlap_p05 5th percentile overlap ratio.

overlap_p25 25th percentile overlap ratio.

overlap_median Median overlap ratio.

overlap_p75 75th percentile overlap ratio.

overlap_p95 95th percentile overlap ratio.

overlap_max Maximum overlap ratio.

`compute_asc_desc_cell_intersection_matrix`
Compute Ascending-Descending Cell Intersection Matrix

Description

Compute Ascending-Descending Cell Intersection Matrix

Usage

```
compute_asc_desc_cell_intersection_matrix(basin_cx)
```

Arguments

`basin_cx`	An object of class 'basin_cx'

Value

A square matrix where (i,j) is the number of shared vertices between cell i and cell j, with labels as lmin_label:lmax_label. Final row and column contain cell sizes.

`confint.graph.spectral.lowess`
Confidence Intervals for Graph Spectral LOWESS

Description

Computes confidence intervals for the smoothed values. Note: This requires bootstrap or other uncertainty quantification methods.

Usage

```
## S3 method for class 'graph.spectral.lowess'
confint(object, parm, level = 0.95, ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`parm`	Parameter specification (currently unused)
`level`	Confidence level (default: 0.95)
`...`	Additional arguments (currently unused)

Value

Matrix with lower and upper confidence bounds

`confint.graph.spectral.ma.lowess` *Confidence Intervals for Graph Spectral MA LOWESS*

Description

Computes confidence intervals for the model-averaged smoothed values. Model averaging naturally provides uncertainty quantification.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
confint(object, parm, level = 0.95, ...)
```

Arguments

`object`	A ’graph.spectral.ma.lowess’ object
`parm`	Parameter specification (currently unused)
`level`	Confidence level (default: 0.95)
`...`	Additional arguments (currently unused)

Value

Matrix with lower and upper confidence bounds

`construct.basin.cx.graph` *Construct Basin Complex Graph*

Description

Constructs a graph representation of a basin complex, where nodes represent basins (local maxima and minima) and edges represent their relationships. Edges are weighted based on basin overlap, function value differences, or other basin metrics.

Usage

```
construct.basin.cx.graph(
  basin_cx,
  min_intersection = 1,
  ms_edges_only = FALSE,
  weight_type = c("dice", "jaccard", "overlap", "y_diff")
)
```

Arguments

`basin_cx`	A basin complex object of class "basin_cx" returned by create.basin.cx()
`min_intersection`	Minimum number of vertices that must be shared between basins for an edge to be created (default: 1)
`ms_edges_only`	Logical; if TRUE, only include edges between ascending and descending basins, ignoring ascending-ascending and descending-descending connections (default: FALSE)
`weight_type`	Character string specifying the edge weight metric to use. Options are: "dice" (Dice-Sørensen similarity), "jaccard" (Jaccard index), "overlap" (overlap size), or "y_diff" (function value difference) (default: "dice")

Details

This function creates a graph where nodes represent basins (either local minima or maxima) and edges represent relationships between them. The resulting graph provides insight into the topological structure of the function on the original graph.

Edge weights can be based on different metrics:

- dice: Dice-Sørensen similarity index $(2|A \cap B|/(|A| + |B|))$
- jaccard: Jaccard similarity index $(|A \cap B|/|A \cup B|)$
- overlap: Raw number of vertices in the intersection
- y_diff: Absolute difference in function value between extrema

When ms_edges_only = TRUE, only edges between an ascending basin (minimum) and a descending basin (maximum) are included, producing a bipartite graph.

Value

A list with class "basin_cx_graph" containing:

adjacency_list List of adjacency lists for the basin complex graph

weights_list List of edge weights

intersection_matrix Matrix where `[i,j]` is the number of vertices shared between basins i and j

similarity_matrix Matrix of similarity values (based on weight_type) between basins

y_diff_matrix Matrix of function value differences between extrema

basin_metadata Data frame with information about each basin (type, extremum index, etc.)

n_ascending Number of ascending basins

n_descending Number of descending basins

weight_type The type of weight metric used

See Also

`create.basin.cx`

construct.function.aware.graph

Construct a Function-Aware Graph

Description

Creates a new graph with edge weights modified based on function values

Usage

```
construct.function.aware.graph(
  adj.list,
  weight.list,
  function.values,
  weight.type = 0,
  epsilon = 1e-06,
  lambda = 1,
  alpha = 1,
  beta = 5,
  tau = 0.1,
  p = 2,
  q = 2,
  r = 2,
  normalize = FALSE,
  weight.thld = -1
)
```

Arguments

`adj.list`	A list where each element contains the indices of vertices adjacent to vertex i. The list should be 1-indexed (as is standard in R).
`weight.list`	A list of the same structure as adj.list, where each element contains the weights of edges connecting vertex i to its adjacent vertices.
`function.values`	A numeric vector containing function values at each vertex of the graph.
`weight.type`	Type of weight modification: 0: Inverse relationship: w_new = w_old / (|f(i) - f(j)| + epsilon) 1: Direct relationship: w_new = w_old * |f(i) - f(j)| 2: Exponential decay: w_new = w_old * exp(-lambda * |f(i) - f(j)|) 3: Power law: w_new = w_old * |f(i) - f(j)|^(-alpha) 4: Sigmoid: w_new = w_old * 1/(1 + exp(beta * (|f(i) - f(j)| - tau))) 5: L_p embedding: w_new = (w_old^p + alpha*|f(i) - f(j)|^q)^(1/r)
`epsilon`	Small constant to avoid division by zero (for weight.type 0)
`lambda`	Decay rate parameter (for weight.type 2)
`alpha`	Power law exponent (for weight.type 3) or scaling factor (for weight.type 5)
`beta`	Sigmoid steepness parameter (for weight.type 4)
`tau`	Sigmoid threshold parameter (for weight.type 4)
`p`	Power for feature distance term (for weight.type 5)
`q`	Power for function difference term (for weight.type 5)

`r`	Power for the overall normalization (for weight.type 5)
`normalize`	Whether to normalize weights after modification
`weight.thld`	Threshold for pruning edges; edges with weight > weight.thld will be pruned. Set to a negative value to disable pruning.

Value

A new graph object with modified weights

`construct.graph.gradient.flow`
Graph Gradient Flow Analysis

Description

Analyzes the structure of a function defined on a weighted graph by computing gradient trajectories, basins, and Morse-Smale pro-cells. This function identifies local extrema (minima and maxima) and constructs trajectories that follow the gradient of the function across the graph.

The gradient flow computation relies on a scale parameter that defines the local neighborhood size for each vertex, within which the gradient direction is determined. Trajectories are constructed by following these gradient directions to connect local minima to local maxima.

Usage

```
construct.graph.gradient.flow(
  adj.list,
  weight.list,
  y,
  scale,
  quantile.scale.thld = 0.025,
  with.trajectories = FALSE
)
```

Arguments

`adj.list`	List of integer vectors where `adj.list[[i]]` contains the indices of vertices adjacent to vertex i (using 1-based indexing).
`weight.list`	List of numeric vectors where `weight.list[[i]][j]` is the weight of the edge from vertex i to `adj.list[[i]][j]`.
`y`	Numeric vector of function values at each graph vertex.
`scale`	Numeric vector of positive scale values for each vertex, controlling the local neighborhood size for gradient computation and extrema detection. If a single value, it is replicated for all vertices.
`quantile.scale.thld`	Numeric scalar between 0 and 1. Scale values are truncated to the interval `(0, quantile(scale, prob = quantile.scale.thld))`. Default is 0.025.
`with.trajectories`	Logical; if `TRUE`, trajectory information is included in the output. Default is `FALSE`.

Details

The gradient flow computation uses the scale parameter to determine the local neighborhood for each vertex. For a vertex `v`, the algorithm finds all shortest paths of length <= `scale[v]` starting at `v`, and selects the path with the steepest rate of change in function value. This approach ensures that the gradient direction is determined at an appropriate scale for each part of the graph.

The returned pro-cells partition the graph into regions, where each region contains trajectories flowing from a specific local minimum to a specific local maximum. This decomposition provides insight into the topological structure of the function defined on the graph.

Value

An object of class `"ggflow"` containing:

local_extrema Data frame with columns:

- **vertex_index** Integer; vertex index (1-based)
- **is_maximum** Integer; 1 for maxima, 0 for minima
- **label** Character; automatically generated label (e.g., "M1", "m2")
- **fn_value** Numeric; function value at the extremum

trajectories List of trajectories (NULL if `with.trajectories = FALSE`). Each trajectory contains:

- **vertices** Integer vector of vertex indices forming the trajectory
- **type** Character string: "LMIN_LMAX", "LMIN_ONLY", "LMAX_ONLY", or "UNKNOWN"
- **label** Character; trajectory label based on endpoints

basins List with two components:

- **ascending** List of basins around local minima
- **descending** List of basins around local maxima

Each basin contains:

- **local_min/local_max** Integer; index of extremum vertex
- **vertices** Integer vector; vertices in the basin
- **label** Character; basin label

cells List of gradient flow pro-cells. Each cell contains:

- **local_min** Integer; index of local minimum vertex
- **local_max** Integer; index of local maximum vertex
- **vertices** Integer vector; vertices in the pro-cell
- **label** Character; cell label

See Also

`summary.ggflow` `basins.merge`, `replace.basin.label`

Examples

```
## Not run:
# Create a simple grid graph
n <- 10
adj.list <- list()
weight.list <- list()

# Build grid adjacency (simplified example)
for(i in 1:(n*n)) {
  neighbors <- integer()
  weights <- numeric()

  # Add horizontal neighbors
  if(i %% n != 1) {
    neighbors <- c(neighbors, i-1)
    weights <- c(weights, 1.0)
  }
  if(i %% n != 0) {
    neighbors <- c(neighbors, i+1)
    weights <- c(weights, 1.0)
  }

  adj.list[[i]] <- neighbors
  weight.list[[i]] <- weights
}

# Define a function on the graph (e.g., a Gaussian peak)
coords <- expand.grid(x = 1:n/n, y = 1:n/n)
z <- exp(-10 * ((coords$x - 0.5)^2 + (coords$y - 0.5)^2))

# Compute gradient flow
flow <- construct.graph.gradient.flow(
  adj.list,
  weight.list,
  z,
  scale = 2.0,
  with.trajectories = TRUE
)

# Examine results
print(flow$local_extrema)
summary(flow)

## End(Not run)
```

`cont.segments3d` *Add 3D Line Segments Color-Coded by Continuous Variable*

Description

Adds line segments to a 3D plot with colors determined by a continuous variable

Usage

```
cont.segments3d(X, y, offset, ...)
```

Arguments

X	A matrix or data.frame with 3 columns representing 3D coordinates.
y	A named continuous numeric vector.
offset	Numeric vector of length 3 specifying the offset for line segments.
...	Additional arguments passed to segments3d.

Details

This function adds line segments to an existing 3D plot where each segment's color is determined by the rank of the corresponding y value. Colors range from red to violet following the rainbow spectrum.

Value

Invisibly returns NULL.

Examples

```
## Not run:
X <- matrix(rnorm(300), ncol = 3)
y <- runif(100)
names(y) <- rownames(X) <- paste0("Sample", 1:100)

plot3D.plain(X)
cont.segments3d(X, y, offset = c(0, 0, 0.1))

## End(Not run)
```

`convert.adjacency.list.to.adjacency.matrix`
Converts a graph adjacency list to an adjacency matrix

Description

This function converts a given graph adjacency list into an adjacency matrix, where each entry in the matrix indicates whether there is an edge between the corresponding nodes.

Usage

```
convert.adjacency.list.to.adjacency.matrix(adj.list, weights.list = NULL)
```

Arguments

`adj.list`	A list representing the adjacency list of a graph. Each element of the list is a vector of node indices that the corresponding node is connected to.
`weights.list`	(optional) A list of the same length as `adj.list`, where each element is a vector of weights for the edges connecting the corresponding node to the nodes in `adj.list`. If not provided, the adjacency matrix will be binary (1 for edges and 0 for no edges).

Value

A matrix representing the adjacency matrix of the graph.

Examples

```
adj.list <- list(c(2, 4), c(1, 3), c(2, 4), c(1, 3))
weights.list <- list(c(2, 4), c(2, 3), c(3, 1), c(4, 1))
convert.adjacency.list.to.adjacency.matrix(adj.list)
convert.adjacency.list.to.adjacency.matrix(adj.list, weights.list)
```

---

convert.adjacency.matrix.to.adjacency.list

Converts a graph adjacency matrix to an adjacency list

Description

This function converts a given graph adjacency matrix into an adjacency list, where each entry in the list corresponds to a node and contains the indices of nodes it is connected to.

Usage

```
convert.adjacency.matrix.to.adjacency.list(M)
```

Arguments

`M`	A matrix representing the adjacency matrix of a graph.

Value

A list representing the adjacency list of the graph.

---

convert.adjacency.to.edge.matrix

Convert Adjacency List to Edge Matrix

Description

This function converts an adjacency list representation of a graph to an edge matrix. It can handle both weighted and unweighted graphs.

Usage

```
convert.adjacency.to.edge.matrix(adj.list, weights.list = NULL)
```

Arguments

`adj.list`	A list where each element is a vector of integers representing the nodes adjacent to the node indexed by the element's position.
`weights.list`	An optional list of numeric vectors representing the weights of the edges. If provided, each element should correspond to the weights of the edges in the same position in adj.list.

Details

The function converts the input adjacency list to 0-based indexing before passing it to the C++ function. The returned edge matrix uses 1-based indexing. For undirected graphs, each edge is included only once, with the lower index always appearing first.

Value

A list with two elements:

`edge.matrix`	A matrix where each row represents an edge, given as a pair of node indices.
`weights`	A numeric vector of weights corresponding to the edges in edge.matrix. This is NULL if weights.list was not provided.

Examples

```
adj.list <- list(c(2,3), c(1,3), c(1,2))
result <- convert.adjacency.to.edge.matrix(adj.list)

# With weights
weights.list <- list(c(0.1, 0.2), c(0.1, 0.3), c(0.2, 0.3))
result_weighted <- convert.adjacency.to.edge.matrix(adj.list, weights.list)
```

`convert.adjacency.to.edgelist` *Convert an Adjacency List to an Edgelist*

Description

This function takes an adjacency list representation of a graph and converts it into an edgelist format suitable for use with plotting functions like `igraph`.

Usage

```
convert.adjacency.to.edgelist(adj.list)
```

Arguments

`adj.list`	An adjacency list where each element is a vector of indices representing the neighbors of a node.

Value

A list where each element is a vector of length two, representing an edge in the graph (source node, target node).

convert.edge.label.list.to.edge.label.vector *Converts an Edge Label List to an Edge Vector*

Description

This function takes a list of edge labels and turns it into a vector.

Usage

```
convert.edge.label.list.to.edge.label.vector(
  edge.label.list,
  rm.duplicates = TRUE
)
```

Arguments

`edge.label.list`	A list of edge labels.
`rm.duplicates`	Set to TRUE to allow for duplicate edges.

Value

A vector of edge labels.

convert.to.undirected *Converts a directed graph to an undirected graph*

Description

This function takes the adjacency list representation of a directed graph and converts it into the adjacency list representation of the corresponding undirected graph.

Usage

```
convert.to.undirected(adj.list)
```

Arguments

`adj.list`	A named list representing the adjacency list of the directed graph. Each element of the list is a character vector containing the names of the adjacent vertices for a given vertex.

Value

A named list representing the adjacency list of the corresponding undirected graph. Each element of the list is a character vector containing the names of the adjacent vertices for a given vertex in the undirected graph.

Examples

```
directed_graph <- list(
  "A" = c("B", "C"),
  "B" = c("C"),
  "C" = c("D"),
  "D" = c("A")
)

undirected_graph <- convert.to.undirected(directed_graph)
print(undirected_graph)
```

`convert.weighted.adjacency.matrix.to.adjacency.list` *Converts a graph weighted adjacency matrix to an adjacency list and weights list*

Description

This function converts a given weighted graph adjacency matrix into two lists:

1. An adjacency list where each entry corresponds to a node and contains the indices of nodes it is connected to.
2. A weights list where each entry corresponds to a node and contains the weights of the edges connecting it to other nodes.

Usage

```
convert.weighted.adjacency.matrix.to.adjacency.list(M)
```

Arguments

`M`	A matrix representing the weighted adjacency matrix of a graph.

Value

A list with two elements: `adjacency.list` and `weights.list`. `adjacency.list` is a list where each element is a vector of connected node indices. `weights.list` is a list where each element is a vector of weights corresponding to the edges in the adjacency list.

`count.edges` *Count Edges in an Undirected Graph*

Description

Computes the total number of edges in an undirected graph represented by an adjacency list. This function works by counting all neighbor connections across the graph and then dividing by 2, since each edge is represented twice in an undirected graph (once from each endpoint).

Usage

```
count.edges(adj.list)
```

Arguments

adj.list	A list where each element adj.list[[i]] contains the neighbors of vertex i as a vector of indices. For an undirected graph, if j is in adj.list[[i]], then i should also be in adj.list[[j]].

Value

A numeric value representing the number of edges in the graph. Returns 0 for an empty graph.

Examples

```
# Create an adjacency list for a simple undirected graph
# with 3 vertices and 2 edges: (1-2) and (2-3)
adj <- list(
  c(2),    # Vertex 1 is connected to vertex 2
  c(1, 3), # Vertex 2 is connected to vertices 1 and 3
  c(2)     # Vertex 3 is connected to vertex 2
)
count.edges(adj)  # Should return 2
```

---

create.2D.grid *Creates a 2D grid of points around X.*

Description

If X is NULL, x1.range and x2.range will be used to create a rectangular grid of n*n points.

Usage

```
create.2D.grid(n, X, f = 0.2, eSDf = 1.5, gRf = 4)
```

Arguments

n	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
X	A set of points around which the grid is created if not NULL.
f	A fraction of x1 and x2 range that the grid is extended to.
eSDf	An edge subdivisioin factor that is a scaling factor, such that edges of mstree(X) are subdivided if their length is greater than eSDf * mode(edge.len).
gRf	A grid radius factor that is a scaling factor, such that points of the rectangular grid that are further than gRf*mode(edge.len) away from the closest subdivision of the minimal spanning tree of X, mstree(X), are eliminated from the grid.

Value

```
A data frame, grid, with n*n rows and 2 columns of the grid points in
        2D. If X is not NULL, grid contains only points nor more than eps away
        from the mstree(X).
```

`create.2D.rect.grid` *Creates a rectangular 2D grid*

Description

Creates a rectangular 2D grid

Usage

```
create.2D.rect.grid(
  n,
  x1.range,
  x2.range,
  type = c("unif", "runif", "norm"),
  f = 0.2
)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`x1.range`	A range of x1 values - the first coordinate.
`x2.range`	A range of x2 values - the second coordinate.
`type`	A type of distribution the points are sampled from.
`f`	A fraction of x1 and x2 range that the grid is extended to.

`create.3D.grid` *Creates a 3D grid of points around X*

Description

Creates a 3D grid of points around X

Usage

```
create.3D.grid(n, X, f = 0.2, min.gSf = 1.5, eSDf = 1.5, gRf = 2)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`X`	A set of points around which the grid is created if not NULL.
`f`	A fraction of x1, x2 and x3 range that the grid is extended to.
`min.gSf`	A scaling factor, indicating the minimal size of the grid around X, with respect to the size of X. That is the minimal size of the grid needs to be at least min.gSf*|X|, where |X| is the size of X. For example, with min.gSf set to 1.5, the mimal size of the grid will be, if possible, at least 1.5*|X|. For this to be possible the initial rectangular grid size has to be at least of that minimal size, but in practice much bigger. It overrides gRf if the size of the grid given by the value of gRf is not

`eSDf`	An edge subdivisioin factor that is a scaling factor, such that edges of mstree(X) are subdivided is their length is greater than eSDf * mode(edge.len).
`gRf`	A scaling factor, such that points of the rectangular grid that are further than gRf*mode.edge.len away from the closest subdivision of the minimal spanning tree of X, mstree(X), are eliminated from the grid.

Value

```
A data frame, grid, with n*n*n rows and 3 columns of the grid points in
      3D. If X is not NULL, grid contains only points nor more than eps away
         from the mstree(X).
```

`create.3D.grid.v2` *Creates a 3D grid of points around X.*

Description

In this version if the min.gSf condition is not satisfied, n is increases until it satisfies that condition. Thus, gRf is never modified.

Usage

```
create.3D.grid.v2(
  n,
  X,
  f = 0.2,
  mst.grid = NULL,
  mode.edge.len = NULL,
  min.gSf = 1.5,
  min.min.gSf = 1,
  gRf = 2,
  min.gRf = 1,
  eSDf = 1.5,
  max.n = 120,
  verbose = FALSE
)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`X`	A set of points around which the grid is created if not NULL.
`f`	A fraction of x1, x2 and x3 range that the grid is extended to.
`mst.grid`	A matrix of 3D points that are subdivisions of edges of the mstree(X).
`mode.edge.len`	The mode of the edge lengths.
`min.gSf`	A scaling factor, indicating the minimal size of the grid around X, with respect to the size of X. That is the minimal size of the grid needs to be at least min.gSf*|X|, where |X| is the size of X. For example, with min.gSf set to 1.5, the mimal size of the grid will be, if possible, at least 1.5*|X|. For this to be possible the initial rectangular grid size has to be at least of that minimal size, but in practice much bigger. It overrides gRf if the size of the grid given by the value of gRf is not

`min.min.gSf`	The minimal allowabel value of min.gSf.
`gRf`	A scaling factor, such that points of the rectangular grid that are further than gRf*mode.edge.len away from the closest subdivision of the minimal spanning tree of X, mstree(X), are eliminated from the grid.
`min.gRf`	The minimal allowabel value of gRf.
`eSDf`	A scaling factor, such that edges of mstree(X) are subdivided is their length is greater than eSDf * mode(edge.len).
`max.n`	The maximal value of n that can be reached as n is adjusted to satisfy the size and radius factor conditions.
`verbose`	Set to TRUE to see info on what is being done.

Value

```
A data frame, grid, with n*n*n rows and 3 columns of the grid points in
      3D. If X is not NULL, grid contains only points nor more than eps away
         from the mstree(X).
```

`create.3D.rect.grid` *Creates a rectangular 3D grid*

Description

Creates a rectangular 3D grid

Usage

```
create.3D.rect.grid(
  n,
  x1.range,
  x2.range,
  x3.range,
  type = c("unif", "runif", "norm"),
  f = 0.2
)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`x1.range`	A range of x1 values - the first coordinate.
`x2.range`	A range of x2 values - the second coordinate.
`x3.range`	A range of x2 values - the second coordinate.
`type`	A type of distribution the points are sampled from.
`f`	A fraction of x1 and x2 range that the grid is extended to.

`create.3D.TN.grid` *Creates a tubular neighborhood 3D grid of X.*

Description

This is a complete rewrite of the predecessor routine create.3D.grid.v2() with gSf replaced by the length of the grid edge (all edges in these grids have the same length).

Usage

```
create.3D.TN.grid(
  X,
  mst.grid,
  dx,
  mode.edge.len,
  gRf,
  f = 0.05,
  verbose = FALSE
)
```

Arguments

`X`	A set of points around which the grid is created if not NULL.
`mst.grid`	A matrix of 3D points that are subdivisions of edges of the mstree(X).
`dx`	The length of the grid edge.
`mode.edge.len`	The mode of the edge lengths.#'
`gRf`	A scaling factor, such that points of the rectangular grid that are further than gRf*mode.edge.len away from the closest subdivision of the minimal spanning tree of X, mstree(X), are eliminated from the grid.
`f`	A fraction of x1, x2 and x3 range that the grid is extended to.
`verbose`	Set to TRUE to see info on what is being done.

Value

```
    A data frame, grid, with n*n*n rows and 3 columns of the grid points in
            3D. If X is not NULL, grid contains only points nor more than eps away
               from the mstree(X).
```

`create.3D.TN.grid.plus` *Creates a tubular neighborhood 3D grid of X returning a list with a grid around X and the distance to the boundary of the grid.*

Description

Creates a tubular neighborhood 3D grid of X returning a list with a grid around X and the distance to the boundary of the grid.

Usage

```
create.3D.TN.grid.plus(
  X,
  mst.grid,
  dx,
  mode.edge.len,
  gRf,
  f = 0.05,
  verbose = FALSE
)
```

Arguments

`X`	A set of points around which the grid is created if not NULL.
`mst.grid`	A matrix of 3D points that are subdivisions of edges of the mstree(X).
`dx`	The length of the grid edge.
`mode.edge.len`	The mode of the edge lengths.#'
`gRf`	A scaling factor, such that points of the rectangular grid that are further than gRf*mode.edge.len away from the closest subdivision of the minimal spanning tree of X, mstree(X), are eliminated from the grid.
`f`	A fraction of x1, x2 and x3 range that the grid is extended to.
`verbose`	Set to TRUE to see info on what is being done.

Value

```
    A data frame, grid, with n*n*n rows and 3 columns of the grid points in
            3D. If X is not NULL, grid contains only points nor more than eps away
               from the mstree(X).
```

`create.adaptive.tiled.X.grid.xD`

Create an Adaptive Tiled Grid Representation of X

Description

This function creates a grid representation of the input data X using an adaptive tiling approach. It's particularly effective for large or complex datasets.

Usage

```
create.adaptive.tiled.X.grid.xD(
  X,
  gSf,
  gRf,
  n = 5,
  n.itrs = 1,
  p.exp = 0.075,
  K = 10,
  wC = 0.1,
  wF = 1.25,
  X.vol.frac = NULL,
  vol.box.grid.size = 10^3,
  max.box.grid.size = 10^7,
  min.tree.obj = NULL,
  verbose = FALSE
)
```

Arguments

`X`	A numeric matrix where each row represents a point in the space.
`gSf`	Grid size factor. Determines the target number of grid points relative to the number of input points.
`gRf`	Grid radius factor. Controls the proximity of grid points to the minimal spanning tree of X.
`n`	Number of boxes for the initial tiling. Default is 5.
`n.itrs`	Number of iterations for the box tiling process. Default is 1.
`p.exp`	Expansion factor for the bounding box. Must be between 0 and 1. Default is 0.075.
`K`	Number of nearest neighbors to consider in the box tiling process. Default is 10.
`wC`	Multiplication factor for temporary grid creation. Must be positive. Default is 0.1.
`wF`	Factor for increasing grid width if necessary. Default is 1.25.
`X.vol.frac`	Pre-computed volume fraction of X, if available. Default is NULL.
`vol.box.grid.size`	Target size for the grid used in volume estimation. Default is 1000.
`max.box.grid.size`	Maximum allowed size for a single box grid. Default is 10^7.
`min.tree.obj`	Pre-computed minimal spanning tree object, if available. Default is NULL.
`verbose`	Logical; if TRUE, print progress information. Default is FALSE.

Value

A list of class "gridX" containing:

`X.grid`	The generated grid points
`d.grid`	Distance values for the grid points
`mst.grid`	Minimal spanning tree of X
`mode.edge.len`	Mode of the edge lengths in the minimal spanning tree
`L`, `R`	Lower and upper bounds of the bounding box
`dim`	Dimension of the space
`X`	Original input data
`X.vol.frac`	Estimated volume fraction of X
`X.vol`	Estimated volume of X
`gRf`, `gSf`, `p.exp`, `wC`	Input parameters

See Also

`create.X.grid.xD` for the original, non-tiled version

Examples

```
## Not run:
X <- matrix(rnorm(1000), ncol = 2)
grid_obj <- create.adaptive.tiled.X.grid.xD(X, gSf = 2, gRf = 1.5, verbose = TRUE)
plot(grid_obj$X.grid, col = "red", pch = 20)
points(X, col = "blue", pch = 1)

## End(Not run)
```

`create.basin.cx` *Create a Gradient Flow Basin Complex on a Weighted Graph*

Description

Constructs a comprehensive topological analysis of a scalar function defined on a graph through identification, clustering, and simplification of gradient flow basins. The function identifies local extrema (minima and maxima), constructs monotonic basins around them, clusters similar basins based on their overlap, and creates a cell complex representing their relationships.

Usage

```
create.basin.cx(
  adj.list,
  weight.list,
  y,
  basin.merge.overlap.thld = 0.1,
  min.asc.desc.cell.size.thld = 1,
  min.asc.asc.cell.size.thld = 1,
  min.desc.desc.cell.size.thld = 1,
  graph.params = list()
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`y`	Numeric vector of length n, the response values at each graph vertex.
`basin.merge.overlap.thld`	Numeric value between 0 and 1 specifying the threshold for basin overlap distance below which basins should be merged (default: 0.1). Lower values result in more conservative merging.
`min.asc.desc.cell.size.thld`	Minimum size threshold for ascending-descending cells (default: 1). Cells with fewer vertices are excluded.

`min.asc.asc.cell.size.thld`	Minimum size threshold for ascending-ascending cells (default: 1). Cells with fewer vertices are excluded.
`min.desc.desc.cell.size.thld`	Minimum size threshold for descending-descending cells (default: 1). Cells with fewer vertices are excluded.
`graph.params`	List of graph parameters.

Details

This function performs the following key operations:

1. Detects local extrema (minima and maxima) on the graph
2. Constructs monotonic basins around each extremum
3. Clusters similar basins based on their vertex overlap
4. Merges basins within clusters to simplify the analysis
5. Constructs a cell complex from intersections between basins

Each basin represents a region of the graph where the function values change monotonically (either increasing or decreasing) from the local extremum. Basins are labeled as "mx" for minima and "Mx" for maxima, where x is a sequential number.

The cell complex represents intersections between basins and provides insights into the topological structure of the function on the graph. Three types of cells are created:

- Ascending-descending cells: Intersections between minimum and maximum basins
- Ascending-ascending cells: Intersections between two minimum basins
- Descending-descending cells: Intersections between two maximum basins

Value

An object of class "basin_cx" containing:

basins A list with elements `ascending` (minima) and `descending` (maxima), each containing the merged basins following clustering.

local_extrema A data frame with vertex indices, types (maximum/minimum), labels, and function values for all extrema.

original_y The original input function values.

harmonic_predictions Numeric vector of repaired function values after simplification.

cluster_assignments A named vector showing which cluster each original basin was assigned to.

cluster_mappings A data frame showing the mapping between original basin labels and merged basin labels.

initial_basin_cx The original basin complex before clustering and merging.

cells A cell complex structure with ascending-descending, ascending-ascending, and descending-descending cells representing basin intersections.

basins_df A data frame summarizing all basins with properties like extremum vertex, function value, relative span, and size.

lmin_basins_df, lmax_basins_df Data frames containing only minima or maxima basins.

***_dist_mat, *_overlap_dists** Various distance and similarity matrices between basins.

See Also

plot.basin_cx

Examples

```
## Not run:
# Create a graph with adjacency list and weights
adj_list <- list(c(2,3), c(1,3,4), c(1,2,5), c(2,5), c(3,4))
weight_list <- list(c(1,2), c(1,1,3), c(2,1,2), c(3,1), c(2,1))

# Define a function on vertices
y <- c(2.5, 1.8, 3.2, 0.7, 2.1)

# Create basin complex
basin_cx <- create.basin.cx(adj_list, weight_list, y, basin.merge.overlap.thld = 0.15)

# Examine results
summary(basin_cx)

# Visualize original vs. simplified function
plot(basin_cx, type = "comparison")

# Extract vertices from a specific basin
m1_vertices <- get_basin_vertices(basin_cx, "m1")

## End(Not run)
```

---

create.bi.kNN.chain.graph

Creates a bi-k-NN chain graph

Description

This function constructs a bi-directional k-nearest neighbor chain graph. Each vertex is connected to up to k neighbors on both sides, forming a chain-like structure. The graph can be based on sorted x-coordinates if provided.

Usage

```
create.bi.kNN.chain.graph(n.vertices = 5, k = 1, x = NULL, y = NULL)
```

Arguments

n.vertices	An integer. The number of vertices in the graph.
k	An integer. The number of neighbors to connect on each side of a vertex.
x	An optional numeric vector of length n.vertices. If provided, vertices will be ordered based on these values.
y	An optional numeric vector of length n.vertices. If provided along with x, it will be sorted according to x.

Value

A list containing:

`adj.list`	A list representing the graph structure. Each element is a vector of indices of neighboring vertices.
`edge.lengths`	A list of edge lengths corresponding to the graph structure. If x is provided, these are based on differences in x values; otherwise, they are set to 1.
`x.sorted`	The sorted x vector if x was provided, otherwise NULL.
`y.sorted`	The y vector sorted according to x if both x and y were provided, otherwise NULL.

Examples

```
# Create a 5-chain graph with 10 vertices
graph <- create.bi.kNN.chain.graph(10, 5)

# Create a graph based on x-coordinates
x <- runif(10)
y <- sin(x)
graph.with.coords <- create.bi.kNN.chain.graph(10, 2, x, y)
```

---

create.bipartite.graph

Create a Bipartite Graph

Description

Generates a bipartite graph with two disjoint sets of nodes.

Usage

```
create.bipartite.graph(n1, n2)
```

Arguments

`n1`	An integer specifying the number of nodes in the first set.
`n2`	An integer specifying the number of nodes in the second set.

Details

In a bipartite graph, nodes are divided into two disjoint sets, and each edge connects a node in the first set to a node in the second set. There are no edges between nodes in the same set.

Value

A list representing the adjacency list of the bipartite graph. The first n1 elements correspond to nodes in the first set, and the next n2 elements correspond to nodes in the second set.

Examples

```
bipartite_graph <- create.bipartite.graph(3, 4)
```

`create.chain.graph` *Create a Chain Graph Structure*

Description

Generates a chain graph where each vertex is connected to its immediate neighbors, forming a linear chain structure. The graph can be constructed based on either the number of vertices or a set of ordered x-coordinates.

Usage

```
create.chain.graph(n.vertices = NULL, x = NULL, y = NULL)
```

Arguments

`n.vertices`	Integer specifying the number of vertices. Required if x is NULL.
`x`	Optional numeric vector. If provided, vertices will be ordered based on these values, and n.vertices will be set to length(x).
`y`	Optional numeric vector corresponding to x values. Must be the same length as x if provided.

Value

A list containing:

adj.list List of adjacent vertices for each vertex

edge.lengths List of edge lengths between adjacent vertices (1 if x not provided, otherwise based on x-coordinate differences)

x.sorted Sorted x values if x was provided, otherwise NULL

y.sorted y values sorted according to x order if both provided, otherwise NULL

Examples

```
# Create a simple 4-vertex chain
chain1 <- create.chain.graph(4)
```

`create.chain.graph.with.offset` *Creates a chain graph with n vertices*

Description

Creates a chain graph with n vertices

Usage

```
create.chain.graph.with.offset(n, offset = 0)
```

Arguments

`n`	The number of vertices in the graph
`offset`	An offset in indexing the vertices of the graph.

Value

A chain graph.

`create.circular.graph` *Creates a circular graph with n vertices*

Description

Creates a circular graph with n vertices

Usage

```
create.circular.graph(n)
```

Arguments

`n`	The number of vertices in the graph

Value

A circular graph.

`create.cmst.graph` *Construct a Minimal Spanning Tree (MST) Completion Graph*

Description

Constructs a completion of the Minimal Spanning Tree (MST) graph from a numeric data matrix. The graph is constructed in two steps: first, a MST is built connecting all data points using minimal total edge length; then, additional edges are added between any pair of points whose Euclidean distance lies below a quantile threshold based on the MST edge length distribution.

Usage

```
create.cmst.graph(
  X,
  q.thld = 0.9,
  pca.dim = 100,
  variance.explained = 0.99,
  verbose = TRUE
)
```

Arguments

`X`	A numeric matrix or data frame of shape $n \times d$, where each row represents a data point in d-dimensional space.
`q.thld`	Numeric scalar between 0 and 1 (exclusive). The quantile threshold for MST completion. Edges are added between points whose distance is below the q.thld-quantile of MST edge weights. Default is 0.9.
`pca.dim`	Positive integer or NULL. If provided and `ncol(X)` > `pca.dim`, dimensionality is reduced to this many principal components before graph construction. Must be less than min(n-1, p) where n is the number of observations and p is the number of variables. Default is 100.
`variance.explained`	Numeric between 0 and 1 (exclusive) or NULL. If provided, selects the minimal number of PCA components such that this proportion of total variance is retained (up to `pca.dim` components). Ignored if `pca.dim` is NULL. Default is 0.99.
`verbose`	Logical. If `TRUE`, prints progress messages during computation. Default is `TRUE`.

Details

The MST completion process enhances connectivity in the original MST by adding edges between vertices that are "close" according to the distance distribution in the MST. This can be useful for creating more robust graph representations of data that maintain local structure while avoiding long-range connections.

When PCA is applied (either through `pca.dim` or `variance.explained`), the data is first centered and projected onto the leading principal components before graph construction. This can significantly reduce computation time for high-dimensional data while preserving the most important variation.

Value

An object of class `mst_completion_graph`, which is a list containing:

`mst_adj_list` A list of length n, where element i contains the indices of vertices adjacent to vertex i in the MST.

`mst_weight_list` A list of length n, where element i contains the edge weights corresponding to the adjacent vertices in `mst_adj_list[[i]]`.

`cmst_adj_list` A list of length n containing adjacency information for the completed MST graph.

`cmst_weight_list` A list of length n containing edge weights for the completed MST graph.

`mst_edge_weights` Numeric vector containing all unique MST edge weights.

The returned object also has the following attributes:

`q_thld` The quantile threshold used for MST completion.

`pca` If PCA was applied, a list containing:

- `original_dim`: The original number of dimensions
- `n_components`: The number of PCA components used
- `variance_explained`: The proportion of variance explained
- `cumulative_variance`: Cumulative variance by component (if applicable)

`call` The matched function call.

References

Gower, J. C., & Ross, G. J. S. (1969). Minimum spanning trees and single linkage cluster analysis. Applied Statistics, 18(1), 54-64.

See Also

`prcomp` for principal component analysis, `pca.optimal.components` for variance-based component selection, `pca.project` for data projection

Examples

```
# Generate sample data
set.seed(123)
X <- matrix(rnorm(100 * 3), nrow = 100, ncol = 3)

# Create MST completion graph with default parameters
graph <- create.cmst.graph(X)

# Create graph with PCA dimensionality reduction
X_high <- matrix(rnorm(100 * 200), nrow = 100, ncol = 200)
graph_pca <- create.cmst.graph(X_high, pca.dim = 50, variance.explained = 0.95)

# Print summary
print(graph_pca)
summary(graph_pca)
```

`create.complete.graph` *Create a Complete Graph*

Description

Generates a complete graph where every node is connected to every other node.

Usage

```
create.complete.graph(n)
```

Arguments

`n`	An integer specifying the number of nodes in the graph.

Details

In a complete graph of n nodes, each node has n-1 connections (to all other nodes). The resulting graph is undirected.

Value

A list representing the adjacency list of the complete graph. Each element of the list corresponds to a node and contains a vector of all other nodes it's connected to.

Examples

```
complete_graph <- create.complete.graph(5)
```

`create.delta1.Delta1.df` *Create Summary Data Frame from First-Order Association Test Results*

Description

Creates a summary data frame containing key statistics from multiple first-order functional association test results.

Usage

```
create.delta1.Delta1.df(res.list, q.thld = 0.1)
```

Arguments

`res.list`	A named list of results from fassoc1.test() or fassoc.test(order=1).
`q.thld`	Numeric FDR threshold for significance. Default is 0.1.

Details

This function processes multiple test results to create a summary suitable for reporting. Results are sorted by RDM (Right Deviation from Mean) and FDR correction is applied to p-values.

Value

A list containing:

d1D1.df Full data frame with all results

sign.d1D1.df Filtered data frame with significant results only

The data frames contain columns: RDM (Right Deviation from Mean), D (ratio), p-val, q-val, delta1, and Delta1.

Examples

```
## Not run:
# Run multiple tests
results <- list()
for (i in 1:10) {
  x <- runif(100)
  y <- sin(2*pi*x) + rnorm(100, sd = 0.3)
  results\code{[[paste0("var", i)]}] <- fassoc.test(x, y, order = 1)
}

# Create summary
summary_df <- create.delta1.Delta1.df(results)
print(summary_df$sign.d1D1.df)

## End(Not run)
```

create.distance.plot *Create Distance Plot*

Description

Creates and saves a plot of edit distances with optional smoothing or regression results.

Usage

```
create.distance.plot(
  x,
  y,
  smooth = TRUE,
  smooth.method = "loess",
  mark.x = NULL,
  xlab = "Number of Nearest Neighbors (k)",
  ylab = "Edit Distance",
  main = NULL,
  width = 6,
  height = 6,
  ...
)
```

Arguments

`x`	Numeric vector of x-axis values (e.g., k values).
`y`	Numeric vector of y-axis values (e.g., edit distances).
`smooth`	Logical indicating whether to add a smoothing line (default: TRUE).
`smooth.method`	Character string specifying the smoothing method. Options include "loess" (default), "lm", "glm", "gam".
`mark.x`	Optional numeric value(s) to mark with vertical lines.
`xlab`	Character string for x-axis label (default: "Number of Nearest Neighbors (k)").
`ylab`	Character string for y-axis label (default: "Edit Distance").
`main`	Character string for plot title (default: NULL, no title).
`width`	Numeric value for plot width in inches when saving (default: 6).
`height`	Numeric value for plot height in inches when saving (default: 6).
`...`	Additional arguments passed to `plot()`.

Details

The function creates a scatter plot with optional smoothing line. If `mark.x` is provided, vertical dashed lines are added at those x-values.

Value

If `file` is specified, returns the absolute path to the saved file. Otherwise returns NULL invisibly.

Examples

```
## Not run:
# Create sample data
k.vals <- seq(5, 50, by = 5)
distances <- 100 / k.vals + rnorm(length(k.vals), sd = 2)

# Basic plot
create.distance.plot(k.vals, distances)

# Plot with marked minimum
min.k <- k.vals[which.min(distances)]
create.distance.plot(k.vals, distances, mark.x = min.k,
                    main = "Edit Distance vs k")

## End(Not run)
```

`create.ED.grid.2D` *Creates a equi-distant (all edges are of equal length) 2D grid using a C routine*

Description

Creates a equi-distant (all edges are of equal length) 2D grid using a C routine

Usage

```
create.ED.grid.2D(dx, x1.range, x2.range, f = 0.2, verbose = FALSE)
```

Arguments

`dx`	The length of each edge.
`x1.range`	A range of x1 values - the first coordinate.
`x2.range`	A range of x2 values - the second coordinate.
`f`	A fraction of x1 and x2 range that the grid is extended to.
`verbose`	If TRUE, some debugging messages will be printed.

`create.ED.grid.3D` *Creates a equi-distant (all edges are of equal length) 3D grid using a C routine*

Description

Creates a equi-distant (all edges are of equal length) 3D grid using a C routine

Usage

```
create.ED.grid.3D(dx, x1.range, x2.range, x3.range, f = 0.1, verbose = FALSE)
```

Arguments

dx	The length of the grid edge.
x1.range	A range of x1 values - the first coordinate.
x2.range	A range of x2 values - the second coordinate.
x3.range	A range of x3 values - the third coordinate.
f	A fraction of x1 and x2 range that the grid is extended to.
verbose	If TRUE, some debugging messages will be printed.

---

create.ED.grid.xD — *Create an Equidistant Grid in x-Dimensional Bounding Box (C Interface)*

Description

This function generates an equidistant grid within a specified x-dimensional bounding box. The grid consists of points that are uniformly spaced, and each edge of the grid has the same length.

Usage

```
create.ED.grid.xD(edge.length, lower.bounds, upper.bounds)
```

Arguments

edge.length	A positive numeric value representing the common length of each edge in the grid.
lower.bounds	A numeric vector containing the left (lower) end of the range for each dimension.
upper.bounds	A numeric vector containing the right (upper) end of the range for each dimension.

Value

A matrix representing the equidistant grid, where each row corresponds to a point in the grid, and each column corresponds to a dimension.

Examples

```
## Not run:
edge.length <- 1.0
lower.bounds <- c(0, 0, 0)
upper.bounds <- c(2, 3, 4)
grid <- create.ED.grid.xD(edge.length, lower.bounds, upper.bounds, round.up = TRUE)
print(grid)

## End(Not run)
```

`create.empty.graph` *Create an Empty Graph*

Description

Generates an empty graph with a specified number of nodes but no edges.

Usage

```
create.empty.graph(n)
```

Arguments

`n`	An integer specifying the number of nodes in the graph.

Details

An empty graph contains nodes but no edges connecting these nodes. This function is useful for initializing graphs or testing edge cases in graph algorithms.

Value

A list representing the adjacency list of the empty graph. Each element of the list is an empty vector, indicating no connections.

Examples

```
empty_graph <- create.empty.graph(5)
```

`create.ENPs.grid.2D` *Creates a equi-number (within each axis) 2D grid using a C routine*

Description

Creates a equi-number (within each axis) 2D grid using a C routine

Usage

```
create.ENPs.grid.2D(n, x1.range, x2.range, f = 0.2)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`x1.range`	A range of x1 values - the first coordinate.
`x2.range`	A range of x2 values - the second coordinate.
`f`	A fraction of x1 and x2 range that the grid is extended to.

`create.ENPs.grid.3D` *Creates a equi-numer (within each axis) 3D grid using a C routine*

Description

Creates a equi-numer (within each axis) 3D grid using a C routine

Usage

```
create.ENPs.grid.3D(n, x1.range, x2.range, x3.range, f = 0.2)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`x1.range`	A range of x1 values - the first coordinate.
`x2.range`	A range of x2 values - the second coordinate.
`x3.range`	A range of x3 values - the third coordinate.
`f`	A fraction of x1 and x2 range that the grid is extended to.

`create.final.grid` *Creates a Final Grid within Selected Boxes*

Description

This function creates a uniform grid within a set of selected sub-boxes, then filters the grid to remove points that are farther than a given distance `epsilon` from the original dataset `X`, and removes duplicated grid elements.

Usage

```
create.final.grid(boxes, X, w, epsilon)
```

Arguments

`boxes`	A list of selected sub-boxes, each represented as a list containing vectors `L` and `R` for left and right boundaries.
`X`	A matrix representing the original dataset for which the grid is to be created.
`w`	A numeric value representing the width of the grid.
`epsilon`	A numeric value representing the maximum distance from `X` that a grid point can be to be included in the final grid.

Value

A matrix representing the final filtered grid.

Examples

```
## Not run:
boxes <- create.ED.boxes(0.5, c(0,0), c(1,1))
X <- matrix(runif(20), ncol = 2)
w <- 0.1
epsilon <- 0.5
final_grid <- create.final.grid(boxes, X, w, epsilon)

## End(Not run)
```

`create.gflow.cx` *Create Graph Flow Complex with Harmonic Extension*

Description

Computes a graph flow complex by identifying local extrema and their neighborhoods, then applies harmonic extension to smooth spurious extrema while preserving significant topological features. The algorithm uses hop index thresholding to identify and smooth spurious extrema through various smoothing methods.

Usage

```
create.gflow.cx(
  adj.list,
  weight.list,
  y,
  hop.idx.thld = 5,
  smoother.type = 0,
  max.outer.iterations = 5,
  max.inner.iterations = 100,
  smoothing.tolerance = 1e-06,
  sigma = 1,
  process.in.order = TRUE,
  verbose = TRUE,
  detailed.recording = FALSE
)
```

Arguments

`adj.list`	A list where each element i contains the indices of vertices adjacent to vertex i. Indices should be 1-based (R convention). Must have the same length as `weight.list` and `y`.
`weight.list`	A list where each element i contains the weights of edges from vertex i to its neighbors. Must have the same structure and length as `adj.list`.
`y`	A numeric vector of function values defined on the graph vertices. Must have the same length as `adj.list`.
`hop.idx.thld`	Numeric scalar specifying the hop index threshold. Extrema with hop index less than or equal to this value are considered spurious and will be smoothed. Default is 5. Must be non-negative.
`smoother.type`	Integer specifying the smoothing algorithm:

- 0: Weighted Mean (default) - Simple weighted averaging
- 1: Harmonic Iterative - Iterative harmonic smoothing
- 2: Harmonic Eigen - Eigenfunction-based harmonic smoothing
- 3: Hybrid Biharmonic-Harmonic - Combined biharmonic and harmonic
- 4: Boundary Smoothed Harmonic - Harmonic with boundary smoothing

`max.outer.iterations`
: Maximum number of outer iterations for the smoothing process. Default is 5.

`max.inner.iterations`
: Maximum number of iterations for each individual smoothing operation. Default is 100.

`smoothing.tolerance`
: Numeric scalar for convergence tolerance in smoothing algorithms. Default is 1e-6.

`sigma`
: Numeric scalar controlling the width of the smoothing kernel. Default is 1.0. Larger values produce more smoothing.

`process.in.order`
: Logical; if TRUE (default), processes extrema in ascending order of hop index, smoothing the most spurious extrema first.

`verbose`
: Logical; if TRUE (default), prints progress information during the computation.

`detailed.recording`
: Logical; if TRUE, records detailed information about each smoothing step for later visualization. Default is FALSE. Note that this increases memory usage.

Details

The algorithm proceeds in several steps:

1. Identifies local minima and maxima in the function values on the graph
2. Computes hop neighborhoods around each extremum
3. Calculates hop indices measuring the significance of each extremum
4. Identifies spurious extrema (those with hop index <= threshold)
5. Applies harmonic extension to smooth out spurious extrema
6. Returns the smoothed function values and extrema information

The hop index measures how many graph hops are needed before the function value exceeds the extremum value, providing a scale-free measure of extremum significance. Spurious extrema typically have small hop indices and represent noise or insignificant features.

Value

An object of class `"gflow_cx"` containing:

harmonic_predictions Numeric vector of smoothed function values at each vertex after applying harmonic extension.

lmin_hop_nbhds List of hop neighborhoods for local minima. Each element contains vertex index, hop index, and neighborhood information.

lmax_hop_nbhds List of hop neighborhoods for local maxima. Each element contains vertex index, hop index, and neighborhood information.

extrema_df Data frame summarizing all extrema with columns: vertex, hop_idx, is_max, label, and fn_value.

extrema_df2 Alternative data frame format with spurious extrema information included.

smoothing_history (If detailed.recording=TRUE) List of records detailing each smoothing step for visualization.

The object also has attributes:

- `smoother.type`: The integer smoother type used
- `smoother.name`: Human-readable name of the smoother
- `hop.idx.threshold`: The hop index threshold used

See Also

`create.hop.nbhd.extrema.df` for extracting extrema information,

Examples

```
## Not run:
# Create a simple triangle graph
adj.list <- list(c(2,3), c(1,3), c(1,2))
weight.list <- list(c(1,1), c(1,1), c(1,1))

# Function with a spurious maximum at vertex 2
y <- c(0.5, 1.0, 0.7)

# Smooth out spurious extrema
result <- create.gflow.cx(
  adj.list, weight.list, y,
  hop.idx.thld = 1,
  smoother.type = 0,  # Weighted Mean
  verbose = FALSE
)

# Check smoothed values
print(result$harmonic_predictions)

# More complex example with detailed recording
# Create a larger graph (grid)
n <- 5
adj.list <- vector("list", n*n)
weight.list <- vector("list", n*n)

for(i in 1:n) {
  for(j in 1:n) {
    idx <- (i-1)*n + j
    neighbors <- c()
    weights <- c()

    # Add edges to grid neighbors
    if(i > 1) { neighbors <- c(neighbors, (i-2)*n + j); weights <- c(weights, 1) }
    if(i < n) { neighbors <- c(neighbors, i*n + j); weights <- c(weights, 1) }
    if(j > 1) { neighbors <- c(neighbors, (i-1)*n + j-1); weights <- c(weights, 1) }
    if(j < n) { neighbors <- c(neighbors, (i-1)*n + j+1); weights <- c(weights, 1) }

    adj.list[[idx]] <- neighbors
    weight.list[[idx]] <- weights
  }
}

# Create function with multiple extrema
y <- sin(seq(0, 2*pi, length.out = n*n)) + 0.1*rnorm(n*n)

# Apply smoothing with detailed recording
result <- create.gflow.cx(
  adj.list, weight.list, y,
  hop.idx.thld = 2,
  smoother.type = 2,  # Harmonic Eigen
  detailed.recording = TRUE,
  verbose = TRUE
)

# Examine extrema
print(result$extrema_df)

## End(Not run)
```

`create.grid` *Create Uniform 2D Grid*

Description

Creates a uniform 2D grid over `[0,1]^2` for function evaluation.

Usage

```
create.grid(axis.n.pts)
```

Arguments

`axis.n.pts`	Integer number of points along each axis

Value

List containing:

`x`	Numeric vector of x coordinates
`y`	Numeric vector of y coordinates
`grid`	Data frame with all (x,y) coordinate pairs

Examples

```
grid <- create.grid(50)
str(grid)
```

create.grid.graph *Create a Refined Graph with Uniformly Spaced Grid Vertices*

Description

Creates a refined version of an input graph by adding grid vertices (points) along its edges. The grid vertices are placed to maintain approximately uniform spacing throughout the graph structure. This function is particularly useful for tasks that require a denser sampling of points along the graph edges, such as graph-based interpolation or spatial analysis.

Usage

```
create.grid.graph(
  adj.list,
  weight.list,
  grid.size,
  start.vertex = 1L,
  snap.tolerance = 0.1
)
```

Arguments

`adj.list`	A list where each element `i` is an integer vector containing the indices of vertices adjacent to vertex `i`. Vertex indices must be 1-based (following R's convention). The graph structure must be undirected, meaning if vertex `j` appears in `adj.list[[i]]`, then vertex `i` must appear in `adj.list[[j]]`.
`weight.list`	A list matching the structure of `adj.list`, where each element contains the corresponding edge weights (typically distances or lengths). `weight.list[[i]][j]` should contain the weight of the edge between vertex `i` and vertex `adj.list[[i]][j]`. Weights must be positive numbers.
`grid.size`	A positive integer specifying the desired number of grid vertices to add. Must be at least 2. The actual number of added vertices may differ slightly from this target due to the distribution of edge lengths in the graph.
`start.vertex`	An integer specifying the starting vertex for graph traversal. Must be between 1 and the number of vertices in the input graph. Defaults to 1.
`snap.tolerance`	A numeric value between 0 and 0.5 controlling the snapping behavior when placing grid vertices near existing vertices. When a grid vertex would be placed within this fraction of an edge length from an existing vertex, it is merged with that vertex instead. Defaults to 0.1.

Details

The function performs a breadth-first traversal starting from `start.vertex` to determine the order in which edges are processed. Grid vertices are distributed across edges based on their relative lengths, with longer edges receiving more grid vertices. The algorithm ensures that the spacing between consecutive vertices (original or grid) along any edge is as uniform as possible.

The `snap.tolerance` parameter helps prevent the creation of vertices that are too close to existing ones, which can cause numerical issues in downstream analyses.

Value

A list with three components:

`adj_list` A list representing the adjacency structure of the refined graph, including both original and grid vertices. The structure follows the same format as the input `adj.list`.

`weight_list` A list containing edge weights for the refined graph, structured to match the new `adj.list`. Edge weights are adjusted to reflect the new distances between connected vertices.

`grid_vertices` An integer vector containing the 1-based indices of the newly added grid vertices in the refined graph.

See Also

`graph_from_adj_list` for converting adjacency lists to igraph objects

Examples

```
# Create a simple path graph with 3 vertices
adj <- list(c(2L), c(1L, 3L), c(2L))
weights <- list(c(1.0), c(1.0, 2.0), c(2.0))

# Add approximately 5 grid vertices
result <- create.grid.graph(adj, weights, grid.size = 5)

# Examine the results
cat("Original graph had", length(adj), "vertices\\n")
cat("Refined graph has", length(result$adj.list), "vertices\\n")
cat("Grid vertices added:", length(result$grid.vertices), "\\n")
cat("Indices of grid vertices:", result$grid.vertices, "\\n")

## Not run:
# Create a more complex graph (a cycle with varying edge weights)
n <- 6
adj <- lapply(1:n, function(i) c(ifelse(i == 1, n, i - 1),
                                 ifelse(i == n, 1, i + 1)))
weights <- lapply(1:n, function(i) runif(2, 0.5, 2.0))

# Refine with more grid vertices
result <- create.grid.graph(adj, weights, grid.size = 20)

g <- igraph::graph_from_adj_list(result$adj.list, mode = "undirected")
plot(g, vertex.color = ifelse(1:length(result$adj.list) %in%
                                result$grid.vertices, "red", "blue"))

## End(Not run)
```

`create.hHN.graph` *Create a k-hop Neighborhood (hHN) Graph*

Description

Generates a k-hop neighborhood graph from an input graph represented by an adjacency list and corresponding edge lengths. The k-hop neighborhood of a vertex includes all vertices reachable within at most k hops, with edges representing the shortest paths between vertices.

Usage

```
create.hHN.graph(graph, edge.lengths, h)
```

Arguments

`graph`	A list of numeric vectors representing the adjacency list of the input graph. Each element `graph[[i]]` contains the indices of vertices adjacent to vertex i. Note: Uses 1-based indexing as standard in R.
`edge.lengths`	A list of numeric vectors representing the edge weights of the input graph. `edge.lengths[[i]][j]` is the weight of the edge from vertex i to its j-th neighbor in `graph[[i]]`.
`h`	An integer specifying the maximum number of hops to consider for neighborhood connectivity. Must be greater than or equal to 1.

Value

A list containing two elements:

adj_list A list of integer vectors representing the adjacency list of the hHN graph. Indices use 1-based indexing.

dist_list A list of numeric vectors representing the edge weights (shortest path distances) of the hHN graph.

The time complexity is O(n * (m + n log n)), where n is the number of vertices and m is the number of edges in the original graph.

See Also

`bbmwd.over.hHN.graphs` for computing BBMWD over hHN graphs

Examples

```
# Create a simple 4-vertex graph
graph <- list(
  c(2, 3),       # Vertex 1 connects to vertices 2 and 3
  c(1, 3, 4),    # Vertex 2 connects to vertices 1, 3, and 4
  c(1, 2, 4),    # Vertex 3 connects to vertices 1, 2, and 4
  c(2, 3)        # Vertex 4 connects to vertices 2 and 3
)

edge.lengths <- list(
  c(1, 2),       # Edge weights from vertex 1
  c(1, 1, 3),    # Edge weights from vertex 2
  c(2, 1, 1),    # Edge weights from vertex 3
  c(3, 1)        # Edge weights from vertex 4
)

# Generate the 2-hop neighborhood graph
h <- 2
khn.graph <- create.hHN.graph(graph, edge.lengths, h)

# Access the results
khn.graph$adj_list   # Adjacency list of the hHN graph
khn.graph$dist_list  # Shortest path distances in the hHN graph
```

`create.hop.nbhd.extrema.df`
Create Data Frame of Extrema Information from Hop Neighborhoods

Description

Extracts and formats extrema information from hop neighborhood results into a convenient data frame format. This function works with results from `create.gflow.cx`.

Usage

```
create.hop.nbhd.extrema.df(
  result,
  include_spurious = TRUE,
  threshold = NULL,
  sort_by_value = FALSE,
  vertex_column_name = "vertex"
)
```

Arguments

`result`	An object containing hop neighborhood information, typically from `create.gflow.cx()`. Must contain components named `lmin_hop_nbhds` and `lmax_hop_nbhds`.
`include_spurious`	Logical; whether to include spurious extrema in the output. Default is TRUE. Spurious extrema are those with hop index less than or equal to the threshold.
`threshold`	Numeric scalar specifying the threshold for identifying spurious extrema. Extrema with hop_idx <= threshold are marked as spurious. If NULL (default), attempts to extract the threshold from the result object or uses 2.
`sort_by_value`	Logical; if TRUE, orders extrema by their function values (ascending for minima, descending for maxima). If FALSE (default), preserves the original order.
`vertex_column_name`	Character string specifying the name for the vertex column in the output data frame. Default is "vertex".

Details

Labels are assigned to extrema based on their function values:

- Minima are labeled "m1", "m2", etc., in ascending order of function value
- Maxima are labeled "M1", "M2", etc., in descending order of function value

This labeling scheme ensures that "m1" is the global minimum among detected minima, and "M1" is the global maximum among detected maxima.

Value

A data frame with the following columns:

vertex Integer vertex indices (1-based)

hop_idx Numeric hop index values for each extremum

is_max Integer indicator: 0 for minima, 1 for maxima

label Character labels for extrema (e.g., "m1", "m2" for minima, "M1", "M2" for maxima)

value Numeric function values at extrema (NA if not available)

spurious Logical indicating whether the extremum is spurious

The vertex column may be renamed based on `vertex_column_name`.

Examples

```
## Not run:
# Create example graph and compute hop neighborhoods
adj.list <- list(c(2,3), c(1,3,4), c(1,2,4), c(2,3))
weight.list <- list(c(1,1), c(1,1,1), c(1,1,1), c(1,1))
y <- c(0, 1, 0.5, 2)  # Minima at vertices 1,3 and maxima at vertices 2,4

# Using with create.gflow.cx
result <- create.gflow.cx(adj.list, weight.list, y, verbose = FALSE)
extrema_df <- create.hop.nbhd.extrema.df(result)
print(extrema_df)

# Exclude spurious extrema
significant_extrema <- create.hop.nbhd.extrema.df(
  result,
  include_spurious = FALSE,
  threshold = 2
)
print(significant_extrema)

## End(Not run)
```

`create.iknn.graphs` *Create Intersection k-Nearest Neighbor Graphs with Dual Pruning Methods*

Description

Computes a sequence of intersection-weighted k-nearest neighbor graphs for k in [kmin, kmax] with two different edge pruning methods: geometric pruning and intersection-size pruning. The function can optionally perform dimensionality reduction via PCA before graph construction.

Usage

```
create.iknn.graphs(
  X,
  kmin,
  kmax,
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  compute.full = TRUE,
  pca.dim = 100,
  variance.explained = 0.99,
  verbose = FALSE
)
```

Arguments

`X`	numeric matrix where rows represent observations and columns represent features. Cannot be a data frame.
`kmin`	integer, minimum number of nearest neighbors (>= 1)
`kmax`	integer, maximum number of nearest neighbors (> kmin)
`max.path.edge.ratio.deviation.thld`	numeric, threshold for geometric pruning based on path-to-edge ratio. If > 0, removes edges where the ratio of alternative path length to direct edge length minus 1.0 is less than this value. If <= 0, geometric pruning uses a default method. Must be in the interval [0, 0.2).
`path.edge.ratio.percentile`	numeric in $[0, 1]$, percentile threshold for edge lengths considered in geometric pruning. Only edges with length greater than this percentile are evaluated for path-ratio pruning.
`compute.full`	logical, if TRUE returns all pruned graphs, if FALSE returns only edge statistics
`pca.dim`	Maximum number of principal components to use if dimensionality reduction is applied (default: 100). Set to NULL to skip dimensionality reduction.
`variance.explained`	Percentage of variance to be explained by the principal components (default: 0.99). If this threshold can be met with fewer components than pca.dim, the smaller number will be used. Set to NULL to use exactly pca.dim components.
`verbose`	Logical. If TRUE, print progress messages and timing information. Default is FALSE.

Details

The function applies two different pruning methods to construct efficient graph representations:

1. Geometric pruning: Based on path-to-edge ratios
   - Removes edges where alternative paths exist with similar or better geometric properties
   - Controlled by max.path.edge.ratio.deviation.thld and path.edge.ratio.percentile parameters
2. Intersection-size pruning: Based on intersection sizes of k-NN sets
   - Removes edges where alternative paths exist with all edges having larger intersection sizes
   - Uses a fixed maximum alternative path length of 2

Note: The current implementation does not compute edge birth/death times. The birth_death_matrix and double_birth_death_matrix fields may be present but will be empty.

Value

A list of class "iknn_graphs" containing:

k_statistics Matrix with columns: k, number of edges in original graph, number of edges after geometric pruning, number of removed edges, edge reduction ratio, number of edges after intersection-size pruning, additional edges removed in intersection-size pruning, intersection-size edge reduction ratio

geom_pruned_graphs If compute_full=TRUE, list of geometrically pruned graphs for each k. Each graph contains adjacency lists and edge weights. If compute_full=FALSE, NULL

isize_pruned_graphs If compute_full=TRUE, list of intersection-size pruned graphs for each k. Each graph contains adjacency lists and edge weights. If compute_full=FALSE, NULL

edge_pruning_stats List of matrices, one for each k value, containing edge pruning statistics including edge lengths and path-to-edge length ratios

Examples

```
## Not run:
# Generate sample data
X <- matrix(rnorm(100 * 5), 100, 5)

# Basic usage
result <- create.iknn.graphs(
  X, kmin = 3, kmax = 10,
  compute.full = FALSE
)

# With custom pruning parameters
result <- create.iknn.graphs(
  X, kmin = 3, kmax = 10,
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  compute.full = TRUE,
  verbose = TRUE
)

# View statistics for each k
print(result$k_statistics)

## End(Not run)
```

---

create.intersection.graph

Create an Intersection Graph

Description

Constructs an intersection graph from an input graph based on neighborhood intersections. In the output graph, an edge exists between two nodes if their neighborhoods in the input graph have a significant intersection.

Usage

```
create.intersection.graph(adj.list, p.thld, n.itrs = 1)
```

Arguments

`adj.list`	A graph adjacency list of integer vectors or a matrix of integers. Each vector/row represents the neighbors of a node in the input graph. If using vectors, they should be sorted in ascending order.

`p.thld`	A numeric value between 0 and 1. Determines the threshold for considering an intersection significant. For nodes i and j, the threshold is calculated as: `thld = p.thld * min(length(adj.list[[i]]), length(adj.list[[j]]))`
`n.itrs`	Integer. Number of iterations to perform (default is 1).

Details

The function performs the following steps:

1. Validates input parameters.
2. Converts input to 0-based indexing for C++ processing.
3. Calls the C++ implementation to create the intersection graph.
4. Returns the result as a list of integer vectors.

For multiple iterations (n.itrs > 1), each iteration uses the result of the previous iteration as its input graph.

Value

A list of integer vectors representing the intersection graph. Each vector contains the indices of neighbors for the corresponding node in the new graph.

Examples

```
adj.list <- list(c(1,2), c(0,2,3), c(0,1), c(1))
result <- create.intersection.graph(adj.list, 0.5)
print(result)
```

`create.latex.table` *Create LaTeX Table from Matrix or Data Frame*

Description

Generates a LaTeX table from a matrix or data frame with proper formatting for inclusion in LaTeX documents.

Usage

```
create.latex.table(data, file, label = NA, caption = "")
```

Arguments

`data`	matrix or data.frame to convert to LaTeX format
`file`	character, path to the output LaTeX file
`label`	character or NA, LaTeX label for cross-referencing. If not NA, will be appended with ":tbl" suffix
`caption`	character, table caption (default: "")

Details

The function creates a properly formatted LaTeX table with:

- Centered alignment
- Column headers from the data's column names
- Row names as the first column
- Optional caption and label for cross-referencing
- Proper escaping of special LaTeX characters

Value

Invisibly returns the LaTeX content as a character vector

Examples

```
## Not run:
# Create example data
mat <- matrix(1:12, nrow = 3)
colnames(mat) <- paste0("Group", 1:4)
rownames(mat) <- paste0("Category", 1:3)

# Generate LaTeX table
create.latex.table(
  data = mat,
  file = "output.tex",
  label = "results",
  caption = "Example results table"
)

## End(Not run)
```

`create.lmin.lmax.contingency.table`
Create Contingency Table of Local Maxima and Minima Relationships

---

Description

Generates a contingency table showing the relationships between local maxima and minima in a Morse-Smale complex, ordered by marginal frequencies. Also provides a LaTeX caption describing the table contents.

Usage

```
create.lmin.lmax.contingency.table(MS.res, lmin.labels, lmax.labels)
```

Arguments

`MS.res`	List containing Morse-Smale complex results with component MS_cx
`lmin.labels`	Named character vector of labels for local minima
`lmax.labels`	Named character vector of labels for local maxima

Details

The function:

1. Maps critical points to their labels
2. Creates a contingency table of label combinations
3. Orders rows and columns by marginal frequencies
4. Generates an appropriate LaTeX caption

Value

A list with two components:

`contingency_table`	Table showing frequencies of local maxima-minima combinations
`caption`	LaTeX caption describing the contingency table

Examples

```
## Not run:
result <- create.lmin.lmax.contingency.table(MS.res, lmin.labels, lmax.labels)
print(result$contingency_table)
cat(result$caption)

## End(Not run)
```

`create.lmin.lmax.frequency.tables`
Create Frequency Tables and LaTeX Captions for Morse-Smale Complex Critical Points

---

Description

Generates detailed frequency tables and corresponding LaTeX captions for local maxima and minima in a Morse-Smale complex. The tables include basin of attraction sizes, relative conditional expectation statistics, and relationships between critical points.

Usage

```
create.lmin.lmax.frequency.tables(
  MS.res,
  lmin.labels,
  lmax.labels,
  rel.condEy,
  freq.thld = 100,
  outcome.name = "outcome"
)
```

Arguments

`MS.res`	List containing Morse-Smale complex results with component MS_cx
`lmin.labels`	Named character vector of labels for local minima
`lmax.labels`	Named character vector of labels for local maxima
`rel.condEy`	Numeric vector of relative conditional expectations
`freq.thld`	Frequency threshold for filtering (default: 100)
`outcome.name`	Character string describing the outcome variable (default: "outcome")

Details

For each critical point type, the function computes:

1. Basin of attraction sizes
2. Maximum and minimum relative conditional expectations within each basin
3. Range of relative conditional expectations (Delta)
4. Number of complementary critical points (minima for maxima and vice versa)

Value

A list with four components:

`lmax.freq.df`	Data frame containing local maxima statistics
`lmin.freq.df`	Data frame containing local minima statistics
`lmax.caption`	LaTeX caption for local maxima table
`lmin.caption`	LaTeX caption for local minima table

`create.lmin.lmax.label.indicators`
Create Label Tables and Indicator Vectors for Morse-Smale Complex Critical Points

---

Description

Creates label tables and indicator vectors for local minima and maxima in a Morse-Smale complex. The label tables map point IDs to their corresponding labels, while indicator vectors mark the presence/absence of critical points in the state space.

Usage

```
create.lmin.lmax.label.indicators(lmin.labels, lmax.labels, state.space)
```

Arguments

`lmin.labels`	Named character vector of labels for local minima
`lmax.labels`	Named character vector of labels for local maxima
`state.space`	State space matrix where rownames correspond to point IDs

Details

The function processes both local minima and maxima labels to create:

1. Label tables that map point IDs to their corresponding labels
2. Binary indicator vectors marking the presence (1) or absence (0) of critical points All vectors maintain the same length as the number of rows in state.space and use consistent naming.

Value

A list with four components:

`lmin.lab.tbl`	Named character vector mapping point IDs to local minima labels
`lmax.lab.tbl`	Named character vector mapping point IDs to local maxima labels
`lmin.ind`	Numeric vector indicating local minima (1) and non-minima (0)
`lmax.ind`	Numeric vector indicating local maxima (1) and non-maxima (0)

Examples

```
## Not run:
# Given lmin.labels, lmax.labels and state space matrix state.space
result <- create.lmin.lmax.label.indicators(lmin.labels, lmax.labels, state.space)
print(head(result$lmin.lab.tbl))
print(sum(result$lmin.ind)) # Number of local minima

## End(Not run)
```

`create.lmin.lmax.labels` *Create Labels for Local Maxima and Minima in Morse-Smale Complex*

Description

Creates unique labels for local maxima and minima in a Morse-Smale complex by concatenating two-letter shortcuts of the most abundant taxa. The function ensures uniqueness of labels by iteratively including additional taxa if needed.

Usage

```
create.lmin.lmax.labels(
  MS.res,
  state.space,
  taxonomy,
  freq.thld = 100,
  min.relAb.thld = 0.05
)
```

Arguments

`MS.res`	A list containing Morse-Smale complex results with component MS_cx
`state.space`	Matrix of ASV counts/abundances with samples as columns and ASVs as rows
`taxonomy`	Taxonomy information for ASVs
`freq.thld`	Threshold for frequency filtering (default: 100)
`min.relAb.thld`	Minimum relative abundance threshold for including taxa in labels (default: 0.05)

Details

The function processes both local maxima and minima, creating unique labels by:

1. Filtering points by frequency threshold
2. For each point, identifying taxa above the relative abundance threshold
3. Creating initial labels using two-letter shortcuts
4. Ensuring uniqueness by incorporating additional taxa if needed

Value

A list with four components:

`lmax.labels`	Named character vector of local maxima labels
`lmin.labels`	Named character vector of local minima labels
`lmax.profiles`	Named list of relative abundance profiles for local maxima, where names are point indices and each profile is a matrix with taxa in rows and two columns: species names and their relative abundances
`lmin.profiles`	Named list of relative abundance profiles for local minima, where names are point indices and each profile is a matrix with taxa in rows and two columns: species names and their relative abundances

Examples

```
## Not run:
labels <- create.lmin.lmax.labels(MS.res, state.space, taxonomy,
                                  freq.thld = 100, min.relAb.thld = 0.05)
print(labels$lmax.labels)
print(labels$lmin.labels)

## End(Not run)
```

`create.maximal.packing` *Create a Maximal Packing of Vertices in a Graph*

Description

Creates a maximal packing of vertices based on a specified grid size. The algorithm places vertices in the graph using a separation distance that produces a vertex set packing of size approximately equal to the given grid size. A vertex packing is a set of vertices where each pair is separated by at least a specified distance.

Usage

```
create.maximal.packing(
  adj.list,
  weight.list,
  grid.size,
  max.iterations = 20,
  precision = 0.1
)
```

Arguments

`adj.list`	A list where each element is a vector of adjacent vertex indices (1-based) for the corresponding vertex. Must represent an undirected graph.
`weight.list`	A list where each element is a vector of edge weights corresponding to the adjacencies in `adj.list`. All weights must be positive.
`grid.size`	A positive integer (>= 2) specifying the approximate separation distance between vertices in the packing.
`max.iterations`	Maximum number of iterations for the algorithm to converge. Default is 20.
`precision`	Precision threshold for convergence of the algorithm. Must be between 0 and 0.5. Default is 0.1.

Details

The function computes a maximal packing by iteratively selecting vertices that are approximately `grid.size` apart from each other. The process starts from one of the graph's diameter endpoints (determined automatically) and continues until no more vertices can be added to the packing without violating the distance constraint.

```
The returned \code{graph_diameter} represents the longest shortest path
in the graph, which provides insight into the graph's overall structure
and extent.

The \code{max_packing_radius} value indicates the minimum distance that
separates any two vertices in the packing. This value is determined
through binary search to achieve a packing of approximately the
requested size.

The function validates input parameters and ensures that the graph
structure is properly specified before computing the packing. The graph
must be undirected (symmetric adjacency) and connected.
```

Value

A list with class "maximal_packing" containing five components:

`adj_list`	A list of adjacency vectors for each vertex in the resulting grid graph
`weight_list`	A list of weight vectors corresponding to each adjacency
`grid_vertices`	An integer vector of vertex indices that form the maximal packing
`graph_diameter`	A numeric value representing the maximum shortest path distance between any two vertices in the graph

`max_packing_radius`	A numeric value representing the optimal radius used for the final packing, which is the minimum guaranteed distance between any two vertices in the packing

See Also

`validate.maximal.packing`, `verify.maximal.packing`

Examples

```
## Not run:
# Create a simple triangle graph
adj.list <- list(c(2, 3), c(1, 3), c(1, 2))
weight.list <- list(c(1, 1), c(1, 1), c(1, 1))

# Create grid graph with grid size 2
result <- create.maximal.packing(adj.list, weight.list, grid.size = 2)

# View the vertices in the maximal packing
print(result$grid_vertices)

# Access the graph diameter and packing radius
cat("Graph diameter:", result$graph_diameter, "\n")
cat("Packing radius:", result$max_packing_radius, "\n")

## End(Not run)
```

`create.mknn.graph` *Compute a Mutual k-Nearest Neighbor Graph with Weights*

Description

Creates an undirected graph where two points are connected by an edge if and only if they are present in each other's k-nearest neighbor (kNN) lists. The weight of each edge represents the distance between the connected vertices.

Usage

```
create.mknn.graph(X, k)
```

Arguments

`X`	A numeric matrix or data frame where each row represents a data point and each column represents a feature/dimension.
`k`	A positive integer specifying the number of nearest neighbors to consider. Must be at least 2.

Details

The mutual k-nearest neighbor graph is a symmetric graph where an edge between vertices i and j exists if and only if:

- j is among the k nearest neighbors of i, AND
- i is among the k nearest neighbors of j

This mutual relationship ensures that the resulting graph is undirected and typically sparser than a standard k-nearest neighbor graph.

Value

A list with class "mknn_graph" containing:

adj_list A list where element i contains the indices of vertices connected to vertex i by an edge.

weight_list A list where element i contains the distances between vertex i and its connected neighbors (in the same order as `adj_list[[i]]`).

n_vertices The number of vertices in the graph.

n_edges The total number of edges in the graph.

k The k value used to construct the graph.

See Also

`create.mknn.graphs` for creating multiple graphs with different k values, `summary.mknn_graphs` for summarizing graph properties

Examples

```
## Not run:
# Generate sample 2D data
set.seed(123)
X <- matrix(rnorm(100 * 2), ncol = 2)

# Create mutual 5-NN graph
graph <- create.mknn.graph(X, k = 5)

# Print basic statistics
cat("Number of vertices:", graph$n_vertices, "\n")
cat("Number of edges:", graph$n_edges, "\n")

# Examine connections for first vertex
cat("Vertex 1 is connected to:", graph$adj_list[[1]], "\n")
cat("With distances:", round(graph$weight_list[[1]], 3), "\n")

## End(Not run)
```

`create.mknn.graphs` *Create Multiple Mutual kNN Graphs with Geometric Pruning*

Description

Constructs a series of mutual k-nearest neighbor (MkNN) graphs across a range of k values, with optional geometric pruning to remove redundant edges based on alternative path analysis.

Usage

```
create.mknn.graphs(
  X,
  kmin,
  kmax,
  max.path.edge.ratio.thld = 1.2,
  path.edge.ratio.percentile = 0.5,
  compute.full = FALSE,
  pca.dim = 100,
  variance.explained = 0.99,
  verbose = FALSE
)
```

Arguments

`X`	A numeric matrix where rows represent data points and columns represent features. Data frames will be converted to matrices.
`kmin`	Minimum k value to consider (positive integer, at least 2).
`kmax`	Maximum k value to consider (must be >= kmin).
`max.path.edge.ratio.thld`	Maximum acceptable ratio of alternative path length to direct edge length for pruning. Edges with alternative paths having ratio <= this value will be pruned. Default is 1.2. Set to 0 or negative to disable pruning.
`path.edge.ratio.percentile`	Percentile threshold (0.0-1.0) for edge lengths to consider for pruning. Only edges with length greater than this percentile are evaluated. Default is 0.5.
`compute.full`	Logical; if TRUE, returns complete graph structures for each k value. If FALSE, returns only summary statistics. Default is FALSE.
`pca.dim`	Maximum number of principal components to use if dimensionality reduction is applied. Default is 100. Set to NULL to skip dimensionality reduction.
`variance.explained`	Target percentage of variance to be explained by the principal components (0-1). Default is 0.99. If this can be achieved with fewer than pca.dim components, the smaller number is used. Set to NULL to use exactly pca.dim components.
`verbose`	Logical; if TRUE, displays progress messages. Default is FALSE.

Details

This function performs the following steps for each k value:

1. **Mutual kNN Graph Construction**: Creates a graph where edges exist only between mutually nearest neighbors.
2. **Geometric Pruning** (optional): Removes edges where alternative paths exist with acceptable length ratios. An edge (i,j) is pruned if there exists a path i->k->j such that (d(i,k) + d(k,j)) / d(i,j) <= threshold.
3. **Dimensionality Reduction** (optional): If the data has more than pca.dim dimensions, PCA is applied before graph construction to improve computational efficiency and reduce noise.

The geometric pruning step helps create sparser graphs by removing edges that can be well-approximated by two-hop paths, which can improve graph quality for clustering and visualization tasks.

Value

An object of class "mknn_graphs" containing:

k_statistics A data frame with columns: k (k value), n_edges (edges before pruning), n_edges_pruned (edges after pruning), n_removed (number of removed edges), reduction_ratio (proportion of edges removed).

pruned_graphs If compute.full=TRUE, a named list of pruned graphs for each k value. NULL otherwise.

edge_pruning_stats A list of pruning statistics for each k value, containing edge lengths and path ratios.

The returned object also has the following attributes:

- kmin, kmax: The k range used
- max.path.edge.ratio.thld: The pruning threshold used
- path.edge.ratio.percentile: The percentile threshold used
- pca: If PCA was performed, contains dimensionality reduction details

See Also

`create.mknn.graph` for creating a single MkNN graph, `summary.mknn_graphs` for summarizing the results

Examples

```
## Not run:
# Generate sample data with clusters
set.seed(123)
n <- 150
X <- rbind(
  matrix(rnorm(n * 2, mean = 0), ncol = 2),
  matrix(rnorm(n * 2, mean = 5), ncol = 2),
  matrix(rnorm(n * 2, mean = c(2.5, 5)), ncol = 2)
)

# Create MkNN graphs for k from 5 to 15 with pruning
result <- create.mknn.graphs(
  X,
  kmin = 5,
  kmax = 15,
  max.path.edge.ratio.thld = 1.2,
  path.edge.ratio.percentile = 0.5,
  compute.full = TRUE,
  verbose = TRUE
)

# Examine the summary statistics
print(result$k_statistics)

# Get the graph for k=10
k10_graph <- result$pruned_graphs[["10"]]
cat("Graph with k=10 has", k10_graph$n_edges, "edges\n")

## End(Not run)

# High-dimensional example with PCA
## Not run:
# Generate high-dimensional data
set.seed(456)
X_highdim <- matrix(rnorm(200 * 1000), nrow = 200, ncol = 1000)

# Apply PCA before graph construction
result_pca <- create.mknn.graphs(
  X_highdim,
  kmin = 10,
  kmax = 20,
  pca.dim = 50,
  variance.explained = 0.95,
  verbose = TRUE
)

# Check PCA information
pca_info <- attr(result_pca, "pca")
cat("Used", pca_info$n_components, "components explaining",
    round(pca_info$variance_explained * 100, 2), "% of variance\n")

## End(Not run)
```

`create.morse.smale.complex`

Create Morse-Smale Complex Object

Description

Constructs complete Morse-Smale complex from grid function values.

Usage

```
create.morse.smale.complex(f.grid)
```

Arguments

`f.grid`	Matrix of function values

Value

S3 object of class "morse.smale.complex" with components:

`cell_lookup`	List mapping cell IDs to point coordinates
`cell_summary`	Data frame summarizing each cell
`local_maxima`	Matrix of local maximum coordinates
`local_minima`	Matrix of local minimum coordinates
`find_cell`	Function to find cell ID for a grid point
`get_neighboring_cells`	Function to find neighboring cells

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
complex <- create.morse.smale.complex(f.grid)
str(complex)

## End(Not run)
```

`create.nerve.complex` *Creates a Nerve Complex Associated With kNN Covering of a Dataset*

Description

This function creates a nerve complex from a set of points in R^n based on their k-nearest neighbor covering.

Usage

```
create.nerve.complex(coords, k, max.dim = 2)
```

Arguments

`coords`	Matrix of point coordinates, where rows are points and columns are dimensions
`k`	Number of nearest neighbors to use for the covering
`max.dim`	Maximum dimension of simplices to compute

Value

A nerve complex object

`create.path.graph` *Create a Path Graph with Limited Hop Distance*

Description

Constructs a path graph from an input graph by finding all reachable vertices within a specified maximum number of hops. For each pair of vertices, stores the shortest path information along with edge weights and hop counts.

Usage

```
create.path.graph(graph, edge.lengths, h)
```

Arguments

`graph`	A list where each element `i` is a numeric vector containing the indices of vertices adjacent to vertex `i`. Indices should be 1-based (following R convention). Each vertex `i` should have an entry in the list, even if it has no neighbors (empty vector).
`edge.lengths`	A list with the same structure as `graph` where each element `i` contains the weights/lengths of the edges specified in `graph[[i]]`. Must contain numeric values > 0. The order of weights must correspond to the order of vertices in `graph[[i]]`.
`h`	Integer >= 1 specifying the maximum number of hops allowed in the path graph. A hop is a single edge traversal.

Details

The function uses Dijkstra's algorithm to compute shortest paths from each vertex to all reachable vertices within the specified hop limit. The algorithm is implemented in C++ for performance. The resulting path graph can be used for various network analyses where connectivity within a limited number of steps is relevant.

Value

An S3 object of class `"path.graph"` containing:

adj.list A list where `adj.list[[i]]` contains all vertices reachable from vertex `i` within `h` hops

edge.length.list A list where `edge.length.list[[i]]` contains the shortest path lengths to each vertex in `adj.list[[i]]`

hop.list A list where `hop.list[[i]]` contains the number of hops required to reach each vertex in `adj.list[[i]]`

shortest.paths A list with components:

- **i** Source vertex indices
- **j** Target vertex indices
- **paths** List of vertex sequences for each shortest path

Note

- All vertex indices in input and output are 1-based (R convention)
- The function internally converts indices to 0-based for C++ processing
- Empty adjacency lists are allowed for isolated vertices
- The graph can be directed or undirected
- Self-loops are not allowed

See Also

shortest_paths for alternative shortest path calculations, create.path.graph.series for creating multiple path graphs efficiently

Examples

```
## Not run:
# Create a simple graph with 3 vertices
graph <- list(
  c(2),    # Vertex 1 connected to 2
  c(1, 3), # Vertex 2 connected to 1 and 3
  c(2)     # Vertex 3 connected to 2
)

# Define edge lengths
edge.lengths <- list(
  c(1.0),      # Length of edge 1->2
  c(1.0, 2.0), # Lengths of edges 2->1 and 2->3
  c(2.0)       # Length of edge 3->2
)

# Create path graph with maximum 2 hops
pg <- create.path.graph(graph, edge.lengths, h = 2)

# Print the path graph
print(pg)

# Get shortest path between vertices 1 and 3
path <- get.shortest.path(pg, 1, 3)

## End(Not run)
```

---

create.path.graph.series

Create a Series of Path Graphs with Different Hop Limits

Description

Efficiently constructs multiple path graphs from an input graph, each with a different maximum hop limit. This is more efficient than creating path graphs separately as it reuses computations.

Usage

```
create.path.graph.series(graph, edge.lengths, h.values)
```

Arguments

`graph`	A list where each element `i` is a numeric vector containing the indices of vertices adjacent to vertex `i`. Indices should be 1-based.
`edge.lengths`	A list with the same structure as `graph` where each element `i` contains the weights/lengths of the edges specified in `graph[[i]]`.
`h.values`	Integer vector specifying the hop limits for which to compute path graphs. All values must be >= 1. Values will be sorted in ascending order internally.

Details

The function computes path graphs for multiple hop limits efficiently by building upon results from smaller hop values. This is particularly useful for analyzing how network connectivity changes with increasing hop limits.

Value

A list of path.graph objects, one for each value in `h.values`, with class "path.graph.series". Each path graph has an additional attribute "h" storing its hop limit.

See Also

`create.path.graph`, `compare.paths`

Examples

```
## Not run:
# Create a simple graph
graph <- list(
  c(2),    # Vertex 1 connected to 2
  c(1, 3), # Vertex 2 connected to 1 and 3
  c(2)     # Vertex 3 connected to 2
)
edge.lengths <- list(c(1.0), c(1.0, 2.0), c(2.0))

# Create path graphs for hop limits 1, 2, and 3
pgs <- create.path.graph.series(graph, edge.lengths, h.values = c(1, 2, 3))

# Access individual path graphs
pg.h2 <- pgs[[2]]  # Path graph with h=2

## End(Not run)
```

`create.plm.graph` *Create a Path Length Matrix Graph Structure*

Description

Creates a PLM (Path Length Matrix) graph structure from an adjacency list representation of an undirected graph with edge weights. This structure is optimized for computing path-based statistics and contains precomputed paths of specified length.

Usage

```
create.plm.graph(graph, edge.lengths, h)
```

Arguments

`graph`	A list where each element `i` is a numeric vector containing the indices of vertices adjacent to vertex `i`. Vertex indices should be 1-based. For undirected graphs, edges should be present in both directions.
`edge.lengths`	A list of the same structure as `graph` where each element contains the weights/lengths of the corresponding edges in `graph`.
`h`	An odd positive integer specifying the maximum path length (in hops) to be precomputed. Common values are 3, 5, or 7.

Details

The PLM (Path Length Matrix) structure is designed for efficient computation of path-based network statistics. It precomputes all paths up to length `h` and stores them in a format that allows rapid queries. The restriction to odd values of `h` is often used in certain network analysis contexts where paths of odd length have special significance.

For undirected graphs, ensure that if edge (i,j) exists, then edge (j,i) also exists in the input with the same weight.

Value

An S3 object of class "path.graph.plm" containing:

adj_list Adjacency list representation of the graph

edge_length_list List of edge weights

hop_list List of hop counts for paths

shortest_paths Precomputed paths information

vertex_paths Information about paths containing each vertex

h The hop limit used

Note

This function is memory-intensive for large graphs or large values of `h`, as it stores all paths up to length `h`.

See Also

`create.path.graph` for standard path graph creation

Examples

```
## Not run:
# Create a simple path graph with 3 vertices: 1 -- 2 -- 3
graph <- list(c(2), c(1, 3), c(2))
edge.lengths <- list(c(1), c(1, 1), c(1))

# Create PLM graph with paths up to length 3
plm_graph <- create.plm.graph(graph, edge.lengths, h = 3)
```

```
print(plm_graph)

## End(Not run)
```

---

create.pruned.isize.list

Create Pruned Intersection Size List

Description

This function generates a list of intersection sizes for the edges of a pruned adjacency list. It takes the original adjacency list, intersection size list, and pruned adjacency list as inputs, and returns a new list containing intersection sizes corresponding to the pruned graph structure.

Usage

```
create.pruned.isize.list(adj.list, isize.list, pruned.adj.list)
```

Arguments

`adj.list`	A list where each element represents a node and contains indices of its neighbors in the original graph.
`isize.list`	A list of the same length as adj.list, where each element contains intersection sizes corresponding to the edges in adj.list.
`pruned.adj.list`	A list representing the pruned graph, where each element contains indices of neighbors after removing redundant edges.

Details

The function iterates through each node in the pruned adjacency list. For each node, it identifies the neighbors in the pruned graph and retrieves their corresponding intersection sizes from the original isize.list. The resulting pruned.isize.list maintains the structure of pruned.adj.list but contains intersection sizes instead of neighbor indices.

Value

A list of the same length as pruned.adj.list, where each element contains intersection sizes corresponding to the edges in the pruned graph.

Note

- The function assumes that all nodes present in pruned.adj.list are also present in adj.list and isize.list.
- If a neighbor in pruned.adj.list is not found in the original adj.list (which should not happen under normal circumstances), the corresponding intersection size will be set to NA.

Examples

```
## Not run:
# Example usage:
adj.list <- list(c(2,3,4), c(1,3), c(1,2,4), c(1,3))
isize.list <- list(c(2,1,3), c(2,1), c(1,1,2), c(3,2))
pruned.adj.list <- list(c(2,4), c(1), c(4), c(1,3))
pruned.isize.list <- create.pruned.isize.list(adj.list, isize.list, pruned.adj.list)
print(pruned.isize.list)

## End(Not run)
```

`create.random.graph` *Create a Random Graph*

Description

Creates a random graph with the specified number of vertices and average degree (neighbors per vertex).

Usage

```
create.random.graph(n_vertices, avg_degree, connected = TRUE)
```

Arguments

`n_vertices`	Number of vertices in the graph
`avg_degree`	Average number of neighbors per vertex
`connected`	Logical; if TRUE, ensure the graph is connected

Value

A list with adjacency and weight lists

Examples

```
graph <- create.random.graph(100, 4)
```

`create.single.iknn.graph` *Create a Single Intersection-weighted k-Nearest Neighbors Graph*

Description

Creates and prunes an intersection-weighted k-nearest neighbors (IWD-kNN) graph from input data. The graph is constructed based on feature space distances and intersection sizes between neighbor sets.

Usage

```
create.single.iknn.graph(
  X,
  k,
  pruning.thld = 0.1,
  compute.full = FALSE,
  pca.dim = 100,
  variance.explained = 0.99,
  verbose = TRUE
)
```

Arguments

`X`	A numeric matrix where rows represent data points and columns represent features. Data frames will be coerced to matrices.
`k`	Integer scalar. The number of nearest neighbors to consider for each point. Must be positive and less than the number of data points.
`pruning.thld`	numeric, controlling intensity of the geometric edge pruning. Edge weight relative deviation is computed as rel_dev = (w_ij + w_jk) / w_ik - 1.0; Geometric pruning is performed on all edges with rel_dev < pruning_thld
`compute.full`	Logical scalar. If TRUE, computes additional graph components and metrics. If FALSE, computes only essential components. Default value: FALSE.
`pca.dim`	Maximum number of principal components to use if dimensionality reduction is applied (default: 100). Set to NULL to skip dimensionality reduction.
`variance.explained`	Percentage of variance to be explained by the principal components (default: 0.99). If this threshold can be met with fewer components than pca.dim, the smaller number will be used. Set to NULL to use exactly pca.dim components.
`verbose`	Logical. If TRUE, print progress messages. Default is TRUE.

Value

An object of class "IkNN" (inheriting from "list") containing:

pruned_adj_list Adjacency lists after pruning (1-based indices)

pruned_weight_list Distances for edges in pruned graph

n_edges Total number of edges in original graph

n_edges_in_pruned_graph Number of edges after pruning

n_removed_edges Number of edges removed by pruning

edge_reduction_ratio Proportion of edges removed

call The matched function call

k Number of nearest neighbors used

n Number of data points

d Number of features

If compute.full = TRUE, additional components include:

adj_list Original adjacency lists (1-based indices)

isize_list Intersection sizes for original edges

weight_list Distances for original edges

conn_comps Connected components identification

connected_components Alternative format of connected components

Examples

```
# Create sample data
set.seed(123)
X <- matrix(rnorm(100), ncol = 2)
result <- create.single.iknn.graph(X, k = 3)
summary(result)
```

`create.star.graph` *Create a star graph by joining chain graphs*

Description

This function creates a star graph by joining chain graphs of specified sizes to a central vertex. The resulting star graph consists of a central vertex connected to multiple chains of vertices.

Usage

```
create.star.graph(sizes)
```

Arguments

`sizes`	A vector of positive integers specifying the sizes of the chain graphs to be joined to the central vertex. Each size represents the number of vertices in a chain graph.

Value

A list representing the adjacency list of the created star graph.

Examples

```
star_graph <- create.star.graph(c(3, 4, 2))
print(star_graph)
```

create.subgraph — *Create a subgraph from a given graph*

Description

This function creates a subgraph from a given graph based on specified indices or IDs.

Usage

```
create.subgraph(
  S.graph,
  id.indices = NULL,
  ids = NULL,
  S = NULL,
  use.sequential.indices = FALSE
)
```

Arguments

`S.graph`	A list containing the original graph structure with adjacency and distance lists.
`id.indices`	A vector of indices to include in the subgraph. Default is NULL.
`ids`	A vector of IDs to include in the subgraph. Default is NULL.
`S`	A data frame or matrix where rownames correspond to node IDs. Required if `ids` is provided. Default is NULL.
`use.sequential.indices`	Logical. If TRUE, renumber the indices in the subgraph to be 1:length(id.indices). Default is FALSE.

Details

The function can create a subgraph based on either `id.indices` or `ids`. If `ids` is provided, `S` must also be provided to map the IDs to indices. The `use.sequential.indices` parameter allows for renumbering the indices in the subgraph to be sequential, which can be useful for certain applications.

Value

A list containing the subgraph structure with adjacency and distance lists.

Examples

```
X <- runif.sphere(20, 2)
graph <- create.single.iknn.graph(X, k = 3, compute.full = TRUE, verbose = FALSE)
graph$dist_list <- graph$weight_list
graph$weight_list <- NULL
subgraph <- create.subgraph(graph, id.indices = c(1:10))
subgraph_sequential <- create.subgraph(graph, id.indices = c(1:10, 15:16),
                                       use.sequential.indices = TRUE)
```

create.taxonomic.labels *Create Taxonomic Labels for Vertices*

Description

Creates unique shortcut labels for vertices based on their taxonomic profiles. The function generates two-letter shortcuts from species names and combines them to create concise labels that represent the most abundant taxa in each vertex.

Usage

```
create.taxonomic.labels(vertices, state.space, taxonomy, min.relAb.thld = 0.05)
```

Arguments

`vertices`	A numeric vector of vertex indices to be labeled.
`state.space`	A matrix or data frame containing the state space data, where rows correspond to different states/samples.
`taxonomy`	A data structure containing taxonomic information used by the prof.fn function to generate taxonomic profiles.
`min.relAb.thld`	Numeric value (default: 0.05). Minimum relative abundance threshold. Taxa with abundance below this threshold will not be included in the label.

Details

The function first creates taxonomic profiles for each vertex using the external prof.fn function. It then generates labels based on the most abundant taxa above the specified threshold. If multiple vertices get the same label, the function ensures uniqueness by adding additional taxa information. The labels consist of two-letter shortcuts for each taxon name, created by taking the first letter of the first two parts of the underscore-separated name.

Value

A list with two components:

`labels`	A named character vector containing the generated labels, with names corresponding to vertex indices.
`profiles`	A list of data frames containing the taxonomic profiles for each vertex.

Note

This function requires an external prof.fn function that is not defined here. The prof.fn function should take an ID, state.space, and taxonomy as arguments and return a data frame with taxa names in the first column and their abundances in the second column.

`create.threshold.distance.graph`
Create Threshold Distance Graph from Distance Matrix

Description

This function constructs a graph where vertices are connected by an edge if and only if their distance is less than a specified threshold. The weight of each edge is the corresponding distance.

Usage

```
create.threshold.distance.graph(dist.matrix, threshold, include.names = TRUE)
```

Arguments

`dist.matrix`	A symmetric distance matrix where rows and columns correspond to vertices
`threshold`	A numeric threshold value; vertices i and j are connected if dist(i,j) < threshold
`include.names`	Logical; whether to include vertex names in the output (default: TRUE)

Value

A list with two components:

`adj_list`	A list of integer vectors. Each vector contains the indices of vertices adjacent to the corresponding vertex.
`weight_list`	A list of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj_list.

Examples

```
# Example distance matrix
dist <- matrix(c(
  0.0000000, 0.02834008, 0.05050505, 0.12500000, 0.1086957,
  0.02834008, 0.00000000, 0.88888889, 0.54166667, 1.0000000,
  0.05050505, 0.88888889, 0.00000000, 0.04166667, 0.1086957,
  0.12500000, 0.54166667, 0.04166667, 0.00000000, 1.0000000,
  0.10869565, 1.00000000, 0.10869565, 1.00000000, 0.0000000
), nrow=5, byrow=TRUE)
rownames(dist) <- colnames(dist) <- c("M1", "M2", "M3", "M4", "M5")

# Force symmetry by averaging with transpose
dist <- (dist + t(dist)) / 2

# Create graph with threshold 0.15
graph <- create.threshold.distance.graph(dist, 0.15)
```

create.tubular.nbhd.of.a.geodesic

Creates a tubular neighborhood of a geodesic

Description

Creates a tubular neighborhood of a geodesic

Usage

```
create.tubular.nbhd.of.a.geodesic(
  smoothed.geodesic,
  S,
  d.thld = "auto",
  dist.to.geodesic.q = 0.25
)
```

Arguments

`smoothed.geodesic`	A matrix of smoothed geodesic points in S.
`S`	A 3d model of a state space.
`d.thld`	A distance to the geodesic threshold. Set to 'auto' for automatic determination of the threshold.
`dist.to.geodesic.q`	The quantile of the distances to the geodesic to which the density function of the distances will be restricted to.

Details

This function creates a tubular neighborhood around a geodesic path by identifying all points in the state space S that are within a specified distance threshold of the geodesic. When d.thld is set to "auto", the threshold is determined based on the distribution of distances to the geodesic.

Value

A list containing:

`nn.i`	A vector of indices indicating the nearest point on the geodesic for each point in S.
`dist.to.geodesic`	A vector of distances from each point in S to the geodesic.
`strand.i`	Indices of points in S that are within the tubular neighborhood.
`strand`	The subset of S containing only points within the tubular neighborhood.
`nn.geodesic.vertex.i`	Indices of the nearest geodesic vertices for points in the strand.
`dist.along.strand.geodesic`	Normalized distances along the geodesic for points in the strand.
`d.thld`	The distance threshold used to define the tubular neighborhood.

create.vertex.label.indicators

Create Label Tables and Indicator Vectors for Vertices

Description

Creates a label table and indicator vector for specified vertices in the state space. The label table maps point IDs to their corresponding labels, while the indicator vector marks the presence/absence of vertices in the state space.

Usage

```
create.vertex.label.indicators(vertex.labels, state.space)
```

Arguments

`vertex.labels`	Named character vector of labels for vertices
`state.space`	State space matrix where rownames correspond to point IDs

Details

The function processes vertex labels to create:

1. A label table that maps point IDs to their corresponding labels
2. A binary indicator vector marking the presence (1) or absence (0) of vertices The vector maintains the same length as the number of rows in state.space and uses consistent naming.

Value

A list with two components:

`lab.tbl`	Named character vector mapping point IDs to vertex labels
`ind`	Numeric vector indicating vertices (1) and non-vertices (0)

Examples

```
## Not run:
# Given vertex.labels and state space matrix state.space
result <- create.vertex.label.indicators(vertex.labels, state.space)
print(head(result$lab.tbl))
print(sum(result$ind)) # Number of vertices

## End(Not run)
```

create.vertex.labels *Create Labels for Vertices in State Space*

Description

Creates unique labels for specified vertices in state space by concatenating two-letter shortcuts of the most abundant taxa. The function ensures uniqueness of labels by iteratively including additional taxa if needed.

Usage

```
create.vertex.labels(
  vertices,
  state.space,
  taxonomy = NULL,
  min.relAb.thld = 0.05,
  profile.length = 5
)
```

Arguments

`vertices`	Vector of indices corresponding to rows in state.space
`state.space`	Matrix of ASV counts/abundances. When taxonomy is NULL, column names should contain species names (e.g., "Genus_species")
`taxonomy`	Taxonomy information for ASVs (default: NULL). If NULL, species names are taken directly from state.space column names
`min.relAb.thld`	Minimum relative abundance threshold for including taxa in labels (default: 0.05)
`profile.length`	Integer specifying the number of top species to keep in each profile (default: 5)

Details

The function processes the specified vertices, creating unique labels by:

1. For each vertex, identifying taxa above the relative abundance threshold
2. Creating initial labels using two-letter shortcuts
3. Ensuring uniqueness by incorporating additional taxa if needed

When taxonomy is NULL, the function uses column names of state.space directly as species names. The profile.length parameter controls how many species are kept in each profile, keeping the most abundant ones.

Value

A list with two components:

`labels`	Named character vector of vertex labels
`profiles`	Named list of relative abundance profiles for vertices, where names are point indices and each profile is a matrix containing the top profile.length species

Examples

```
## Not run:
# Keep only top 5 species in profiles
result <- create.vertex.labels(c(1,3,5), state.space, taxonomy,
                              min.relAb.thld = 0.05, profile.length = 5)

## End(Not run)
```

`create.X.grid` *Creates a grid around a state space.*

Description

This function generates a uniform grid in a tubular neighborhood of a state space.

Usage

```
create.X.grid(X, gSf, gRf, min.K = 10, med.dK.divf = 5, max.dx.C = 1)
```

Arguments

`X`	A state space for which the grid is to be created.
`gSf`	A grid size scaling factor.
`gRf`	A grid radius scaling factor. Points on the rectangular grid that are farther than gRf*mode.edge.len from the closest subdivision of the minimal spanning tree of X, mstree(X), are eliminated from the grid.
`min.K`	The minimum number of nearest neighbors ('x' NNs) that must be present in each window.
`med.dK.divf`	A division factor for the lower bound of bandwidth (bw) and dx. It is equal to med.dK/med.dK.divf, where `med.dK=median(dK)` and `dK=nn.dist[,min.K]`.
`max.dx.C`	A division factor for estimating max.dx, defined as the length of the diagonal in the X enclosing box divided by max.dx.C.

Value

A list with the following components:

- `X.grid`: A uniform grid in a tubular neighborhood of X.
- `d.grid`: The distance to the boundary of the grid.
- `mst.grid`: A uniform grid over the edges of the minimum spanning tree of X.
- `opt.dx`: The optimal value of the dx parameter for given gSf and gRf values.
- `mode.edge.len`: The mode of the lengths of the edges of the minimum spanning tree of X.

Examples

```
## Not run:
# Let X be a low-dimensional model of a state space.
res <- create.X.grid(X, gSf=5, gRf=5, min.K=10, med.dK.divf=5, max.dx.C=1)
str(res)

## End(Not run)
```

create.X.grid.xD *Creates a Uniform Grid in a Tubular Neighborhood of a State Space*

Description

This function generates a uniform grid in a tubular neighborhood of a given state space X, considering specified parameters for grid size, radius scaling, and other factors.

Usage

```
create.X.grid.xD(
  X,
  gSf,
  gRf,
  p.exp = 0.05,
  wC = 0.1,
  wF = 1.25,
  X.vol.frac = NULL,
  max.tmp.grid.size = 10^7,
  min.tree.obj = NULL,
  n.threads = 1,
  verbose = FALSE
)
```

Arguments

`X`	A numeric matrix representing the state space for which the grid is to be created.
`gSf`	A numeric value representing the grid size scaling factor, such that the grid size $n(X.grid) = gSf * n(X)$, where $n(X)$ is the number of rows of X.
`gRf`	A numeric value representing the grid radius scaling factor. Points on the rectangular grid that are farther than $gRf * mode.edge.len$ from the closest subdivision of the minimal spanning tree of X, $mstree(X)$, are eliminated from the grid.
`p.exp`	A numeric value representing the expansion factor of the bounding box. Must be between 0 and 1. Default is 0.05.
`wC`	A numeric value representing a multiplication factor controlling the precision of the estimation of the volume of X. Must be a positive number. Default is 0.1.
`wF`	A numeric value representing a weighting factor. Default is 1.25.
`X.vol.frac`	The volume fraction of X in its bounding box.
`max.tmp.grid.size`	Maximum size of temporary grid. Default is 10^7.
`min.tree.obj`	A minimal spanning tree object. This parameter is useful when running this function with different values of gSf and gRf parameters and so only the first call may have min.tree.obj set to NULL and all remaining may use a minimal tree object from the first call.
`n.threads`	The number of threads to use in the corresponding C function call.
`verbose`	A logical value. Set to TRUE to see information on what is being done. Default is FALSE.

Value

A list with class "gridX" and the following components:

- X.grid: A numeric matrix representing a uniform grid in a tubular neighborhood of X.
- d.grid: A numeric vector representing the distance to the boundary of the grid.
- mst.grid: A numeric matrix representing a uniform grid over the edges of the minimum spanning tree of X.
- mode.edge.len: A numeric value representing the mode of the lengths of the edges of the minimum spanning tree of X.
- L: A numeric vector representing the left end of the range for each dimension.
- R: A numeric vector representing the right end of the range for each dimension.
- dim: An integer representing the number of dimensions.
- X: A numeric matrix representing the original state space.
- X.vol.frac: A numeric value representing the fraction of the volume of X in the bounding box.
- X.vol: A numeric value representing the estimated volume of X.
- gRf: A numeric value representing the grid radius scaling factor.
- gSf: A numeric value representing the grid size scaling factor.
- p.exp: A numeric value representing the expansion factor of the bounding box.
- wC: A numeric value representing the multiplication factor controlling the precision of the volume estimation.

Examples

```
## Not run:
# Let X be a low-dimensional model of a state space.
res <- create.X.grid.xD(X, gSf=5, gRf=5, p.exp=0.05, wC=0.1)
str(res)

## End(Not run)
```

---

critical.points.plot *Plot Critical Points*

Description

Plots critical points (maxima, minima, saddles) from continuous analysis.

Usage

```
critical.points.plot(
  critical_points,
  xlim = c(0, 1),
  ylim = c(0, 1),
  main = "Critical Points Analysis",
  add = FALSE,
  ...
)
```

Arguments

`critical_points`	List with maxima, minima, and saddles matrices
`xlim`	X-axis limits
`ylim`	Y-axis limits
`main`	Plot title
`add`	Logical, whether to add to existing plot
`...`	Additional arguments passed to plot

Value

Invisible NULL

Examples

```
# Create a function with known critical points
f <- function(x, y) x^2 - y^2  # Saddle at origin
gradient <- function(x, y) c(2*x, -2*y)
grid <- create.grid(30)
critical <- find.critical.points.continuous(gradient, grid)
critical.points.plot(critical)
```

`cross.prod` *Computes a cross product between two 3D vectors.*

Description

This routine calculates the cross product between two vectors in `R^3`.

Usage

```
cross.prod(x, y)
```

Arguments

`x`	A 3D vector.
`y`	A 3D vector.

Value

The cross product of x and y.

Examples

```
## Not run:
x <- c(0,1,3)
y <- c(2,3,4)
z <- cross.prod(x, y)

## End(Not run)
```

`cv.imputation` *Cross-Validation Imputation on Graphs*

Description

Performs cross-validation imputation on graph-structured data using various methods. This function supports both binary and continuous data, and offers different imputation strategies including local mean thresholding, neighborhood matching, and iterative approaches.

Usage

```
cv.imputation(
  test.set,
  graph,
  edge.lengths,
  y,
  y.binary,
  imputation.method = 1,
  max.iterations = 10,
  convergence.threshold = 1e-06,
  apply.binary.threshold = TRUE,
  binary.threshold = 0.5,
  kernel = "Epanechnikov",
  dist.normalization.factor = 1.01
)
```

Arguments

`test.set`	A numeric vector of indices for the test set vertices (1-based).
`graph`	A list of numeric vectors representing the graph structure. Each element of the list corresponds to a vertex and contains the indices of its neighbors.
`edge.lengths`	A list of edge lengths. The structure should match that of `graph`.
`y`	A numeric vector of original vertex values.
`y.binary`	A logical value indicating whether the data is binary (TRUE) or continuous (FALSE).
`imputation.method`	A string or integer specifying the imputation method. Options are: - 0 or "local_mean_threshold": Uses the mean of y computed over the training vertices (default). - 1 or "neighborhood_matching": Uses a matching method based on local neighborhood statistics. - 2 or "iterative_neighborhood_matching": Uses an iterative version of the neighborhood matching method. - 3 or "supplied_threshold": Uses a user-supplied threshold value. - 4 or "global_mean_threshold": Uses the global mean of y across all vertices.
`max.iterations`	An integer specifying the maximum number of iterations for iterative methods.
`convergence.threshold`	A numeric value specifying the convergence threshold for iterative methods.
`apply.binary.threshold`	A logical value indicating whether to apply binary thresholding to the results.

`binary.threshold`	A numeric value between 0 and 1 specifying the threshold for binary classification.
`kernel`	A character string specifying the kernel function for distance weighting. Options are "Box", "Triangular", "Epanechnikov", or "Gaussian".
`dist.normalization.factor`	A numeric value greater than 1 used for normalizing distances.

Details

This function serves as an R interface to a C++ implementation of graph-based imputation methods. It handles various input checks and data type conversions before calling the underlying C++ function.

The choice of imputation method, kernel function, and other parameters allows for flexibility in addressing different types of graph-structured data and imputation scenarios.

Value

A numeric vector containing the imputed values for the test set vertices.

Note

- The function assumes that the graph structure is consistent and that all vertex indices in test.set and training.set are valid.
- For binary data (y.binary = TRUE), the imputed values will be either 0 or 1.
- For continuous data, the imputed values may fall outside the range of the original y values, depending on the chosen method.

See Also

For more details on graph-based imputation methods, see the documentation of the underlying C++ implementation.

Examples

```
## Not run:
# Example with a small graph
graph <- list(c(2,3), c(1,3), c(1,2,4), c(3))
y <- c(1, 0, 1, 0)
test.set <- c(2, 4)
result <- cv.imputation(test.set, graph, y = y, y.binary = TRUE,
                        imputation.method = "LOCAL_MEAN_THRESHOLD")
print(result)

## End(Not run)
```

deg0.loo.llm.1D — *Degree 0 case of Leave-One-Out (LOO) cross-validation of 1D local linear models*

Description

Degree 0 case of Leave-One-Out (LOO) cross-validation of 1D local linear models

Usage

```
deg0.loo.llm.1D(nx, nn.i, nn.w, nn.y)
```

Arguments

nx	The number of elements of x.
nn.i	A matrix of indices of x NN's of xgrid.
nn.w	A matrix of NN weights.
nn.y	The values of y over nn.i.

Value

predictions

deg0.lowess.graph.smoothing — *Iterative Degree 0 LOWESS Graph Smoothing*

Description

Apply iterative degree 0 LOWESS smoothing to a graph and its associated data matrix. At each iteration, conditional expectations of features are estimated using locally weighted averages, and a new graph is constructed from the smoothed data.

Usage

```
deg0.lowess.graph.smoothing(
  adj.list,
  weight.list,
  X,
  max.iterations = 10,
  convergence.threshold = 1e-04,
  convergence.type = 1,
  k = 10,
  pruning.thld = 0.1,
  n.bws = 10,
  log.grid = TRUE,
  min.bw.factor = 0.05,
  max.bw.factor = 0.5,
  dist.normalization.factor = 1,
  kernel.type = 1,
  n.folds = 5,
  use.uniform.weights = FALSE,
  outlier.thld = 0.1,
  with.bw.predictions = FALSE,
  switch.to.residuals.after = NULL,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list[[i]]` contains the indices of vertices adjacent to vertex i.
`weight.list`	A list of numeric vectors with edge weights corresponding to adjacencies. Each element `weight.list[[i]][j]` is the weight of the edge from vertex i to `adj.list[[i]][j]`.
`X`	A numeric matrix where rows are samples and columns are features
`max.iterations`	Maximum number of iterations to perform
`convergence.threshold`	Threshold for convergence
`convergence.type`	Type of convergence criteria: 1 = maximum absolute difference 2 = mean absolute difference 3 = maximum relative change
`k`	Number of nearest neighbors for kNN graph construction
`pruning.thld`	Threshold for pruning edges in graph construction
`n.bws`	Number of candidate bandwidths for LOWESS
`log.grid`	Logical, whether to use logarithmic spacing for bandwidth grid
`min.bw.factor`	Factor for minimum bandwidth (multiplied by graph diameter)
`max.bw.factor`	Factor for maximum bandwidth (multiplied by graph diameter)
`dist.normalization.factor`	Factor for normalizing distances in kernel weight calculation
`kernel.type`	Type of kernel function (1 = Gaussian, 2 = Exponential, etc.)
`n.folds`	Number of cross-validation folds for bandwidth selection
`use.uniform.weights`	Logical, whether to use uniform weights instead of kernel weights
`outlier.thld`	Maximum proportion of vertices that can be identified as outliers in a single iteration before terminating the smoothing process. When the graph has multiple connected components, vertices not in the largest component are considered outliers. If the proportion of outliers exceeds this threshold, the algorithm stops to prevent excessive removal of data points. Must be between 0 and 0.5. Default is 0.1 (10% of vertices).
`with.bw.predictions`	Logical, whether to compute predictions for all bandwidths
`switch.to.residuals.after`	Number of iterations to perform direct smoothing before switching to residual smoothing (boosting mode). Default is max.iterations (never switch). Set to 0 to use residual smoothing from the start.
`verbose`	Logical, whether to print progress information

Value

A list containing:

`smoothed.graphs`	List of smoothed graphs at each iteration
`smoothed.X`	List of smoothed data matrices at each iteration
`convergence.metrics`	Numeric vector of convergence metrics at each iteration
`iterations.performed`	Number of iterations actually performed
`used.boosting`	Logical, whether boosting (residual smoothing) was used
`vertex.mappings`	List of vertex mappings at each iteration
`outlier.indices`	Integer vector of indices of identified outlier vertices

Examples

```
## Not run:
# Create a graph and data matrix
graph <- create.iknn.graph(X, k = 10, pruning.thld = 0.1)

# Apply degree 0 LOWESS graph smoothing - traditional approach
result1 <- deg0.lowess.graph.smoothing(
  adj.list = graph$adj_list,
  weight.list = graph$weight_list,
  X = X,
  max.iterations = 10,
  convergence.threshold = 1e-4,
  convergence.type = 1,  # MAX.ABSOLUTE.DIFF
  k = 10,
  pruning.thld = 0.1,
  n.bws = 10,
  n.folds = 5,
  verbose = TRUE
)

# Apply degree 0 LOWESS graph smoothing with boosting
result2 <- deg0.lowess.graph.smoothing(
  adj.list = graph$adj_list,
  weight.list = graph$weight_list,
  X = X,
  max.iterations = 10,
  convergence.threshold = 1e-4,
  convergence.type = 1,  # MAX.ABSOLUTE.DIFF
  k = 10,
  pruning.thld = 0.1,
  n.bws = 10,
  n.folds = 5,
  switch.to.residuals.after = 2,  # Switch to boosting after 2 iterations
  verbose = TRUE
)

# Access final smoothed data matrix
```

```
X.smoothed <- result2$smoothed.X[[length(result2$smoothed.X)]]

# Plot convergence metrics for both approaches
plot(result1$convergence.metrics, type = "b", col = "blue",
     xlab = "Iteration", ylab = "Convergence Metric")
lines(result2$convergence.metrics, type = "b", col = "red")
legend("topright", legend = c("Traditional", "Boosting"),
       col = c("blue", "red"), lty = 1)

## End(Not run)
```

`delta.indices` *Calculate Higher-Order Functional Association Indices*

Description

Computes Delta and delta functional association indices up to order k from conditional mean estimates.

Usage

```
delta.indices(Eyg, k = 10)
```

Arguments

`Eyg`	Numeric vector of conditional mean estimates E_x(y) over a uniform grid.
`k`	Integer specifying the highest order of indices to compute. Default is 10. Must be positive.

Details

For each order i from 1 to k:

- `Delta[i]` represents the total signed change at order i
- `delta[i]` represents the total absolute change at order i

Higher orders capture increasingly fine-scale variation in the functional relationship.

Value

A list containing:

Delta Numeric vector of Delta indices from order 1 to k

delta Numeric vector of delta indices from order 1 to k

Examples

```
## Not run:
# Generate example conditional mean curve
x <- seq(0, 1, length.out = 100)
Eyg <- sin(4*pi*x) + 0.5*x

# Calculate indices up to order 5
indices <- delta.indices(Eyg, k = 5)
print(indices$Delta)
print(indices$delta)

# Plot delta values by order
plot(1:5, indices$delta, type = "b",
     xlab = "Order", ylab = "delta",
     main = "Functional Association by Order")

## End(Not run)
```

`derivative.second.order.method` *Second-order accurate method of derivative of a function estimate*

Description

Computes the derivative using the second-order accurate finite difference method:

$$f'(t_i) \approx \frac{-f(t_{i+2}) + 8f(t_{i+1}) - 8f(t_{i-1}) + f(t_{i-2})}{12\Delta t}$$

Usage

```
derivative.second.order.method(y, dx)
```

Arguments

`y`	A numeric vector of response values defined over a uniform grid.
`dx`	The distance between consecutive points of the grid over which y is defined.

Details

It is assumed that y is defined over a uniform grid. The method uses:

- Forward difference for the first point
- Central difference for the second and second-to-last points
- Backward difference for the last point
- Five-point stencil for interior points

Value

A numeric vector of the same length as y containing the derivative estimates.

Examples

```
## Not run:
# Example with a quadratic function
x <- seq(0, 2*pi, length.out = 100)
y <- sin(x)
dx <- x[2] - x[1]
dy <- derivative.second.order.method(y, dx)

# Compare with analytical derivative
dy_true <- cos(x)
plot(x, dy, type = "l", col = "blue", main = "Numerical vs Analytical Derivative")
lines(x, dy_true, col = "red", lty = 2)
legend("topright", c("Numerical", "Analytical"), col = c("blue", "red"), lty = c(1, 2))

## End(Not run)
```

`destination.minimum.identify`

Identify Destination Minimum

Description

Identifies which minimum a gradient descent trajectory converges to by finding the closest known minimum to the trajectory endpoint. This function is useful for classifying trajectories in Morse-Smale cell construction.

Usage

```
destination.minimum.identify(trajectory, minima_coordinates, tolerance = 0.05)
```

Arguments

`trajectory`	Matrix of trajectory points where each row is an (x, y) coordinate
`minima_coordinates`	Matrix of known minima locations where each row is an (x, y) coordinate
`tolerance`	Maximum distance between trajectory endpoint and minimum to consider a match

Value

Integer index of the destination minimum (row number in minima_coordinates), or NA if no minimum is within tolerance

`detect.local.extrema` *Detect Local Extrema in a Graph*

Description

Identifies local maxima or minima in a graph based on vertex function values. A vertex is considered a local extremum if it has the highest (for maxima) or lowest (for minima) function value within a neighborhood of specified radius, and the neighborhood contains at least a minimum number of vertices.

Usage

```
detect.local.extrema(
  adj.list,
  weight.list,
  y,
  max.radius,
  min.neighborhood.size,
  detect.maxima = TRUE,
  custom.prefix = NULL
)
```

Arguments

`adj.list`	A list where each element contains integer indices of vertices adjacent to the corresponding vertex. Must have length equal to the number of vertices in the graph.
`weight.list`	A list where each element contains numeric weights of edges from the corresponding vertex. `weight.list[[i]][j]` is the weight of the edge from vertex `i` to vertex `adj.list[[i]][j]`.
`y`	A numeric vector of function values at each vertex. Must have the same length as `adj.list`.
`max.radius`	Positive numeric value specifying the maximum radius for neighborhood search.
`min.neighborhood.size`	Positive integer specifying the minimum number of vertices required in a neighborhood for a vertex to be considered an extremum.
`detect.maxima`	Logical; if `TRUE` (default), detect local maxima; if `FALSE`, detect local minima.
`custom.prefix`	Character string to use as prefix for extrema labels. If `NULL` (default), uses "M" for maxima and "m" for minima.

Details

The algorithm uses a graph-based approach to identify local extrema by examining neighborhoods defined by graph distance. For each vertex, it searches within increasing radii up to `max.radius` to find a neighborhood where the vertex has the extreme value among all vertices in that neighborhood.

The implementation uses a C++ backend for computational efficiency, particularly beneficial for large graphs.

Value

An object of class "local_extrema", which is a list containing:

vertices Integer vector of vertex indices identified as extrema

values Numeric vector of function values at the extrema

radii Numeric vector of neighborhood radii where extremum property holds

neighborhood_sizes Integer vector of the number of vertices in each extremum's neighborhood

is_maxima Logical vector indicating whether each extremum is a maximum (TRUE) or minimum (FALSE)

type Character vector with values "Maximum" or "Minimum" for each extremum

labels Character vector of labels for each extremum (e.g., "M1", "M2" for maxima)

See Also

summary.local_extrema for summarizing results, plot.local_extrema for visualization

Examples

```
# Create a simple chain graph
adj.list <- list(c(2), c(1,3), c(2,4), c(3,5), c(4))
weight.list <- list(c(1), c(1,1), c(1,1), c(1,1), c(1))
y <- c(1, 3, 2, 5, 1)  # Function values with peaks at vertices 2 and 4

# Detect maxima
maxima <- detect.local.extrema(adj.list, weight.list, y,
                               max.radius = 2,
                               min.neighborhood.size = 2)
print(maxima$vertices)  # Should identify vertices 2 and 4

# Detect minima
minima <- detect.local.extrema(adj.list, weight.list, y,
                               max.radius = 2,
                               min.neighborhood.size = 2,
                               detect.maxima = FALSE)
print(minima$vertices)  # Should identify vertices 1, 3, and 5
```

---

deviance.mabilog *Compute Deviance for Mabilog Model*

Description

Computes the deviance of the fitted model

Usage

```
## S3 method for class 'mabilog'
deviance(object, ...)
```

Arguments

`object`	A 'mabilog' object
`...`	Additional arguments (currently unused)

Value

Deviance value

`disk.to.sphere` *Maps points from the interior of a unit disk onto the unit sphere of the same dimension as the disk*

Description

This function transforms points from the interior of an n-dimensional unit disk (points with L2 norm < 1) onto the surface of an (n+1)-dimensional unit sphere using a stereographic-like projection. Points at the center of the disk map to the "north pole" of the sphere.

Usage

```
disk.to.sphere(X)
```

Arguments

`X`	A matrix or data frame specifying a set of points within the unit disk, where each row represents a point and each column represents a dimension. All points must have L2 norm strictly less than 1.

Details

The mapping uses the following transformation:

- For a point p with radius $r = ||p||$, the angle $\phi = \pi r$
- The sphere coordinates are: $(\sin(\phi)p/r, \cos(\phi))$
- Points at the origin (r = 0) map to (0, 0, ..., 0, 1)

Value

A numeric matrix with nrow(X) rows and ncol(X)+1 columns, where each row represents a point on the unit sphere in (n+1)-dimensional space. The returned points all have L2 norm equal to 1.

Examples

```
# Example 1: Map 2D disk points to 3D sphere
X <- matrix(c(0.5, 0.3, -0.2, 0.4, 0, 0), nrow = 3, ncol = 2, byrow = TRUE)
sphere_points <- disk.to.sphere(X)
# Verify all points are on unit sphere
apply(sphere_points, 1, function(x) sqrt(sum(x^2)))

# Example 2: Center point maps to north pole
origin <- matrix(c(0, 0), nrow = 1)
disk.to.sphere(origin)  # Returns (0, 0, 1)
```

`dist.to.knn` *Producess k-NN distance and index matrices associated with a distance matrix.*

Description

Producess k-NN distance and index matrices associated with a distance matrix.

Usage

```
dist.to.knn(d, k)
```

Arguments

d	A distance matrix.
k	The number of nearest neighbors.

Details

For each point (row) in the distance matrix, this function identifies the k nearest neighbors and their corresponding distances. The neighbors are ordered from closest to farthest.

Value

A list containing two matrices:

nn.i	An n x k matrix where element `[i,j]` contains the index of the j-th nearest neighbor of point i.
nn.d	An n x k matrix where element `[i,j]` contains the distance to the j-th nearest neighbor of point i.

Examples

```
# Create a simple distance matrix
d <- as.matrix(dist(matrix(rnorm(20), ncol=2)))
knn_result <- dist.to.knn(d, k=3)
# knn_result$nn.i contains indices of 3 nearest neighbors for each point
# knn_result$nn.d contains distances to those neighbors
```

`dlaplace` *Laplace Distribution Density Function*

Description

Calculates the density function for the Laplace distribution.

Usage

```
dlaplace(x, location = 0, scale = 1, log = FALSE)
```

Arguments

`x`	Vector of quantiles.
`location`	The location parameter μ. Default is 0.
`scale`	The scale parameter b. Must be positive. Default is 1.
`log`	Logical; if TRUE, the log density is returned. Default is FALSE.

Value

A vector of density values (or log-density if log = TRUE).

Examples

```
x <- seq(-5, 5, by = 0.1)
y <- dlaplace(x, location = 0, scale = 1)
plot(x, y, type = "l", main = "Laplace Density")
```

`draw.3d.line` *Draw 3D Line Segment*

Description

Plots a line segment in 3D space from origin in the direction of a vector

Usage

```
draw.3d.line(x, length = 2, col = "gray")
```

Arguments

`x`	Numeric vector of length 3 specifying direction.
`length`	Length of the line segment.
`col`	Color of the line segment.

Details

This function draws a line segment from the origin (0,0,0) in the direction of vector x, with the specified length.

Value

Invisibly returns NULL.

Examples

```
## Not run:
plot3D.plain(matrix(0, ncol = 3))
draw.3d.line(c(1, 1, 1), length = 2, col = "red")
draw.3d.line(c(1, 0, 0), length = 1.5, col = "blue")

## End(Not run)
```

`draw.axes` *Draw 3D Axes*

Description

Creates and plots 3D axes with labels and arrows

Usage

```
draw.axes(
  delta = 0.5,
  axes.color = "black",
  axes.lab.cex = 2,
  edge.lwd = 10,
  half.axes = TRUE,
  x.lab = "x_1",
  y.lab = "x_2",
  z.lab = "x_3"
)
```

Arguments

`delta`	Extension length beyond unit length for axes.
`axes.color`	Color of the axes.
`axes.lab.cex`	Character expansion factor for axis labels.
`edge.lwd`	Line width of axes.
`half.axes`	Logical. If TRUE, only positive half-axes are drawn.
`x.lab`	Label for x-axis.
`y.lab`	Label for y-axis.
`z.lab`	Label for z-axis.

Details

This function draws a set of 3D axes with optional arrows and labels. The axes can be drawn as full axes or half-axes (positive direction only).

Value

Invisibly returns NULL.

Examples

```
## Not run:
plot3D.plain(matrix(rnorm(30), ncol = 3))
draw.axes(delta = 0.5, axes.color = "black", half.axes = TRUE)

## End(Not run)
```

`draw.dashed.line3d` *Draw Dashed Line in 3D Space*

Description

Creates a dashed line between two points in 3D space

Usage

```
draw.dashed.line3d(
  x1,
  y1,
  z1,
  x2,
  y2,
  z2,
  n = 10,
  dashLength = 0.1,
  gapLength = 0.1,
  col = "black",
  lwd = 2
)
```

Arguments

`x1, y1, z1`	Coordinates of the starting point.
`x2, y2, z2`	Coordinates of the ending point.
`n`	Deprecated parameter kept for compatibility.
`dashLength`	Length of each dash.
`gapLength`	Length of gaps between dashes.
`col`	Color of the line.
`lwd`	Line width.

Details

This function creates a dashed line by drawing multiple short line segments with gaps between them.

Value

Invisibly returns NULL.

Examples

```
## Not run:
# Draw a dashed line from origin to (1, 1, 1)
plot3D.plain(matrix(c(0,0,0,1,1,1), ncol = 3, byrow = TRUE))
draw.dashed.line3d(0, 0, 0, 1, 1, 1, col = "red", lwd = 3)

## End(Not run)
```

`dx.vs.gRf.mod.gSf` *Given gSf and gRf, dx are found so that corresponding grid(X) has give gSf. The resulting data is used to build models allowing to construct grid(X) with the prespecified values of gRf and gSf.*

Description

Given gSf and gRf, dx are found so that corresponding grid(X) has give gSf. The resulting data is used to build models allowing to construct grid(X) with the prespecified values of gRf and gSf.

Usage

```
dx.vs.gRf.mod.gSf(
  X,
  min.K = 15,
  med.dK.divf = 10,
  max.dx.C = 5,
  max.dx = NULL,
  min.gRf = 3,
  max.gRf = 20,
  n.gRfs = 20,
  min.gSf = 1,
  max.gSf = 5,
  n.gSfs = 10,
  n.cores = 10,
  verbose = TRUE
)
```

Arguments

`X`	A state space matrix.
`min.K`	The number of nearest neighbors used to estimate median min.K.
`med.dK.divf`	A med.dK division factor for the lower bound of bw and dx to be med.dK/med.dK.divf, where `med.dK=median(dK)` with `dK=nn.dist[,min.K]`.
`max.dx.C`	A division factor for max.dx estimate such that max.dx = (length of the diagonal in the X encolosing box) / max.dx.C.
`max.dx`	The maximal value of dx.
`min.gRf`	The minimum of gRf.
`max.gRf`	The maximum of gRf.
`n.gRfs`	The number of values of gRf between min.gRf and max.gRf.
`min.gSf`	The minimum of gSf.
`max.gSf`	The maximum of gSf.
`n.gSfs`	The number of values of gSf between min.gSf and max.gSf.
`n.cores`	The number of cores in foreach loop.
`verbose`	A logical flag. Set it to TRUE to report timing of the foreach loop.

E.geodesic.F — *Estimates and the mean prevalence of a factors along a geodesic.*

Description

Given a list of subject-level factors, subj.factors, and IDs of the given branch, bch.ids, as well as the distance along the geodesic, gamma, vector and a file prefix, this routine estimates the mean prevalence of the given factor, E_gamma_F, along the geodesic and saves the results of the estimates to a file the given prefix and suffix: Fid.rda.

Usage

```
E.geodesic.F(
  subj.factors,
  subjID,
  bch.ids,
  dist.along.bch.geodesic,
  file.prefix
)
```

Arguments

`subj.factors`	A list of subject-level factors.
`subjID`	A vector of subject IDs.
`bch.ids`	Branch IDs.
`dist.along.bch.geodesic`	A distance along the geodesic of the given branch.
`file.prefix`	The prefix of an rda file to which the results of E_gamma_F estimates will be save. Example: "~/projects/rllm/projects/ZB/data/asv_branches/Lc2Gv6_".

Details

For each factor in subj.factors, this function:

1. Identifies subjects present in the branch
2. Calculates the mean prevalence along the geodesic distance
3. Saves the results to an RDA file named `{file.prefix}{Fid}.rda`

The saved RDA files contain the E_gamma_F estimates which can be loaded for further analysis or visualization.

Value

A named character vector mapping factor IDs to their corresponding saved file paths. Each element name is a factor ID and its value is the full path to the RDA file containing the E_gamma_F estimation results for that factor.

E.geodesic.X *Estimate Mean Abundance Along a Geodesic*

Description

Estimates the mean abundance of covariates along a geodesic path. Given a matrix of covariates defined over a subset of samples from the state space containing the strand, this function estimates the mean of each column as a function of distance along the geodesic for columns meeting prevalence criteria. Results are saved to the specified data directory.

Usage

```
E.geodesic.X(
  X,
  strand.ids,
  dist.along.strand.geodesic,
  data.dir,
  winsorize = TRUE,
  winsorize.p = 0.025,
  min.max.normalize = TRUE,
  prev.thld = 0.9,
  min.sample.count = 40,
  rerun = FALSE,
  n.partitions = 1,
  partition.i = 1,
  verbose = TRUE
)
```

Arguments

`X`	A numeric matrix of covariates with samples as rows and variables as columns. Row names should contain sample IDs.
`strand.ids`	Character vector of sample IDs belonging to the strand.
`dist.along.strand.geodesic`	Numeric vector of distances along the geodesic for each sample in the strand. Names should correspond to sample IDs.
`data.dir`	Character string specifying the directory path where results will be saved.
`winsorize`	Logical. If `TRUE` (default), performs right winsorization on columns of `X` before normalization.
`winsorize.p`	Numeric value between 0 and 1. Proportion of values at the right tail to winsorize. Values above the (1 - `winsorize.p`) quantile are set to that quantile value. Default is 0.025.
`min.max.normalize`	Logical. If `TRUE` (default), applies min-max normalization to each column of `X`.
`prev.thld`	Numeric value between 0 and 1. Prevalence threshold for column filtering. Only columns with non-zero values in at least `prev.thld` proportion of samples are retained. Default is 0.9.
`min.sample.count`	Integer. Minimum number of samples required in common between `X` and the strand to proceed with analysis. Default is 40.

`rerun`	Logical. If `TRUE`, reruns `fassoc1.test()` even if results already exist. Default is `FALSE`.
`n.partitions`	Integer. Number of partitions to split the column processing into. Useful for parallel processing. Default is 1.
`partition.i`	Integer. Index of the partition to process when `n.partitions > 1`. Must be between 1 and `n.partitions`.
`verbose`	Logical. If `TRUE` (default), prints progress messages during execution.

Details

The function performs the following steps:

1. Identifies samples common to both `X` and `strand.ids`
2. Filters columns based on the prevalence threshold
3. Optionally applies winsorization and normalization
4. Runs `fassoc1.test()` on each retained column
5. Saves results to individual files in the specified directory

When `n.partitions > 1`, the function processes only a subset of columns specified by `partition.i`, enabling distributed computation.

Value

A list with the following components:

`X`	The processed matrix after sample and column filtering, with optional winsorization and normalization applied.
`dist.along.strand.geodesic`	Numeric vector of geodesic distances for the samples common to both `X` and the strand.
`data.dir`	Character string of the data directory path.
`save.files`	Named character vector mapping standardized column names to file paths where results are saved.
`name.tbl`	Named character vector mapping standardized column names to original column names from `X`.

Note

Results for each column are saved as RDA files in `data.dir` with filenames based on column indices. Each file contains `Ey` (expected values) and `pval` (p-value) from `fassoc1.test()`.

See Also

`fassoc1.test`, `right.winsorize`, `minmax.normalize`

Examples

```
## Not run:
# Create example data
set.seed(123)
X <- matrix(rnorm(1000), nrow=100, ncol=10)
rownames(X) <- paste0("sample", 1:100)
colnames(X) <- paste0("var", 1:10)

# Define strand
strand.ids <- paste0("sample", 1:80)
dist.along.strand.geodesic <- seq(0, 1, length.out=80)
names(dist.along.strand.geodesic) <- strand.ids

# Run analysis
result <- E.geodesic.X(X = X,
                       strand.ids = strand.ids,
                       dist.along.strand.geodesic = dist.along.strand.geodesic,
                       data.dir = tempdir(),
                       prev.thld = 0.5)

# Check results
str(result)

## End(Not run)
```

`ecdf.cpp` *Empirical Cumulative Distribution Function (ECDF)*

Description

Calculates the empirical cumulative distribution function (ECDF) for a given vector of data.

Usage

```
ecdf.cpp(x)
```

Arguments

`x`	A numeric vector of data.

Details

The ECDF is a step function that represents the cumulative distribution of a sample of data. It is defined as the proportion of observations in the sample that are less than or equal to a given value. This function calculates the ECDF for each unique value in the input vector.

Value

A numeric vector representing the ECDF values for each unique value in the input vector. The length of the return value is the same as the number of unique values in the input vector. The names attribute of the return value is set to "ecdf".

Note

The input vector is sorted internally to compute the ECDF. If the input vector contains missing values (NA), they will be removed before computing the ECDF.

Examples

```
x <- c(1.0, 2.0, 3.0, 4.0, 5.0)
result <- ecdf.cpp(x)
print(result)
```

`edge.diff` *Compute Edge Difference Between Two Graphs*

Description

This function takes two graphs represented as adjacency lists and computes the edge difference from graph1 to graph2. It returns a list where each component contains the edges present in graph1 but not in graph2 for each vertex.

Usage

```
edge.diff(graph1, graph2)
```

Arguments

`graph1`	A list representing the adjacency list of the first graph. Each element of the list corresponds to a vertex and contains the indices of its neighbors.
`graph2`	A list representing the adjacency list of the second graph. It must have the same number of vertices as graph1.

Value

A list of the same length as the input graphs. Each element contains the neighbors of the corresponding vertex that are present in graph1 but not in graph2.

Examples

```
graph1 <- list(c(2,3), c(1,3), c(1,2), c(5), c(4))
graph2 <- list(c(2), c(1), c(), c(5), c(4))
diff <- edge.diff(graph1, graph2)
print(diff)
```

`effective.models` *Effective Number of Models in Graph Spectral MA LOWESS*

Description

Calculates the effective number of models contributing to each vertex's prediction based on the weight distribution entropy.

Usage

```
effective.models(object, ...)

## S3 method for class 'graph.spectral.ma.lowess'
effective.models(object, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

Numeric vector with effective number of models per vertex

`eigen.ulogit` *Fit Univariate Logistic Regression Using Eigen*

Description

Fits a univariate logistic regression model using the Eigen linear algebra library for enhanced numerical stability. This function supports both linear and quadratic models with optional observation weights and provides efficient leave-one-out cross-validation.

Usage

```
eigen.ulogit(
  x,
  y,
  w = NULL,
  fit.quadratic = FALSE,
  with.errors = TRUE,
  max.iterations = 100L,
  ridge.lambda = 0.002,
  tolerance = 1e-08
)
```

Arguments

`x`	Numeric vector of predictor values. Must contain only finite values (no NA, NaN, or Inf).
`y`	Binary response vector containing only 0 or 1 values. Must be the same length as `x`.
`w`	Optional numeric vector of non-negative observation weights. If `NULL` (default), all observations are given equal weight. Must be the same length as `x` if provided.
`fit.quadratic`	Logical; if `TRUE`, includes a quadratic term (x^2) in the model. Default is `FALSE`.
`with.errors`	Logical indicating whether to compute leave-one-out cross-validation errors. Default is `TRUE`. Setting to `FALSE` can improve performance for large datasets.
`max.iterations`	Maximum number of Newton-Raphson iterations. Must be a positive integer. Default is 100.
`ridge.lambda`	Ridge regularization parameter for numerical stability. Must be non-negative. Default is 0.002. Applied to all non-intercept coefficients.
`tolerance`	Convergence tolerance for Newton-Raphson algorithm. The algorithm stops when the relative change in log-likelihood is less than this value. Must be positive. Default is 1e-8.

Details

For the linear model (`fit.quadratic = FALSE`), the function fits:

$$logit(p_i) = \beta_0 + \beta_1 x_i$$

For the quadratic model (`fit.quadratic = TRUE`), it fits:

$$logit(p_i) = \beta_0 + \beta_1 x_i + \beta_2 x_i^2$$

The implementation uses the Eigen C++ library for linear algebra operations, providing enhanced numerical stability compared to standard BLAS/LAPACK routines, particularly for ill-conditioned problems.

Ridge regularization is applied to non-intercept coefficients to prevent overfitting and improve numerical stability. The penalty term added to the negative log-likelihood is $\lambda(\beta_1^2 + \beta_2^2)$.

Value

A list of class `"eigen.ulogit"` containing:

predictions Numeric vector of fitted probabilities

converged Logical indicating whether the algorithm converged

iterations Integer giving the number of iterations used

beta Numeric vector of coefficients: intercept, linear coefficient, and (if `fit.quadratic = TRUE`) quadratic coefficient

errors If `with.errors = TRUE`, numeric vector of leave-one-out cross-validation errors; otherwise `NULL`

loglik Final log-likelihood value

aic Akaike Information Criterion

bic Bayesian Information Criterion

call The matched call

model Character string indicating model type ("linear" or "quadratic")

References

McCullagh, P., & Nelder, J. A. (1989). Generalized Linear Models (2nd ed.). Chapman and Hall/CRC.

See Also

`ulogit` for the standard implementation, `glm` for general linear models, `predict.eigen.ulogit` for predictions on new data

Examples

```
# Generate example data with non-linear relationship
set.seed(456)
x <- runif(150, -3, 3)
true_prob <- 1/(1 + exp(-(1 + 2*x - 0.5*x^2)))
y <- rbinom(150, 1, prob = true_prob)

# Fit linear model
fit_linear <- eigen.ulogit(x, y)

# Fit quadratic model
fit_quad <- eigen.ulogit(x, y, fit.quadratic = TRUE)

# Compare models using AIC
cat("Linear model AIC:", fit_linear$aic, "\n")
cat("Quadratic model AIC:", fit_quad$aic, "\n")

# Plot both fits
ord <- order(x)
plot(x, y, pch = 16, col = ifelse(y == 1, "blue", "red"),
     main = "Linear vs Quadratic Logistic Regression")
lines(x[ord], fit_linear$predictions[ord], lwd = 2, col = "green")
lines(x[ord], fit_quad$predictions[ord], lwd = 2, col = "purple")
legend("topleft", c("y = 1", "y = 0", "Linear", "Quadratic"),
       col = c("blue", "red", "green", "purple"),
       pch = c(16, 16, NA, NA), lty = c(NA, NA, 1, 1))

# Example with weights and without errors for efficiency
w <- runif(150, 0.5, 2)
fit_fast <- eigen.ulogit(x, y, w = w, with.errors = FALSE)

# Access coefficients
cat("Quadratic model coefficients:\n")
cat("  Intercept:", fit_quad$beta[1], "\n")
cat("  Linear:", fit_quad$beta[2], "\n")
cat("  Quadratic:", fit_quad$beta[3], "\n")
```

`elapsed.time` *Format and Print Elapsed Time*

Description

This function calculates the elapsed time since a given start time, formats it, and prints it with an optional message.

Usage

```
elapsed.time(start.time, message = "DONE", with.brackets = TRUE)
```

Arguments

`start.time`	A proc.time() object representing the start time.
`message`	Character string to be printed before the elapsed time. Default is "DONE".
`with.brackets`	Logical, if TRUE (default), the time is enclosed in parentheses.

Value

None. This function is called for its side effect of printing.

Examples

```
start <- proc.time()
Sys.sleep(2)  # Do some work
elapsed.time(start, "Processing complete")
```

`energy.distance` *Compute Energy Distance Between Two Datasets*

Description

This function calculates the energy distance between two datasets. The energy distance is a statistical distance between the distributions of random vectors, which is zero if and only if the distributions are identical.

Intuitively, the energy distance can be thought of as a measure of the "tension" between two distributions. If we imagine the points in each distribution as particles with the same charge, the energy distance is analogous to the potential energy of the system. When the distributions are identical, the "tension" or "energy" is minimized.

The energy distance has several desirable properties:

1. It's simple to compute, requiring only pairwise distances between points.
2. It's applicable to data of any dimension.
3. It doesn't require density estimation, making it suitable for high-dimensional data.

While not a direct proxy for relative entropy, the energy distance provides a robust measure of distributional differences and can be used in similar contexts.

Usage

```
energy.distance(X, Y)
```

Arguments

`X`	A matrix or data frame representing the first dataset.
`Y`	A matrix or data frame representing the second dataset.

Value

A numeric value representing the energy distance between X and Y.

References

Székely, G. J., & Rizzo, M. L. (2013). Energy statistics: A class of statistics based on distances. Journal of statistical planning and inference, 143(8), 1249-1272.

Examples

```
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- energy.distance(X, Y)
print(result)
```

---

entropy — *Entropy of a vector of non-negative values*

Description

Entropy of a vector of non-negative values

Usage

```
entropy(x, base = 10)
```

Arguments

`x`	A vector of non-negative values
`base`	The base of the logarithm in the entropy formula.

Value

entropy with log 'base'

---

entropy.difference — *Estimate Relative Entropy Using Entropy Difference*

Description

This function estimates the relative entropy between two datasets by computing the difference between the joint entropy and the average of individual entropies. The intuition behind this method comes from the relationship between mutual information and entropy:

I(X;Y) = H(X) + H(Y) - H(X,Y) = KL(P(X,Y) || P(X)P(Y))

Where I(X;Y) is the mutual information, H() is entropy, and KL() is the Kullback-Leibler divergence.

The algorithm works by:

1. Discretizing the continuous data into bins.
2. Estimating the entropy of X, Y, and their union separately.
3. Computing the difference to approximate the relative entropy.

This method provides a rough approximation of the relative entropy and can be useful when dealing with high-dimensional data where direct density estimation is challenging. However, it's sensitive to the choice of binning and may not be as accurate as some other methods.

Usage

```
entropy.difference(X, Y, num.bins = 10)
```

Arguments

`X`	A matrix or data frame representing the first dataset.
`Y`	A matrix or data frame representing the second dataset.
`num.bins`	The number of bins to use for discretization (default is 10).

Value

A numeric value representing the estimated relative entropy.

References

Cover, T. M., & Thomas, J. A. (2006). Elements of information theory. John Wiley & Sons.

See Also

`discretize` for the discretization method,

Examples

```
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- entropy.difference(X, Y)
print(result)
```

`epanechnikov.kernel` *Epanechnikov kernel function*

Description

The function equals to 1-t^2 for t in (-1,1) and 0 otherwise; t = abs(x/bw)

Usage

```
epanechnikov.kernel(x, bw = 1)
```

Arguments

`x`	A numeric vector.
`bw`	A bandwidth numeric parameter.

`estimate.geodesic.distances` *Estimate Geodesic Distances Between Points*

Description

This function estimates the shortest path distances (geodesic distances) between points in a given dataset. It can use either a k-nearest neighbors graph or a minimal spanning tree.

Usage

```
estimate.geodesic.distances(points, k = 5, graph = NULL, method = "knn.graph")
```

Arguments

`points`	A matrix or data frame where each row represents a point in the feature space.
`k`	Positive integer. The number of nearest neighbors to use when constructing the graph. When k=1 or method='mst', a minimal spanning tree is used instead. Default is 5.
`graph`	An optional igraph object representing the graph structure of the points. If provided, this graph will be used instead of constructing a new one.
`method`	Character string specifying the method to use for constructing the graph. Must be either "knn.graph" (default) or "mst" (minimal spanning tree).

Details

Geodesic distances represent the shortest path between points through a graph structure, which can better capture the intrinsic geometry of data lying on a manifold compared to Euclidean distances.

The function supports two methods for graph construction:

- **knn.graph**: Connects each point to its k nearest neighbors based on Euclidean distance
- **mst**: Creates a minimal spanning tree of all points

Value

A distance matrix containing the estimated geodesic distances between all pairs of points. The matrix has dimensions n x n where n is the number of points. Row and column names are preserved from the input if present.

Note

The k-NN graph construction creates a directed graph that is then treated as undirected. This may result in some points having more than k connections if they appear in other points' k-nearest neighbor lists.

Examples

```
# Generate sample data on a Swiss roll manifold
n <- 100
t <- seq(0, 4*pi, length.out = n)
points <- cbind(
  x = t * cos(t),
  y = 10 * runif(n),
  z = t * sin(t)
)

# Estimate geodesic distances using k-NN graph
geo_dist_knn <- estimate.geodesic.distances(points, k = 5)

# Estimate geodesic distances using MST
geo_dist_mst <- estimate.geodesic.distances(points, method = "mst")

# Compare with Euclidean distances
euc_dist <- as.matrix(dist(points))

# Geodesic distances are typically larger than Euclidean for manifold data
mean(geo_dist_knn > euc_dist, na.rm = TRUE)
```

`estimate.optimal.bandwidth.from.extrema.elbow` *Estimate Optimal Bandwidth Using Local Extrema Count Elbow Method*

Description

Estimates the optimal bandwidth for kernel smoothing by analyzing the relationship between bandwidth and the number of local extrema. The method identifies an "elbow point" in this relationship using piecewise linear regression and adjusts it based on model residuals.

Usage

```
estimate.optimal.bandwidth.from.extrema.elbow(
  extrema.counts,
  sd.multiplier = 1,
  plot.results = FALSE
)
```

Arguments

`extrema.counts`	A numeric vector of local extrema counts corresponding to different bandwidths, ordered from smallest to largest bandwidth. Must contain at least 5 values.
`sd.multiplier`	The number of residual standard deviations to add to the elbow point. Must be non-negative. Default is 1.0.
`plot.results`	Logical. If `TRUE`, produces a diagnostic plot showing the fitted model and selected bandwidth. Default is `FALSE`.

Details

The number of local extrema in a kernel-smoothed function typically decreases as bandwidth increases, often exhibiting an elbow-shaped curve. This function:

1. Fits a piecewise linear model to the extrema counts
2. Identifies the breakpoint (elbow) in the relationship
3. Adds a specified multiple of the residual standard deviation to the breakpoint location

The optimal bandwidth is often slightly beyond the elbow point, as the exact elbow typically represents the transition from undersmoothing to appropriate smoothing.

Value

An integer representing the index of the estimated optimal bandwidth in the input vector.

References

Muggeo, V. M. (2003). Estimating regression models with unknown break-points. *Statistics in Medicine*, 22(19), 3055-3071.

See Also

`graph.kernel.smoother` for the main smoothing function.

Examples

```
# Simulated extrema counts for increasing bandwidths
extrema.counts <- c(150, 142, 128, 95, 72, 58, 45, 38, 35, 33, 32, 31)

# Estimate optimal bandwidth
opt.idx <- estimate.optimal.bandwidth.from.extrema.elbow(
  extrema.counts,
  sd.multiplier = 1.5,
  plot.results = TRUE
)

print(paste("Optimal bandwidth index:", opt.idx))
```

`estimate_density` *Define a function to estimate the density rho_S(X)*

Description

This is a placeholder for a density estimation function. In practice, this might use methods like kernel density estimation.

Usage

```
estimate_density(point, X)
```

Arguments

`point`	A numeric vector representing a point in the state space.
`X`	A matrix where each row represents a point in the state space, used as the reference dataset for density estimation.

Details

This function is intended to estimate the density of points in a high-dimensional space. The current implementation returns a constant value of 1, serving as a placeholder. In production use, this should be replaced with an appropriate density estimation method such as:

- Kernel density estimation (KDE)
- k-nearest neighbors density estimation
- Gaussian mixture models

Value

A numeric value representing the estimated density at the given point. Currently returns 1 as a placeholder implementation.

Note

This is a placeholder implementation. Users should replace this with an appropriate density estimation method for their specific use case.

Examples

```
# Current placeholder usage
X <- matrix(rnorm(100), ncol=2)
point <- c(0, 0)
density <- estimate_density(point, X)  # Returns 1
```

`euclidean.distance` *Function to calculate the Euclidean distance between two points*

Description

Function to calculate the Euclidean distance between two points

Usage

```
euclidean.distance(p1, p2)
```

Arguments

`p1`	First point.
`p2`	Second point.

Details

This function computes the Euclidean distance between two points in n-dimensional space using the formula: $\sqrt(\sum((p1-p2)^2))$. The points $p1$ and $p2$ should be numeric vectors of the same length.

Value

A numeric value representing the Euclidean distance between p1 and p2.

Examples

```
# Distance between two 2D points
point1 <- c(0, 0)
point2 <- c(3, 4)
euclidean.distance(point1, point2)  # Returns 5

# Distance between two 3D points
point1 <- c(1, 2, 3)
point2 <- c(4, 6, 8)
euclidean.distance(point1, point2)  # Returns sqrt(50)
```

`evaluate.function.on.grid`

Evaluate Function on Grid

Description

Evaluates a bivariate function on a uniform grid.

Usage

```
evaluate.function.on.grid(f, grid)
```

Arguments

`f`	Function taking two arguments (x, y)
`grid`	List containing grid data (as created by create.grid)

Value

Matrix of function values

Examples

```
## Not run:
grid <- create.grid(50)
f <- function(x, y) sin(x) * cos(y)
f.grid <- evaluate.function.on.grid(f, grid)
image(grid$x, grid$y, f.grid)

## End(Not run)
```

`evaluate.function.on.grid.as.vector` *Evaluate Function on Grid as Vector*

Description

Evaluates a bivariate function at each point of a grid, returning results as a vector.

Usage

```
evaluate.function.on.grid.as.vector(f, grid)
```

Arguments

`f`	Function taking two arguments (x, y)
`grid`	List containing grid data (as created by create.grid)

Value

Numeric vector of function values

Examples

```
grid <- create.grid(50)
f <- function(x, y) x^2 + y^2
values <- evaluate.function.on.grid.as.vector(f, grid)
```

`evenness` *Computes Shannon Evenness of a discreate probability distribution*

Description

Computes normalized Shannon entropy (evenness) of a numeric vector defined as entropy(x) / logB(length(x)), where B is the base of the logarithm.

Usage

```
evenness(x, base = 2)
```

Arguments

`x`	Numeric vector of counts or abundances
`base`	Logarithm base (default: 2)

Value

A number in $[0, 1]$ quantifying evenness (1 = perfectly even)

`expand.box` *Expands a bounding box given by the opposite vertices (L, R)*

Description

Expands a bounding box given by the opposite vertices (L, R)

Usage

```
expand.box(L, R, p.exp, margin = NULL)
```

Arguments

`L`	A numeric vector representing the left vertix that is the point of the box with the smallest coordinates among all other points of the box.
`R`	A numeric vector representing the right vertix that is the point of the box with the largest coordinates among all other points of the box.
`p.exp`	An expansion factor for the bounding box. Each edge of the box is scaled by the factor (1 + p.exp).
`margin`	A numeric value specifying a fixed margin to add to the box. If NULL (default), margin is calculated based on p.exp.

`ext.graph.diffusion.smoother` *Extended Graph Diffusion Smoother*

Description

Performs graph diffusion smoothing on a graph signal using an iterative diffusion process. This method is useful for denoising graph signals, interpolating missing values, and enhancing patterns in network data while preserving the underlying graph structure.

Usage

```
ext.graph.diffusion.smoother(
  graph,
  edge.lengths,
  y,
  weights = NULL,
  n.time.steps = 100,
  step.factor = 0.1,
  normalize = 0,
  preserve.local.maxima = FALSE,
  local.maximum.weight.factor = 1,
  preserve.local.extrema = FALSE,
  imputation.method = 0,
  max.iterations = 10,
  convergence.threshold = 1e-06,
  apply.binary.threshold = TRUE,
  binary.threshold = 0.5,
  kernel = 1,
  dist.normalization.factor = 1.01,
  n.CVs = 0,
  n.CV.folds = 10,
  epsilon = 1e-10,
  seed = 0,
  verbose = TRUE
)
```

Arguments

`graph`	A list of integer vectors where each element contains the indices of neighbors for the corresponding vertex (1-based indexing).
`edge.lengths`	A list of numeric vectors where each element contains the edge lengths corresponding to the neighbors in `graph`.
`y`	A numeric vector of signal values at each vertex.
`weights`	An optional numeric vector of weights for each vertex. If `NULL`, all vertices are weighted equally. Default is `NULL`.
`n.time.steps`	An integer specifying the number of diffusion steps. Default is 100.
`step.factor`	A numeric value between 0 and 1 controlling the magnitude of each diffusion step. Smaller values lead to more gradual smoothing. Default is 0.1.
`normalize`	An integer (0, 1, or 2) specifying the normalization method: • 0: No normalization • 1: Range adjustment to $[0, 1]$ • 2: Mean adjustment to preserve the mean of the signal. Default is 0.
`preserve.local.maxima`	A logical value. If `TRUE`, local maxima are preserved during diffusion by adjusting weights based on neighborhood values. Default is `FALSE`.
`local.maximum.weight.factor`	A numeric value between 0 and 1 controlling the degree of preservation for local maxima. Default is 1.0.
`preserve.local.extrema`	A logical value. If `TRUE`, both local maxima and minima are preserved. Cannot be `TRUE` simultaneously with `preserve.local.maxima`. Default is `FALSE`.
`imputation.method`	An integer (0-4) specifying the imputation method: • 0: LOCAL_MEAN_THRESHOLD • 1: NEIGHBORHOOD_MATCHING • 2: ITERATIVE_NEIGHBORHOOD_MATCHING • 3: SUPPLIED_THRESHOLD • 4: GLOBAL_MEAN_THRESHOLD. Default is 0.
`max.iterations`	A positive integer for the maximum number of iterations in iterative imputation methods. Default is 10.

`convergence.threshold`	A positive numeric value for convergence criteria. Default is 1e-6.
`apply.binary.threshold`	A logical value indicating whether to apply binary thresholding to the output. Default is `TRUE`.
`binary.threshold`	A numeric value between 0 and 1 for binary thresholding. Default is 0.5.
`kernel`	An integer (0-4) specifying the kernel type for distance weighting: • 0: Uniform kernel • 1: Gaussian kernel • 2: Laplacian kernel • 3: Exponential kernel • 4: Cauchy kernel Default is 1.
`dist.normalization.factor`	A numeric value >= 1 for distance normalization. Default is 1.01.
`n.CVs`	A non-negative integer for the number of cross-validation runs. If 0, no cross-validation is performed. Default is 0.
`n.CV.folds`	An integer > 1 for the number of folds in cross-validation. Default is 10.
`epsilon`	A positive numeric value for numerical stability. Default is 1e-10.
`seed`	An integer for the random number generator seed. Default is 0.
`verbose`	A logical value. If `TRUE`, prints progress information. Default is `TRUE`.

Details

The graph diffusion process iteratively updates vertex values based on their neighbors' values, weighted by edge lengths and kernel functions. The algorithm can preserve local extrema to maintain important features while smoothing noise.

Cross-validation uses a masking approach where test vertices don't influence the diffusion but their values are predicted to evaluate performance.

Value

A list of class `"ext.gds"` containing:

`y.traj`	A matrix where each column represents the signal at a time step.
`y.optimal`	The optimally smoothed signal (based on CV if performed).
`cv.errors`	A matrix of cross-validation errors (rows: time steps, columns: CV runs).
`mean.cv.errors`	Mean CV errors across runs for each time step.
`Rmean.cv.errors`	R-computed mean CV errors (for verification).
`Rmedian.cv.errors`	Median CV errors across runs.
`optimal.time.step`	The time step with minimum CV error.
`min.cv.error`	The minimum mean CV error.

References

Coifman, R. R., & Lafon, S. (2006). Diffusion maps. Applied and computational harmonic analysis, 21(1), 5-30.

See Also

graph.diffusion.matrix.smoother, instrumented.gds

Examples

```
## Not run:
# Create a simple chain graph
n <- 20
graph <- vector("list", n)
edge.lengths <- vector("list", n)
for(i in 1:(n-1)) {
  graph[[i]] <- c(graph[[i]], i+1)
  graph[[i+1]] <- c(graph[[i+1]], i)
  edge.lengths[[i]] <- c(edge.lengths[[i]], 1)
  edge.lengths[[i+1]] <- c(edge.lengths[[i+1]], 1)
}

# Create noisy signal
true.signal <- sin(seq(0, 2*pi, length.out = n))
y <- true.signal + rnorm(n, 0, 0.2)

# Apply diffusion smoothing
result <- ext.graph.diffusion.smoother(
  graph = graph,
  edge.lengths = edge.lengths,
  y = y,
  n.time.steps = 50,
  step.factor = 0.1,
  n.CVs = 5,
  verbose = TRUE
)

# Plot results
plot(y, type = "b", col = "red", main = "Graph Diffusion Smoothing")
lines(result$y.optimal, type = "b", col = "blue")
legend("topright", c("Original", "Smoothed"), col = c("red", "blue"), lty = 1)

## End(Not run)
```

`extract.derivatives` *Extract Derivative Information from assoc1 Object*

Description

Extracts and summarizes derivative information from the first-order functional association test results.

Usage

```
extract.derivatives(
  object,
  type = c("estimate", "credible.interval", "both"),
  probs = c(0.025, 0.975)
)
```

Arguments

`object`	An object of class "assoc1".
`type`	Character string specifying what to extract: "estimate" (default), "credible.interval", or "both".
`probs`	Numeric vector of probabilities for credible intervals. Default is c(0.025, 0.975) for 95% intervals.

Value

Depending on type:

`estimate`	A numeric vector of derivative estimates
`credible.interval`	A matrix with lower and upper bounds
`both`	A list containing both estimate and credible.interval

Examples

```
## Not run:
result <- fassoc1.test(x, y)
deriv.est <- extract.derivatives(result)
deriv.ci <- extract.derivatives(result, type = "credible.interval")

## End(Not run)
```

`extract.extrema.subgraphs` *Extract Local Subgraphs Around Extrema*

Description

Creates a list of subgraphs centered around each local extremum in the input data. Each subgraph includes vertices within the hop radius of the extremum plus an additional offset.

Usage

```
extract.extrema.subgraphs(
  adj.list,
  weight.list,
  y,
  predictions,
  extrema.df,
  hop.offset = 2,
  min.hop.idx = 0,
  max.hop.idx = Inf
)
```

Arguments

`adj.list`	A list where each element contains the neighbors of the corresponding vertex. Must be a valid adjacency list representation of an undirected graph.
`weight.list`	A list of numeric vectors where each element contains the edge weights corresponding to the neighbors in `adj.list`. Must have the same structure as `adj.list`.
`y`	Numeric vector of function values on vertices. Length must equal the number of vertices in the graph.
`predictions`	Numeric vector of smoothed predictions (e.g., from spectral filtering). Must have the same length as `y`.
`extrema.df`	Data frame containing extrema information with required columns: • `vertex`: Integer vertex index • `hop_idx`: Integer hop radius of the extremum • `is_max`: Logical indicating if extremum is a maximum • `label`: Character label for the extremum • `fn_value`: Numeric function value at the extremum
`hop.offset`	Integer specifying additional hop distance to include beyond the extremum's hop radius (default: 2). Must be non-negative.
`min.hop.idx`	Integer specifying minimum hop index for extrema to process (default: 0). Must be non-negative.
`max.hop.idx`	Integer or `Inf` specifying maximum hop index for extrema to process (default: `Inf`).

Value

A list where each element corresponds to a processed extremum and contains:

adj_list Adjacency list of the subgraph (1-indexed)

weight_list Weight list of the subgraph

y Original function values for subgraph vertices

predictions Predicted function values for subgraph vertices

extremum_info Data frame row containing information about the central extremum, with vertex index mapped to subgraph numbering

region_vertices Integer vector of subgraph indices for vertices within the extremum's hop radius

boundary_vertices Integer vector of subgraph indices for vertices at exactly hop_idx+1 distance

boundary_values Named numeric vector of `y` values at boundary vertices, names are subgraph indices

vertex_mapping List with two components:

- `orig.to.new`: Named integer vector mapping original to subgraph indices
- `new.to.orig`: Integer vector mapping subgraph to original indices

hop_distances Numeric vector of hop distances from center for all subgraph vertices

extrema_df Data frame of all extrema within the subgraph with vertex indices mapped to subgraph numbering

orig_region_vertices Integer vector of original indices for region vertices

orig_boundary_vertices Integer vector of original indices for boundary vertices

orig_boundary_values Named numeric vector of y values with original vertex indices

Examples

```
## Not run:
# Create a simple graph
adj_list <- list(c(2, 3), c(1, 3, 4), c(1, 2, 4), c(2, 3))
weight_list <- list(c(1, 1), c(1, 1, 1), c(1, 1, 1), c(1, 1))
y <- c(1, 2, 3, 1)
predictions <- c(1.1, 2.1, 2.9, 1.1)

# Create extrema data frame
extrema_df <- data.frame(
  vertex = 3,
  hop_idx = 1,
  is_max = TRUE,
  label = "Max1",
  fn_value = 3,
  stringsAsFactors = FALSE
)

# Extract subgraph
result <- extract.extrema.subgraphs(
  adj_list, weight_list, y, predictions, extrema_df, hop.offset = 1
)

## End(Not run)
```

`extract.skeleton.graph` *Extract 1-Skeleton Graph from Nerve Complex*

Description

This function extracts the 1-skeleton of the nerve complex as a graph.

Usage

```
extract.skeleton.graph(complex)
```

Arguments

complex	A nerve complex object

Value

A graph representation of the 1-skeleton `extract.xy` *Extracts Numerical x and y Values from a Character Vector.*

Description

This function takes a character vector of the form `c("x1,y1", "x2,y2", ..., "xN,yN")` and extracts two numerical vectors, `x` and `y`, containing the corresponding x and y values.

Usage

```
extract.xy(s)
```

Arguments

`s`	A character vector containing strings of the form "xi,yi". Each string must contain exactly one comma, and the values before and after the comma must be numerical.

Value

A list containing two numerical vectors:

- `x`: A vector containing the x values (c(x1, x2, ..., xN)).
- `y`: A vector containing the y values (c(y1, y2, ..., yN)).

Examples

```
## Not run:
s <- c("1,2", "3,4", "5,6")
result <- extract_xy(s)
x <- result$x
y <- result$y

## End(Not run)
```

`fassoc.test` *Functional Association Test*

Description

Tests for the existence of non-trivial functional association between two variables x and y using either zero-order or first-order functional association measures. This function serves as a unified interface to both fassoc0.test() and fassoc1.test().

Usage

```
fassoc.test(
  x,
  y,
  order = 1,
  two.sample.test = c("norm", "Wasserstein", "KS"),
  bw = NULL,
  n.perms = 1000,
  n.BB = 1000,
  grid.size = 400,
  min.K = 5,
  n.cores = 7,
  plot.it = TRUE,
  xlab = "x",
  ylab = "y",
  Eyg.col = "red",
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.lines.col = "gray95",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  verbose = TRUE
)

functional.association.test(
  x,
  y,
  order = 1,
  two.sample.test = c("norm", "Wasserstein", "KS"),
  bw = NULL,
  n.perms = 1000,
  n.BB = 1000,
  grid.size = 400,
  min.K = 5,
  n.cores = 7,
  plot.it = TRUE,
  xlab = "x",
  ylab = "y",
  Eyg.col = "red",
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.lines.col = "gray95",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  verbose = TRUE
)
```

Arguments

`x`	A numeric vector of predictor values.
`y`	A numeric vector of response values (same length as x). Can be binary or continuous.
`order`	Integer specifying the order of the functional association test. Can be 0 (zero-order) or 1 (first-order). Default is 1.
`two.sample.test`	Character string specifying the test for comparing distributions. Options are "Wasserstein", "KS", or "norm". Default is "norm".
`bw`	Numeric bandwidth parameter for smoothing. If NULL (default), bandwidth is automatically selected.
`n.perms`	Integer specifying the number of permutations for null distribution. Default is 1000.
`n.BB`	Integer specifying the number of Bayesian bootstrap samples. Default is 1000.
`grid.size`	Integer specifying the size of evaluation grid. Default is 400.
`min.K`	Integer specifying minimum number of observations for local estimation. Default is 5.
`n.cores`	Integer specifying the number of CPU cores for parallel computation. Default is 7.
`plot.it`	Logical indicating whether to produce diagnostic plots. Default is TRUE.
`xlab`	Character string for x-axis label. Default is "x".
`ylab`	Character string for y-axis label. Default is "y".
`Eyg.col`	Color specification for conditional mean curve. Default is "red".
`pt.col`	Color specification for data points. Default is "black".
`pt.pch`	Integer or character specifying point type. Default is 1.
`CrI.as.polygon`	Logical indicating whether to draw credible intervals as polygons (TRUE) or lines (FALSE). Default is TRUE.
`CrI.polygon.col`	Color specification for credible interval polygon. Default is "gray90".
`null.lines.col`	Color specification for null distribution lines. Default is "gray95".
`CrI.line.col`	Color specification for credible interval lines. Default is "gray".
`CrI.line.lty`	Line type for credible interval lines. Default is 2.
`verbose`	Logical indicating whether to print progress messages. Default is TRUE.

Details

This function provides a unified interface to test for functional associations between variables. The zero-order measure (order=0) quantifies deviations of the conditional mean from the marginal mean, while the first-order measure (order=1) quantifies the variability in the derivative of the conditional mean.

The mathematical notation for these measures:

- Zero-order: $\text{fassoc}_0(x, y) = \int |E_x(y) - E(y)| dx$
- First-order: $\text{fassoc}_1(x, y) = \int |dE_x(y)/dx| dx$

This function can also be called using the alias:

- `functional.association.test()` - Full descriptive name

For direct access to specific orders, use:

- `fassoc0.test()` or `zofam.test()` - Zero-Order Functional Association Measure
- `fassoc1.test()` or `fofam.test()` - First-Order Functional Association Measure

Value

An object of class "assoc0" (if order=0) or "assoc1" (if order=1) containing test results. See fassoc0.test() and fassoc1.test() for details on the return values.

See Also

`fassoc0.test`, `fassoc1.test`

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 200
x <- runif(n)
y <- sin(2*pi*x) + rnorm(n, sd = 0.3)

# Test for zero-order functional association
result0 <- fassoc.test(x, y, order = 0, n.cores = 2)

# Test for first-order functional association
result1 <- fassoc.test(x, y, order = 1, n.cores = 2)

# Compare p-values
print(result0$p.value)
print(result1$p.value)

## End(Not run)
```

`fassoc0.test` *Tests for Existence of Non-trivial Functional Association Between Two Variables*

Description

Computes and tests the zero-order functional association measure between two variables x and y. The function estimates the integral of |E_x(y) - E(y)|, where E_x(y) is the conditional mean of y given x, and E(y) is the marginal mean of y. Statistical significance is assessed using permutation tests with Bayesian bootstrap.

Usage

```
fassoc0.test(
  x,
  y,
  two.sample.test = c("norm", "Wasserstein", "KS"),
  bw = NULL,
  n.perms = 10000,
  n.BB = 1000,
  grid.size = 400,
  min.K = 5,
  n.cores = 7,
  plot.it = TRUE,
  xlab = "x",
  ylab = "y",
  Eyg.col = "red",
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.lines.col = "gray95",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  verbose = TRUE
)

zofam.test(
  x,
  y,
  two.sample.test = c("norm", "Wasserstein", "KS"),
  bw = NULL,
  n.perms = 10000,
  n.BB = 1000,
  grid.size = 400,
  min.K = 5,
  n.cores = 7,
  plot.it = TRUE,
  xlab = "x",
  ylab = "y",
  Eyg.col = "red",
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.lines.col = "gray95",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  verbose = TRUE
)
```

Arguments

x	A numeric vector of predictor values.

`y`	A numeric vector of response values (same length as x). Can be binary or continuous.
`two.sample.test`	Character string specifying the test for comparing distributions. Options are "Wasserstein", "KS", or "norm". Default is "norm".
`bw`	Numeric bandwidth parameter for smoothing. If NULL (default), bandwidth is automatically selected.
`n.perms`	Integer specifying the number of permutations for null distribution. Default is 10000.
`n.BB`	Integer specifying the number of Bayesian bootstrap samples. Default is 1000.
`grid.size`	Integer specifying the size of evaluation grid. Default is 400.
`min.K`	Integer specifying minimum number of observations for local estimation. Default is 5.
`n.cores`	Integer specifying the number of CPU cores for parallel computation. Default is 7.
`plot.it`	Logical indicating whether to produce diagnostic plots. Default is TRUE.
`xlab`	Character string for x-axis label. Default is "x".
`ylab`	Character string for y-axis label. Default is "y".
`Eyg.col`	Color specification for conditional mean curve. Default is "red".
`pt.col`	Color specification for data points. Default is "black".
`pt.pch`	Integer or character specifying point type. Default is 1.
`CrI.as.polygon`	Logical indicating whether to draw credible intervals as polygons (TRUE) or lines (FALSE). Default is TRUE.
`CrI.polygon.col`	Color specification for credible interval polygon. Default is "gray90".
`null.lines.col`	Color specification for null distribution lines. Default is "gray95".
`CrI.line.col`	Color specification for credible interval lines. Default is "gray".
`CrI.line.lty`	Line type for credible interval lines. Default is 2.
`verbose`	Logical indicating whether to print progress messages. Default is TRUE.

Details

The function tests for functional association by comparing the observed measure of association against a null distribution generated through permutation. The conditional mean function E_x(y) is estimated using local polynomial methods with Bayesian bootstrap for uncertainty quantification.

Value

An object of class "assoc0" containing:

`delta`	The observed functional association measure
`p.value`	The p-value for the test of association
`log.p.value`	Natural logarithm of the p-value
`null.delta`	Vector of null distribution values
`null.delta.boxcox`	Box-Cox transformed null distribution (if applicable)
`BB.delta`	Bayesian bootstrap samples of delta

`BB.delta.boxcox`	Box-Cox transformed bootstrap samples (if applicable)
`delta.boxcox`	Box-Cox transformed observed delta (if applicable)
`Ey.res`	Results from conditional mean estimation
`x`	Input predictor values
`y`	Input response values
`Ey`	Marginal mean of y
`xg`	Grid points for evaluation
`Exyg`	Conditional mean estimates at grid points

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 200
x <- runif(n)
y <- sin(2*pi*x) + rnorm(n, sd = 0.3)

# Test for functional association
result <- fassoc0.test(x, y, n.cores = 2, n.perms = 1000)

# Custom plot
plot(result, plot = "Exy")

## End(Not run)
```

`fassoc1.test` *Tests for First-Order Functional Association Between Two Variables*

Description

Computes and tests the first-order functional association measure between two variables x and y. The function estimates the integral of |dE_x(y)/dx|, where E_x(y) is the conditional mean of y given x. This measure captures the strength of the derivative relationship between variables.

Usage

```
fassoc1.test(
  x,
  y,
  two.sample.test = c("norm", "Wasserstein", "KS"),
  bw = NULL,
  n.perms = 1000,
  n.BB = 1000,
  grid.size = 400,
  min.K = 5,
  n.cores = 7,
  plot.it = TRUE,
  xlab = "x",
  ylab = "y",
  Eyg.col = "red",
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.lines.col = "gray95",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  verbose = TRUE
)

fofam.test(
  x,
  y,
  two.sample.test = c("norm", "Wasserstein", "KS"),
  bw = NULL,
  n.perms = 1000,
  n.BB = 1000,
  grid.size = 400,
  min.K = 5,
  n.cores = 7,
  plot.it = TRUE,
  xlab = "x",
  ylab = "y",
  Eyg.col = "red",
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.lines.col = "gray95",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  verbose = TRUE
)
```

Arguments

`x`	A numeric vector of predictor values.
`y`	A numeric vector of response values (same length as x). Can be binary or continuous.
`two.sample.test`	Character string specifying the test for comparing distributions. Options are "Wasserstein", "KS", or "norm". Default is "norm".
`bw`	Numeric bandwidth parameter for smoothing. If NULL (default), bandwidth is automatically selected.
`n.perms`	Integer specifying the number of permutations for null distribution. Default is 1000.
`n.BB`	Integer specifying the number of Bayesian bootstrap samples. Default is 1000.
`grid.size`	Integer specifying the size of evaluation grid. Default is 400.

`min.K`	Integer specifying minimum number of observations for local estimation. Default is 5.
`n.cores`	Integer specifying the number of CPU cores for parallel computation. Default is 7.
`plot.it`	Logical indicating whether to produce diagnostic plots. Default is TRUE.
`xlab`	Character string for x-axis label. Default is "x".
`ylab`	Character string for y-axis label. Default is "y".
`Eyg.col`	Color specification for conditional mean curve. Default is "red".
`pt.col`	Color specification for data points. Default is "black".
`pt.pch`	Integer or character specifying point type. Default is 1.
`CrI.as.polygon`	Logical indicating whether to draw credible intervals as polygons (TRUE) or lines (FALSE). Default is TRUE.
`CrI.polygon.col`	Color specification for credible interval polygon. Default is "gray90".
`null.lines.col`	Color specification for null distribution lines. Default is "gray95".
`CrI.line.col`	Color specification for credible interval lines. Default is "gray".
`CrI.line.lty`	Line type for credible interval lines. Default is 2.
`verbose`	Logical indicating whether to print progress messages. Default is TRUE.

Details

The first-order functional association measure captures the variability in the derivative of the conditional mean function. Unlike the zero-order measure which looks at deviations from the mean, this measure is sensitive to the rate of change in the relationship between x and y. The test is performed by comparing the observed measure against a null distribution generated through permutation with Bayesian bootstrap for uncertainty quantification.

Value

An object of class "assoc1" containing:

`Delta1`	The total change in conditional mean (E_y(x_max) - E_y(x_min))
`delta1`	The first-order functional association measure
`p.value`	The p-value for the test of association
`log.p.value`	Natural logarithm of the p-value
`null.delta1`	Vector of null distribution values
`null.delta1.boxcox`	Box-Cox transformed null distribution (if applicable)
`BB.delta1`	Bayesian bootstrap samples of delta1
`BB.delta1.boxcox`	Box-Cox transformed bootstrap samples (if applicable)
`delta1.boxcox`	Box-Cox transformed observed delta1 (if applicable)
`null.Ey`	Matrix of null conditional mean estimates
`null.dEy`	Matrix of null conditional mean derivatives
`BB.dEy`	Matrix of bootstrap conditional mean derivatives
`Ey.res`	Results from conditional mean estimation

x	Input predictor values
y	Input response values
Ey	Conditional mean estimates at grid points
grid.size	Number of grid points used

Examples

```
## Not run:
# Generate example data with nonlinear relationship
set.seed(123)
n <- 200
x <- runif(n)
y <- sin(4*pi*x) + rnorm(n, sd = 0.2)

# Test for first-order functional association
result <- fassoc1.test(x, y, n.cores = 2, n.perms = 500)

# Plot derivative of conditional mean
plot(result, plot = "dExy")

## End(Not run)
```

`fb.magelo` *1D MAGELO (Model Averaged Grid-based Epsilon LOwess) with fixed boundary condition. Given a vector 'y', this routine finds 'predictions' such that the first element of 'predictions' equals the first element of 'y', and the last element of 'predictions' equals the last element of 'y'.*

Description

1D MAGELO (Model Averaged Grid-based Epsilon LOwess) with fixed boundary condition. Given a vector 'y', this routine finds 'predictions' such that the first element of 'predictions' equals the first element of 'y', and the last element of 'predictions' equals the last element of 'y'.

Usage

```
fb.magelo(
  x,
  y,
  grid.size = 400,
  degree = 1,
  min.K = 5,
  f = NULL,
  bw = NULL,
  min.bw.f = 0.025,
  method = ifelse(length(x) < 1000, "LOOCV", "CV"),
  n.bws = 100,
  n.BB = 1000,
  get.predictions.CrI = TRUE,
  get.gpredictions.CrI = TRUE,
  get.BB.predictions = FALSE,
  get.BB.gpredictions = FALSE,
  level = 0.95,
  n.C.itr = 0,
  C = 6,
  stop.C.itr.on.min = TRUE,
  n.cv.folds = 10,
  n.cv.reps = 20,
  nn.kernel = "epanechnikov",
  y.binary = FALSE,
  cv.nNN = 3,
  n.perms = 0,
  n.cores = 1,
  use.binloss = FALSE,
  verbose = FALSE,
  fb.C = 1
)
```

Arguments

`x`	A numeric vector of a predictor variable.
`y`	A numeric vector of an outcome variable.
`grid.size`	The number of grid points; default 400 which provides good balance between accuracy and computation speed.
`degree`	A degree of the polynomial of x in the linear regression; 0 means weighted mean, 1 is regular linear model lm(y ~ x), and deg = d is lm(y ~ poly(x, d)). The only allowed values are 0, 1 and 2.
`min.K`	The minimal number of x points that must be present in each disk neighborhood.
`f`	The proportion of the range of x that is used within a moving window to train the model. If NULL, the optimal value of f will be found using cross-validation optimization algorithm.
`bw`	A bandwidth parameter (radius of disk neighborhood).
`min.bw.f`	The min.bw factor, such that, min.bw = min.bw.f * dx, where dx <- diff(x.range). The default value is 0.025.
`method`	A method of estimating the optimal value of the bandwidth. Possible choices are "LOOCV" (for small datasets) and "CV".
`n.bws`	The number of bandwidths in the optimization process. Default: 100.
`n.BB`	The number of Bayesian bootstrap (BB) iterations for estimates of CI's.
`get.predictions.CrI`	A logical parameter. If TRUE, BB with quantile determined by the value of 'level' will be used to determine the upper and lower limits of CI's.
`get.gpredictions.CrI`	Set to TRUE if gpredictions.CI's need to be estimated.
`get.BB.predictions`	Set to TRUE if a matrix of Bayesian bootstrap estimates of predictions is to be returned.
`get.BB.gpredictions`	Set to TRUE if a matrix of Bayesian bootstrap estimates of gpredictions is to be returned.

`level`	A confidence level.
`n.C.itr`	The number of Cleveland's absolute residue based reweighting iterations for robust estimates of mean y values.
`C`	A scaling of |res| parameter changing |res| to |res|/C before applying ae.kernel to |res|'s.
`stop.C.itr.on.min`	A logical variable, if TRUE, the Cleveland's iterative reweighting stops when the maximum of the absolute values of differences of the old and new predictions estimates reach a local minimum.
`n.cv.folds`	The number of cross-validation folds. Used only when f = NULL. Default value: 10.
`n.cv.reps`	The number of repetitions of cross-validation. Used only when f = NULL. Default value: 20.
`nn.kernel`	A kernel for weighting neighbors. Options: "epanechnikov", "triangular", "tr.exponential", "normal".
`y.binary`	A logical variable. If TRUE, bw optimization is going to use a binary loss function mean(y(1-p) + (1-y)p).
`cv.nNN`	The number of nearest neighbors in interpolation used to find predictions given gpredictions in the cv routines.
`n.perms`	Number of y permutations for p-value calculation. Default: 0.
`n.cores`	Number of CPU cores for parallel processing. Default: 1.
`use.binloss`	Logical; if TRUE, use binary loss function for optimization.
`verbose`	Prints info about what is being done.
`fb.C`	The number of bw's away from the boundary points of the domain of x that the adjusting of gpredictions will take place, so that gpredictions satisfies the boundary condition.

Value

A list of class "magelo" containing input parameters as well as fitted values, credible intervals, bootstrap estimates, and model coefficients

---

`fig.E.geodesic.F` *Generates figures associated with E.geodesic.F estimates.*

Description

Generates figures associated with E.geodesic.F estimates.

Usage

```
fig.E.geodesic.F(save.file.prefix, rlabs, pics.file.prefix)
```

Arguments

`save.file.prefix`	A prefix of the save() files.
`rlabs`	Labels of the subject factors with levels.
`pics.file.prefix`	A prefix pdf output files.

Details

This function loads the results from E.geodesic.F estimation (saved as RDA files) and creates visualization plots saved as PDF files. For each factor label in rlabs, it looks for a file named `{save.file.prefix}{lab}.rda` and if found, generates a plot saved as `{pics.file.prefix}{lab}_Exy.pdf`.

The plots show the expected values (Ey) along the geodesic distance (Ex) with confidence intervals, providing a visual representation of how the factor varies along the geodesic path.

Value

NULL. This function is called for its side effects of generating PDF plots. One PDF file is created for each factor in rlabs that has a corresponding saved RDA file.

See Also

`E.geodesic.F` for generating the data files used by this function

`find.ascending.trajectory.from.point` *Find Ascending Trajectory from Point*

Description

Traces gradient ascent path from a starting point using analytical gradient.

Usage

```
find.ascending.trajectory.from.point(
  x,
  y,
  step,
  mixture,
  max_steps = 1000,
  grad_threshold = 1e-05,
  angle_threshold = -0.5,
  domain = list(x_min = 0, x_max = 1, y_min = 0, y_max = 1)
)
```

Arguments

`x`	Starting x coordinate
`y`	Starting y coordinate
`step`	Step size for gradient ascent
`mixture`	Object with gradient method
`max_steps`	Maximum number of steps (default 1000)
`grad_threshold`	Gradient magnitude threshold (default 1e-5)
`angle_threshold`	Angle change threshold (default -0.5)
`domain`	Domain boundaries as list

Value

Matrix of (x,y) coordinates along trajectory

`find.box.containing.x` *Finds the box within a box list covering some data set that contains a given point.*

Description

This function finds the sub-box within a given set of boxes that contains a specified point `x`.

Usage

```
find.box.containing.x(boxes, x)
```

Arguments

`boxes`	A list of boxes, where each box is a list containing vectors `L` and `R` representing the left and right boundaries of the box.
`x`	A vector representing the point for which the containing sub-box is to be found.

Value

A list representing the sub-box containing `x`, or `NULL` if `x` is not found within any sub-box.

Examples

```
## Not run:
boxes <- create.ED.boxes(0.5, c(0,0), c(1,1))
x <- c(0.2, 0.3)
containing_box <- find.box.containing.x(boxes, x)

## End(Not run)
```

`find.boxes.containing.x`
Finds boxes within a box list covering some data set that (after expansion by a factor p.exp) contain a given point.

Description

Finds boxes within a box list covering some data set that (after expansion by a factor p.exp) contain a given point.

Usage

```
find.boxes.containing.x(boxes, x, p.exp = 0.1, margin = NULL)
```

Arguments

`boxes`	A list of boxes, where each box is a list containing vectors `L` and `R` representing the left and right boundaries of the box.
`x`	A vector representing the point for which the containing sub-box is to be found.
`p.exp`	An expansion factor for the bounding box. Each edge of the box is scaled by the factor (1 + p.exp). Default is 0.1.
`margin`	An explicity expansion amount of the box in along each axis.

Value

A list representing the sub-box containing `x`, or `NULL` if `x` is not found within any sub-box.

Examples

```
## Not run:
boxes <- create.ED.boxes(0.5, c(0,0), c(1,1))
x <- c(0.2, 0.3)
containing_box <- find.box.containing.x(boxes, x)

## End(Not run)
```

`find.critical.points` *Find Critical Points in Function Grid*

Description

Identifies local maxima and minima in a gridded function.

Usage

```
find.critical.points(f.grid)
```

Arguments

`f.grid`	Matrix of function values

Value

List containing:

`maxima`	Matrix of (i,j) coordinates for local maxima
`minima`	Matrix of (i,j) coordinates for local minima

Examples

```
## Not run:
grid <- create.grid(50)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
critical <- find.critical.points(f.grid)
print(critical)

## End(Not run)
```

find.critical.points.continuous
Find Critical Points with Continuous Coordinates

Description

Identifies critical points (maxima, minima, saddles) using continuous gradient estimation.

Usage

```
find.critical.points.continuous(gradient, grid)
```

Arguments

gradient	Function returning gradient vector at (x,y)
grid	Grid object created by create.grid

Value

List containing matrices of critical point coordinates

find.descending.trajectory.from.point
Find Descending Trajectory from Point

Description

Traces gradient descent path from a starting point using analytical gradient.

Usage

```
find.descending.trajectory.from.point(
  x,
  y,
  step = 0.01,
  mixture,
  max_steps = 1000,
  grad_threshold = 1e-05,
  angle_threshold = -0.5,
  domain = list(x_min = 0, x_max = 1, y_min = 0, y_max = 1)
)
```

Arguments

x	Starting x coordinate
y	Starting y coordinate
step	Step size for gradient descent
mixture	Object with gradient method

`max_steps`	Maximum number of steps (default 1000)
`grad_threshold`	Gradient magnitude threshold (default 1e-5)
`angle_threshold`	Angle change threshold (default -0.5)
`domain`	Domain boundaries as list

Value

Matrix of (x,y) coordinates along trajectory

`find.gflow.basins` *Find Gradient-Flow Basins on a Weighted Graph*

Description

Identifies local minima and maxima of a scalar field on a graph and computes their basins via geodesic flow analysis.

Usage

```
find.gflow.basins(
  adj.list,
  weight.list,
  y,
  min.basin.size = 1L,
  min.path.size = 2L,
  q.edge.thld = 0.5
)
```

Arguments

`adj.list`	A list of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices must be 1-based.
`weight.list`	A list of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in `adj.list`. Must have the same structure as `adj.list`.
`y`	A numeric vector of length n, the response values at each graph vertex.
`min.basin.size`	An integer scalar, the minimum number of vertices a basin must contain to be considered significant. Default is 1.
`min.path.size`	An integer scalar, the minimum number of vertices along a shortest path for which monotonicity is tested. Default is 2.
`q.edge.thld`	A numeric scalar in $[0, 1]$, the quantile threshold for maximum allowed edge weight along geodesics. Default is 0.5.

Value

An object of class "gflow_basins", which is a list with the following components:

basins A list with two elements:

- **ascending** A list of local-minimum basins
- **descending** A list of local-maximum basins

Each basin is a list with components:

- **vertex** Integer, the 1-based vertex index of the extremum
- **value** Numeric, the function value at the extremum
- **basin** A matrix with columns "vertex" and "distance"
- **label** Character, a unique label for the extremum

local_extrema A data frame summarizing all detected extrema with columns:

- **vertex_index** Integer, 1-based vertex index
- **is_maximum** Integer, 1 if maximum, 0 if minimum
- **label** Character, unique label (e.g., "M1", "m1")
- **fn_value** Numeric, function value at extremum

See Also

summary.gflow_basins, find.local.extrema

Examples

```
# Create a simple graph
adj.list <- list(c(2, 3), c(1, 3, 4), c(1, 2, 4), c(2, 3))
weight.list <- list(c(1, 1), c(1, 1, 1), c(1, 1, 1), c(1, 1))
y <- c(1, 0, 2, 1.5)


# Find basins (requires compiled C++ backend)
result <- find.gflow.basins(adj.list, weight.list, y,
                            min.basin.size = 1,
                            min.path.size = 2,
                            q.edge.thld = 0.5)
summary(result)
```

---

find.inflection.pts *Finds inflection points of a vector*

Description

Identifies inflection points in a numeric vector by computing the second derivative using either the MAGELO (Monotonic Averaging with Gaussian Error and Linear Order) or MABILO (Monotonic Averaging with Binomial Error and Linear Order) smoothing methods. Inflection points are where the second derivative equals zero.

Usage

```
find.inflection.pts(y, x = NULL, method = c("magelo", "mabilo"))
```

Arguments

`y`	A numeric vector of values for which to find inflection points.
`x`	Optional numeric vector of x-coordinates corresponding to y values. If NULL (default), uses indices 1:length(y). Must be the same length as y.
`method`	Character string specifying the smoothing method to use. Either "magelo" (Gaussian error model) or `mabilo` (Binomial error model). Default is "magelo". Only the first element is used if a vector is provided.

Details

The function uses a three-step process:

1. Smooths the original data using the specified method
2. Computes and smooths the first derivative (differences of smoothed values)
3. Computes and smooths the second derivative
4. Finds zeros of the second derivative using root-finding

If x values are provided, the data is first sorted by x before analysis.

Value

A list containing:

- `y`: The sorted y values (reordered if x was provided)
- `r0`: Result from the initial smoothing of y
- `r1`: Result from smoothing the first derivative
- `r2`: Result from smoothing the second derivative
- `infl.pts`: Integer indices of inflection points in the sorted data
- `infl.pts.vals`: The y values at the inflection points

Note

- Requires the 'rootSolve' package for finding zeros of the second derivative
- Also requires either the 'magelo' or `mabilo` function to be available
- The inflection point indices are floored to integers, which may introduce slight imprecision
- When x is NULL, the function sorts y values, which changes the interpretation of results

See Also

`uniroot.all` for root finding, `magelo` and `mabilo` for the smoothing methods

Examples

```
## Not run:
# Example 1: Find inflection points in a sigmoid-like curve
x <- seq(-5, 5, length.out = 100)
y <- 1 / (1 + exp(-x)) + 0.1 * sin(2*x)  # Sigmoid with oscillation

result <- find.inflection.pts(y, x, method = "magelo")

# Plot the results
```

```
plot(x, y, type = "l", main = "Inflection Points")
points(x[result$infl.pts], result$infl.pts.vals, col = "red", pch = 19)

# Example 2: Using indices as x-coordinates
y <- c(1, 2, 5, 10, 15, 18, 19, 19.5, 19.8, 20)  # Growth curve
result <- find.inflection.pts(y, method = "mabilo")
print(result$infl.pts)

## End(Not run)
```

find.line.intersection *Find Line Intersection*

Description

Finds the intersection point of two line segments.

Usage

```
find.line.intersection(P1, P2, Q1, Q2)
```

Arguments

P1	First point of first line segment c(x, y)
P2	Second point of first line segment c(x, y)
Q1	First point of second line segment c(x, y)
Q2	Second point of second line segment c(x, y)

Value

Coordinates of intersection point or NULL if no intersection

find.local.extrema *Detect Local Extrema on a Graph with Labeled Basins*

Description

Identifies local minima and maxima of a scalar function defined on the vertices of a graph, using a basin-based definition of extrema.

Usage

```
find.local.extrema(adj.list, weight.list, y, min.basin.size = 1L)
```

Arguments

`adj.list`	A list of integer vectors, where each element represents the neighbors of a vertex (1-based indices). Must match the length of `y`.
`weight.list`	A list of numeric vectors representing edge weights corresponding to the adjacency list structure.
`y`	A numeric vector of function values defined on the vertices of the graph.
`min.basin.size`	An integer specifying the minimum number of vertices required for a valid basin to be considered an extremum. Default is 1.

Details

This function uses a simpler algorithm than `find.gflow.basins`, focusing on local extrema detection without the geodesic flow analysis. Each extremum is associated with a basin of attraction obtained via a gradient flow process.

Value

An object of class `"local_extrema_basins"`, which is a list with the following components:

basins A list with elements `ascending` (minima) and `descending` (maxima), each containing a list of basins with vertex index, function value, and basin information.

local_extrema A data frame summarizing all detected extrema with columns: `vertex_index`, `is_maximum`, `label`, and `fn_value`.

See Also

`find.gflow.basins` for more comprehensive basin analysis

Examples

```
# Create a simple graph
adj.list <- list(c(2, 3), c(1, 3, 4), c(1, 2, 4), c(2, 3))
weight.list <- list(c(1, 1), c(1, 1, 1), c(1, 1, 1), c(1, 1))
y <- c(1, 0, 2, 1.5)


# Find local extrema (requires compiled C++ backend)
result <- find.local.extrema(adj.list, weight.list, y, min.basin.size = 1)
print(result$local_extrema)
```

`find.local.minima` *Find Local Minima in Distance Sequence*

Description

Identifies local minima in a sequence of edit distances, which can indicate stable graph configurations.

Usage

```
find.local.minima(
  distances,
  k.values = NULL,
  threshold = 0,
  na.rm = FALSE,
  sort.by.prominence = TRUE
)
```

Arguments

`distances`	Numeric vector of distances.
`k.values`	Optional numeric vector of corresponding k values. If NULL, indices are used.
`threshold`	Numeric value for the minimum prominence of a local minimum to be considered significant (default: 0, all local minima).
`na.rm`	Logical. Should NA values be handled? If TRUE, NA values are interpolated; if FALSE (default), function fails if NA values are present.
`sort.by.prominence`	Logical. Should results be sorted by prominence? If TRUE (default), results are sorted with most prominent first. If FALSE, results are returned in order of occurrence.

Details

A local minimum is defined as a point that is smaller than both its neighbors. Prominence is calculated as the minimum height the function must climb to reach a higher value.

If `na.rm` = `TRUE`, NA values are linearly interpolated before finding minima. Minima at originally NA positions are excluded from results.

Value

A data frame with columns:

`index`	Integer indices of local minima
`k`	Corresponding k values (or indices if k.values is NULL)
`distance`	Distance values at local minima
`prominence`	Prominence of each local minimum

Examples

```
# Create sample data with clear minima
k <- 1:20
dist <- sin(k / 3) + 0.1 * rnorm(20)

# Find local minima
minima <- find.local.minima(dist, k)
print(minima)

# Plot with minima marked
plot(k, dist, type = "b")
points(minima$k, minima$distance, col = "red", pch = 19, cex = 1.5)
```

```
# Example with NA handling
dist_na <- dist
dist_na[c(5, 15)] <- NA
minima_na <- find.local.minima(dist_na, k, na.rm = TRUE)
```

find.longest.true.stretch

Finds the Longest Stretch of Consecutive TRUE Values in a Logical Vector

Description

This function takes a logical vector as input and returns the starting index and length of the longest subinterval where the values are TRUE.

Usage

```
find.longest.true.stretch(l)
```

Arguments

`l`	A logical vector.

Value

A list with two elements:

start_index The index at which the longest stretch of consecutive TRUE values starts.

length The length of the longest stretch of consecutive TRUE values.

Note

If there are multiple disjoined stretch of TRUE values of the same length, then the first one is returned.

Examples

```
l <- c(TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE)
result <- find.longest.true.stretch(l)
print(result)
# $start_index
# [1] 4
# $length
# [1] 3

## Two longest TRUE stretches
l <- c(TRUE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE)
find.longest.true.stretch(l)
# $start.index
# [1] 1
# $length
# [1] 3
```

```
## No TRUE values
l <- c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)
find.longest.true.stretch(l)
# $start.index
# [1] NA
# $length
# [1] 0
```

`find.optimal.k` *Find Optimal k Parameter Using Edge Persistence Analysis*

Description

Analyzes the stability of graph structures across different k values by examining the persistence patterns of edges. The function combines multiple stability metrics to identify the k value that produces the most stable graph structure.

Usage

```
find.optimal.k(birth.death.matrix, kmin, kmax, matrix_type = "geom")
```

Arguments

`birth.death.matrix`	A numeric matrix with columns "birth_time" and "death_time" containing the birth and death times of edges. Each row represents one edge. For edges that persist through kmax, death_time equals kmax + 1.
`kmin`	The minimum k value to consider
`kmax`	The maximum k value to consider
`matrix_type`	Character string describing the type of matrix (default: "geom"). Used for informative warning messages.

Details

The function calculates stability scores for each k value using three components:

1. Average persistence (lifetime) of edges present at k
2. Proportion of persistent edges (those that continue to kmax+1)
3. Edge stability measure that penalizes recently born or soon-to-die edges

The final stability score for each k is the product of these three components. Higher scores indicate more stable graph structures.

Value

A list containing:

stability.scores Numeric vector of stability scores for each k value

k.values Vector of k values analyzed (kmin:kmax)

opt.k The k value with highest stability score

See Also

`create.iknn.graphs` for generating the birth-death matrix

Examples

```
## Not run:
# Assuming we have birth-death data from create.iknn.graphs()
result <- create.iknn.graphs(X, kmin = 3, kmax = 10)

# Find optimal k
stability <- find.optimal.k(result$birth_death_matrix, kmin = 3, kmax = 10)

# Plot stability scores
plot(stability$k.values, stability$stability.scores, type = "l",
     xlab = "k", ylab = "Stability Score")
abline(v = stability$opt.k, col = "red", lty = 2)

## End(Not run)
```

`find.points.within.box` *Find the points of a set X within a box.*

Description

This function finds ids of the points of X that are found within the given box.

Usage

```
find.points.within.box(X, box, eps)
```

Arguments

X	A numeric matrix corresponding to a set of points in some Euclidean space.
box	A matrix or data frame containing the points in the space.
eps	A distance threshold (non-negative real number) such that each box is expanded by eps and then tested for the presence of the elements of X.

Value

A vector of row names of X corresponding to the points fund within the given box.

Examples

```
## Not run:
X <- matrix(runif(100), ncol = 2)
box <- list()
box$L <- c(0.2, 0.2)
box$R <- c(0.5, 0.5)
points_within_box <- find.points.within.box(X, box, eps = 0)

## End(Not run)
```

`find.separatrices` *Find Separatrices from Saddle Points*

Description

Computes separatrices emanating from saddle points.

Usage

```
find.separatrices(saddle_point, mixture, eps = 0.01, step = 0.01)
```

Arguments

`saddle_point`	Coordinates of saddle point c(x, y)
`mixture`	Object with gradient method
`eps`	Small offset from saddle (default 0.01)
`step`	Step size for trajectory (default 0.01)

Value

List of separatrix trajectories

`find.shortest.paths.within.radius` *Find All Shortest Paths Within a Radius*

Description

Finds all shortest paths from a starting vertex to other vertices in a weighted graph, limited by a maximum distance (radius). This function implements a modified version of Dijkstra's algorithm optimized for local neighborhood exploration.

Usage

```
find.shortest.paths.within.radius(adj.list, weight.list, start, radius)
```

Arguments

`adj.list`	A list of integer vectors, where each vector contains the indices of vertices adjacent to the vertex at that position in the list. For example, `adj.list[[1]]` contains the indices of vertices connected to vertex 1.
`weight.list`	A list of numeric vectors containing the weights of edges in the corresponding `adj.list`. `weight.list[[i]][j]` is the weight of the edge between vertex i and vertex `adj.list[[i]][j]`.
`start`	The index of the starting vertex (1-based indexing as is standard in R).
`radius`	The maximum allowed distance for paths from the start vertex. Paths with a total weight greater than this value will not be included in the results.

Details

The function converts the R graph representation to a C++ graph, and then uses an efficient implementation of a bounded Dijkstra's algorithm to find all shortest paths within the specified radius. The algorithm works in two phases:

1. A bounded Dijkstra's algorithm to find distances and predecessors
2. Path reconstruction to generate actual paths from the collected information

The resulting paths are sorted in descending order of their total weights.

Value

A list containing:

`paths`	A list of integer vectors, where each vector represents a path from the start vertex to another vertex in the graph. Each element of the vector is a vertex index.
`reachable.vertices`	An integer vector containing all vertices that are reachable from the start vertex within the given radius.
`vertex.to.path.map`	A matrix with three columns: "vertex", "path.index", and "total.weight". Each row maps a reachable vertex to its corresponding path in the paths list and includes the total weight (distance) to that vertex. This provides efficient O(1) lookup for finding the shortest path to any specific vertex.

See Also

For graph creation and manipulation, consider using the `igraph` package.

Examples

```
# Create a simple graph with 5 vertices
adj.list <- list(c(2, 3), c(1, 3, 4), c(1, 2, 5), c(2, 5), c(3, 4))
weight.list <- list(c(1, 2), c(1, 1, 3), c(2, 1, 2), c(3, 1), c(2, 1))

# Find all shortest paths within radius 3 from vertex 1
(res <- find.shortest.paths.within.radius(adj.list, weight.list, 1, 3))
```

---

find_optimal_breakpoints

Find Optimal Number of Breakpoints

Description

Find Optimal Number of Breakpoints

Usage

```
find_optimal_breakpoints(
  x,
  y,
  max_breakpoints = 5,
  method = c("aic", "bic", "davies"),
  alpha = 0.05
)
```

Arguments

`x`	Numeric vector containing the predictor variable values
`y`	Numeric vector containing the response variable values
`max_breakpoints`	Maximum number of breakpoints to consider (default = 5)
`method`	Character string specifying the selection method ("aic", "bic", or "davies")
`alpha`	Significance level for Davies' test (default = 0.05)

Value

A list containing the optimal number of breakpoints and model comparison results

`fit.pwlm` *Fit a Piecewise Linear Model*

Description

Fits a piecewise linear model to data using the segmented package. Can fit models with single or multiple breakpoints. If the segmented regression fails, falls back to simple linear regression.

Usage

```
fit.pwlm(x, y, n_breakpoints = 1, breakpoint_bounds = NULL)
```

Arguments

`x`	Numeric vector containing the predictor variable values
`y`	Numeric vector containing the response variable values
`n_breakpoints`	Integer specifying the number of breakpoints to estimate (default = 1)
`breakpoint_bounds`	List of vectors specifying the bounds for each breakpoint (optional)

Value

An object of class "pwlm" containing:

model	Either a segmented model object or a linear model object if segmented regression failed
breakpoints	Numeric vector of estimated breakpoints. NA if using fallback linear model
x	The original x values
y	The original y values
type	Character string indicating "segmented" or "linear"
n_breakpoints	Number of breakpoints specified

Examples

```
# Generate sample data
x <- 1:20
y <- x + 2*x^2 + rnorm(20, 0, 10)

# Single breakpoint
fit1 <- fit.pwlm(x, y)

# Two breakpoints
fit2 <- fit.pwlm(x, y, n_breakpoints = 2)
```

---

fit.pwlm.optimal   *Fit a Piecewise Linear Model with Optimal Number of Breakpoints*

Description

Fits a piecewise linear model to data, automatically determining the optimal number of breakpoints using model selection criteria.

Usage

```
fit.pwlm.optimal(
  x,
  y,
  max_breakpoints = 5,
  method = c("aic", "bic", "davies"),
  alpha = 0.05,
  plot_selection = FALSE
)
```

Arguments

x	Numeric vector containing the predictor variable values
y	Numeric vector containing the response variable values
max_breakpoints	Maximum number of breakpoints to consider (default = 5)
method	Character string specifying the selection method ("aic", "bic", or "davies")
alpha	Significance level for Davies' test (default = 0.05)
plot_selection	Logical indicating whether to plot selection criteria (default = FALSE)

Value

An object of class "pwlm" with additional component 'selection_results'

---

fitted.graph.spectral.lowess

Extract Fitted Values from Graph Spectral LOWESS

Description

Extracts the fitted (smoothed) values from a graph spectral LOWESS object

Usage

```
## S3 method for class 'graph.spectral.lowess'
fitted(object, ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`...`	Additional arguments (currently unused)

Value

Numeric vector of fitted values

---

fitted.graph.spectral.ma.lowess

Extract Fitted Values from Graph Spectral MA LOWESS

Description

Extracts the fitted (smoothed) values from a graph spectral model-averaged LOWESS object. These values represent the weighted average across multiple bandwidth models.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
fitted(object, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

Numeric vector of fitted values

`fitted.mabilog` *Extract Fitted Values from Mabilog Model*

Description

Extracts fitted probability values from a mabilog object

Usage

```
## S3 method for class 'mabilog'
fitted(object, type = c("response", "link"), ...)
```

Arguments

`object`	A 'mabilog' object
`type`	Character string specifying which values to return: "response" for probabilities (default), "link" for logit scale
`...`	Additional arguments (currently unused)

Value

Numeric vector of fitted values

`fitted.mabilo_plus` *Extract Fitted Values from Mabilo Plus Model*

Description

Extracts fitted values from a mabilo_plus object

Usage

```
## S3 method for class 'mabilo_plus'
fitted(object, type = c("ma", "sm", "both"), ...)
```

Arguments

`object`	A 'mabilo_plus' object
`type`	Character string specifying which predictions to return: "ma" (default), "sm", or "both"
`...`	Additional arguments (currently unused)

Value

Numeric vector or matrix of fitted values

function.aware.weights.plot

Plot Weight Modification Comparison

Description

Creates a visualization comparing how different weighting schemes modify edge weights based on function value differences.

Usage

```
function.aware.weights.plot(
  adj.list,
  weight.list,
  function.values,
  weight.types = 0:5,
  epsilon = 1e-06,
  lambda = 1,
  alpha = 1,
  beta = 5,
  tau = 0.1,
  p = 2,
  q = 2,
  r = 2,
  log.scale = TRUE
)
```

Arguments

`adj.list`	A list where each element contains the indices of vertices adjacent to vertex i.
`weight.list`	A list of the same structure as adj.list, with edge weights.
`function.values`	Vector of function values at each vertex
`weight.types`	Vector of weight types to analyze (default: 0:5)
`epsilon`	Small constant to avoid division by zero (for weight.type 0)
`lambda`	Decay rate parameter (for weight.type 2)
`alpha`	Power law exponent (for weight.type 3) or scaling factor (for weight.type 5)
`beta`	Sigmoid steepness parameter (for weight.type 4)
`tau`	Sigmoid threshold parameter (for weight.type 4)
`p`	Power for feature distance term (for weight.type 5)
`q`	Power for function difference term (for weight.type 5)
`r`	Power for the overall normalization (for weight.type 5)
`log.scale`	Whether to use log scale for the y-axis

Value

Invisibly returns the data frame used for plotting `function.contours.plot` *Plot Function Contours*

Description

Creates a contour plot of function values on a grid.

Usage

```
function.contours.plot(
  grid,
  f.grid,
  main = "Function Contours",
  nlevels = 20,
  xlab = "",
  ylab = "",
  axes = FALSE,
  add = FALSE,
  ...
)
```

Arguments

`grid`	Grid object from create.grid
`f.grid`	Matrix of function values
`main`	Plot title
`nlevels`	Number of contour levels (default 20)
`xlab`	X-axis label
`ylab`	Y-axis label
`axes`	Logical, whether to show axes
`add`	Logical, whether to add to existing plot
`...`	Additional arguments passed to contour

Value

Invisible NULL

Examples

```
## Not run:
grid <- create.grid(50)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
function.contours.plot(grid, f.grid)

## End(Not run)
```

gaussian.mixture.xD *Creates a Synthetic Function with Specified Number of Local Maxima over a subset of R^d, d > 1.*

Description

Generates a synthetic function with specified number of local maxima using a mixture of Gaussians and a partition of unity. This function is useful for creating complex, non-linear synthetic data for analysis and testing.

Usage

```
gaussian.mixture.xD(
  S,
  X.lmax = NULL,
  n.lmax = 3,
  y.lmax = NULL,
  y.min = 1,
  y.max = 5,
  C = 2,
  q = 1,
  with.partition.of.unity = FALSE
)
```

Arguments

`S`	A matrix or data frame of points over which a the values of the constructed synthetic function will be evaluated.
`X.lmax`	A matrix or data frame of the localtion of local maxima.
`n.lmax`	The number of local maxima. Used when X.lmax is NULL. If this is the case, a random set of n.lmax rows of S is taken to be X.lmax.
`y.lmax`	A vector of values the function being created supposet to have at the local maxima. A function that will be constructed is designed to have the values at the local maxima specified by y.lmax, but there are no warranty that it is going to be the case.
`y.min`	Minimum y value for local maxima (defaults to 1).
`y.max`	Maximum y value for local maxima (defaults to 5).
`C`	A scaling factor to control the spread of the bump functions. Defaults to 2.
`q`	A q parameter in radial q-exponential Gaussian functions. Controls the steepness of the radial Gaussian function.
`with.partition.of.unity`	Set to TRUE to use a partition of unity.

Details

The function generates a set of local maxima and their corresponding y-values. It then constructs a partition of unity and a matrix of radial Gaussian functions centered at these local maxima. The final synthetic function is the sum of these radial Gaussians weighted by the partition of unity.

Value

A list containing the following components:

- `X.lmax`: A matrix of the locations of the local maxima.
- `y.lmax`: A vector of function values at the local maxima.
- `y`: A vector of values of the created synthetic function.

`gaussian.second.derivative`
Calculate Second Derivative of Gaussian Function

Description

This function computes the second derivative of the unnormalized Gaussian function:

$$f(x) = \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

with respect to x. The second derivative identifies inflection points of the Gaussian curve, which occur at $\mu \pm \sigma$.

Usage

```
gaussian.second.derivative(x, mu, sd)
```

Arguments

`x`	Numeric vector of x values at which to evaluate the second derivative
`mu`	Mean parameter (location) of the Gaussian function
`sd`	Standard deviation parameter (scale) of the Gaussian function. Must be positive.

Details

The second derivative is computed using the formula:

$$f''(x) = \frac{(z^2 - 1) \cdot \exp(-z^2/2)}{\sigma^2}$$

where $z = (x - \mu)/\sigma$ is the standardized value.

Key properties:

- Zero at x = mu ± sd (inflection points)
- Maximum at x = mu (where f''(mu) = -1/sd^2)
- Approaches 0 as $x \to \pm\infty$

Value

Numeric vector of second derivative values with the same length as x. The second derivative equals zero at the inflection points (x = mu ± sd), is negative between them (concave down), and positive outside them (concave up).

Note

This computes the second derivative of the unnormalized Gaussian (without the $1/(\sigma\sqrt{2\pi})$ normalization factor). For the normalized version, multiply the result by $1/(\sigma\sqrt{2\pi})$.

See Also

dnorm for the normalized Gaussian density function, find.inflection.pts for finding inflection points in general curves

Examples

```
# Example 1: Basic usage
x <- seq(-5, 5, length.out = 100)
result <- gaussian.second.derivative(x, mu = 0, sd = 1)

# Example 2: Verify inflection points at mu ± sd
mu <- 2
sd <- 1.5
x_inflection <- c(mu - sd, mu + sd)
gaussian.second.derivative(x_inflection, mu, sd)  # Should be ~0

# Example 3: Visualize the Gaussian and its second derivative
x <- seq(-4, 4, length.out = 200)
f <- exp(-(x^2)/2)  # Original function (mu=0, sd=1)
f_double_prime <- gaussian.second.derivative(x, mu = 0, sd = 1)

par(mfrow = c(2, 1))
plot(x, f, type = "l", main = "Gaussian Function", ylab = "f(x)")
abline(v = c(-1, 1), col = "red", lty = 2)  # Inflection points
plot(x, f_double_prime, type = "l", main = "Second Derivative", ylab = "f''(x)")
abline(h = 0, col = "gray")
abline(v = c(-1, 1), col = "red", lty = 2)  # Zero crossings
```

---

gaussian.xD *Gaussian function in arbitrary dimension*

Description

Gaussian function in arbitrary dimension

Usage

```
gaussian.xD(x, mu, sigma, C)
```

Arguments

x	Numeric vector of input values
mu	Numeric vector of means (same length as x)
sigma	Numeric vector of standard deviations (same length as x)
C	Amplitude/scaling factor

Value

Numeric value of the Gaussian function

`gaussian.xD.grad` *Gradient of the Gaussian function in arbitrary dimension*

Description

Gradient of the Gaussian function in arbitrary dimension

Usage

```
gaussian.xD.grad(x, mu, sigma, C)
```

Arguments

`x`	Numeric vector of input values
`mu`	Numeric vector of means (same length as x)
`sigma`	Numeric vector of standard deviations (same length as x)
`C`	Amplitude/scaling factor

Value

Numeric vector of gradient values

`gdensity` *Kernel Density Estimation on a Uniform Grid*

Description

Estimates probability density using kernel density estimation on a uniform grid. Similar to base R's density() but uses a fixed uniform grid for estimation.

Usage

```
gdensity(
  x,
  grid.size = 512L,
  poffset = 0.1,
  bw = 0,
  kernel = 5L,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of observations
`grid.size`	Integer specifying the number of points in the estimation grid (default: 512)
`poffset`	Numeric value between 0 and 1 specifying the proportion of data range to add as padding on each end (default: 0.1)
`bw`	Bandwidth for kernel estimation. If <= 0, bandwidth is automatically selected (default: 0)
`kernel`	Integer specifying kernel type: 1 = Epanechnikov, 2 = Triangular, 4 = Laplace, 5 = Normal, 6 = Biweight, 7 = Tricube
`verbose`	Logical indicating whether to print progress information (default: FALSE)

Value

A list containing:

- density - Vector of density estimates at grid points
- bw - Bandwidth used for estimation
- bw_auto_selected - Logical indicating if bandwidth was automatically selected
- offset - Actual offset added to data range
- start - Starting point of the grid
- end - End point of the grid

Examples

```
data <- rnorm(100)
result <- gdensity(data, grid.size = 200, poffset = 0.1, bw = 0, kernel = 5)
plot(result$x, result$density, type = "l")
```

`generate.1d.circular.gaussian.mixture`
Generate a 1D Circular Gaussian Mixture

Description

This function generates a smooth, periodic function over a circle by creating a mixture of Gaussian distributions. The function ensures continuity and smoothness at all points, including the 0/2pi connection point.

Usage

```
generate.1d.circular.gaussian.mixture(
  n.points = 100,
  x.knots = c(0, pi, 1.5 * pi),
  y.knots = c(5, 8, 2.5),
  sd.knot = 0.5,
  out.dir = NULL
)
```

Arguments

`n.points`	An integer specifying the number of points to generate along the circle. Must be positive. Default is 100.
`x.knots`	A numeric vector specifying the positions of the Gaussian centers on the circle, in radians. Values are automatically wrapped to `[0, 2pi]`. Must have the same length as y.knots.
`y.knots`	A numeric vector specifying the heights of the Gaussian centers. Must have the same length as x.knots.
`sd.knot`	A positive numeric value specifying the standard deviation for all Gaussian distributions. Default is 0.5.
`out.dir`	An optional character string specifying the directory to save the results. If NULL (default), results are not saved to a file.

Value

A list containing the following elements:

`x`	A numeric vector of x values (angles in radians) from 0 to 2*pi.
`y`	A numeric vector of corresponding y values of the mixture function.
`file.name`	A character string with the name of the file where results are saved (if out.dir is specified).
`params`	A list of all input parameters used to generate the mixture.

Examples

```
# Generate a simple circular Gaussian mixture
result <- generate.1d.circular.gaussian.mixture(
  n.points = 200,
  x.knots = c(0, pi, 1.5*pi),
  y.knots = c(5, 8, 2.5),
  sd.knot = 0.3
)

# Plot the result
plot(result$x, result$y, type = "l", xlab = "Angle (radians)", ylab = "Value")
```

`generate.1d.gaussian.mixture` *Generate a 1D Gaussian Mixture with Direct Knot Specification*

Description

A convenience function that generates a 1D Gaussian mixture by directly specifying component positions and amplitudes. This function provides a simpler interface to create Gaussian mixtures when exact component locations are known.

Usage

```
generate.1d.gaussian.mixture(
  n.points = 100,
  x.knots = c(-5, 0, 10),
  y.knots = c(5, 8, 2.5),
  sd.knot = 1,
  x.offset = 3,
  add.noise = FALSE,
  noise.fraction = 0.1,
  out.dir = NULL,
  verbose = FALSE
)
```

Arguments

`n.points`	Integer. The number of points in the output grid. Default is 100.
`x.knots`	Numeric vector. The x-coordinates of the Gaussian components' centers. Default is c(-5, 0, 10).
`y.knots`	Numeric vector. The amplitudes of the Gaussian components. Must have the same length as x.knots. Default is c(5, 8, 2.5).
`sd.knot`	Numeric. The standard deviation for all Gaussian components. Default is 1.0.
`x.offset`	Numeric. The range offset to extend beyond the minimum and maximum x.knots. Default is 3.
`add.noise`	Logical. Whether to add noise to the generated mixture. Default is FALSE.
`noise.fraction`	Numeric. If add.noise is TRUE, the noise standard deviation as a fraction of the y range. Default is 0.1.
`out.dir`	Character string or NULL. If provided, specifies the directory to save the results. Default is NULL (results not saved).
`verbose`	Logical. Whether to print the output file path if saving. Default is FALSE.

Details

This function provides a simplified interface for generating Gaussian mixtures when the user wants to specify exact component locations and amplitudes. Since `get.gaussian.mixture` doesn't support custom x positions directly, this function generates the mixture by evaluating the Gaussian components at the specified positions. It's particularly useful for creating test cases or reproducing specific mixture configurations.

Note: Unlike `get.gaussian.mixture`, this function directly computes the mixture values rather than using positioning strategies.

Value

A list containing:

`x`	Numeric vector. The x-coordinates of the generated points.
`y`	Numeric vector. The y-coordinates of the generated Gaussian mixture.
`y.true`	Numeric vector. The true y-coordinates without noise (if noise added).
`x.knots`	Numeric vector. The x-coordinates of the components.
`y.knots`	Numeric vector. The amplitudes of the components.

`sd.knots`	Numeric vector. The standard deviations of the components.
`file.name`	Character string. The file name if results were saved, NULL otherwise.
`params`	List. All input parameters used to generate the mixture.

See Also

`get.gaussian.mixture` for more flexible mixture generation options

Examples

```
# Generate mixture with default parameters
result1 <- generate.1d.gaussian.mixture()
plot(result1$x, result1$y, type = 'l', main = "Default 1D Gaussian Mixture")

# Generate mixture with custom parameters
result2 <- generate.1d.gaussian.mixture(
  n.points = 200,
  x.knots = c(-10, 0, 5, 15),
  y.knots = c(2, 5, 8, 3),
  sd.knot = 0.8,
  x.offset = 5
)

# Generate mixture with noise
result3 <- generate.1d.gaussian.mixture(
  x.knots = c(-2, 2),
  y.knots = c(1, -1),
  add.noise = TRUE,
  noise.fraction = 0.15
)
```

`generate.binary.sample.from.gaussian.mixture`
Generate Binary Sample from Gaussian Mixture

Description

Transforms a Gaussian mixture into a probability function using logistic transformation and generates binary samples based on these probabilities. This is useful for creating synthetic binary response data with complex non-linear relationships.

Usage

```
generate.binary.sample.from.gaussian.mixture(
  gaussian.mixture.result,
  n.sample.points = 50,
  sample.method = c("uniform", "regular", "custom"),
  x.sample = NULL,
  use.true.y = TRUE,
  transform.method = c("logit", "direct"),
  logit.range = c(-3, 3),
  out.dir = NULL,
  verbose = FALSE,
  seed = NULL
)
```

Arguments

`gaussian.mixture.result`	List or character. Either the result from `generate.1d.gaussian.mixture` or `get.gaussian.mixture`, or a path to a file containing such results.
`n.sample.points`	Integer. Number of points to sample. Default is 50.
`sample.method`	Character. Method for selecting sample points: • "uniform": Random uniform sampling within x range (default) • "regular": Equally spaced points • "custom": User-specified x locations (requires x.sample parameter)
`x.sample`	Numeric vector. Custom x locations when sample.method = "custom".
`use.true.y`	Logical. If TRUE, use y.true (without noise) if available. Default is TRUE.
`transform.method`	Character. Method to transform y values to probabilities: • "logit": Normalize to logit.range then apply inverse logit (default) • "direct": Normalize directly to $[0, 1]$
`logit.range`	Numeric vector of length 2. When transform.method = "logit", the range to normalize y values before applying inverse logit. Default is c(-3, 3).
`out.dir`	Character or NULL. Directory to save results. If NULL, results are not saved. Default is NULL.
`verbose`	Logical. Whether to print file path when saving. Default is FALSE.
`seed`	Integer or NULL. Random seed for reproducibility. Default is NULL.

Details

The function performs the following steps:

1. Extracts the smooth function from the Gaussian mixture
2. Selects sample points according to sample.method
3. Interpolates y values at sample points
4. Transforms y values to probabilities using specified method
5. Generates binary outcomes using binomial sampling

For the logit transformation method, y values are first normalized to logit.range, then the inverse logit function is applied: p = 1/(1 + exp(-y_normalized)). This creates a smooth probability function that approaches 0 and 1 asymptotically.

Value

A list containing:

`x`	Numeric vector. The x-coordinates of the sampled points.
`y.binary`	Integer vector. The binary sample (0 or 1).

`y.prob`	Numeric vector. The probability of Y=1 for each point.
`y.smooth`	Numeric vector. The interpolated y values before transformation.
`gaussian.mixture`	List. The original Gaussian mixture result.
`params`	List. Parameters used for generation.

Examples

```
# First generate a Gaussian mixture
gm <- generate.1d.gaussian.mixture(
  x.knots = c(-2, 0, 2),
  y.knots = c(-1, 2, -1.5)
)

# Generate binary sample with default settings
binary.data <- generate.binary.sample.from.gaussian.mixture(gm)

# Plot the results
plot(gm$x, gm$y, type = 'l', col = 'blue',
     main = "Gaussian Mixture and Binary Sample")
points(binary.data$x, binary.data$y.binary, pch = 19,
       col = ifelse(binary.data$y.binary == 1, "green", "red"))
lines(binary.data$x, binary.data$y.prob, col = 'orange', lwd = 2)

# Generate regular grid sample with wider logit range
binary.regular <- generate.binary.sample.from.gaussian.mixture(
  gm,
  n.sample.points = 100,
  sample.method = "regular",
  logit.range = c(-5, 5)  # Steeper probability transitions
)
```

`generate.circle` *Generate Equally Spaced Points on a Circle without any noise.*

Description

This function creates a set of points that are equally spaced along a circle. The first and last point are the same.

Usage

```
generate.circle(n, radius = 1)
```

Arguments

`n`	The desired number of points to generate.
`radius`	The radius of the circle (default is 1).

Value

A data frame with two columns, x and y, containing the coordinates of the generated points.

`generate.circle.data` *Generate Points on a Circle*

Description

This function creates a set of points along a circle. The points can be either equally spaced or randomly distributed along the circumference. Optional Gaussian or Laplace noise can be added to the radial component.

Usage

```
generate.circle.data(
  n,
  radius = 1,
  noise = 0.1,
  type = "random",
  noise.type = "laplace",
  seed = NULL
)
```

Arguments

`n`	An integer specifying the desired number of points to generate.
`radius`	A positive numeric value specifying the radius of the circle (default is 1).
`noise`	A non-negative numeric value specifying the level of noise to add to the points (default is 0.1).
`type`	A character string, either "uniform" for equally spaced points or "random" for randomly distributed points along the circle.
`noise.type`	A character string, either "normal" for Gaussian noise or "laplace" for Laplace (double exponential) noise.
`seed`	An integer for the random seed. Default is NULL.

Value

A data frame with two columns, `x` and `y`, containing the coordinates of the generated points.

Examples

```
# Generate 100 equally spaced points on a circle with radius 2
df <- generate.circle.data(100, radius = 2)

# Generate 50 random points with Laplace noise
df_noisy <- generate.circle.data(50, noise = 0.1, type = "random", noise.type = "laplace")
```

`generate.circle.graph` *Generate a Circle Graph*

Description

Creates a circle graph where each vertex is connected to the vertices to its left and right. The edge weight is the angular distance (shortest angle) between connected vertices.

Usage

```
generate.circle.graph(n, type = "random", seed = NULL)
```

Arguments

`n`	A positive integer specifying the number of vertices in the graph.
`type`	Character string specifying the type of angle distribution. Either "uniform" for equally spaced angles or "random" for random angles. Default is "random".
`seed`	Optional seed for the random number generator. Default is NULL.

Value

A list with two components:

`adj.list`	Adjacency list of the circle graph. Each element `adj.list[[i]]` contains the vertices adjacent to vertex i.
`weight.list`	Weight list corresponding to the adjacency list. Each element `weight.list[[i]]` contains the weights of the edges connecting vertex i to the vertices in `adj.list[[i]]`.

Examples

```
# Generate a circle graph with 5 vertices and uniform angles
graph <- generate.circle.graph(5, type = "uniform")

# Generate a circle graph with 10 vertices and random angles
graph <- generate.circle.graph(10, type = "random", seed = 123)
```

`generate.clustered.data` *Generates clustered data (diagonal covariance)*

Description

Generates clustered data (diagonal covariance)

Usage

```
generate.clustered.data(
  n.clusters,
  cluster.sizes,
  dimensions = 2,
  means = NULL,
  covariances = NULL,
  separation = 2
)
```

Arguments

`n.clusters`	Integer, number of clusters.
`cluster.sizes`	Integer vector of length n.clusters with sizes per cluster.
`dimensions`	Integer, number of dimensions (columns).
`means`	Optional numeric matrix `[n.clusters` x `dimensions]` of cluster means.
`covariances`	Optional list of length n.clusters; each element either (i) a diagonal covariance matrix `(dimensions` x `dimensions)`, or (ii) a numeric vector of variances of length `dimensions`. If NULL, uses 0.5 * I.
`separation`	Numeric, separation scale used when `means` is NULL.

Value

Numeric matrix with `sum(cluster.sizes)` rows and `dimensions` columns.

`generate.graph.gaussian.mixture`
Generate a Gaussian Mixture Function on a Graph

Description

Creates a function on a graph that is a mixture of Gaussian-like components centered at specified vertices. Each component follows a decay based on the shortest path distance from its center.

Usage

```
generate.graph.gaussian.mixture(
  adj.list,
  weight.list,
  centers,
  amplitudes = rep(1, length(centers)),
  sigmas = rep(2, length(centers)),
  normalize = TRUE
)
```

Arguments

`adj.list`	List of adjacency lists, where `adj.list[[i]]` contains indices of vertices adjacent to vertex `i`.
`weight.list`	List of edge weights, where `weight.list[[i]][j]` is the weight of the edge from vertex `i` to `adj.list[[i]][j]`.
`centers`	Vector of vertex indices to use as centers for the Gaussian components
`amplitudes`	Vector of amplitudes for each Gaussian component
`sigmas`	Vector of sigma values (spread) for each Gaussian component
`normalize`	Logical; if TRUE, normalize the resulting function to `[0,1]`.

Value

A numeric vector of function values at each vertex of the graph

Examples

```
## Not run:
# Create a grid graph
grid <- create.grid.graph(10, 10)

# Generate a mixture of two Gaussians
centers <- c(1, 100)  # Corner and center vertices
amplitudes <- c(1.0, 0.7)
sigmas <- c(2.0, 3.0)

y <- generate.graph.gaussian.mixture(
  grid$adj.list,
  grid$weight.list,
  centers,
  amplitudes,
  sigmas
)

## End(Not run)
```

`generate.noisy.circle.embedding` *Generate Noisy Circle Data in Higher Dimensions*

Description

This function generates data points from a noisy circle embedded in a higher-dimensional space.

Usage

```
generate.noisy.circle.embedding(
  n,
  dim = 3,
  radius = 1,
  noise = 0,
  type = "random",
  noise.type = "laplace"
)
```

Arguments

`n`	An integer. The number of data points to generate.
`dim`	An integer greater than 2. The dimension of the space to embed the circle in.
`radius`	A positive number. The radius of the circle.
`noise`	A non-negative number. The scale of the noise to add to the data.
`type`	A string, either "uniform" or "random". Determines how angles are generated.
`noise.type`	A string, either "normal" or "laplace". Determines the distribution of the noise.

Value

A matrix with n rows and dim columns. Each row represents a point in R^dim.

Examples

```
# Generate 100 points on a noisy circle in 3D space
data <- generate.noisy.circle.embedding(100, dim = 3, radius = 1, noise = 0.1)
```

`generate.partition` *Generate Random Partition of an Integer*

Description

This function generates a random partition of an integer n into a fixed number of parts (a), where each part is at least m. It uses a sampling-based approach to generate the partition.

Usage

```
generate.partition(n, a, m)
```

Arguments

`n`	A positive integer to be partitioned
`a`	The number of parts in the partition
`m`	The minimum value for each part

Details

The algorithm works by first calculating r = n - (a * m), then randomly sampling (a-1) indices from 1 to r to create partition points. The differences between consecutive partition points (plus m) become the values in the final partition.

Value

A numeric vector of length a containing the partition, where each element is greater than or equal to m and the sum equals n. Returns NULL if the partition is impossible (when n < a * m).

Examples

```
generate.partition(20, 4, 3)  # Generates a partition of 20 into 4 parts, each >= 3
generate.partition(10, 3, 2)  # Generates a partition of 10 into 3 parts, each >= 2
```

`generate.star.dataset` *Generate Synthetic Dataset on a Star-Shaped Geometric Space*

Description

Creates a synthetic dataset consisting of three main components:

1. A geometric realization of a star graph embedded in a d-dimensional Euclidean space
2. A smooth function defined on the vertices of this geometric realization
3. Noisy observations obtained by adding random noise to the smooth function values

The star graph consists of n.arms line segments (arms) meeting at a central point, with each arm having potentially different lengths. Points are distributed along these arms to create the graph vertices.

Usage

```
generate.star.dataset(
  n.points,
  n.arms = 3,
  min.n.pts.within.arm = 3,
  min.arm.length = 0.5,
  max.arm.length = 2,
  arm.lengths = NULL,
  dim = 2,
  fn = "exp",
  fn.center = NULL,
  fn.scale = 1,
  noise = "norm",
  noise.sd = 0.1,
  rand.directions = FALSE,
  eps = 0.05
)
```

Arguments

`n.points`	Integer. Total number of points to generate in the star graph, including the center point and arm endpoints. Must be greater than n.arms * min.n.pts.within.arm
`n.arms`	Integer. Number of arms in the star graph. Must be at least 2
`min.n.pts.within.arm`	Integer. Minimum number of interior points to place within each arm
`min.arm.length`	Numeric. Minimum length for graph arms. Must be positive
`max.arm.length`	Numeric. Maximum length for graph arms. Must be greater than min.arm.length

`arm.lengths`	Numeric vector or NULL. If provided, must be of length n.arms with values between min.arm.length and max.arm.length. If NULL, lengths are randomly sampled from a uniform distribution
`dim`	Integer. Dimension of the Euclidean space (>= 2) in which the star graph is embedded
`fn`	Character. Type of smooth function to use: "exp" - Gaussian function exp(-((x - center)^2)/(2 * scale^2)) "poly" - Cubic polynomial peaking at specified center "sin" - Damped sinusoidal function "wave" - Mexican hat wavelet
`fn.center`	Controls the center point of the smooth function. Can be specified in three ways: 1. NULL: Centers the function at the origin (graph center) 2. Single number: Can be either: - An integer: Uses the coordinates of the vertex at that index - A non-integer: Places the center along the first arm at a fractional distance (e.g., 1.3 places it 30% along the first arm) 3. Numeric vector of length dim: Uses these exact coordinates as the center Default is NULL, centering the function at the origin.
`fn.scale`	Numeric. Scale parameter controlling the spread of the function. For "exp": standard deviation of the Gaussian For "poly": controls the rate of polynomial decay For "sin": controls the frequency and damping rate For "wave": controls the width of the wavelet
`noise`	Character. Type of noise to add: "norm" - Normal distribution "laplace" - Laplace distribution "t" - Student's t-distribution with 3 degrees of freedom "unif" - Uniform distribution
`noise.sd`	Numeric. Scale parameter for the noise distribution
`rand.directions`	Logical. If TRUE, generates random directions for the arms. If FALSE and dim=2, uses evenly spaced points on a circle. Default is FALSE.
`eps`	Numeric. Minimum relative distance between points within arms (0 < eps < 1)

Value

List containing:

`points`	Matrix of vertex coordinates, where rows correspond to vertices
`adj.list`	List of adjacency lists defining the graph structure
`edge.lengths`	List of edge lengths corresponding to adj.list
`y.smooth`	Vector of smooth function values at the vertices
`y.noisy`	Vector of noisy observations (y.smooth + noise)
`arm.lengths`	Vector of actual arm lengths used
`center`	Coordinates of the star center (origin)
`endpoints`	Matrix of arm endpoints coordinates
`directions`	Matrix of unit vectors corresponding to the directions of the arms
`partition`	An integer vector showing the number of elements in each arm
`fn.center`	An index of vector of the center of the y.smooth

Examples

```
# Generate a 2D star graph with 5 arms and exponential function centered at origin
result <- generate.star.dataset(
  n.points = 20,
  n.arms = 5,
  min.n.pts.within.arm = 3,
  min.arm.length = 1,
  max.arm.length = 3,
  dim = 2,
  fn = "exp",
  fn.center = c(0, 0),
  fn.scale = 1,
  noise = "norm",
  noise.sd = 0.1
)
```

`generate.synthetic.function.higher.dim` *Generate a Synthetic Smooth Function in Higher Dimensions*

Description

This function creates a synthetic smooth function based on a set of points in a multi-dimensional space. It utilizes Gaussian functions centered at each point with standard deviations determined by the minimum distance to other points. The function allows for both positive and negative Gaussians.

Usage

```
generate.synthetic.function.higher.dim(X, y = NULL)
```

Arguments

`X`	A numeric matrix where each row represents a point in a multi-dimensional space (R^d).
`y`	An optional numeric vector of response values corresponding to each row in `X`. If not provided, random values are generated.

Details

The function calculates the standard deviation for each Gaussian centered at a point in `X` as half the minimum distance to any other point in `X`. It generates a random vector of +1 or -1 to multiply with the Gaussians, allowing the inclusion of negative values. The output is a function that, when evaluated at any given point in R^d, provides the corresponding value of the synthetic smooth function.

Value

A function that represents the synthetic smooth function. This function takes a numeric vector (a point in R^d) as input and returns a numeric value.

Examples

```
## Not run:
# Example usage
X <- matrix(runif(20), ncol = 2) # Random 2D points
synthetic_fn <- generate_synthetic_function_higher_dim(X)
point <- c(0.5, 0.5) # A point in 2D space
value <- synthetic_fn(point) # Evaluate the function at the given point

## End(Not run)
```

generate.trefoil.knot *Generate a Trefoil Knot in 3D*

Description

Generate a Trefoil Knot in 3D

Usage

```
generate.trefoil.knot(n = 100, scale = 1, type = "uniform")
```

Arguments

n	The desired number of points to generate.
scale	The scale of the knot.
type	A character vector: "uniform" or "random".

geodesic.knn *Estimates geodesic (shortest path) nearest neighbors.*

Description

Estimates geodesic (shortest path) nearest neighbors.

Usage

```
geodesic.knn(X, k, K = 5, G = NULL)
```

Arguments

X	A numeric matrix.
k	The number of nearest neighbors to be returned.
K	The number of neighbors in knn.graph() to build a knn graph. When k=1, the knn graph is the minimal spanning tree.
G	A graph associated with X.

Details

This function computes k-nearest neighbors based on geodesic distances (shortest paths through a graph) rather than Euclidean distances. The geodesic distance is particularly useful for data that lies on a manifold, where the straight-line distance between points may not accurately reflect their true similarity.

The function first constructs or uses a provided graph G to compute geodesic distances between all pairs of points, then identifies the k nearest neighbors for each point based on these distances.

Value

A list containing two matrices:

`nn.index`	An n x k matrix where element `[i,j]` contains the index of the j-th geodesic nearest neighbor of point i.
`nn.dist`	An n x k matrix where element `[i,j]` contains the geodesic distance to the j-th nearest neighbor of point i.

See Also

`estimate.geodesic.distances` for computing geodesic distances `dist.to.knn` for converting distance matrix to k-NN format

Examples

```
## Not run:
# Generate sample data on a spiral
t <- seq(0, 4*pi, length.out = 100)
X <- cbind(t*cos(t), t*sin(t))

# Find 5 geodesic nearest neighbors
gnn <- geodesic.knn(X, k = 5, K = 10)

# Compare with Euclidean nearest neighbors
enn <- get.knn(X, k = 5)

## End(Not run)
```

`geodesic.knnx` *Estimates geodesic (shortest path) nearest neighbors of X.grid points in X. That is for each point of X.grid the k-NN's within X are returned.*

Description

Estimates geodesic (shortest path) nearest neighbors of X.grid points in X. That is for each point of X.grid the k-NN's within X are returned.

Usage

```
geodesic.knnx(X, X.grid, k, method = "knn.graph", K = 5)
```

Arguments

`X`	A numeric matrix.
`X.grid`	A grid associated with X.
`k`	The number of nearest neighbors to be returned.
`method`	A method for building a graph distances within which are going to give shortest path distance estimates. This parameter is not used now, but in the nearest future I am envisioning other methods of constructing the graph.
`K`	The number of neighbors in knn.graph() to build a knn graph. When k=1, the knn graph is the minimal spanning tree.

Details

This function finds k-nearest neighbors from dataset X for each point in X.grid based on geodesic distances. It's particularly useful for interpolation or extrapolation tasks where you want to find the closest data points to query points on a manifold.

The algorithm works by:

1. Creating a combined graph that includes both X.grid and X points
2. Establishing edges within X.grid based on grid structure
3. Connecting X points to their nearest grid points
4. Computing geodesic distances through this combined graph
5. Extracting k-nearest neighbors from X for each X.grid point

Value

A list containing:

`V`	Combined vertex matrix containing both X.grid and X points.
`E`	Edge matrix defining the graph structure.
`nn.index`	An nrow(X.grid) x k matrix where element `[i,j]` contains the index (in X) of the j-th geodesic nearest neighbor of grid point i.
`nn.dist`	An nrow(X.grid) x k matrix where element `[i,j]` contains the geodesic distance from grid point i to its j-th nearest neighbor in X.

Note

The parameter K is automatically set to 2^ncol(X) to ensure adequate connectivity between the data points and the grid. The grid connectivity is based on the dimensionality of the data (2 neighbors in 1D, 4 in 2D, 6 in 3D, etc.).

See Also

`geodesic.knn` for finding geodesic nearest neighbors within a single dataset `get.knn` for Euclidean k-nearest neighbors `get.knnx` for Euclidean k-nearest neighbors between two datasets

Examples

```
## Not run:
# Create sample data
X <- matrix(rnorm(200), ncol=2)

# Create a regular grid
x_seq <- seq(min(X[,1]), max(X[,1]), length.out=10)
y_seq <- seq(min(X[,2]), max(X[,2]), length.out=10)
X.grid <- as.matrix(expand.grid(x_seq, y_seq))

# Find 3 nearest neighbors in X for each grid point
result <- geodesic.knnx(X, X.grid, k=3)

# Access the nearest neighbor indices and distances
nn_indices <- result$nn.index
nn_distances <- result$nn.dist

## End(Not run)
```

`get.1d.binary.models.MABs`
Estimate Mean Absolute Bias for Multiple Non-linear Regression Models with Binary Outcome

---

Description

Compares the performance of different non-linear regression models for binary outcomes by estimating their Mean Absolute Bias (MAB) on synthetic data.

Usage

```
get.1d.binary.models.MABs(
  xt,
  df,
  y.min = -3,
  y.max = 3,
  n.subsamples = 100,
  verbose = TRUE,
  n.cores = 10
)
```

Arguments

`xt`	A numeric vector of predictor values. Its length must equal the number of rows of df.
`df`	A data frame or matrix of smooth function values (columns) over xt values. These will be transformed via min-max normalization and inverse logit function.
`y.min`	The target minimum for min-max transformation (default = -3).
`y.max`	The target maximum for min-max transformation (default = 3).

`n.subsamples`	The number of elements to randomly sample from the predictor and response variables. Must be less than or equal to the number of rows of df (default = 100).
`verbose`	Logical indicating whether to print progress messages (default = TRUE).
`n.cores`	The number of cores to use for parallel computation (default = 10).

Details

This function tests the following models for binary outcomes:

- rf: Random Forest
- magelo1: Robust Local Linear Model (degree 1)
- magelo2: Robust Local Linear Model (degree 2)
- spline: GAM spline regression
- glmnet.poly: Regularized polynomial regression
- loess: Local polynomial regression

The function transforms the smooth functions in df to probabilities using min-max normalization followed by the inverse logit transformation, then generates binary outcomes by sampling from these probabilities.

Value

A list containing:

MAB.df A data frame of MAB estimates with rows corresponding to columns of df and columns corresponding to different models

RMSE.df A data frame of RMSE estimates with the same structure as MAB.df

model.run.times.df A data frame of run times for each model (if n.cores = 1)

Examples

```
## Not run:
# Generate synthetic data
set.seed(123)
n <- 200
xt <- seq(0, 10, length.out = n)
df <- data.frame(
  linear = xt,
  quadratic = xt^2 / 10,
  sine = sin(xt) * 5,
  exponential = exp(xt/5) - 1
)

# Compare models for binary outcomes
results <- get.1d.binary.models.MABs(xt, df, n.subsamples = 100, n.cores = 1)
print(results$MAB.df)

## End(Not run)
```

get.1d.models.MABs *Estimate Mean Absolute Bias for Multiple Non-linear Regression Models*

Description

Compares the performance of different non-linear regression models by estimating their Mean Absolute Bias (MAB) on synthetic data with added noise.

Usage

```
get.1d.models.MABs(
  xt,
  df,
  error = "norm",
  sd = 0.5,
  n.subsamples = 100,
  x.min = 0,
  x.max = 10,
  n.bw = 100,
  min.bw.p = 0.01,
  max.bw.p = 0.9,
  n.cores = 10,
  verbose = TRUE
)
```

Arguments

`xt`	A numeric vector of true predictor values. Its length must equal the number of rows of df.
`df`	A data frame or matrix with rows corresponding to predictor values and columns corresponding to different functional relationships between predictor and response variables.
`error`	A character string indicating the error distribution (currently only "norm" supported).
`sd`	The standard deviation of the normal error distribution (default = 0.5).
`n.subsamples`	The number of elements to randomly sample from the predictor and response variables. Must be less than or equal to the number of rows of df (default = 100).
`x.min`	The minimum of the range of x values (default = 0).
`x.max`	The maximum of the range of x values (default = 10).
`n.bw`	The number of bandwidth values in the bandwidth grid (default = 100).
`min.bw.p`	The minimal proportion of the x range to use as bandwidth (default = 0.01).
`max.bw.p`	The maximal proportion of the x range to use as bandwidth (default = 0.9).
`n.cores`	The number of cores to use for parallel computation (default = 10).
`verbose`	Logical indicating whether to print progress messages (default = TRUE).

Details

This function tests the following models:

- rf: Random Forest
- magelo1: Robust Local Linear Model (degree 1)
- magelo2: Robust Local Linear Model (degree 2)
- smooth.spline: Smooth spline regression
- loess: Local polynomial regression
- locpoly: Local polynomial regression (KernSmooth)
- svm: Support Vector Machine
- krr: Kernel Ridge Regression

Value

A list containing:

MAB.df A data frame of MAB estimates with rows corresponding to columns of df and columns corresponding to different models

RMSE.df A data frame of RMSE estimates with the same structure as MAB.df

model.run.times.df A data frame of run times for each model (if n.cores = 1)

Examples

```
## Not run:
# Generate synthetic data
set.seed(123)
n <- 200
xt <- seq(0, 10, length.out = n)
df <- data.frame(
  linear = xt,
  quadratic = xt^2 / 10,
  sine = sin(xt),
  exponential = exp(xt/5) - 1
)

# Compare models
results <- get.1d.models.MABs(xt, df, sd = 0.5, n.subsamples = 100, n.cores = 1)
print(results$MAB.df)

## End(Not run)
```

---

get.3D.rect.grid.NN *Creates a rectangular 3D grid and returns a vector of NN's*

Description

Creates a rectangular 3D grid and returns a vector of NN's

Usage

```
get.3D.rect.grid.NN(
  n,
  X,
  f = 0.2,
  eSDf = 1.5,
  mst.grid = NULL,
  mode.edge.len = NULL
)
```

Arguments

`n`	The number of uniformly spaced points, seq(min(xi), max(xi), length=n), on each axis that are the basis of the grid.
`X`	A set of points around which the grid is created if not NULL.
`f`	A fraction of x1 and x2 range that the grid is extended to.
`eSDf`	A scaling factor, such that edges of mstree(X) are subdivided is their length is greater than eSDf * mode(edge.len).
`mst.grid`	A matrix of 3D points that are subdivisions of edges of the mstree(X).
`mode.edge.len`	The mode of the edge lengths.

`get.basin.vertices` *Extract vertices belonging to a specific basin from a basin complex*

Description

This function extracts all vertex indices that belong to a specified basin from a gradient flow basin complex object. The basin is identified by its label (e.g., "M1" or "m2").

Usage

```
get.basin.vertices(x, id)
```

Arguments

`x`	An object of class 'basin_cx', typically returned by `create.basin.cx()`
`id`	A character string specifying the basin label to extract (e.g., "M1" for first maximum, "m2" for second minimum)

Details

Basin labels follow the naming convention of the basin complex: "Mx" for maxima basins and "mx" for minima basins, where x is a sequential number assigned based on function value ordering.

Value

An integer vector containing the sorted vertex indices belonging to the specified basin

See Also

`create.basin.cx`

get.BB.gpredictions *Generates Bayesian bootstrap estimates of gpredictions*

Description

Generates Bayesian bootstrap estimates of gpredictions

Usage

```
get.BB.gpredictions(
  n.BB,
  nn.i,
  nn.x,
  nn.y,
  y.binary,
  nn.w,
  nx,
  max.K,
  degree,
  grid.nn.i,
  grid.nn.x,
  grid.nn.w,
  grid.max.K
)
```

Arguments

`n.BB`	The number of Bayesian bootstrap iterations
`nn.i`	A matrix of NN indices.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`y.binary`	Set to TRUE if y values are within the interval `[0,1]`.
`nn.w`	A matrix of NN weights.
`nx`	The number of elements of x.
`max.K`	A vector of **indices** indicating the range where weights are not 0.
`degree`	The degree of the local models.
`grid.nn.i`	A matrix of grid asociated NN indices.
`grid.nn.x`	A matrix of grid asociated x values over NNs.
`grid.nn.w`	A matrix of grid asociated NN weights.
`grid.max.K`	A vector of grid asociated max.K values.

get.bws — *Gets bandwidths defined as radii of linear model's disks of support*

Description

Finds a bandwidth for each column of d, such that `bws[i] = bw` if `d[iminK + ir] < bw` and `bws[i] = d[iminK + 1 + ir] = d[minK + ir]`, otherwise.

Usage

```
get.bws(d, min.K, bw)
```

Arguments

`d`	A numeric matrix nn.d of distances to NNs.
`min.K`	The mininum number of elements of each row with weights > 0.
`bw`	A bendwidth.

Value

A vector of bendwidths.

get.bws.with.minK — *Gets bandwidths defined as radia of linear model's disks of support*

Description

The difference between this function and get.bws() is that in the former, min.K is a constant, which is replaced by a vector minK in here. This function is not meant to be accessible by user and serves as an R interface to get_bws_with_minK_a() for unit testing this C function.

Usage

```
get.bws.with.minK(d, minK, bw)
```

Arguments

`d`	A numeric matrix.
`minK`	A vector such that minK[i] is the required mininum number of elements in the row of the i-th column of d with weights > 0.
`bw`	A bendwidth.

Value

A vector of bendwidths.

get.edge.weights *Get Unique Edge Weights from a Weighted Graph in Parallel*

Description

Returns a vector containing the weights of all edges in a weighted graph without duplication, using parallel processing for improved performance on large graphs.

Usage

```
get.edge.weights(adj.list, weight.list, n.cores = 12)
```

Arguments

`adj.list`	A list where each element contains the adjacency list for a vertex. Element i contains a vector of vertices that are adjacent to vertex i.
`weight.list`	A list with the same structure as `adj.list`, where `weight.list[[i]][j]`. contains the weight of the edge between vertex i and its j-th neighbor.
`n.cores`	Number of cores to use for parallel processing. Default is 2.

Details

The function parallelizes the processing by dividing the vertices among different cores. It handles undirected graphs by only processing edges where i < neighbor to ensure each edge is counted exactly once. The function requires the foreach and doParallel packages.

Value

A numeric vector containing the weights of all unique edges in the graph.

Examples

```
# Create a simple undirected weighted graph
adj.list <- list(c(2, 3), c(1, 3), c(1, 2))
weight.list <- list(c(5, 10), c(5, 7), c(10, 7))

# Get weights of all edges using 2 cores
edge_weights <- get.edge.weights(adj.list, weight.list, n.cores = 2)
```

get.gam.spline.MAB *Estimate Mean Absolute Bias (MAB) of GAM Spline Model*

Description

Fits a Generalized Additive Model (GAM) with spline smoothing to a given dataset and evaluates its performance. For binary outcomes, uses logistic GAM.

Usage

```
get.gam.spline.MAB(x, y, xt, yt, folds, y.binary = FALSE)
```

Arguments

`x`	A numeric vector representing the predictor variable for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A numeric vector representing the predictor variable for the test set.
`yt`	A numeric vector representing the response variable for the test set.
`folds`	A list of indices for cross-validation folds (currently not used in implementation).
`y.binary`	Logical indicating if y is a binary variable (default = FALSE).

Details

This function fits a GAM model using spline smoothing. For binary outcomes, it uses a binomial family with logit link and returns predicted probabilities. For continuous outcomes, it uses a Gaussian family.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the GAM model on the test set

residuals Absolute residuals

parameters Empty list (for consistency with other functions)

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- runif(100, 0, 10)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- seq(0, 10, length.out = 50)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Run the function
result <- get.gam.spline.MAB(x, y, xt, yt, folds)
print(result$MAB)

# Example with binary data
y.binary <- rbinom(100, 1, plogis(sin(x)))
yt.binary <- plogis(sin(xt))
result.binary <- get.gam.spline.MAB(x, y.binary, xt, yt.binary, folds, y.binary = TRUE)
print(result.binary$MAB)

## End(Not run)
```

get.gaussian.mixture *Generate Synthetic Data from a Mixture of Gaussians*

Description

Creates synthetic data by generating a mixture of Gaussian components with flexible positioning, scaling, and noise characteristics. This function provides extensive control over the component placement, standard deviations, amplitudes, and noise properties, making it suitable for testing and benchmarking smoothing methods. Supports both absolute and sigma-relative positioning strategies.

Usage

```
get.gaussian.mixture(
  n.points = 100,
  n.components = 5,
  x.range = c(0, 1),
  y.range = c(-1, 1),
  x.knots.strategy = "uniform",
  jitter.fraction = 0.25,
  edge.buffer = 0.1,
  sigma = 1.5,
  xmin.factor = 3,
  xmax.factor = 7,
  x.offset.factor = 2,
  sd.strategy = "fixed",
  sd.knot = 0.1,
  sd.factor = 0.5,
  sd.random.range = c(0.5, 2),
  custom.sd.knots = NULL,
  y.strategy = "mixed",
  y.min.fraction = 0.1,
  custom.y.knots = NULL,
  noise.fraction = 0.15,
  error.distribution = "norm",
  seed = NULL,
  verbose = FALSE
)
```

Arguments

`n.points`	Integer specifying the number of data points to generate
`n.components`	Integer specifying the number of Gaussian components
`x.range`	Numeric vector c(min, max) specifying the range for x values. Ignored when x.knots.strategy = "offset"
`y.range`	Numeric vector c(min, max) specifying the range for y values
`x.knots.strategy`	Character string specifying how to position components. Must be one of:

- "uniform": equally spaced components

- "random": uniformly random positions
- "jittered": perturbed uniform spacing
- "offset": sequential spacing with sigma-relative gaps

`jitter.fraction`
: Numeric between 0 and 0.5 specifying maximum jitter as fraction of local spacing when x.knots.strategy = "jittered"

`edge.buffer`
: Numeric between 0 and 0.5 specifying minimum distance from range edges as fraction of total range (not used for "offset" strategy)

`sigma`
: Numeric specifying the scale parameter when using offset strategy. Also informs the error distribution scale. Default is 1.5

`xmin.factor`
: Numeric specifying minimum spacing between components in units of sigma when x.knots.strategy = "offset". Default is 3

`xmax.factor`
: Numeric specifying maximum spacing between components in units of sigma when x.knots.strategy = "offset". Default is 7

`x.offset.factor`
: Numeric factor to extend x-range beyond components when x.knots.strategy = "offset". Default is 2

`sd.strategy`
: Character string specifying how to generate component SDs. Must be one of:

- "fixed": constant SD for all components
- "adaptive": SD based on distance to neighbors
- "stoch.adaptive": randomized adaptive SDs
- "custom": user-specified SD values

`sd.knot`
: Numeric specifying the SD value when sd.strategy = "fixed"

`sd.factor`
: Numeric between 0 and 1 specifying the fraction of neighbor distance to use for SD in adaptive strategies

`sd.random.range`
: Numeric vector c(min, max) specifying multipliers for random SD generation in "stoch.adaptive" strategy

`custom.sd.knots`
: Optional numeric vector of length n.components specifying custom SD values

`y.strategy`
: Character string specifying how to generate component amplitudes. Must be one of:

- "mixed": mix of positive and negative amplitudes
- "positive": only positive amplitudes
- "custom": user-specified amplitudes

`y.min.fraction`
: Numeric between 0 and 1 specifying minimum absolute amplitude as fraction of y range

`custom.y.knots`
: Optional numeric vector of length n.components specifying custom amplitude values

`noise.fraction`
: Numeric specifying noise standard deviation as fraction of y range

`error.distribution`
: Character string specifying noise distribution. Must be one of:

- "norm": Gaussian noise
- "laplace": Laplace noise

`seed`
: Optional integer for random number generation

`verbose`
: Logical indicating whether to print generation details

Value

A list containing:

x Numeric vector of x coordinates

y Numeric vector of noisy y values

y.true Numeric vector of true y values (without noise)

x.knots Numeric vector of component x positions

y.knots Numeric vector of component amplitudes

sd.knots Numeric vector of component standard deviations

parameters List of all input parameters used

Positioning Strategies

The function supports four strategies for positioning components along the x-axis:

uniform Components are equally spaced within x.range with edge buffers

random Components are randomly positioned within x.range with edge buffers

jittered Components start on a uniform grid then are randomly perturbed. The jitter.fraction parameter controls the maximum displacement while maintaining order

offset Components are sequentially positioned with random gaps between xmin.factor*sigma and xmax.factor*sigma. The first component starts at 0, and the x-range is automatically determined

Jittering Details

When using jittered positioning, components maintain their relative order:

- Interior components can move up to jitter.fraction times the distance to their neighbors
- Edge components have full freedom toward the range boundaries
- jitter.fraction must be ≤ 0.5 to prevent crossing

Examples

```
# Basic usage with default parameters
data <- get.gaussian.mixture()

# Custom mixture with specific parameters
data <- get.gaussian.mixture(
  n.points = 200,
  n.components = 3,
  x.knots.strategy = "jittered",
  sd.strategy = "stoch.adaptive",
  y.strategy = "mixed",
  noise.fraction = 0.1
)

# Using sigma-scaled offset positioning
data <- get.gaussian.mixture(
  n.components = 4,
  x.knots.strategy = "offset",
  sigma = 2,
  xmin.factor = 2,
  xmax.factor = 5,
  sd.strategy = "adaptive"
)

# Example with fixed standard deviations and custom amplitudes
data <- get.gaussian.mixture(
  n.components = 2,
  x.knots.strategy = "uniform",
  sd.strategy = "custom",
  custom.sd.knots = c(0.05, 0.15),
  y.strategy = "custom",
  custom.y.knots = c(1, -0.5)
)
```

---

get.gaussian.mixture.2d

Generate 2D Gaussian Mixture

Description

Creates a 2D Gaussian mixture function with flexible component placement and characteristics. Returns an object of class "gaussian_mixture" with associated plotting methods.

Usage

```
get.gaussian.mixture.2d(
  n.components = 3,
  x.range = c(0, 1),
  y.range = c(0, 1),
  centers.strategy = c("random", "grid", "custom"),
  custom.centers = NULL,
  amplitudes.strategy = c("mixed", "positive", "custom"),
  custom.amplitudes = NULL,
  amplitude.range = c(0.5, 2),
  sd.strategy = c("fixed", "random", "custom"),
  sd.value = 0.1,
  sd.range = c(0.05, 0.2),
  custom.sds = NULL,
  correlation = 0,
  edge.buffer = 0.1,
  seed = NULL,
  verbose = FALSE
)
```

Arguments

`n.components`	Integer specifying the number of Gaussian components
`x.range`	Numeric vector c(min, max) specifying the range for x values
`y.range`	Numeric vector c(min, max) specifying the range for y values
`centers.strategy`	Character string specifying how to position components:

	• "random": uniformly random positions (default) • "grid": regular grid arrangement • "custom": user-specified centers
`custom.centers`	Matrix with n.components rows and 2 columns (x, y coordinates) when centers.strategy = "custom"
`amplitudes.strategy`	Character string specifying amplitude generation: • "mixed": positive and negative amplitudes (default) • "positive": only positive amplitudes • "custom": user-specified amplitudes
`custom.amplitudes`	Numeric vector of length n.components when amplitudes.strategy = "custom"
`amplitude.range`	Numeric vector c(min, max) for amplitude magnitudes
`sd.strategy`	Character string specifying how to generate component SDs: • "fixed": constant SD for all components (default) • "random": random SDs within range • "custom": user-specified SDs
`sd.value`	Numeric specifying SD when sd.strategy = "fixed"
`sd.range`	Numeric vector c(min, max) when sd.strategy = "random"
`custom.sds`	Numeric vector or 2x2 matrix. If vector, creates isotropic Gaussians; if matrix, specifies covariance for each component
`correlation`	Numeric between -1 and 1 specifying correlation between x and y for all components (ignored if custom.sds is a matrix)
`edge.buffer`	Numeric between 0 and 0.5 specifying buffer from edges
`seed`	Optional integer for random number generation
`verbose`	Logical indicating whether to print generation details

Value

An object of class "gaussian_mixture" containing:

- `f`: Function f(x, y) that evaluates the mixture at given coordinates
- `n.components`: Number of components
- `centers`: Matrix of component centers (n.components x 2)
- `amplitudes`: Vector of component amplitudes
- `covariances`: List of 2x2 covariance matrices for each component
- `x.range`: X range used
- `y.range`: Y range used
- `parameters`: List of all parameters used

Examples

```
## Not run:
# Generate random 2D mixture
gm1 <- get.gaussian.mixture.2d(n.components = 3)
plot(gm1)

# Generate grid arrangement with custom amplitudes
gm2 <- get.gaussian.mixture.2d(
  n.components = 4,
  centers.strategy = "grid",
  amplitudes.strategy = "custom",
  custom.amplitudes = c(1, -0.5, 0.8, -0.3)
)

# Custom centers with anisotropic Gaussians
gm3 <- get.gaussian.mixture.2d(
  n.components = 2,
  centers.strategy = "custom",
  custom.centers = matrix(c(0.3, 0.3, 0.7, 0.7), ncol = 2, byrow = TRUE),
  correlation = 0.5
)

## End(Not run)
```

---

get.gausspr.MAB *Estimate Mean Absolute Bias (MAB) of Gaussian Process Regression Model*

Description

Fits a Gaussian Process Regression (GPR) model to a given dataset and evaluates its performance. The Mean Absolute Bias (MAB) is calculated using the test set.

Usage

```
get.gausspr.MAB(x, y, xt, yt, folds, kernel = "rbfdot")
```

Arguments

x	A matrix or data frame representing the predictor variables for the training set.
y	A numeric vector representing the response variable for the training set.
xt	A matrix or data frame representing the predictor variables for the test set.
yt	A numeric vector representing the response variable for the test set.
folds	A list of indices for cross-validation folds (currently not used in implementation).
kernel	A kernel function to be used in GPR (default = 'rbfdot'). Options include 'rbfdot', 'polydot', 'laplacedot', etc. from kernlab package.

Details

The function uses Gaussian Process Regression to model the relationship between the predictors and the response. Users can specify different kernel functions to explore various types of non-linear relationships in the data.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the GPR model on the test set

residuals Absolute residuals

parameters List containing the kernel used

Examples

```
## Not run:
# Generate example data
library(kernlab)
set.seed(123)
x <- matrix(rnorm(100), ncol = 1)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- matrix(rnorm(50), ncol = 1)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Run the function
result <- get.gausspr.MAB(x, y, xt, yt, folds)
print(result$MAB)

## End(Not run)
```

`get.glmnet.poly.MAB` *Estimate Mean Absolute Bias (MAB) of Regularized Polynomial Regression Model*

Description

Fits a regularized polynomial regression model using elastic net regularization via the glmnet package. The function creates polynomial features (x, x^2, x^3) and uses cross-validation to select the optimal regularization parameter lambda.

Usage

```
get.glmnet.poly.MAB(
  x,
  y,
  xt,
  yt,
  folds,
  lambda.sequence = NULL,
  alpha = 1,
  y.binary = TRUE
)
```

Arguments

`x`	A numeric vector representing the predictor variable for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A numeric vector representing the predictor variable for the test set.
`yt`	A numeric vector representing the true response values for the test set.
`folds`	A list of indices for cross-validation folds (currently not used).
`lambda.sequence`	A numeric vector of lambda values to test (default = NULL, uses glmnet default).
`alpha`	The elastic net mixing parameter (default = 1 for lasso, 0 for ridge).
`y.binary`	Logical indicating if y is binary (default = TRUE).

Details

This function creates polynomial features (x, x^2, x^3) to capture non-linear relationships. It uses cv.glmnet for automatic cross-validation to select the optimal lambda parameter. The alpha parameter controls the type of regularization:

- alpha = 1: Lasso regression (L1 penalty)
- alpha = 0: Ridge regression (L2 penalty)
- 0 < alpha < 1: Elastic net (combination of L1 and L2)

For binary outcomes, it uses binomial family with logit link. For continuous outcomes, it uses gaussian family.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the glmnet model on the test set

residuals Absolute residuals

parameters List containing optimal lambda value

Note

Consider renaming this function to get.glmnet.poly.MAB to better reflect that it uses polynomial features rather than splines.

Examples

```
## Not run:
# Example with continuous outcome
set.seed(123)
x <- seq(-2, 2, length.out = 100)
y <- x^2 + rnorm(100, sd = 0.5)
xt <- seq(-2, 2, length.out = 50)
yt <- xt^2

# Create cross-validation folds (not used internally)
```

```
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Fit model
result <- get.glmnet.poly.MAB(x, y, xt, yt, folds, alpha = 1, y.binary = FALSE)
print(result$MAB)

# Example with binary outcome
y.binary <- rbinom(100, 1, plogis(x^2))
yt.binary <- plogis(xt^2)
result.binary <- get.glmnet.poly.MAB(x, y.binary, xt, yt.binary, folds, y.binary = TRUE)
print(result.binary$MAB)

## End(Not run)
```

get.gpredictions.CrI *Generates Bayesian bootstrap credible intervals of gpredictions.*

Description

Generates Bayesian bootstrap credible intervals of gpredictions.

Usage

```
get.gpredictions.CrI(
  n.BB,
  nn.i,
  nn.x,
  nn.y,
  nn.w,
  nx,
  max.K,
  degree,
  grid.nn.i,
  grid.nn.x,
  grid.nn.w,
  grid.max.K,
  y.binary = FALSE,
  alpha = 0.05
)
```

Arguments

`n.BB`	The number of Bayesian bootstrap iterations
`nn.i`	A matrix of indices of X of the nearest neighbors of each point of X.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`nn.w`	A matrix of NN weights.
`nx`	The number of elements of x.

`max.K`	A vector of **indices** indicating the range where weights are not 0.
`degree`	The degree of the local models.
`grid.nn.i`	A matrix of grid asociated NN indices.
`grid.nn.x`	A matrix of grid asociated x values over NNs.
`grid.nn.w`	A matrix of grid asociated NN weights.
`grid.max.K`	A vector of grid asociated max.K values.
`y.binary`	A logical variable. If TRUE, bw optimization is going to use a binary loss function mean(y(1-p) + (1-y)p).
`alpha`	The confidence level.

Value

A matrix of Bayesian bootstrap credible intervals of gpredictions.

Examples

```
# TBD
```

`get.gpredictionss` *Estimates gpredictions Means Matrix Over a Uniform Grid for Different bandwidths.*

Description

Given a vector, bws, of bandwidths this routine estimates a matrix of gpredictions's where each column corresponds to the gpredictions estimate for a different bandwidth. This is an R interface to a C routine, C_get_gpredictionss(), where the estimates of gpredictions's are done. In practice, it is not intended to be used directly from R, but rather to test the correctness of C_get_gpredictionss(). This routine is used in different bandwidth optimization processes based on gpredictions characteristics (like total absolute curvature or the number of inflection points).

Usage

```
get.gpredictionss(bws, nn.i, nn.d, nn.x, nn.y, y.binary, degree, min.K, xgrid)
```

Arguments

`bws`	A vector of bandwidths.
`nn.i`	A matrix of indices for K nearest neighbors.
`nn.d`	A matrix of distances to the nearest neighbors.
`nn.x`	A matrix of x values over K nearest neighbors of each grid point, where K is determined in the parent routine.
`nn.y`	A matrix of y values over K nearest neighbors of each grid point.
`y.binary`	Set to TRUE if y values are in the interval `[0,1]`.
`degree`	The degree of the local models.
`min.K`	The minimal number of elements in each set of nearest neighbors.
`xgrid`	A vector of grid points.

Value

A matrix of the estimates of the means, gpredictions, of y over a uniform grid where the i-th column consists of gpredictions estimates using the i-th value of the 'bws' vector.

Examples

```
## Not run:
res <- get.gpredictionss(bws, nn.i, nn.d, nn.x, nn.y, y.binary, degree, min.K, xgrid)
str(res)

## End(Not run)
```

`get.krr.MAB` *Estimate Mean Absolute Bias (MAB) of Kernel Ridge Regression Model*

Description

Fits a Kernel Ridge Regression (KRR) model to a given dataset and evaluates its performance using cross-validation. The Mean Absolute Bias (MAB) is calculated using the test set with optimized hyperparameters.

Usage

```
get.krr.MAB(x, y, xt, yt, kernel = "rbfdot")
```

Arguments

`x`	A matrix or data frame representing the predictor variables for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A matrix or data frame representing the predictor variables for the test set.
`yt`	A numeric vector representing the response variable for the test set.
`kernel`	A kernel function to be used in KRR (default = 'rbfdot').

Details

The function uses Kernel Ridge Regression to model the relationship between the predictors and the response. It performs cross-validation to find optimal hyperparameters (sigma and lambda) and computes the MAB using the true x,y values. Users can specify different kernel functions to explore various types of non-linear relationships in the data.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the KRR model on the test set

residuals Absolute residuals

parameters List containing optimal sigma and lambda values

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- matrix(rnorm(100), ncol = 1)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- matrix(rnorm(50), ncol = 1)
yt <- sin(xt)

# Run the function
result <- get.krr.MAB(x, y, xt, yt)
print(result$MAB)

## End(Not run)
```

---

get.krr.MAB.xD *Estimate Mean Absolute Bias (MAB) of Kernel Ridge Regression for Multi-dimensional Data*

Description

Fits a Kernel Ridge Regression (KRR) model to multi-dimensional data and evaluates its performance using cross-validation to select optimal hyperparameters.

Usage

```
get.krr.MAB.xD(
  X,
  y,
  yt,
  kernel = "rbfdot",
  lambdas = 10^(-8:0),
  sigmas = 10^((1:9)/3)
)
```

Arguments

X	A matrix or data frame of predictor variables (dimension > 1).
y	A numeric vector of response values.
yt	A numeric vector of true response values for evaluation.
kernel	A kernel function to be used in KRR (default = 'rbfdot').
lambdas	A numeric vector of regularization parameters to test (default = 10^(-8:0)).
sigmas	A numeric vector of kernel width parameters to test (default = 10^((1:9)/3)).

Details

The function uses Kernel Ridge Regression to model the relationship between the predictors and the response. It performs 10-fold cross-validation to find optimal hyperparameters (sigma and lambda) and computes the MAB using in-sample predictions.

Value

A list containing:

MAB Mean Absolute Bias calculated on the dataset

RMSE Root Mean Square Error

predictions Predictions made by the KRR model

residuals Absolute residuals

parameters List containing optimal sigma and lambda values

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 100
X <- matrix(rnorm(n * 3), ncol = 3)
y <- X[,1] + X[,2]^2 + rnorm(n, sd = 0.1)
yt <- X[,1] + X[,2]^2  # True values

# Fit KRR model and compute MAB
result <- get.krr.MAB.xD(X, y, yt)
print(result$MAB)

# With custom parameters
result2 <- get.krr.MAB.xD(X, y, yt, kernel = 'polydot',
                          lambdas = 10^seq(-6, -2, length = 5))
print(result2$MAB)

## End(Not run)
```

`get.locpoly.MAB` *Estimate Mean Absolute Bias (MAB) of Local Polynomial Regression*

Description

Fits a local polynomial regression model to the given dataset and evaluates its performance using cross-validation. The mean absolute error (MAE) is used as the performance metric, and the function selects the optimal bandwidth for the local polynomial regression.

Usage

```
get.locpoly.MAB(x, y, xt, yt, folds, bandwidths)
```

Arguments

`x`	A numeric vector representing the predictor variable for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A numeric vector representing the predictor variable for the test set.
`yt`	A numeric vector representing the response variable for the test set.
`folds`	A list of indices for cross-validation folds
`bandwidths`	A numeric vector of bandwidth values to be tested.

Details

The function performs cross-validation to find the optimal bandwidth that minimizes the MAE. It uses the `locpoly` function from the KernSmooth package for fitting local polynomial models. The optimal bandwidth is then used to fit the final model and make predictions on the test set.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the local polynomial regression model on the test set

residuals Absolute residuals

parameters List containing the optimal bandwidth

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- runif(100, 0, 10)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- seq(0, 10, length.out = 50)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Define bandwidth values
bandwidths <- seq(0.1, 1, by = 0.1)

# Run the function
result <- get.locpoly.MAB(x, y, xt, yt, folds, bandwidths)
print(result$MAB)

## End(Not run)
```

`get.loess.MAB` *Estimate Mean Absolute Bias (MAB) of LOESS Model*

Description

Estimates the Mean Absolute Bias (MAB) of a LOESS (Locally Estimated Scatterplot Smoothing) model with the span parameter chosen based on cross-validated MAE estimates.

Usage

```
get.loess.MAB(x, y, xt, yt, folds, deg = 2)
```

Arguments

`x`	A numeric vector of predictor values.
`y`	A numeric vector of response values.
`xt`	A numeric vector of true predictor values for evaluation.
`yt`	A numeric vector of true response values for evaluation.
`folds`	A list of indices for cross-validation folds
`deg`	The degree of the local polynomial used (default = 2).

Value

A list containing:

MAB Mean Absolute Bias

RMSE Root Mean Square Error

predictions Predicted values at xt

residuals Absolute residuals

parameters List containing the optimal span

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- runif(100, 0, 10)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- seq(0, 10, length.out = 50)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Fit LOESS model and compute MAB
result <- get.loess.MAB(x, y, xt, yt, folds, deg = 2)
print(result$MAB)

## End(Not run)
```

`get.magelo.MAB` *Estimate Mean Absolute Bias (MAB) of MAGELO Model*

Description

Estimates the Mean Absolute Bias (MAB) of a MAGELO model.

Usage

```
get.magelo.MAB(x, y, xt, yt, deg = 2, y.binary = FALSE, n.cores = 10)
```

Arguments

`x`	A numeric vector of predictor values.
`y`	A numeric vector of response values.
`xt`	A numeric vector of true predictor values for evaluation.
`yt`	A numeric vector of true response values for evaluation.
`deg`	The degree of the MAGELO polynomial model (default = 2).
`y.binary`	Logical indicating if y is a binary variable (default = FALSE).
`n.cores`	Number of cores to use for parallel computation (default = 10).

Value

A list containing:

MAB Mean Absolute Bias

RMSE Root Mean Square Error

predictions Predicted values at xt

residuals Absolute residuals

parameters List containing the degree used

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- runif(100, 0, 10)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- seq(0, 10, length.out = 50)
yt <- sin(xt)

# Fit magelo model and compute MAB
result <- get.magelo.MAB(x, y, xt, yt, deg = 2)
print(result$MAB)

## End(Not run)
```

`get.np.MAB.xD` *Estimate Mean Absolute Bias (MAB) of Nonparametric Kernel Regression Model*

Description

Fits a nonparametric kernel regression model to multi-dimensional data and evaluates its performance using in-sample predictions.

Usage

```
get.np.MAB.xD(X, y, yt, y.binary = FALSE)
```

Arguments

`X`	A matrix or data frame of predictor variables (dimension > 1).
`y`	A numeric vector of response values.
`yt`	A numeric vector of true response values for evaluation.
`y.binary`	Logical indicating if y is a binary variable (default = FALSE).

Details

This function fits a nonparametric kernel regression model using the np package. For continuous outcomes, it uses local linear regression with bandwidth selection via cross-validated AIC. For binary outcomes, it uses conditional mode estimation. Note that bandwidth selection can be computationally intensive for large datasets.

Value

A list containing:

MAB Mean Absolute Bias calculated on the dataset

RMSE Root Mean Square Error

predictions Predictions made by the nonparametric model

residuals Absolute residuals

parameters Empty list (for consistency with other functions)

Examples

```
## Not run:
# Generate example data
library(np)
set.seed(123)
n <- 100
X <- matrix(rnorm(n * 2), ncol = 2)
colnames(X) <- c("x1", "x2")
y <- X[,1] + X[,2]^2 + rnorm(n, sd = 0.1)
yt <- X[,1] + X[,2]^2  # True values

# Fit nonparametric model and compute MAB
result <- get.np.MAB.xD(X, y, yt)
print(result$MAB)

# Example with binary data
y.binary <- rbinom(n, 1, plogis(y))
yt.binary <- plogis(yt)
result.binary <- get.np.MAB.xD(X, y.binary, yt.binary, y.binary = TRUE)
print(result.binary$MAB)

## End(Not run)
```

get.path.data *Compute Graph Path Data with Kernel Weights*

Description

Analyzes paths in a graph centered around a reference vertex, computing distances, kernel weights, and associated values along these paths. The function identifies both single paths and composite paths that meet the minimum size requirement, with the reference vertex serving as a central point in the path structure.

Usage

```
get.path.data(
  adj.list,
  weight.list,
  y,
  ref.vertex,
  bandwidth,
  dist.normalization.factor = 1.01,
  min.path.size = 5L,
  diff.threshold = 5L,
  kernel.type = 7L,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the graph's adjacency structure. Each element `i` contains the vertices adjacent to vertex `i` (1-based indexing).
`weight.list`	A list of numeric vectors containing edge weights. Each element `i` contains weights corresponding to edges in `adj.list[[i]]`.
`y`	A numeric vector of values associated with each vertex in the graph.
`ref.vertex`	An integer specifying the reference vertex around which paths are constructed (1-based indexing).
`bandwidth`	A positive numeric value specifying the maximum allowable path distance from the reference vertex.
`dist.normalization.factor`	A numeric value between 0 and 1 for normalizing distances in kernel calculations (default: 1.01).
`min.path.size`	An integer specifying the minimum number of vertices required in a valid path (default: 5).
`diff.threshold`	An integer specifying the number of vertices after the ref vertex that two paths have to have different (set intersection is empty) to produce a composite path from these two paths. Default is 5. If out of valid range, will be auto-adjusted to the midpoint of the valid range.
`kernel.type`	An integer specifying the kernel function type: - 1: Epanechnikov - 2: Triangular - 4: Laplace - 5: Normal - 6: Biweight - 7: Tricube (default)
`verbose`	Logical indicating whether to print progress information. Default is FALSE.

Value

A list where each element represents a path and contains:

`vertices`	Integer vector of path vertices (1-based indices)
`ref_vertex`	Integer indicating the reference vertex (1-based index)
`rel_center_offset`	Numeric value indicating relative position of reference vertex (0 = center, 0.5 = endpoint)
`total_weight`	Numeric value representing total path length
`x_path`	Numeric vector of cumulative distances along path from start
`w_path`	Numeric vector of kernel weights for each vertex
`y_path`	Numeric vector of y-values for path vertices

Note

- All vertex indices must be positive integers
- Edge weights must be non-negative
- The length of adj.list and weight.list must match
- The reference vertex must exist in the graph

See Also

The C++ implementation details can be found in the source code of S_get_path_data

Examples

```
## Not run:
# Create a simple graph with 5 vertices
adj.list <- list(c(2,3), c(1,3,4), c(1,2,5), c(2), c(3))
weight.list <- list(c(1,1), c(1,1,1), c(1,1,1), c(1), c(1))
y <- c(1.5, 2.0, 0.5, 1.0, 1.5)

# Find paths centered around vertex 2
paths <- get.path.data(adj.list, weight.list, y,
                      ref.vertex = 2, bandwidth = 2)

## End(Not run)
```

`get.random.forest.MAB` *Estimate Mean Absolute Bias (MAB) of Random Forest Model*

Description

Fits a Random Forest model to a given dataset and evaluates its performance by computing the Mean Absolute Bias on test data. Supports both regression and classification tasks with optional optimization of the number of trees.

Usage

```
get.random.forest.MAB(
  x,
  y,
  xt,
  yt,
  ntree = 500,
  optimize.ntree = FALSE,
  max.trees = 1000,
  plot.oob = FALSE,
  ...
)
```

Arguments

`x`	A vector, matrix, or data frame of predictor values for training.
`y`	A vector of response values (numeric for regression, factor for classification).
`xt`	A vector, matrix, or data frame of predictor values for testing.
`yt`	A vector of true response values for testing. For classification, these should be probabilities of the positive class.
`ntree`	Number of trees to grow in the random forest (default = 500).
`optimize.ntree`	Logical. If TRUE, uses OOB error to find optimal number of trees between 100 and max.trees (default = FALSE).
`max.trees`	Maximum number of trees to consider when optimize.ntree = TRUE (default = 1000).
`plot.oob`	Logical. If TRUE and optimize.ntree = TRUE, plots the OOB error curve (default = FALSE).
`...`	Additional arguments passed to randomForest().

Details

For classification tasks (when `y` is a factor), the function returns predicted probabilities for the second class level. For regression tasks, it returns the predicted values directly.

When optimize.ntree = TRUE, the function fits a random forest with max.trees and examines the OOB error curve to find where the error stabilizes. It selects the smallest number of trees where the OOB error is within 1% of the minimum.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the random forest model on the test set

residuals Absolute residuals

model The fitted random forest model object

parameters List containing ntree value used and optimal.ntree if optimization was performed

Examples

```
## Not run:
# Regression example
set.seed(123)
x <- matrix(rnorm(200), ncol = 2)
y <- x[,1] + x[,2]^2 + rnorm(100, sd = 0.5)
xt <- matrix(rnorm(100), ncol = 2)
yt <- xt[,1] + xt[,2]^2

# Basic usage
result <- get.random.forest.MAB(x, y, xt, yt, ntree = 500)
print(result$MAB)

# With optimization
result.opt <- get.random.forest.MAB(x, y, xt, yt, optimize.ntree = TRUE)
print(result.opt$parameters$optimal.ntree)

# Classification example
y.class <- factor(ifelse(y > median(y), "high", "low"))
yt.prob <- pnorm(yt, mean = mean(yt), sd = sd(yt))
result.class <- get.random.forest.MAB(x, y.class, xt, yt.prob)
print(result.class$MAB)

## End(Not run)
```

`get.random.forest.MAB.xD` *Estimate Mean Absolute Bias (MAB) of Random Forest Model for Multi-dimensional Data*

Description

Fits a Random Forest model to multi-dimensional data and evaluates its performance by computing the Mean Absolute Bias on the same data (in-sample evaluation).

Usage

```
get.random.forest.MAB.xD(X, y, yt, ntree = 500)
```

Arguments

X	A matrix or data frame of predictor variables (dimension > 1).
y	A vector of response values (numeric for regression, factor for classification).
yt	A vector of true response values for evaluation.
ntree	Number of trees to grow in the random forest (default = 500).

Details

For classification tasks (when y is a factor), the function returns predicted probabilities for the second class level. For regression tasks, it returns the predicted values directly. Note that this function evaluates the model on the training data (in-sample).

Value

A list containing:

MAB Mean Absolute Bias calculated on the dataset

RMSE Root Mean Square Error

predictions Predictions made by the random forest model

residuals Absolute residuals

model The fitted random forest model object

parameters List containing ntree value used

Examples

```
## Not run:
# Regression example
set.seed(123)
n <- 100
X <- matrix(rnorm(n * 3), ncol = 3)
y <- X[,1] + X[,2]^2 + rnorm(n, sd = 0.1)
yt <- X[,1] + X[,2]^2  # True values

result <- get.random.forest.MAB.xD(X, y, yt, ntree = 500)
print(result$MAB)

# Classification example
y.class <- factor(ifelse(y > median(y), "high", "low"))
yt.prob <- plogis(yt - median(yt))  # True probabilities
result.class <- get.random.forest.MAB.xD(X, y.class, yt.prob, ntree = 500)
print(result.class$MAB)

## End(Not run)
```

`get.region.boundary` *Get Boundary Vertices of a Region in a Graph*

Description

Identifies the boundary vertices of a specified region in a graph, defined as vertices in the region that have at least one neighbor outside the region.

Usage

```
get.region.boundary(adj.list, region)
```

Arguments

`adj.list`	A list of integer vectors, where each vector contains the indices of vertices adjacent to the corresponding vertex. Indices must be 1-based.
`region`	An integer vector of vertex indices (1-based) defining the region for which to find boundary vertices.

Details

This function determines which vertices in a specified region are positioned at the boundary - meaning they have at least one neighbor that is not part of the region. These boundary vertices are often treated differently in graph-based algorithms, such as harmonic smoothing, where their values are typically fixed as constraints.

The boundary definition used matches the one implemented in the C++ harmonic smoothing functions, focusing on vertices that have connections to the outside of the region.

Value

An integer vector containing the indices of the boundary vertices, which is a subset of the input `region` vector. Returns an empty integer vector if no boundary vertices exist.

See Also

`perform.harmonic.smoothing`, `harmonic.smoother`

Examples

```
## Not run:
# Create a simple grid graph adjacency list (5x5 grid)
create_grid_adj_list <- function(n_rows, n_cols) {
  n <- n_rows * n_cols
  adj_list <- vector("list", n)
  for (i in 1:n_rows) {
    for (j in 1:n_cols) {
      v <- (i-1) * n_cols + j
      neighbors <- numeric(0)

      # Add neighbors (up, down, left, right)
      if (i > 1) neighbors <- c(neighbors, (i-2) * n_cols + j)  # up
      if (i < n_rows) neighbors <- c(neighbors, i * n_cols + j)  # down
      if (j > 1) neighbors <- c(neighbors, (i-1) * n_cols + (j-1))  # left
      if (j < n_cols) neighbors <- c(neighbors, (i-1) * n_cols + (j+1))  # right

      adj_list[[v]] <- neighbors
    }
  }
  return(adj_list)
}

# Create a 5x5 grid graph
grid_adj_list <- create_grid_adj_list(5, 5)

# Define a region (central 3x3 subgrid)
central_region <- c(7:9, 12:14, 17:19)

# Find boundary vertices of the central region
boundary <- get.region.boundary(grid_adj_list, central_region)
print(boundary)
# Expected output: c(7, 8, 9, 12, 14, 17, 18, 19)
# (all except the center vertex 13)

## End(Not run)
```

`get.shortest.path` *Get Shortest Path Between Two Vertices*

Description

Retrieves the shortest path between two vertices in a path graph object.

Usage

```
get.shortest.path(pg, from, to)
```

Arguments

`pg`	A path.graph object created by `create.path.graph`
`from`	Source vertex index (1-based)
`to`	Target vertex index (1-based)

Value

A list containing:

path Integer vector of vertex indices representing the path

length Numeric value of the total path length

hops Integer number of hops in the path

Returns NULL if no path exists between the vertices.

Examples

```
## Not run:
# Create a path graph
pg <- create.path.graph(graph, edge.lengths, h = 3)

# Get shortest path from vertex 1 to vertex 3
path_info <- get.shortest.path(pg, 1, 3)
if (!is.null(path_info)) {
  cat("Path:", path_info$path, "\n")
  cat("Length:", path_info$length, "\n")
}

## End(Not run)
```

`get.simplex.counts` *Get Simplex Counts for Nerve Complex*

Description

This function returns the number of simplices in each dimension.

Usage

```
get.simplex.counts(complex)
```

Arguments

`complex`	A nerve complex object

Value

Vector of simplex counts

`get.smooth.spline.MAB` *Estimate Mean Absolute Bias (MAB) of Smooth Spline Model*

Description

Fits a smooth spline model to a given dataset and evaluates its performance using cross-validation. The mean absolute error (MAE) is used as the performance metric. The function selects the optimal degrees of freedom for the spline model based on MAE.

Usage

```
get.smooth.spline.MAB(x, y, xt, yt, folds)
```

Arguments

`x`	A numeric vector representing the predictor variable for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A numeric vector representing the predictor variable for the test set.
`yt`	A numeric vector representing the response variable for the test set.
`folds`	A list of indices for cross-validation folds

Details

The function internally performs cross-validation to select the optimal degrees of freedom (between 3 and 5) that minimizes the MAE. It then fits a smooth spline model with the optimal degrees of freedom to the entire dataset and evaluates its performance on the test set.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the smooth spline model on the test set

residuals Absolute residuals

parameters List containing the optimal degrees of freedom

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- runif(100, 0, 10)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- seq(0, 10, length.out = 50)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Run the function
result <- get.smooth.spline.MAB(x, y, xt, yt, folds)
print(result$MAB)

## End(Not run)
```

`get.sphere.degree.props` *Calculate Degree Distribution Properties for Random Points on a Sphere*

Description

Simulates points uniformly on a sphere and computes the degree distribution properties of their k-nearest neighbor graph. For each simulation, points are generated, a k-NN graph is constructed, and the proportion of vertices with each degree is calculated. The function returns mean proportions and confidence intervals across all simulations.

Usage

```
get.sphere.degree.props(n.pts = 1000, n.sims = 100, k = 10, dim = 2)
```

Arguments

`n.pts`	numeric; Number of points to generate on the sphere for each simulation.
`n.sims`	numeric; Number of simulations to run.
`k`	numeric; Number of nearest neighbors to use in constructing the graph.
`dim`	numeric; Dimension of the sphere (e.g., 2 for circle, 3 for sphere).

Details

The function generates uniform random points on a sphere using `runif.sphere` and constructs k-nearest neighbor graphs using `create.single.iknn.graph`. For each simulation, it computes the proportion of vertices with each degree. The final results include means and 95% confidence intervals for these proportions across all simulations.

The confidence intervals are computed using the normal approximation: mean ± 1.96 * (standard deviation / sqrt(n.sims))

Value

A list containing:

mean.props Vector of mean proportions for each degree

ci.lower Vector of lower 95% confidence interval bounds

ci.upper Vector of upper 95% confidence interval bounds

degrees Vector of degree values corresponding to the proportions

Examples

```
## Not run:
# Calculate degree distribution properties for 1000 points on a circle
circle_props <- get.sphere.degree.props(n.pts = 1000, n.sims = 100, k = 10, dim = 2)

# Calculate for points on a sphere
sphere_props <- get.sphere.degree.props(n.pts = 1000, n.sims = 100, k = 10, dim = 3)

## End(Not run)
```

`get.spline.MAB` *Estimate Mean Absolute Bias (MAB) of Spline Model*

Description

Fits a spline model to a given dataset and evaluates its performance using cross-validation. The function supports both continuous and binary outcomes, using smooth splines for continuous data and B-splines with GLM for binary data. The optimal degrees of freedom is selected based on minimizing the Mean Absolute Error.

Usage

```
get.spline.MAB(x, y, xt, yt, folds, y.binary = FALSE)
```

Arguments

`x`	A numeric vector representing the predictor variable for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A numeric vector representing the predictor variable for the test set.
`yt`	A numeric vector representing the true response values for the test set.
`folds`	A list of indices for cross-validation folds.
`y.binary`	Logical indicating if y is a binary variable (default = FALSE).

Details

For continuous outcomes (`y.binary = FALSE`), the function uses `smooth.spline` to fit smoothing splines with different degrees of freedom (3, 4, 5) and selects the optimal value via cross-validation.

For binary outcomes (`y.binary = TRUE`), the function uses generalized linear models with B-spline basis functions via `glm` and `bs`. The degrees of freedom parameter controls the number of basis functions.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the spline model on the test set

residuals Absolute residuals

parameters List containing the optimal degrees of freedom

Examples

```
## Not run:
# Example with continuous outcome
set.seed(123)
x <- seq(0, 2*pi, length.out = 100)
y <- sin(x) + rnorm(100, sd = 0.2)
xt <- seq(0, 2*pi, length.out = 50)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 5, list = TRUE, returnTrain = TRUE)

# Fit spline model
result <- get.spline.MAB(x, y, xt, yt, folds, y.binary = FALSE)
print(result$MAB)

# Example with binary outcome
y.binary <- rbinom(100, 1, plogis(sin(x)))
yt.binary <- plogis(sin(xt))
result.binary <- get.spline.MAB(x, y.binary, xt, yt.binary, folds, y.binary = TRUE)
print(result.binary$MAB)

## End(Not run)
```

`get.spline.MAB.xD` *Estimate Mean Absolute Bias (MAB) of GAM Spline Model for Multi-dimensional Data*

Description

Fits a Generalized Additive Model (GAM) with spline smoothing to multi-dimensional data and evaluates its performance using in-sample predictions.

Usage

```
get.spline.MAB.xD(X, y, yt, folds, y.binary = FALSE)
```

Arguments

X	A matrix or data frame of predictor variables (dimension > 1).
y	A numeric vector of response values.
yt	A numeric vector of true response values for evaluation.
folds	A list of indices for cross-validation folds (currently not used in implementation).
y.binary	Logical indicating if y is a binary variable (default = FALSE).

Details

This function fits a GAM model with smooth terms for each predictor. For binary outcomes, it uses a binomial family with logit link and returns predicted probabilities. For continuous outcomes, it uses a Gaussian family. Note that this creates a tensor product smooth of all predictors.

Value

A list containing:

MAB Mean Absolute Bias calculated on the dataset

RMSE Root Mean Square Error

predictions Predictions made by the GAM model

residuals Absolute residuals

parameters Empty list (for consistency with other functions)

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 100
X <- matrix(rnorm(n * 2), ncol = 2)
colnames(X) <- c("x1", "x2")
y <- X[,1] + X[,2]^2 + rnorm(n, sd = 0.1)
yt <- X[,1] + X[,2]^2  # True values

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Fit GAM model and compute MAB
result <- get.spline.MAB.xD(X, y, yt, folds)
print(result$MAB)

# Example with binary data
y.binary <- rbinom(n, 1, plogis(y))
yt.binary <- plogis(yt)
result.binary <- get.spline.MAB.xD(X, y.binary, yt.binary, folds, y.binary = TRUE)
print(result.binary$MAB)

## End(Not run)
```

`get.subj.ids` *Get subject-specific visit information and indices*

Description

Extracts visit information and corresponding row indices in S.3d for a specific subject.

Usage

```
get.subj.ids(sID, S.3d, subjID, visit)
```

Arguments

`sID`	A subject ID to search for
`S.3d`	A matrix/array with rownames containing subject identifiers
`subjID`	A vector of subject IDs corresponding to rows in S.3d
`visit`	A vector of visit numbers corresponding to subjID entries

Value

A list containing:

`sVisit`	Named vector of visit numbers for the subject
`ii`	Indices of the subject's rows in S.3d
`ids`	Character vector of row names from S.3d for this subject

Examples

```
# Example usage:
# S.3d <- matrix(1:12, nrow=4, dimnames=list(c("s1_v1", "s1_v2", "s2_v1", "s2_v2")))
# subjID <- c("s1", "s1", "s2", "s2")
# visit <- c(1, 2, 1, 2)
# get.subj.ids("s1", S.3d, subjID, visit)
```

`get.svm.MAB` *Estimate Mean Absolute Bias (MAB) of Support Vector Machine Model*

Description

Fits a Support Vector Machine (SVM) model to a given dataset and evaluates its performance using cross-validation. The Mean Absolute Bias (MAB) is used as the performance metric. The function selects the optimal hyperparameters for the SVM model based on cross-validated MAE.

Usage

```
get.svm.MAB(
  x,
  y,
  xt,
  yt,
  folds,
  cost = 10^seq(-1, 1, length.out = 3),
  gamma = 10^seq(-1, 1, length.out = 3),
  y.binary = FALSE
)
```

Arguments

`x`	A matrix, data frame, or vector representing the predictor variables for the training set.
`y`	A numeric vector representing the response variable for the training set.
`xt`	A matrix, data frame, or vector representing the predictor variables for the test set.
`yt`	A numeric vector representing the response variable for the test set.
`folds`	A list of indices for cross-validation folds
`cost`	A numeric vector of candidate cost values to evaluate (default = 10^seq(-1, 1, length.out = 3)).
`gamma`	A numeric vector of candidate gamma values for the radial basis kernel (default = 10^seq(-1, 1, length.out = 3)).
`y.binary`	Logical indicating if y is binary (default = FALSE).

Details

The function iterates over a range of candidate cost and gamma values, evaluates each model using cross-validation, and selects the hyperparameters that minimize the MAE. Finally, the function fits an SVM model with the optimal hyperparameters to the entire dataset and evaluates its performance on the test set.

Value

A list containing:

MAB Mean Absolute Bias calculated on the test set

RMSE Root Mean Square Error

predictions Predictions made by the SVM model on the test set

residuals Absolute residuals

model The fitted SVM model object

parameters List containing optimal cost and gamma values

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- matrix(rnorm(100), ncol = 1)
y <- sin(x) + rnorm(100, sd = 0.1)
xt <- matrix(rnorm(50), ncol = 1)
yt <- sin(xt)

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Define hyperparameter grid
cost <- 10^seq(-1, 1, length.out = 3)
gamma <- 10^seq(-1, 1, length.out = 3)

# Run the function
result <- get.svm.MAB(x, y, xt, yt, folds, cost, gamma)
print(result$MAB)

## End(Not run)
```

---

get.svm.MAB.xD — *Estimate Mean Absolute Bias (MAB) of SVM Model for Multi-dimensional Data*

Description

Fits a Support Vector Machine (SVM) model to multi-dimensional data and evaluates its performance using cross-validation. The optimal hyperparameters are selected based on cross-validated MAE.

Usage

```
get.svm.MAB.xD(X, y, yt, folds, cost, gamma, y.binary = FALSE)
```

Arguments

`X`	A matrix or data frame of predictor variables (dimension > 1).
`y`	A numeric vector of response values.
`yt`	A numeric vector of true response values for evaluation.
`folds`	A list of indices for cross-validation folds.
`cost`	A numeric vector of candidate cost values to evaluate.
`gamma`	A numeric vector of candidate gamma values for the radial basis kernel.
`y.binary`	Logical indicating if y is binary (default = FALSE).

Details

The function performs grid search over cost and gamma parameters using cross-validation to find the optimal values. It then fits a final SVM model with these parameters and evaluates on the training data (in-sample).

Value

A list containing:

MAB Mean Absolute Bias calculated on the dataset

RMSE Root Mean Square Error

predictions Predictions made by the SVM model

residuals Absolute residuals

model The fitted SVM model object

parameters List containing optimal cost and gamma values

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 100
X <- matrix(rnorm(n * 3), ncol = 3)
y <- X[,1] + X[,2]^2 + rnorm(n, sd = 0.1)
yt <- X[,1] + X[,2]^2  # True values

# Create cross-validation folds
folds <- create.folds(y, k = 10, list = TRUE, returnTrain = TRUE)

# Define hyperparameter grid
cost <- 10^seq(-1, 1, length.out = 3)
gamma <- 10^seq(-1, 1, length.out = 3)

# Fit SVM model and compute MAB
result <- get.svm.MAB.xD(X, y, yt, folds, cost, gamma)
print(result$MAB)

## End(Not run)
```

`get.TN.S1.degree.props` *Generate Degree Distribution Properties in Tubular Neighborhood of Unit Circle*

Description

Generates random samples in a tubular neighborhood of the unit circle and computes degree distribution properties of the resulting k-nearest neighbor graph. The function performs multiple simulations to estimate mean proportions and confidence intervals for each degree.

Usage

```
get.TN.S1.degree.props(
  n.pts = 1000,
  n.sims = 100,
  k = 10,
  noise = 0.1,
  noise.type = "laplace"
)
```

Arguments

n.pts	Positive integer. Number of points to generate in each simulation.
n.sims	Positive integer. Number of simulations to run.
k	Positive integer. Number of nearest neighbors for graph construction. Must be less than n.pts.
noise	Non-negative numeric. Standard deviation of the noise added to the radius.
noise.type	Character string. Type of noise distribution to use ("laplace" or "normal").

Details

The function uses `generate.circle.data()` to create points and `create.single.iknn.graph()` to construct the k-nearest neighbor graph. It computes degree distributions for each simulation and aggregates the results to estimate population parameters.

Value

A list containing:

- mean.props: Vector of mean proportions for each degree
- ci.lower: Vector of lower 95% confidence interval bounds
- ci.upper: Vector of upper 95% confidence interval bounds
- degrees: Vector of degrees corresponding to the proportions

Examples

```
# Generate degree distribution properties with default parameters
results <- get.TN.S1.degree.props()

# Use custom parameters
results <- get.TN.S1.degree.props(
  n.pts = 500,
  n.sims = 50,
  k = 5,
  noise = 0.05,
  noise.type = "normal"
)
```

get.torus.degree.props

Calculate Degree Distribution Properties for Random Points on a Torus

Description

Simulates points uniformly on a torus and computes the degree distribution properties of their k-nearest neighbor graph. For each simulation, points are generated, a k-NN graph is constructed, and the proportion of vertices with each degree is calculated. The function returns mean proportions and confidence intervals across all simulations.

Usage

```
get.torus.degree.props(n.pts = 1000, n.sims = 100, k = 10, dim = 1)
```

Arguments

`n.pts`	numeric; Number of points to generate on the torus for each simulation.
`n.sims`	numeric; Number of simulations to run.
`k`	numeric; Number of nearest neighbors to use in constructing the graph.
`dim`	numeric; Dimension of the torus (e.g., 1 for circle, 2 for 2-torus).

Details

The function generates uniform random points on a torus using `runif.torus` and constructs k-nearest neighbor graphs using `create.single.iknn.graph`. For each simulation, it computes the proportion of vertices with each degree. The final results include means and 95% confidence intervals for these proportions across all simulations.

The confidence intervals are computed using the normal approximation: mean$\pm 1.96\times$(standard deviation$/\sqrt{n.sims}$)

Value

A list containing:

mean.props Vector of mean proportions for each degree

ci.lower Vector of lower 95% confidence interval bounds

ci.upper Vector of upper 95% confidence interval bounds

degrees Vector of degree values corresponding to the proportions

Examples

```
## Not run:
# Calculate degree distribution properties for 1000 points on a circle
circle_props <- get.torus.degree.props(n.pts = 1000, n.sims = 100, k = 10, dim = 1)

# Calculate for points on a 2-torus
torus_props <- get.torus.degree.props(n.pts = 1000, n.sims = 100, k = 10, dim = 2)

## End(Not run)
```

ggaussian *Generalized Gaussian Function*

Description

Computes the values of a generalized Gaussian function, which allows for adjusting the shape of the curve by altering the power of the absolute difference term.

Usage

```
ggaussian(x, offset = 0, h = 1, p = 2)
```

Arguments

`x`	A numeric vector or a single numeric value where the generalized Gaussian function is evaluated.
`offset`	The center of the Gaussian function (defaults to 0).
`h`	The scale parameter of the Gaussian function (defaults to 1).
`p`	The power to which the absolute difference is raised (defaults to 2).

Details

The generalized Gaussian function is defined as exp(-((abs(x - offset))^p) / h^p). The parameter `p` allows for adjusting the shape of the Gaussian curve: a value of `p` = 2 gives the standard Gaussian, `p` < 2 gives a curve with heavier tails, and `p` > 2 gives a curve with lighter tails. This function can be useful in various statistical applications where different shapes of Gaussian-like curves are required.

Value

A numeric vector or a single numeric value representing the value(s) of the generalized Gaussian function at the input `x`. The function's shape changes depending on the value of `p`.

Examples

```
## Not run:
x <- seq(-5, 5, length.out = 100)
plot(x, ggaussian(x, p = 2), type = "l")  # Standard Gaussian
plot(x, ggaussian(x, p = 1), type = "l")  # Laplace distribution (heavier tails)
plot(x, ggaussian(x, p = 3), type = "l")  # Lighter tails than Gaussian

## End(Not run)
```

`ggraph` *Create a gflow graph object*

Description

Create a gflow graph object

Usage

```
ggraph(adj.list, weight.list = NULL)
```

Arguments

`adj.list`	List where element i contains indices of vertices adjacent to vertex i
`weight.list`	List where element i contains weights of edges from vertex i

Value

An object of class "ggraph"

`gradient.trajectories.plot` *Plot Gradient Trajectories*

Description

Visualizes gradient flow trajectories from sample points on a contour plot.

Usage

```
gradient.trajectories.plot(grid, f.grid, sample.points = NULL, spacing = 10)
```

Arguments

`grid`	Grid object from create.grid
`f.grid`	Matrix of function values
`sample.points`	Matrix of (i,j) indices for sample trajectories (NULL for automatic)
`spacing`	Integer spacing between sample points (default 10)

Value

Invisible NULL

Examples

```
## Not run:
grid <- create.grid(50)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
gradient.trajectories.plot(grid, f.grid, spacing = 15)

## End(Not run)
```

gradient.trajectory.plot

Plot Gradient Trajectory

Description

Plots a single gradient trajectory with ascending and descending paths.

Usage

```
gradient.trajectory.plot(
  grid,
  trajectory,
  ascending.color = "orange",
  descending.color = "purple",
  line.width = 2,
  add = FALSE,
  ...
)
```

Arguments

`grid`	Grid object from create.grid
`trajectory`	Trajectory object from compute.gradient.trajectory
`ascending.color`	Color for ascending path (default "orange")
`descending.color`	Color for descending path (default "purple")
`line.width`	Line width (default 2)
`add`	Logical, whether to add to existing plot
`...`	Additional arguments passed to plot

Value

Invisible NULL

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) -((x-0.5)^2 + (y-0.5)^2)
f.grid <- evaluate.function.on.grid(f, grid)
traj <- compute.gradient.trajectory(15, 15, f.grid)
gradient.trajectory.plot(grid, traj)

## End(Not run)
```

`graph.3d` *Create a graph.3d object*

Description

Create a graph.3d object

Usage

```
graph.3d(plot.result, z)
```

Arguments

`plot.result`	A list returned by plot.graph()/plot.ggraph(), containing the graph and layout elements
`z`	A numeric vector containing z values for each vertex in the graph

Value

An object of class "graph.3d"

`graph.adj.mat` *Creates weights adjacency matrix given a matrix of edges and position matrix of the corresponding vertices.*

Description

The weights adjacency matrix A has the property that the (i,j) entry is the weight of the edge between i and j, if such edge exists, and 0 otherwise. The weights are distances in $R^{ncol(X)}$ between corresponding vertices (rows of X).

Usage

```
graph.adj.mat(X, E)
```

Arguments

`X`	A position matrix of vertices of the graph where each row represents a vertex position in R^d space (d = ncol(X)).
`E`	A two-column matrix where each row represents an edge. `E[i,1]` and `E[i,2]` are the indices of the vertices (rows of X) connected by the i-th edge.

Details

This function constructs a weighted adjacency matrix for a graph where:

- Vertices are represented by their positions in X
- Edges are specified by the index pairs in E
- Edge weights are the Euclidean distances between connected vertices

The resulting matrix is symmetric since the graph is treated as undirected. Self-loops (edges from a vertex to itself) will have weight 0.

Value

A symmetric n x n adjacency matrix where n = nrow(X). Element `A[i,j]` contains the Euclidean distance between vertices `i` and `j` if they are connected by an edge, and 0 otherwise.

Note

The function assumes 1-based indexing for the edge matrix E. All indices in E must be between 1 and nrow(X).

Examples

```
# Create a simple triangle graph
X <- matrix(c(0,0, 1,0, 0,1), ncol=2, byrow=TRUE)
E <- matrix(c(1,2, 2,3, 3,1), ncol=2, byrow=TRUE)
A <- graph.adj.mat(X, E)
print(A)
# Should show distances: A[1,2]=A[2,1]=1, A[2,3]=A[3,2]=sqrt(2), A[3,1]=A[1,3]=1
```

`graph.cltr.evenness` *Compute local cluster evenness with full cluster support*

Description

Computes the evenness (normalized entropy) of cluster label distribution in the expanded neighborhood of each vertex. The entropy is computed using the full set of cluster labels, including those not present in the neighborhood (i.e., with zero frequency), to ensure comparability across vertices.

Usage

```
graph.cltr.evenness(adj.list, cltr)
```

Arguments

`adj.list`	List of adjacency vectors (1-based indices) for each vertex.
`cltr`	A vector of cluster labels (factor, character, or numeric), one per vertex.

Value

Numeric vector of evenness values for each vertex.

graph.connected.components
Assign Vertices to Connected Components

Description

This function takes a graph representation and assigns each vertex to a connected component.

Usage

```
graph.connected.components(adj.list)
```

Arguments

adj.list	A list representing the adjacency list of the graph. Each element of the list corresponds to a vertex and contains the indices of its neighbors.

Value

An integer vector where each element represents the connected component ID for the corresponding vertex.

Examples

```
adj.list <- list(c(2,3), c(1), c(1), c(5), c(4))
components <- graph.connected.components(adj.list)
print(components)
```

graph.constrained.gradient.flow.trajectories
Compute Constrained Gradient Flow Trajectories

Description

Computes gradient flow trajectories on a graph while preventing basin-jumping between different regions of attraction by enforcing monotonicity along paths in a core graph.

Usage

```
graph.constrained.gradient.flow.trajectories(graph, core.graph, Ey)
```

Arguments

graph	List of integer vectors representing the h-th power graph pIG_k^h(X)
core.graph	List of integer vectors representing the core graph pIG_k(X)
Ey	Numeric vector of function values at vertices

Details

The function ensures that trajectories follow paths that are monotonic in the core graph, preventing undesirable jumps between basins of attraction of different local maxima. Indices in graph and core.graph are converted from 1-based (R) to 0-based (C++) indexing.

Value

List containing:

- lext - Matrix with two columns containing local minima and maxima for each vertex
- trajectories - List of integer vectors containing complete trajectories

Examples

```
## Not run:
trajectories <- graph.constrained.gradient.flow.trajectories(graph, core.graph, Ey)
local.extrema <- trajectories$lext
paths <- trajectories$trajectories

## End(Not run)
```

`graph.deg0.lowess` *Degree 0 LOWESS (Locally Weighted Average) on Graphs*

Description

Performs local constant fitting (degree 0 LOWESS) on graph data using a fixed bandwidth.

Usage

```
graph.deg0.lowess(
  adj.list,
  weight.list,
  y,
  bandwidth,
  kernel.type = 7L,
  dist.normalization.factor = 1.1,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list[[i]]` contains the indices of vertices adjacent to vertex i.
`weight.list`	A list of numeric vectors with edge weights corresponding to adjacencies. Each element `weight.list[[i]][j]` is the weight of the edge from vertex i to `adj.list[[i]][j]`.
`y`	A numeric vector of response values for each vertex in the graph.
`bandwidth`	A numeric value specifying the fixed bandwidth (radius) to use for all local neighborhoods.

`kernel.type`	Integer specifying the kernel function for weighting vertices: • 1: Epanechnikov • 2: Triangular • 4: Laplace • 5: Normal • 6: Biweight • 7: Tricube (default)
`dist.normalization.factor`	Numeric factor for normalizing distances when calculating kernel weights (default: 1.0).
`verbose`	Logical indicating whether to display progress information (default: FALSE).

Details

This function implements a simplified version of LOWESS for graph data where only degree 0 local models (weighted averages) are fit. For each vertex, the function:

1. Finds all vertices within the specified bandwidth radius
2. Computes a weighted average of response values using kernel weights
3. Returns the smoothed prediction

Value

A numeric vector of smoothed predictions for each vertex.

Examples

```
## Not run:
# Create a simple ring graph
n <- 100
set.seed(123)

# Create a ring graph
adj.list <- vector("list", n)
weight.list <- vector("list", n)

for (i in 1:n) {
  neighbors <- c(i-1, i+1)
  # Handle wrap-around for ring structure
  neighbors[neighbors == 0] <- n
  neighbors[neighbors == n+1] <- 1

  adj.list[[i]] <- neighbors
  weight.list[[i]] <- rep(1, length(neighbors))
}

# Generate response values with spatial pattern plus noise
y <- sin(2*pi*(1:n)/n) + rnorm(n, 0, 0.2)

# Apply degree 0 LOWESS with different bandwidths
bw1 <- 5
bw2 <- 10
bw3 <- 20
```

```
result1 <- graph.deg0.lowess(adj.list, weight.list, y, bw1)
result2 <- graph.deg0.lowess(adj.list, weight.list, y, bw2)
result3 <- graph.deg0.lowess(adj.list, weight.list, y, bw3)

# Plot results
plot(y, type="p", col="gray", main="Graph Degree 0 LOWESS")
lines(result1, col="blue", lwd=2)
lines(result2, col="red", lwd=2)
lines(result3, col="green", lwd=2)
legend("topright", legend=c(paste0("BW=", bw1),
                            paste0("BW=", bw2),
                            paste0("BW=", bw3)),
       col=c("blue", "red", "green"), lwd=2)

## End(Not run)
```

graph.deg0.lowess.cv.mat *Matrix Version of Graph-Based LOWESS with Cross-Validation*

Description

This function implements a matrix version of graph-based LOWESS (Locally Weighted Smoothing) of degree 0 with spatially-stratified cross-validation for bandwidth selection. It processes multiple response variables simultaneously, finding the optimal bandwidth for each response variable. This is more efficient than calling the vector version for each response variable separately.

Usage

```
graph.deg0.lowess.cv.mat(
  adj.list,
  weight.list,
  Y,
  min.bw.factor = 0.1,
  max.bw.factor = 0.5,
  n.bws = 10,
  log.grid = TRUE,
  kernel.type = 7L,
  dist.normalization.factor = 1.1,
  use.uniform.weights = FALSE,
  n.folds = 5,
  with.bw.predictions = FALSE,
  precision = 1e-06,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors. Each vector contains the indices of vertices adjacent to the vertex at the corresponding list position.

`weight.list`	A list of numeric vectors. Each vector contains the weights (distances) of edges to the adjacent vertices specified in adj.list.
`Y`	A list of numeric vectors. Each vector represents a response variable measured at each vertex of the graph.
`min.bw.factor`	Minimum bandwidth as a factor of graph diameter. The actual minimum bandwidth will be min.bw.factor * graph.diameter.
`max.bw.factor`	Maximum bandwidth as a factor of graph diameter. The actual maximum bandwidth will be max.bw.factor * graph.diameter.
`n.bws`	Number of bandwidths to test in the grid between min.bw and max.bw.
`log.grid`	Logical, if TRUE, use logarithmic spacing for the bandwidth grid; if FALSE, use linear spacing.
`kernel.type`	Integer specifying the kernel function to use. Possible values: 0=uniform, 1=triangular, 2=epanechnikov, 3=quartic, 4=triweight, 5=tricube, 6=gaussian, 7=cosine.
`dist.normalization.factor`	Factor for normalizing distances in kernel weights. A typical value is 1.1, which ensures all normalized distances fall within the effective support of most kernel functions.
`use.uniform.weights`	Whether to use uniform weights instead of kernel weights
`n.folds`	Number of cross-validation folds.
`with.bw.predictions`	Logical, if TRUE, compute and return predictions for all bandwidths; if FALSE, only return predictions for the optimal bandwidths.
`precision`	Numeric precision parameter for binary search and comparisons.
`verbose`	Logical, if TRUE, print progress information during computation.

Details

The algorithm performs the following steps:

1. Creates a maximal packing of vertices to serve as fold seed points
2. Assigns all vertices to the nearest seed point to form spatially coherent folds
3. For each candidate bandwidth, performs cross-validation across the folds for each response variable
4. Selects the bandwidth with the lowest cross-validation error for each response variable
5. Fits the final model with the optimal bandwidth for each response variable

Value

A list containing:

predictions A list of numeric vectors, where `predictions[[j]][i]` is the smoothed value for the jth response variable at the ith vertex

bw.predictions A nested list structure, where `bw.predictions[[j]][[bw.idx]]` contains predictions for the jth response variable using the bandwidth at index bw.idx. Only included if with.bw.predictions is TRUE.

bw.errors A list of numeric vectors, where `bw.errors[[j]][bw.idx]` is the cross-validation error for the jth response variable using the bandwidth at index bw.idx

bws Numeric vector of bandwidths used in cross-validation

opt.bws Numeric vector of optimal bandwidths for each response variable

opt.bw.idxs Integer vector of indices of the optimal bandwidths in the bws vector for each response variable (1-based indices)

Examples

```
## Not run:
# Create a simple graph with 3 response variables
adj.list <- list(c(2,3), c(1,3), c(1,2))
weight.list <- list(c(1,1), c(1,1), c(1,1))
Y <- list(c(1,2,3), c(4,5,6), c(7,8,9))

# Run the algorithm
result <- graph.deg0.lowess.cv.mat(
  adj.list, weight.list, Y,
  min.bw.factor = 0.1, max.bw.factor = 0.5,
  n.bws = 10, log.grid = TRUE, kernel.type = 2,
  dist.normalization.factor = 1.1, n.folds = 3,
  with.bw.predictions = FALSE, precision = 1e-6, verbose = TRUE
)

# Access results
result$predictions  # Smoothed values for each response variable
result$opt.bws      # Optimal bandwidths for each response variable

## End(Not run)
```

`graph.diffusion.matrix.smoother`
Graph Diffusion Matrix Smoother

Description

Applies graph diffusion smoothing to each column of a matrix independently. This is useful for smoothing multiple features or time series on the same graph structure simultaneously.

Usage

```
graph.diffusion.matrix.smoother(
  X,
  graph,
  edge.lengths,
  weights = NULL,
  n.time.steps = 100,
  step.factor = 0.5,
  normalize = 0,
  imputation.method = "local_mean_threshold",
  max.iterations = 10,
  convergence.threshold = 1e-06,
  ikernel = 1,
  dist.normalization.factor = 1.01,
  n.CVs = 0,
  n.CV.folds = 10,
  epsilon = 1e-10,
  seed = 0
)
```

Arguments

`X`	A numeric matrix where rows represent vertices and columns represent features to be smoothed.
`graph`	A list of integer vectors representing the adjacency structure (1-based indexing).
`edge.lengths`	A list of numeric vectors containing edge lengths.
`weights`	An optional numeric vector of vertex weights. If `NULL`, all vertices are weighted equally.
`n.time.steps`	An integer specifying the number of diffusion steps. Default is 100.
`step.factor`	A numeric value between 0 and 1 for step size. Default is 0.5.
`normalize`	An integer (0, 1, or 2) for normalization method. Default is 0.
`imputation.method`	Either an integer (0-4) or a character string: • 0 or "local_mean_threshold": Uses local mean threshold (default) • 1 or "neighborhood_matching": Neighborhood-based matching • 2 or "iterative_neighborhood_matching": Iterative version • 3 or "supplied_threshold": User-supplied threshold • 4 or "global_mean_threshold": Global mean threshold
`max.iterations`	Maximum iterations for iterative methods. Default is 10.
`convergence.threshold`	Convergence threshold. Default is 1e-6.
`ikernel`	An integer (0-4) for kernel type. Default is 1.
`dist.normalization.factor`	Distance normalization factor > 1. Default is 1.01.
`n.CVs`	Number of cross-validation runs. Default is 0.
`n.CV.folds`	Number of CV folds. Default is 10.
`epsilon`	Numerical tolerance. Default is 1e-10.
`seed`	Random seed for CV. Default is 0.

Value

A list containing:

`X.traj`	A list of matrices representing the diffusion trajectory.
`mean.cv.error`	A vector of mean CV errors per time step.

Examples

```
## Not run:
# Create example data
n <- 50
p <- 5
X <- matrix(rnorm(n * p), n, p)

# Create a ring graph
graph <- vector("list", n)
edge.lengths <- vector("list", n)
for(i in 1:n) {
  graph[[i]] <- c(ifelse(i == 1, n, i-1), ifelse(i == n, 1, i+1))
  edge.lengths[[i]] <- c(1, 1)
}

result <- graph.diffusion.matrix.smoother(
  X = X,
  graph = graph,
  edge.lengths = edge.lengths,
  n.time.steps = 20
)

## End(Not run)
```

`graph.diffusion.smoother`
Graph Diffusion Smoother

Description

A simplified interface for graph diffusion smoothing with cross-validation. This function provides a more accessible API while maintaining full functionality.

Usage

```
graph.diffusion.smoother(
  adj.list,
  weight.list,
  y,
  n.time.steps,
  step.factor,
  binary.threshold = 0.5,
  ikernel = 1,
  dist.normalization.factor = 1.1,
  n.CVs = 0,
  n.CV.folds = 10,
  epsilon = 1e-10,
  verbose = FALSE,
  seed = 0
)
```

Arguments

`adj.list`	A list where each element contains neighbor indices.
`weight.list`	A list where each element contains edge weights.
`y`	A numeric vector of values to smooth.
`n.time.steps`	Number of diffusion steps. Default is 100.
`step.factor`	Step size factor (0, 1). Default is 0.5.
`binary.threshold`	Threshold for binary classification. Default is 0.5.
`ikernel`	Kernel type (0-4). Default is 1.
`dist.normalization.factor`	Distance normalization > 1. Default is 1.1.
`n.CVs`	Number of CV runs. Default is 0.
`n.CV.folds`	Number of CV folds. Default is 10.
`epsilon`	Numerical tolerance. Default is 1e-10.
`verbose`	Print progress information. Default is FALSE.
`seed`	Random seed. Default is 0.

Value

A list containing smoothing results and CV information.

Examples

```
## Not run:
# Create simple chain graph
adj.list <- list(2, c(1,3), c(2,4), c(3,5), 4)
weight.list <- lapply(adj.list, function(x) rep(1, length(x)))
y <- c(1, 0, 1, 0, 1) + rnorm(5, 0, 0.1)

result <- graph.diffusion.smoother(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  n.time.steps = 20,
  n.CVs = 3,
  verbose = TRUE
)

## End(Not run)
```

`graph.edit.distance` *Calculate Graph Edit Distance (Pure R Implementation)*

Description

Computes the graph edit distance between two weighted graphs with identical vertex sets using a pure R implementation. This function considers edge insertions, deletions, and weight modifications.

Usage

```
graph.edit.distance(
  graph1.adj.list,
  graph1.weights,
  graph2.adj.list,
  graph2.weights,
  edge.cost = 1,
  weight.cost.factor = 0.1
)
```

Arguments

`graph1.adj.list`	A list of integer vectors representing the adjacency list of the first graph. Each element contains the neighbors of the corresponding vertex.
`graph1.weights`	A list of numeric vectors representing the edge weights of the first graph. Must have the same structure as `graph1.adj.list`.
`graph2.adj.list`	A list of integer vectors representing the adjacency list of the second graph.
`graph2.weights`	A list of numeric vectors representing the edge weights of the second graph.
`edge.cost`	A positive numeric scalar representing the cost of adding or removing an edge (default: 1).
`weight.cost.factor`	A non-negative numeric scalar representing the factor to scale weight differences (default: 0.1).

Details

The graph edit distance is calculated as:

- For edges present in one graph but not the other: `edge.cost`
- For edges present in both graphs with different weights: `abs(weight1 - weight2) * weight.cost.factor`

Both graphs must have the same number of vertices. The function assumes undirected graphs.

Value

A non-negative numeric value representing the graph edit distance between the two input graphs.

Examples

```
# Create two simple graphs
graph1.adj <- list(c(2, 3), c(1, 3), c(1, 2))
graph1.wts <- list(c(1, 2), c(1, 3), c(2, 3))

graph2.adj <- list(c(2, 3), c(1, 3), c(1, 2))
graph2.wts <- list(c(1.5, 2), c(1.5, 2.5), c(2, 2.5))

# Calculate distance
dist <- graph.edit.distance(graph1.adj, graph1.wts,
                            graph2.adj, graph2.wts)
print(dist)  # Weight differences only
```

graph.embedding *Create a 2D or 3D Embedding of a Graph*

Description

This function takes an adjacency list representation of a graph and produces a 2D or 3D embedding using either the Fruchterman-Reingold or Kamada-Kawai algorithm. It can handle both weighted and unweighted graphs.

Usage

```
graph.embedding(
  adj.list,
  weights.list = NULL,
  invert.weights = TRUE,
  dim = 3,
  method = c("fr", "kk"),
  verbose = TRUE
)
```

Arguments

`adj.list`	A list representing the adjacency list of the graph.
`weights.list`	An optional list of numeric vectors representing the weights of the edges. If provided, each element should correspond to the weights of the edges in the same position in adj.list.
`invert.weights`	Logical, if TRUE (default), inverts the weights (1/weight) before applying them to the Fruchterman-Reingold algorithm. Has no effect on Kamada-Kawai algorithm or when weights are not provided.
`dim`	Integer, either 2 or 3, specifying the dimension of the embedding. Default is 3.
`method`	Character string specifying the layout algorithm to use. Either "fr" for Fruchterman-Reingold or "kk" for Kamada-Kawai. Default is "fr".
`verbose`	Logical, if TRUE (default), print progress messages.

Details

The function can handle both weighted and unweighted graphs. When weights are provided, they are interpreted differently by the two layout algorithms:

- Fruchterman-Reingold (FR): Weights are interpreted as spring constants. Higher weights mean stronger springs, pulling vertices closer together. By default, weights are inverted (1/weight) so that higher original weights result in greater distances between vertices, matching the intuition of weights as distances.
- Kamada-Kawai (KK): Weights are interpreted as distances. Higher weights mean greater distances between vertices.

The `invert.weights` parameter allows control over this behavior for the FR algorithm. When TRUE (default), it inverts the weights for FR to match the distance interpretation. When FALSE, it uses the weights as-is, where higher weights pull vertices closer.

Value

A matrix representing the graph embedding, with each row corresponding to a vertex and containing its coordinates.

Examples

```
# Unweighted graph
adj.list <- list(c(2, 3), c(1, 3), c(1, 2))
embedding <- graph.embedding(adj.list, dim = 2, method = "fr")

# Weighted graph
weights.list <- list(c(0.1, 0.2), c(0.1, 0.3), c(0.2, 0.3))
embedding_weighted <- graph.embedding(adj.list, weights.list, dim = 2, method = "kk")
```

`graph.kernel.smoother` *Graph-Based Kernel Smoother with Buffer Zone Cross-Validation*

Description

Performs graph-based locally weighted smoothing (LOWESS-like) with adaptive bandwidth selection using spatially-aware cross-validation. The method implements buffer zones around test vertices during cross-validation to prevent spatial autocorrelation from biasing bandwidth selection.

Usage

```
graph.kernel.smoother(
  adj.list,
  weight.list,
  y,
  min.bw.factor = 0.01,
  max.bw.factor = 0.5,
  n.bws = 20,
  log.grid = TRUE,
  vertex.hbhd.min.size = 1,
  dist.normalization.factor = 1.1,
  use.uniform.weights = FALSE,
  buffer.hops = 2,
  auto.buffer.hops = TRUE,
  kernel.type = 7L,
  n.folds = 5,
  with.bw.predictions = FALSE,
  precision = 1e-06,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list[[i]]` contains the indices of vertices adjacent to vertex `i`. Vertices should be indexed from 1 to n.

`weight.list`	A list of numeric vectors with edge weights corresponding to adjacencies. Each element `weight.list[[i]][j]` is the weight of the edge from vertex `i` to `adj.list[[i]][j]`. Must be positive values.
`y`	A numeric vector of response values at each vertex. Length must match the number of vertices in the graph.
`min.bw.factor`	Minimum bandwidth as a factor of graph diameter. Must be between 0 and 1. Default is 0.01.
`max.bw.factor`	Maximum bandwidth as a factor of graph diameter. Must be between `min.bw.factor` and 1. Default is 0.5.
`n.bws`	Number of bandwidths to test. Must be at least 2. Default is 20.
`log.grid`	Logical. If `TRUE`, use logarithmic spacing for bandwidth grid; if `FALSE`, use linear spacing. Default is `TRUE`.
`vertex.hbhd.min.size`	Integer. Minimum number of vertices required in any neighborhood. Must be between 1 and 10% of total vertices. Default is 1.
`dist.normalization.factor`	Positive factor for normalizing distances in kernel weight computation. Values > 1 result in wider effective bandwidths. Default is 1.1.
`use.uniform.weights`	Logical. If `TRUE`, use uniform weights instead of kernel-based weights. Default is `FALSE`.
`buffer.hops`	Integer. Number of hops for buffer zone around test vertices during cross-validation. Must be non-negative. Default is 2.
`auto.buffer.hops`	Logical. If `TRUE`, automatically determine optimal buffer size based on spatial autocorrelation analysis. Default is `TRUE`.
`kernel.type`	Integer specifying the kernel function: • 1: Uniform kernel • 2: Triangular kernel • 3: Epanechnikov kernel • 4: Quartic (biweight) kernel • 5: Triweight kernel • 6: Gaussian kernel • 7: Tricube kernel (default)
`n.folds`	Integer. Number of cross-validation folds. Default is 5. The effective range is $[2, \lfloor n/2 \rfloor]$, where n is the number of vertices in the input graph. Values outside this range are automatically clamped: if `n.folds < 2`, it is reset to 2; if `n.folds > floor(n/2)`, it is reset to $\lfloor n/2 \rfloor$. A message is issued when values are clamped if `verbose = TRUE`.
`with.bw.predictions`	Logical. If `TRUE`, compute and store predictions for all tested bandwidths. Default is `FALSE`.
`precision`	Numeric. Precision for bandwidth grid computation. Must be positive. Default is 1e-6.
`verbose`	Logical. If `TRUE`, print progress information during computation. Default is `FALSE`.

Details

This function implements a graph-based extension of LOWESS (degree-0, i.e., locally weighted average) with spatially-stratified cross-validation using buffer zones. The algorithm proceeds as follows:

1. **Spatial fold creation**: Creates a maximal packing of vertices to serve as fold seed points, then assigns all vertices to the nearest seed point to form spatially coherent folds.
2. **Buffer zone construction**: For each fold, creates a buffer zone around test vertices by excluding vertices within a specified graph distance (hops) from the training set.
3. **Cross-validation**: For each candidate bandwidth, performs cross-validation across the spatially-separated folds.
4. **Bandwidth selection**: Selects the bandwidth that minimizes the cross-validation error.
5. **Final fitting**: Fits the final model using all data with the optimal bandwidth.

The kernel smoother at vertex i computes:

$$\hat{y}_i = \sum_{j \in N(i,h)} w_{ij} y_j / \sum_{j \in N(i,h)} w_{ij}$$

where $N(i,h)$ is the neighborhood of vertex i within bandwidth h, and w_{ij} are kernel weights based on graph distance.

Value

An object of class `"graph_kernel_smoother"`, which is a list containing:

`predictions` Numeric vector of smoothed predictions at each vertex.

`bw_predictions` If `with.bw.predictions = TRUE`, a list of prediction vectors for each tested bandwidth; otherwise `NULL`.

`bw_errors` Numeric vector of cross-validation errors for each tested bandwidth.

`bws` Numeric vector of bandwidth values tested.

`opt_bw` Numeric. The optimal bandwidth value selected by cross-validation.

`opt_bw_idx` Integer. Index of the optimal bandwidth in `bws`.

`buffer_hops_used` Integer. Number of hops used for the buffer zone (useful when `auto.buffer.hops = TRUE`).

Examples

```
n.pts <- 100
gm <- generate.1d.gaussian.mixture(
    n.points = n.pts,
    x.knot = c(0, 10),
    y.knot = c(10, 2.5),
    sd.knot = 1.5,
    x.offset = 3)
x <- sort(runif(n.pts, min = min(gm$x), max = max(gm$x)))
x.graph <- create.bi.kNN.chain.graph(k = 1, x = x, y = gm$y)

y.smooth <- approx(gm$x, gm$y, xout = x)$y
sigma <- 1
eps <- rnorm(n.pts, 0, sigma)
y <- y.smooth + eps
```

```
g <- ggraph(x.graph$adj.list, x.graph$edge.lengths)
plot(g, y.smooth)

plot(gm$x, gm$y, type = "l", las = 1, ylim = range(y), col = "red", xlab = "x", ylab = "y")
points(x, y)
legend("topright", legend = c("y.smooth", "y"), lty = c(1,NA), pch = c(NA,1),
col = c("red", "black"), inset = 0.1)

gks.res <- graph.kernel.smoother(x.graph$adj.list,
                                 x.graph$edge.lengths,
                                 y,
                                 min.bw.factor = 0.025,
                                 max.bw.factor = 0.5,
                                 n.bws = 20,
                                 log.grid = TRUE,
                                 vertex.hbhd.min.size = 3,
                                 dist.normalization.factor = 1.1,
                                 use.uniform.weights = FALSE,
                                 buffer.hops = 1,
                                 auto.buffer.hops = FALSE,
                                 kernel.type = 7L,
                                 n.folds = 10,
                                 with.bw.predictions = TRUE,
                                 verbose = TRUE)

# View results
print(gks.res)
summary(gks.res)

# Computing Mean Absolute and Mean Squared Errors
mae <- c()
for (i in seq(ncol(gks.res$bw_predictions))) {
    mae[i] <- mean(abs(y.smooth - gks.res$bw_predictions[,i]))
}
plot(mae, las = 1, type = 'b', xlab = "Bandwidth indices", ylab = "Mean Absolute Error")
which.min(mae)
gks.res$opt_bw_idx

plot(gks.res$bw_mean_abs_errors, las = 1, type = "b")
abline(v = gks.res$opt_bw_idx, lty = 2)
which.min(gks.res$bw_mean_abs_errors)

plot(gm$x, gm$y, type = "l", las = 1, ylim = range(y), col = "red")
points(x, y)
lines(x, gks.res$predictions, col = "blue")
```

graph.kmean *Compute the Kernel-weighted Graph Neighbor Mean*

Description

Calculates the kernel-weighted mean value of the neighbors for each vertex in a graph. The graph is represented as an adjacency list, with distances provided as edge lengths, and values associated with each vertex.

Usage

```
graph.kmean(
  adj.list,
  edge.lengths,
  y,
  kernel = 1,
  dist.normalization.factor = 1.01
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element contains the indices of neighboring vertices (1-based indexing).
`edge.lengths`	A list of numeric vectors containing the edge lengths (distances) corresponding to the neighbors in `adj.list`. Must have the same structure as `adj.list`.
`y`	A numeric vector of values associated with each vertex in the graph. Length must equal the number of vertices.
`kernel`	An integer specifying the kernel function to use: • 1 = Epanechnikov kernel • 2 = Triangular kernel • 3 = Truncated exponential kernel • 4 = Normal (Gaussian) kernel. Default is 1.
`dist.normalization.factor`	A positive numeric value (>= 1) used to normalize distances in the kernel computation. Default is 1.01.

Details

The function computes weighted averages where weights are determined by a kernel function applied to the edge distances. Smaller distances result in larger weights. The distance normalization factor controls the bandwidth of the kernel.

Value

A numeric vector containing the kernel-weighted neighbor mean for each vertex.

See Also

`graph.kmean.cv` for cross-validation

Examples

```
## Not run:
# Simple triangle graph
graph <- list(c(2, 3), c(1, 3), c(1, 2))
edge.lengths <- list(c(0.1, 0.2), c(0.1, 0.3), c(0.2, 0.3))
y <- c(1.0, 2.0, 3.0)

# Compute kernel-weighted means with Epanechnikov kernel
y.kwmean <- graph.kmean(graph, edge.lengths, y, kernel = 1)
print(y.kwmean)
```

```
## End(Not run)
```

`graph.kmean.cv` *Cross-validation for Kernel-weighted Graph Neighbor Mean*

Description

Performs k-fold cross-validation to evaluate the performance of the kernel-weighted neighbor mean algorithm on a graph. This function helps in selecting optimal parameters such as the kernel type and distance normalization factor.

Usage

```
graph.kmean.cv(
  graph,
  edge.lengths,
  y,
  kernel = 1,
  dist.normalization.factor = 1.01,
  n.CVs = 10,
  n.CV.folds = 10,
  seed = 0,
  use.weighted.MAD.error = FALSE
)
```

Arguments

`graph`	A list of integer vectors representing the adjacency list (1-based indexing).
`edge.lengths`	A list of numeric vectors containing edge lengths. Structure must match `graph`.
`y`	A numeric vector of response values at each vertex.
`kernel`	Either an integer (1-4) or a character string specifying the kernel function: • 1 or "epanechnikov" = Epanechnikov kernel (default) • 2 or "triangular" = Triangular kernel • 3 or "truncated_exponential" = Truncated exponential kernel • 4 or "normal" = Normal (Gaussian) kernel
`dist.normalization.factor`	A positive numeric value (>= 1) for distance normalization. Default is 1.01.
`n.CVs`	Number of cross-validation iterations. Default is 10.
`n.CV.folds`	Number of folds for each CV iteration. Must be >= 2. Default is 10.
`seed`	Integer seed for reproducibility. Default is 0.
`use.weighted.MAD.error`	Logical; if TRUE, uses weighted Mean Absolute Deviation for binary classification problems to handle class imbalance. Default is FALSE.

Details

The function performs repeated k-fold cross-validation. In each iteration, the data is randomly partitioned into k folds. Each fold is held out once while the model is trained on the remaining folds, and predictions are made for the held-out vertices.

When `use.weighted.MAD.error = TRUE`, the function applies class weights to handle imbalanced binary classification:

- Weight for class 0: 1 / (1 - q)
- Weight for class 1: 1 / q

where q is the proportion of class 1 samples.

Value

A numeric vector of cross-validation errors for each vertex. Values may be NaN for vertices that were excluded in all CV iterations.

See Also

`graph.kmean` for the main function

Examples

```
# Triangle graph example
graph <- list(c(2, 3), c(1, 3), c(1, 2))
edge.lengths <- list(c(0.1, 0.2), c(0.1, 0.3), c(0.2, 0.3))
y <- c(1.0, 2.0, 3.0)

# Perform cross-validation
cv.errors <- graph.kmean.cv(graph, edge.lengths, y,
                            kernel = "epanechnikov",
                            n.CVs = 5, n.CV.folds = 3)
print(cv.errors)
```

---

graph.low.pass.filter *Compute Low-Pass Filter of a Function Over a Graph*

Description

Computes the low-pass filter of a function over a graph using the Graph Fourier Transform (GFT). The filter is applied by summing the contributions of the eigenvectors starting from a specified index, effectively filtering out high-frequency components.

Usage

```
graph.low.pass.filter(init.ev, evectors, y.gft)
```

Arguments

`init.ev`	An integer specifying the index (1-based) of the first eigenvalue to include in the low-pass filter. Must be between 1 and the number of eigenvectors.
`evectors`	A numeric matrix of eigenvectors of the graph Laplacian. Each column corresponds to an eigenvector, ordered by their associated eigenvalues.
`y.gft`	A numeric matrix representing the Graph Fourier Transform (GFT) of the function over the graph. The first column should contain the GFT coefficients corresponding to each eigenvector.

Details

The Graph Fourier Transform decomposes a signal defined on the vertices of a graph into its frequency components using the eigenvectors of the graph Laplacian. Low-pass filtering retains only the low-frequency components (associated with smaller eigenvalues), which typically represent smooth variations over the graph structure.

Value

A numeric vector of length equal to the number of vertices, representing the filtered function values over the graph.

See Also

`graph.spectrum`

Examples

```
# Create example eigenvectors (2 vertices, 2 eigenvectors)
evectors <- matrix(c(1/sqrt(2), 1/sqrt(2), 1/sqrt(2), -1/sqrt(2)),
                  nrow = 2, ncol = 2)

# Example GFT coefficients
y.gft <- matrix(c(3, 1), nrow = 2, ncol = 1)

# Apply low-pass filter starting from the first eigenvector
filtered <- graph.low.pass.filter(1, evectors, y.gft)

# Apply low-pass filter using only the second eigenvector
filtered_high <- graph.low.pass.filter(2, evectors, y.gft)
```

`graph.mad` *Calculate Median Absolute Deviation for Graph Vertices*

Description

Computes the Median Absolute Deviation (MAD) for each vertex in a graph based on its neighborhood values. The MAD is calculated using the values of each vertex and its immediate neighbors in the graph.

Usage

```
graph.mad(graph, y)
```

Arguments

`graph`	A list where each element is a numeric vector containing the indices of neighboring vertices. The indices should be positive integers representing 1-based vertex numbering (as is standard in R).
`y`	A numeric vector containing the values associated with each vertex.

Details

For each vertex, the function:

- Considers the vertex's value and the values of its immediate neighbors
- Calculates the median of these values
- Computes the absolute deviations from this median
- Returns the median of these absolute deviations

Value

A numeric vector of the same length as `y`, where each element is the MAD value for the corresponding vertex. For vertices with no neighbors, the MAD value will be 0.

Note

- The graph structure should represent an undirected graph
- Vertex indices in the graph list should be valid (between 1 and length(y))
- Isolated vertices (those with no neighbors) will have a MAD value of 0

See Also

`mad` for the standard MAD calculation on vectors

Examples

```
# Create a simple chain graph with 3 vertices
g <- list(c(2), c(1,3), c(2))
values <- c(1, 10, 2)
graph.mad(g, values)
```

graph.MS.cx — *Constructs the Morse-Smale complex of a function defined on graph vertices*

Description

Constructs the Morse-Smale complex of a function defined on graph vertices

Usage

```
graph.MS.cx(graph, hop.list = NULL, core.graph, Ey)
```

Arguments

`graph`	A list representing the graph adjacency list. Each element of the list should be an integer vector specifying the neighboring vertex indices for a given vertex. The vertex indices should be 1-based.
`hop.list`	Optional list of hop constraints for gradient flow. If NULL, no constraints are applied. If provided, should contain hop information for each vertex.
`core.graph`	A list representing the core graph adjacency list used to constrain gradient flow paths. Each element should be an integer vector of 1-based neighbor indices.
`Ey`	A numeric vector of function values at vertices.

Details

This function computes gradient flow trajectories and Morse-Smale complex components for a function Ey defined on vertices of a graph G = pIG_k^h(X).

Key concepts and definitions:

1. Gradient Flow Trajectories:
   - For each vertex v, computes ascending and descending trajectories
   - Ascending trajectory follows steepest ascent path to local maximum
   - Descending trajectory follows steepest descent path to local minimum
   - Trajectories are constrained to valid paths in core graph pIG_k(X)
2. Morse-Smale Pro-cells:
   - A pro-cell is defined by a pair (M,m) where M is a local maximum and m is a local minimum
   - Contains all vertices that lie on gradient trajectories starting at m and ending at M
   - Formally: pro-cell(M,m) = {v ∈ V | there exists a trajectory through v from m to M}
3. Morse-Smale Cells:
   - Connected components of pro-cells after removing their defining extrema
   - For pro-cell(M,m), compute components of subgraph induced by: vertices in pro-cell(M,m) \ {M,m}
   - Unlike classical Morse theory, cells may not be disjoint

Implementation Overview:

1. Trajectory Computation: a. For each vertex v:

   - Compute ascending trajectory following steepest ascent
   - Compute descending trajectory following steepest descent
   - Store endpoints as local extrema
   - Update connectivity maps between extrema

2. Pro-cell Construction:
   - During trajectory computation, for each vertex v:
     - If trajectory connects local minimum m to maximum M
     - Add all trajectory vertices to pro-cell(M,m)

3. Cell Decomposition:
   - For each pro-cell(M,m): a. Remove M and m from vertex set b. Compute connected components of remaining subgraph c. Each component is a Morse-Smale cell

This function computes the Morse-Smale complex components:

- Gradient flow trajectories following steepest ascent/descent paths
- Local extrema (maxima and minima) of the function
- Morse-Smale pro-cells: sets of vertices whose trajectories flow from a specific minimum to a specific maximum
- Morse-Smale cells: connected components of pro-cells with extrema removed

The function converts indices between R's 1-based and C++'s 0-based systems. All returned indices are 1-based following R convention.

Value

An R list of class "graphMScx" with the following components:

- trajectories: A list where element i contains the gradient flow trajectory for vertex i (both descending and ascending paths)
- lmax_to_lmin: A list where element i contains indices of local minima connected to the i-th local maximum by gradient trajectories
- lmin_to_lmax: A list where element i contains indices of local maxima connected to the i-th local minimum by gradient trajectories
- local_maxima: Integer vector containing indices of all local maxima
- local_minima: Integer vector containing indices of all local minima
- procell_keys: List of integer pairs `(max_idx, min_idx)` identifying each pro-cell
- procells: List where element i contains all vertices in the i-th pro-cell
- cells: List where `cells[[i]]` contains the connected components of pro-cell i after removing its defining extrema

Note

Unlike classical Morse-Smale complexes on manifolds, the cells of the graph Morse-Smale complex may not be disjoint.

graph.spectral.embedding
Generate Spectral Embedding of a Graph

Description

Generates a spectral embedding of a graph into R^d using the eigenvectors corresponding to the smallest non-zero eigenvalues of the graph's Laplacian matrix. This embedding preserves the graph's structural properties in a low-dimensional space.

Usage

```
graph.spectral.embedding(evectors, dim, evalues = NULL)
```

Arguments

`evectors`	A numeric matrix of eigenvectors of the graph Laplacian. Each column corresponds to an eigenvector, ordered by their corresponding eigenvalues in descending order (largest to smallest).
`dim`	An integer specifying the dimension of the embedding space (R^{dim}). Must be between 1 and `ncol(evectors) - 1`.
`evalues`	An optional numeric vector of eigenvalues corresponding to the eigenvectors, also in descending order. If provided, the eigenvectors will be scaled by dividing by the square root of their corresponding eigenvalues, which can improve the embedding quality.

Details

Spectral embedding is a dimensionality reduction technique that uses the eigenvectors of the graph Laplacian to embed vertices in a low-dimensional space. The embedding preserves the connectivity structure of the graph, placing connected vertices close to each other in the embedded space.

The function excludes the trivial constant eigenvector (associated with eigenvalue 0) and uses the next `dim` eigenvectors corresponding to the smallest non-zero eigenvalues.

When eigenvalues are provided, the eigenvectors are normalized by dividing by the square root of their eigenvalues, which is a common practice in spectral clustering and can lead to better separation of clusters.

Value

A numeric matrix of dimension `nrow(evectors)` by `dim`, where each row represents the spectral embedding coordinates of a vertex in R^{dim}.

References

von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and Computing*, 17(4), 395-416.

See Also

`graph.spectrum`

Examples

```
# Create example eigenvectors for 5 vertices
set.seed(123)
evectors <- matrix(rnorm(5 * 5), nrow = 5, ncol = 5)

# Apply Gram-Schmidt to ensure orthogonality
evectors <- qr.Q(qr(evectors))

# Generate 2D embedding without eigenvalue scaling
embedding_2d <- graph.spectral.embedding(evectors, dim = 2)

# Generate 3D embedding with eigenvalue scaling
evalues <- c(5, 3, 2, 0.5, 0)  # Example eigenvalues in descending order
embedding_3d_scaled <- graph.spectral.embedding(evectors, dim = 3, evalues)
```

---

graph.spectral.filter *Spectral Filtering for Graph Signals*

Description

Implements a comprehensive framework for spectral filtering of signals defined on graph vertices. Supports multiple types of graph Laplacians, spectral filters, and parameter configurations to achieve different smoothing characteristics.

Usage

```
graph.spectral.filter(
  adj.list,
  weight.list,
  y,
  laplacian.type = 4L,
  filter.type = 1L,
  laplacian.power = 3L,
  kernel.tau.factor = 0.05,
  kernel.radius.factor = 5,
  kernel.type = 0L,
  kernel.adaptive = FALSE,
  kernel.min.radius.factor = 0.05,
  kernel.max.radius.factor = 0.99,
  kernel.domain.min.size = 4L,
  kernel.precision = 1e-06,
  n.evectors.to.compute = min(c(20L, length(y) - 2L)),
  n.candidates = 200L,
  log.grid = TRUE,
  with.t.predictions = TRUE,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list[[i]]` contains the indices of vertices adjacent to vertex `i`. Uses 1-based indexing. Must be non-empty.
`weight.list`	A list of numeric vectors where `weight.list[[i]]` contains the edge weights corresponding to the edges in `adj.list[[i]]`. Must have the same structure as `adj.list`.
`y`	Numeric vector of length equal to the number of vertices containing the signal values to be filtered. NA values are not allowed.
`laplacian.type`	Integer code specifying the type of graph Laplacian to use (default: 4). Valid values are 0-11. See Details section for descriptions of each type.
`filter.type`	Integer code specifying the type of spectral filter to apply (default: 1). Valid values are 0-11. See Details section for descriptions of each type.
`laplacian.power`	Positive integer specifying the power to which the Laplacian is raised (default: 3). Higher powers provide smoother filtering.
`kernel.tau.factor`	Positive numeric value in (0,1] determining kernel bandwidth as a fraction of graph diameter (default: 0.05).
`kernel.radius.factor`	Numeric value >= 1 multiplying the search radius when finding vertices within kernel range (default: 5.0).
`kernel.type`	Integer code (0-8) specifying the kernel function type (default: 0 for Gaussian). See Details section.
`kernel.adaptive`	Logical indicating whether to use locally adaptive kernel bandwidth (default: FALSE).
`kernel.min.radius.factor`	Minimum radius factor in (0,1) as fraction of graph diameter (default: 0.05).
`kernel.max.radius.factor`	Maximum radius factor in (0,1] as fraction of graph diameter (default: 0.99).
`kernel.domain.min.size`	Positive integer specifying minimum number of vertices required within radius for kernel computations (default: 4).
`kernel.precision`	Positive numeric value for radius determination precision (default: 1e-6).
`n.evectors.to.compute`	Positive integer specifying number of Laplacian eigenpairs to compute. Default is `min(20, length(y)-2)`.
`n.candidates`	Positive integer specifying number of filter parameter values to evaluate (default: 200).
`log.grid`	Logical indicating whether to use logarithmically-spaced filter parameters (default: TRUE).
`with.t.predictions`	Logical indicating whether to return smoothed signals for all parameter values (default: TRUE).
`verbose`	Logical indicating whether to print progress information (default: FALSE).

Details

The general spectral filtering process follows these steps:

1. Constructs a specific graph Laplacian operator based on the selected Laplacian type
2. Computes the eigendecomposition of this Laplacian (or its transformation)
3. Projects the signal onto the eigenbasis (Graph Fourier Transform)
4. Applies filter weights based on the eigenvalues and selected filter type
5. Reconstructs smoothed signals for a grid of filter parameter values
6. Selects the optimal parameter using Generalized Cross-Validation (GCV)

The function provides extensive flexibility through different Laplacian constructions, various spectral filter types, and parameter controls for localization and smoothness.

Value

An object of class `"graph_spectral_filter"`, which is a list containing:

evalues Numeric vector of eigenvalues of the Laplacian operator

evectors Matrix of corresponding eigenvectors (columns)

candidate_ts Numeric vector of filter parameter values tested

gcv_scores Numeric vector of Generalized Cross-Validation scores

opt_t_idx Integer index of the optimal parameter value

predictions Numeric vector of smoothed signal at optimal parameter

t_predictions Matrix of smoothed signals at each parameter (if requested)

laplacian_type Factor indicating Laplacian type used

filter_type Factor indicating filter type applied

laplacian_power Integer power to which Laplacian was raised

kernel_params List of kernel parameters used

compute_time_ms Numeric computation time in milliseconds

gcv_min_score Numeric minimum GCV score achieved

Laplacian Types

0 - STANDARD L = D - A, the combinatorial Laplacian. Provides basic smoothing that minimizes first differences across edges. Best for general-purpose smoothing on regularly structured graphs.

1 - NORMALIZED L_norm = D^(-1/2) L D^(-1/2), the normalized Laplacian. Accounts for varying vertex degrees, giving more balanced smoothing on irregular graphs. Recommended for graphs with highly variable connectivity.

2 - RANDOM_WALK L_rw = D^(-1) L, the random walk Laplacian. Similar to normalized Laplacian but with asymmetric normalization. Useful when modeling diffusion processes.

3 - KERNEL L_kernel = D_kernel - W_kernel, a Laplacian constructed using distance-based kernel weights rather than adjacency weights. Provides smoother spectral response, especially on irregular or noisy graphs.

4 - NORMALIZED_KERNEL Normalized version of the kernel Laplacian. Combines benefits of normalization and kernel-based edge weighting. Excellent for irregular graphs where geometric relationships matter.

5 - ADAPTIVE_KERNEL Kernel Laplacian with locally adaptive bandwidth. Adjusts smoothing automatically based on local graph density. Best for graphs with highly variable density.

6 - SHIFTED I - L, the shifted standard Laplacian. Inverts the spectrum to emphasize smooth components. Particularly effective for chain graphs and 1D signals.

7 - SHIFTED_KERNEL I - L_kernel, the shifted kernel Laplacian. Combines kernel-based edge weighting with spectral inversion.

8 - REGULARIZED L + ϵ*I, adds small regularization to ensure positive definiteness. Helps with numerical stability in filtering operations.

9 - REGULARIZED_KERNEL L_kernel + ϵ*I, regularized version of kernel Laplacian.

10 - MULTI_SCALE Weighted combination of kernel Laplacians at different scales. Captures both fine and coarse features simultaneously.

11 - PATH Path Laplacians for specialized graph structures.

Filter Types

0 - HEAT $\exp(-t\lambda)$, classic heat kernel filter. Provides smooth decay across frequencies with more pronounced filtering at higher frequencies.

1 - GAUSSIAN $\exp(-t\lambda^2)$), Gaussian spectral filter. More aggressive decay at higher frequencies than heat kernel. Produces very smooth results with minimal ringing.

2 - NON_NEGATIVE $\exp(-t\max(\lambda, 0))$, truncated heat kernel that only attenuates non-negative eigenvalues.

3 - CUBIC_SPLINE $1/(1 + t\lambda^2)$, filter that mimics cubic spline behavior. Minimizes second derivatives, producing the smoothest results while preserving linear trends.

4 - EXPONENTIAL $\exp(-t\sqrt(\lambda))$, less aggressive decay than heat kernel.

5 - MEXICAN_HAT $\lambda\exp(-t\lambda^2)$, band-pass filter that enhances mid-frequencies.

6 - IDEAL_LOW_PASS 1 for $\lambda < t$, 0 otherwise. Sharp cutoff filter.

7 - BUTTERWORTH $1/(1 + (\lambda/t)^(2n))$, smoother cutoff than ideal filter.

8 - TIKHONOV $1/(1 + t\lambda)$, first-order smoothing filter.

9 - POLYNOMIAL $(1 - \lambda/\lambda_{\max})^p$ for $\lambda < \lambda_{\max}$, polynomial decay filter.

10 - INVERSE_COSINE $\cos(\pi\lambda/(2\lambda_{\max}))$, smooth filter with cosine profile.

11 - ADAPTIVE Data-driven filter that adapts to signal properties.

Kernel Types

0 - GAUSSIAN $\exp(-d^2/\tau^2)$, Classic bell curve, smooth decay from center

1 - EXPONENTIAL $\exp(-d/\tau)$, Sharper peak, heavier tails than Gaussian

2 - HEAT $\exp(\frac{-d^2}{4\tau})$, Similar to Gaussian but with different scaling

3 - TRICUBE $(1 - (d/\tau)^3)^3$ for $d < \tau$, Compact support, smooth transition to zero

4 - EPANECHNIKOV $1 - (d/\tau)^2$ for $d < \tau$, Parabolic shape, optimal in statistical sense

5 - UNIFORM 1 for $d < \tau$, 0 otherwise, Equal weighting within radius

6 - TRIANGULAR $1 - |d/\tau|$ for $d < \tau$, Linear decay from center

7 - QUARTIC $(1 - (d/\tau)^2)^2$ for $d < \tau$, Similar to Gaussian but with compact support

8 - TRIWEIGHT $(1 - (d/\tau)^2)^3$ for $d < \tau$, Higher-order version of quartic

References

Shuman, D. I., Narang, S. K., Frossard, P., Ortega, A., & Vandergheynst, P. (2013). The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. IEEE Signal Processing Magazine, 30(3), 83-98.

See Also

plot.graph_spectral_filter for visualization, predict.graph_spectral_filter for prediction

Examples

```
## Not run:
# Create a simple chain graph with 30 vertices (for CRAN check timing)
n <- 30
adj_list <- vector("list", n)
weight_list <- vector("list", n)

# Build adjacency structure for chain
for (i in seq_len(n)) {
  adj_list[[i]] <- integer(0)
  weight_list[[i]] <- numeric(0)

  if (i > 1) {
    adj_list[[i]] <- c(adj_list[[i]], i - 1L)
    weight_list[[i]] <- c(weight_list[[i]], 1.0)
  }
  if (i < n) {
    adj_list[[i]] <- c(adj_list[[i]], i + 1L)
    weight_list[[i]] <- c(weight_list[[i]], 1.0)
  }
}

# Create a noisy signal
set.seed(123)
x <- seq(0, 1, length.out = n)
y <- sin(2 * pi * x) + rnorm(n, 0, 0.2)

# Standard Laplacian with heat kernel filter
result1 <- graph.spectral.filter(
  adj.list = adj_list,
  weight.list = weight_list,
  y = y,
  laplacian.type = 0L,    # STANDARD
  filter.type = 0L,       # HEAT
  laplacian.power = 1L,
  n.candidates = 50       # Reduced for faster execution
)

# Cubic spline-like smoothing
result2 <- graph.spectral.filter(
  adj.list = adj_list,
  weight.list = weight_list,
  y = y,
  laplacian.type = 0L,    # STANDARD
  filter.type = 3L,       # CUBIC_SPLINE
  laplacian.power = 2L,
  n.candidates = 50
)

# Compare results
plot(x, y, pch = 16, col = "gray", main = "Graph Spectral Filtering",
     xlab = "Position", ylab = "Value")
lines(x, result1$predictions, col = "blue", lwd = 2)
lines(x, result2$predictions, col = "red", lwd = 2)
legend("topright",
       legend = c("Noisy data", "Heat kernel", "Cubic spline"),
       pch = c(16, NA, NA),
       lty = c(NA, 1, 1),
       col = c("gray", "blue", "red"),
       lwd = c(NA, 2, 2))

# Print summary
summary(result1)

## End(Not run)
```

graph.spectral.lowess *Local Regression on Graphs Using Spectral Embedding*

Description

Performs local regression on graph data using spectral embeddings with adaptive bandwidth selection.

Usage

```
graph.spectral.lowess(
  adj.list,
  weight.list,
  y,
  n.evectors = 5,
  n.bws = 20,
  log.grid = TRUE,
  min.bw.factor = 0.05,
  max.bw.factor = 0.5,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  precision = 0.001,
  n.cleveland.iterations = 1L,
  verbose = FALSE
)
```

Arguments

adj.list	A list of integer vectors representing the adjacency list of the graph. Each element adj.list[[i]] contains the indices of vertices adjacent to vertex i.

`weight.list`	A list of numeric vectors with edge weights corresponding to adjacencies. Each element `weight.list[[i]][j]` is the weight of the edge from vertex i to `adj.list[[i]][j]`.
`y`	A numeric vector of response values for each vertex in the graph.
`n.evectors`	Integer specifying the number of eigenvectors to use in the spectral embedding (default: 5).
`n.bws`	Integer specifying the number of candidate bandwidths to evaluate (default: 10).
`log.grid`	Logical indicating whether to use logarithmic spacing for bandwidth grid (default: TRUE).
`min.bw.factor`	Numeric value specifying the minimum bandwidth as a fraction of graph diameter (default: 0.05).
`max.bw.factor`	Numeric value specifying the maximum bandwidth as a fraction of graph diameter (default: 0.25).
`dist.normalization.factor`	Numeric factor for normalizing distances when calculating kernel weights (default: 1.0).
`kernel.type`	Integer specifying the kernel function for weighting vertices: • 1: Epanechnikov • 2: Triangular • 4: Laplace • 5: Normal • 6: Biweight • 7: Tricube (default) Default is 7.
`precision`	Numeric value specifying the precision tolerance for binary search and optimization algorithms (default: 0.001).
`n.cleveland.iterations`	Number of Cleveland's robustness iterations (default: 0)
`verbose`	Logical indicating whether to display progress information (default: FALSE).

Details

This function implements a graph-based extension of LOWESS (Locally Weighted Scatterplot Smoothing) that uses spectral embedding to transform graph distances into a Euclidean space suitable for local linear regression. For each vertex, the function:

1. Finds all vertices within the maximum bandwidth radius
2. Creates a local spectral embedding using graph Laplacian eigenvectors
3. Fits weighted linear models at multiple candidate bandwidths
4. Selects the optimal bandwidth based on leave-one-out cross-validation error
5. Computes smoothed predictions using the optimal model

Value

A list containing:

- `predictions`: Numeric vector of smoothed values for each vertex
- `errors`: Numeric vector of leave-one-out cross-validation errors
- `scale`: Numeric vector of optimal bandwidths (local scales) for each vertex
- `graph.diameter`: Numeric scalar with computed graph diameter

Examples

```
## Not run:
# Create a simple graph with 100 vertices
n <- 100
set.seed(123)

# Create a ring graph
adj.list <- vector("list", n)
weight.list <- vector("list", n)

for (i in 1:n) {
  neighbors <- c(i-1, i+1)
  # Handle wrap-around for ring structure
  neighbors[neighbors == 0] <- n
  neighbors[neighbors == n+1] <- 1

  adj.list[[i]] <- neighbors
  weight.list[[i]] <- rep(1, length(neighbors))
}

# Generate response values with spatial pattern
y <- sin(2*pi*(1:n)/n) + rnorm(n, 0, 0.2)

# Apply spectral LOWESS
result <- graph.spectral.lowess(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  n.evectors = 5,
  verbose = TRUE
)

# Plot results
plot(y, type="l", col="gray", main="Graph Spectral LOWESS")
lines(result$predictions, col="red", lwd=2)

## End(Not run)
```

`graph.spectral.lowess.mat` *Matrix version of Spectral Graph Local Polynomial Regression*

Description

Performs locally weighted regression for multiple response variables simultaneously on a graph structure using spectral embeddings. This function is more efficient than applying the standard `graph_spectral_lowess` to each response variable separately, as it computes the graph Laplacian and eigenvectors only once.

Usage

```
graph.spectral.lowess.mat(
  adj.list,
  weight.list,
  Y,
  n.evectors = 5,
  n.bws = 20,
  log.grid = FALSE,
  min.bw.factor = 0.01,
  max.bw.factor = 0.5,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  precision = 1e-06,
  n.cleveland.iterations = 0L,
  with.errors = FALSE,
  with.scale = FALSE,
  verbose = TRUE
)
```

Arguments

`adj.list`	List where each element contains indices of adjacent vertices
`weight.list`	List where each element contains weights of adjacent edges
`Y`	Matrix or list of numeric vectors where each column/element represents a response variable at each vertex
`n.evectors`	Number of eigenvectors to use for spectral embedding
`n.bws`	Number of candidate bandwidths to evaluate (default: 20)
`log.grid`	Logical; whether to use logarithmic spacing for bandwidth grid (default: FALSE)
`min.bw.factor`	Factor for minimum bandwidth as fraction of graph diameter (default: 0.01)
`max.bw.factor`	Factor for maximum bandwidth as fraction of graph diameter (default: 0.5)
`dist.normalization.factor`	Factor for normalizing distances (default: 1.1)
`kernel.type`	Integer indicating kernel type: 0=gaussian, 1=epanechnikov, etc. (default: 7)
`precision`	Precision threshold for numerical calculations (default: 1e-6)
`n.cleveland.iterations`	Number of iterations for Cleveland's robust fitting (default: 3)
`with.errors`	Logical; whether to compute prediction errors (default: FALSE)
`with.scale`	Logical; whether to compute bandwidth scales (default: FALSE)
`verbose`	Logical; whether to print progress information (default: TRUE)

Details

This function efficiently handles multiple response variables by leveraging the fact that the graph structure processing (Laplacian calculation, spectral decomposition) only needs to be performed once. For each vertex, it identifies an appropriate neighborhood, creates a spectral embedding, and fits robust linear models for each response variable within this embedding space.

The optimal bandwidth is selected independently for each response variable at each vertex, balancing between overfitting (small bandwidth) and underfitting (large bandwidth).

Value

A list with components:

predictions	Matrix where each column contains smoothed values for a response variable
errors	Matrix of prediction errors (if with.errors=TRUE)
scale	Matrix of local bandwidth scales (if with.scale=TRUE)

Examples

```
## Not run:
# Create sample graph
n <- 100
g <- make.example.graph(n)

# Create multiple response variables (e.g., 3 columns)
Y <- matrix(rnorm(n*3), ncol=3)

# Apply spectral lowess to all columns at once
result <- graph.spectral.lowess.mat(g$adj.list, g$weight.list, Y)

# Plot results for first response variable
plot(Y[,1], result$predictions[,1], main="Original vs. Smoothed (Var 1)")
abline(0, 1, col="red", lty=2)

## End(Not run)
```

---

graph.spectral.ma.lowess

Local Regression on Graphs Using Spectral Embedding

Description

Performs local regression on graph data using spectral embeddings with adaptive bandwidth selection and model averaging.

Usage

```
graph.spectral.ma.lowess(
  adj.list,
  weight.list,
  y,
  n.evectors = 5,
  n.bws = 20,
  log.grid = TRUE,
  min.bw.factor = 0.05,
  max.bw.factor = 0.33,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  blending.coef = 0,
  precision = 0.001,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list\[\[i\]\]` contains the indices of vertices adjacent to vertex i.
`weight.list`	A list of numeric vectors with edge weights corresponding to adjacencies. Each element `weight.list\[\[i\]\]\[j\]` is the weight of the edge from vertex i to `adj.list\[\[i\]\]\[j\]`.
`y`	A numeric vector of response values for each vertex in the graph.
`n.evectors`	Integer specifying the number of eigenvectors to use in the spectral embedding (default: 5).
`n.bws`	Integer specifying the number of candidate bandwidths to evaluate (default: 10).
`log.grid`	Logical indicating whether to use logarithmic spacing for bandwidth grid (default: TRUE).
`min.bw.factor`	Numeric value specifying the minimum bandwidth as a fraction of graph diameter (default: 0.05).
`max.bw.factor`	Numeric value specifying the maximum bandwidth as a fraction of graph diameter (default: 0.25).
`dist.normalization.factor`	Numeric factor for normalizing distances when calculating kernel weights (default: 1.0).
`kernel.type`	Integer specifying the kernel function for weighting vertices: • 1: Epanechnikov • 2: Triangular • 4: Laplace • 5: Normal • 6: Biweight • 7: Tricube (default) Default is 7.
`blending.coef`	Numeric value specifying weighting of models in model avaraging.
`precision`	Numeric value specifying the precision tolerance for binary search and optimization algorithms (default: 0.001).
`verbose`	Logical indicating whether to display progress information (default: FALSE).

Details

This function implements a graph-based extension of LOWESS (Locally Weighted Scatterplot Smoothing) that uses spectral embedding to transform graph distances into a Euclidean space suitable for local linear regression. For each vertex, the function:

- Finds all vertices within the maximum bandwidth radius
- Creates a local spectral embedding using graph Laplacian eigenvectors
- Fits weighted linear models at multiple candidate bandwidths
- Selects the optimal bandwidth based on leave-one-out cross-validation error
- Computes smoothed predictions using the optimal model

Value

A list containing:

- `predictions`: Numeric vector of smoothed values for each vertex
- `errors`: Numeric vector of leave-one-out cross-validation errors
- `scale`: Numeric vector of optimal bandwidths (local scales) for each vertex
- `graph.diameter`: Numeric scalar with computed graph diameter

Examples

```
## Not run:
# Create a simple graph with 100 vertices
n <- 100
set.seed(123)

# Create a ring graph
adj.list <- vector("list", n)
weight.list <- vector("list", n)

for (i in 1:n) {
  neighbors <- c(i-1, i+1)
  # Handle wrap-around for ring structure
  neighbors\[neighbors == 0\] <- n
  neighbors\[neighbors == n+1\] <- 1

  adj.list\[\[i\]\] <- neighbors
  weight.list\[\[i\]\] <- rep(1, length(neighbors))
}

# Generate response values with spatial pattern
y <- sin(2*pi*(1:n)/n) + rnorm(n, 0, 0.2)

# Apply spectral LOWESS
result <- graph.spectral.lowess(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  n.evectors = 5,
  verbose = TRUE
)

# Plot results
plot(y, type="l", col="gray", main="Graph Spectral LOWESS")
lines(result$predictions, col="red", lwd=2)

## End(Not run)
```

`graph.spectral.smoother`
Graph Spectral Smoother utilizing spectrum of graph Laplacian

---

Description

Smooths a graph signal using a spectral smoothing technique based on the graph Laplacian. It determines the optimal number of low-eigenvalue eigenvectors for representing the graph function and returns the smoothed signal along with cross-validation errors.

Usage

```
graph.spectral.smoother(
  graph,
  edge.lengths,
  y,
  weights = NULL,
  imputation.method = 0,
  max.iterations = 10,
  convergence.threshold = 1e-06,
  apply.binary.threshold = TRUE,
  binary.threshold = 0.5,
  kernel = 1,
  dist.normalization.factor = 1.1,
  n.CVs = 0,
  n.CV.folds = 10,
  epsilon = 1e-10,
  min.plambda = 0.01,
  max.plambda = 0.2,
  seed = 0,
  verbose = FALSE
)
```

Arguments

`graph`	A list of integer vectors representing the graph's adjacency list. Each vector contains the indices of neighboring vertices (1-based).
`edge.lengths`	A list of edge lengths. Structure should match that of graph.
`y`	A numeric vector representing the graph signal to be smoothed.
`weights`	A numeric vector of weights for each vertex. If NULL, uniform weights are used.
`imputation.method`	Integer specifying the imputation method: 0: LOCAL_MEAN_THRESHOLD (default) 1: NEIGHBORHOOD_MATCHING 2: ITERATIVE_NEIGHBORHOOD_MATCHING 3: SUPPLIED_THRESHOLD 4: GLOBAL_MEAN_THRESHOLD
`max.iterations`	Integer. Maximum iterations for iterative matching method. Default is 10.
`convergence.threshold`	Numeric. Convergence threshold for iterative matching. Default is 1e-6.
`apply.binary.threshold`	Logical. Whether to apply binary thresholding. Default is TRUE.
`binary.threshold`	Numeric. Threshold for binary classification (0-1). Default is 0.5.
`kernel`	Integer specifying the kernel function: 0: Constant, 1: Epanechnikov (default), 2: Triangular, 3: Truncated Exponential, 4: Normal

`dist.normalization.factor`	Numeric. Scaling factor for distance normalization. Default is 1.01.
`n.CVs`	Integer. Number of cross-validation iterations. Default is 0.
`n.CV.folds`	Integer. Number of cross-validation folds. Default is 10.
`epsilon`	Numeric. Small positive constant for numerical stability. Default is 1e-10.
`min.plambda`	Numeric. Lower bound on proportion of eigenvectors to use. Default is 0.01.
`max.plambda`	Numeric. Upper bound on proportion of eigenvectors to use. Default is 0.20.
`seed`	Integer. Seed for random number generation. Default is 0.
`verbose`	Logical. Whether to print additional information. Default is FALSE.

Value

A list containing:

`optimal.num.eigenvectors`	Optimal number of eigenvectors used for smoothing
`y.smoothed`	Smoothed signal
`cv.errors`	Cross-validation errors for each fold
`mean.cv.errors`	Mean cross-validation errors across all folds
`median.cv.errors`	Median cross-validation errors across all folds
`Cmean.cv.errors`	Mean cross-validation errors computed in C++
`min.plambda`, `max.plambda`	Input lambda values
`min.num.eigenvectors`, `max.num.eigenvectors`	Range of eigenvectors used
`evalues`, `evectors`	Eigenvalues and eigenvectors of the graph Laplacian
`low.pass.ys`	Low-pass filtered versions of y for different numbers of eigenvectors

---

`graph.spectrum` *Compute Graph Spectrum*

Description

Computes the first nev eigenvalues and eigenvectors of the Laplacian matrix of a given graph. Supports both R-based and C-based implementations for performance optimization.

Usage

```
graph.spectrum(
  graph,
  nev = NULL,
  use.R = FALSE,
  return.Laplacian = FALSE,
  return.dense = FALSE
)
```

Arguments

`graph`	A list of integer vectors representing the adjacency list of the graph. Each element of the list corresponds to a vertex, and contains the indices of vertices adjacent to it.
`nev`	An integer specifying the number of eigenvalues and eigenvectors to compute. If `NULL` (default), computes all eigenvalues except the trivial zero eigenvalue.
`use.R`	Logical; if `TRUE`, uses the R-based implementation (slower but more portable). If `FALSE` (default), uses the C-based implementation for better performance.
`return.Laplacian`	Logical; if `TRUE`, includes the Laplacian matrix in the return value. Default is `FALSE`.
`return.dense`	Logical; if `TRUE` and `return.Laplacian` is `TRUE`, returns the Laplacian as a dense matrix. If `FALSE` (default), returns it as a sparse matrix. Only applicable when `use.R = FALSE`.

Details

The Laplacian matrix L of a graph is defined as L = D - A, where D is the degree matrix and A is the adjacency matrix. The eigenvalues and eigenvectors of the Laplacian matrix provide important spectral properties of the graph.

The function requires the **igraph** package when `use.R = TRUE`, and the **Matrix** package when returning sparse Laplacian matrices.

When `return.Laplacian = TRUE` on the C path and **Matrix** is not installed, the function automatically returns a **dense** Laplacian (sets `return.dense = TRUE`) to avoid a hard dependency on Matrix.

Value

A list containing:

`evalues`	A numeric vector of eigenvalues in descending order.
`evectors`	A matrix of eigenvectors, where each column corresponds to an eigenvalue.
`laplacian`	(Optional) The Laplacian matrix of the graph, included only if `return.Laplacian = TRUE`.

See Also

`graph.spectral.embedding`, `graph.low.pass.filter`

Examples

```
# Create a simple graph adjacency list
graph <- list(c(2, 3), c(1, 3, 4), c(1, 2, 4), c(2, 3))

# Compute spectrum using R implementation
spec_r <- graph.spectrum(graph, nev = 3, use.R = TRUE)

# Compute spectrum using C implementation
spec_c <- graph.spectrum(graph, nev = 3, use.R = FALSE)

# Get spectrum with Laplacian matrix
spec_lap <- graph.spectrum(graph, return.Laplacian = TRUE)
```

graph.vs.complex.regression.compare

Compare Graph vs Simplicial Complex Regression

Description

Compares regression performance using graph Laplacian vs full simplicial complex Laplacian.

Usage

```
graph.vs.complex.regression.compare(
  n.points = 200,
  k = 5,
  max.dim = 2,
  lambda = 1,
  n.test = 50,
  weight.type = "gaussian",
  weight.params = c(0.2),
  dim.weights = c(1, 0.5, 0.25)
)
```

Arguments

`n.points`	Number of points to generate
`k`	Number of nearest neighbors
`max.dim`	Maximum simplex dimension
`lambda`	Regularization parameter
`n.test`	Number of test points
`weight.type`	Weight scheme to use
`weight.params`	Parameters for weight scheme
`dim.weights`	Weights for each dimension's contribution

Value

A list containing comparison results

greedy.NN.transport.distance.1D

A function to compute the Wasserstein distance between two samples from one-dimensional distributions using nearest neighbors strategy.

Description

The samples are assumed to be of the same size.

Usage

```
greedy.NN.transport.distance.1D(x, y)
```

Arguments

`x`	A sample from a one-dimensional distribution.
`y`	A sample from a one-dimensional distribution.

`grid.critical.points.plot`
Plot Critical Points on Grid

Description

Plots local maxima and minima from grid-based critical point detection.

Usage

```
grid.critical.points.plot(
  grid,
  critical.points,
  max.color = "red",
  min.color = "blue",
  max.pch = 4,
  min.pch = 20,
  main = "Critical Points",
  point.size = 2,
  xlab = "",
  ylab = "",
  axes = FALSE,
  add = FALSE,
  ...
)
```

Arguments

`grid`	Grid object from create.grid
`critical.points`	List with 'maxima' and 'minima' matrices
`max.color`	Color for maxima (default "red")
`min.color`	Color for minima (default "blue")
`max.pch`	Symbol for maxima (default 4, cross)
`min.pch`	Symbol for minima (default 20, filled circle)
`main`	Plot title
`point.size`	Point size multiplier (default 2)
`xlab`	X-axis label
`ylab`	Y-axis label
`axes`	Logical, whether to show axes
`add`	Logical, whether to add to existing plot
`...`	Additional arguments passed to plot

Value

Invisible list with color information

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
critical <- find.critical.points(f.grid)
grid.critical.points.plot(grid, critical)

## End(Not run)
```

`harmonic.smoother` *Perform Harmonic Smoothing with Topology Tracking*

Description

Applies harmonic smoothing to function values defined on vertices of a graph while tracking how the topological structure (local extrema and their basins) evolves during the smoothing process. This helps identify the optimal level of smoothing that reduces noise while preserving significant features.

Usage

```
harmonic.smoother(
  adj.list,
  weight.list,
  values,
  region.vertices,
  max.iterations = 100,
  tolerance = 1e-06,
  record.frequency = 1,
  stability.window = 3,
  stability.threshold = 0.05
)
```

Arguments

`adj.list`	A list of integer vectors, where each vector contains indices of vertices adjacent to the corresponding vertex. Indices must be 1-based.
`weight.list`	A list of numeric vectors containing weights of edges corresponding to adjacencies in `adj.list`.
`values`	A numeric vector of function values defined at each vertex.
`region.vertices`	An integer vector of vertex indices (1-based) defining the region to be smoothed. Boundary vertices will have fixed values.
`max.iterations`	Integer scalar, the maximum number of relaxation iterations to perform. Default is 100.

`tolerance`	Numeric scalar, the convergence threshold for value changes. Default is 1e-6.
`record.frequency`	Integer scalar, how often to record states (every N iterations). Default is 1 (record every iteration).
`stability.window`	Integer scalar, number of consecutive iterations to check for topological stability. Default is 3.
`stability.threshold`	Numeric scalar in $[0, 1]$, maximum allowed difference in topology to consider stable. Default is 0.05.

Details

This function extends standard harmonic smoothing by monitoring the evolution of local extrema during the iterative process. It identifies a "sweet spot" where the topological structure stabilizes, indicating that noise has been removed without over-flattening important features.

The algorithm:

1. Iteratively performs harmonic smoothing on interior vertices
2. Periodically identifies local extrema and their basins
3. Monitors the stability of the topological structure
4. Identifies when the topological structure stabilizes

The function returns comprehensive information about the smoothing process, including all intermediate states and the detected stability point.

Value

A list of class "harmonic_smoother" containing:

`harmonic_predictions`	Numeric vector of smoothed function values
`i_harmonic_predictions`	Matrix of function values at each recorded iteration (columns are iterations)
`i_basins`	List of matrices representing extrema at each iteration
`stable_iteration`	Integer indicating the iteration at which topology stabilized
`topology_differences`	Numeric vector of differences between consecutive recorded iterations

See Also

`perform.harmonic.smoothing` for basic smoothing, `plot.harmonic_smoother` for visualization methods, `summary.harmonic_smoother` for summary statistics

Examples

```
## Not run:
# Create a simple grid graph
grid.graph <- create.graph.from.grid(10, 10)

# Create noisy function values
values <- sin(0.1 * seq_len(100)) + rnorm(100, 0, 0.1)
```

```
# Define a region for smoothing (center of the grid)
region <- 35:65

# Apply harmonic smoothing with topology tracking
result <- harmonic.smoother(
  grid.graph$adj.list,
  grid.graph$weight.list,
  values,
  region,
  max.iterations = 200,
  tolerance = 1e-8,
  record.frequency = 5,  # Record every 5 iterations
  stability.window = 3,
  stability.threshold = 0.05
)

# Get the smoothed values
smoothed.values <- result$harmonic_predictions

# Plot original vs smoothed values
plot(values, type = "l", col = "gray")
lines(smoothed.values, col = "red")

# Plot the evolution of topology differences
plot(result$topology_differences, type = "l",
     xlab = "Iteration", ylab = "Topology Difference")
abline(v = result$stable_iteration, col = "blue", lty = 2)

## End(Not run)
```

hdbscan.cltr *Apply HDBSCAN Clustering*

Description

This function applies HDBSCAN clustering to a given dataset (expressed as a matrix), and evaluates clustering results with different minimum cluster sizes using dunn and connectivity indices.

Usage

```
hdbscan.cltr(
  X,
  method = "dunn",
  min.pts = 5:50,
  min.prop = 0.1,
  max.prop = 0.5,
  n.test.cltr = 10,
  soft.K = 20,
  n.cores = 10,
  verbose = FALSE
)
```

Arguments

`X`	A matrix of the data to be clustered. Each row represents an observation.
`method`	A method (index) to be used for identifying the optimal number of clusters. Default is 'dunn'. Other options: 'connectivity', 'cl0.size' and 'all'.
`min.pts`	A vector of integers indicating the minimum cluster sizes to be evaluated. Default is 5:50. If NULL, the function will generate values based on min.prop, max.prop, and n.test.cltr.
`min.prop`	The minimum proportion of nrow(X) to be used in test clusterings when min.pts is NULL. Default is 0.1.
`max.prop`	The maximum proportion of nrow(X) to be used in test clusterings when min.pts is NULL. Default is 0.5.
`n.test.cltr`	The number of test clusters when min.pts is NULL. Default is 10.
`soft.K`	The number of nearest neighbors to consider when softening the cluster boundaries. Default is 20.
`n.cores`	The number of cores to use for parallel processing. Default is 10.
`verbose`	A logical value indicating whether progress should be displayed. Default is FALSE.

Value

A list containing the details of the clustering for three different metrics:

`cl0.size`	Vector of sizes of cluster labeled as '0' (noise) for each min.pts value
`n.cltrs`	Vector of number of clusters (excluding noise) for each min.pts value
`cl0.cltr`	Cluster assignments using the min.pts that minimizes cl0.size
`cl0.cltr.ext`	Extended cluster assignments after softening (cl0.size method)
`cl0.opt.k`	Optimal min.pts value that minimizes cl0.size
`cl0.cl0.size`	Size of cluster '0' for the cl0.size-optimal clustering
`cl0.n.cltrs`	Number of clusters for the cl0.size-optimal clustering
`cl0.cltr.freq`	Frequency table of clusters for cl0.size method
`cl0.cltr.ext.freq`	Frequency table of extended clusters for cl0.size method
`dunn.idx`	Vector of Dunn indices for each min.pts value
`dunn.cltr`	Cluster assignments using the min.pts that maximizes Dunn index
`dunn.cltr.ext`	Extended cluster assignments after softening (Dunn method)
`dunn.opt.k`	Optimal min.pts value that maximizes Dunn index
`dunn.cl0.size`	Size of cluster '0' for the Dunn-optimal clustering
`dunn.n.cltrs`	Number of clusters for the Dunn-optimal clustering
`dunn.cltr.freq`	Frequency table of clusters for Dunn method
`dunn.cltr.ext.freq`	Frequency table of extended clusters for Dunn method
`connectivity.idx`	Vector of connectivity indices for each min.pts value
`connectivity.cltr`	Cluster assignments using the min.pts that minimizes connectivity

`connectivity.cltr.ext`
: Extended cluster assignments after softening (connectivity method)

`connectivity.opt.k`
: Optimal min.pts value that minimizes connectivity index

`connectivity.cl0.size`
: Size of cluster '0' for the connectivity-optimal clustering

`connectivity.n.cltrs`
: Number of clusters for the connectivity-optimal clustering

`connectivity.cltr.freq`
: Frequency table of clusters for connectivity method

`connectivity.cltr.ext.freq`
: Frequency table of extended clusters for connectivity method

Note

This function requires the packages 'foreach', and 'doParallel'.

The Dunn and connectivity metrics are computed via the suggested package 'clValid'. If 'clValid' is not installed, use `method = "cl0.size"` (or avoid `method = "dunn"`, `"connectivity"`, or `"all"`).

The HDBSCAN clustering is performed via the suggested package 'dbscan'. This function will error if 'dbscan' is not installed.

Examples

```
## Not run:
# Generate sample data
set.seed(123)
X <- rbind(
  matrix(rnorm(100, mean = 0), ncol = 2),
  matrix(rnorm(100, mean = 5), ncol = 2)
)

# Apply HDBSCAN clustering
result <- hdbscan.cltr(X, min.pts = 5:50, soft.K = 20, verbose = TRUE)

# View optimal clustering for Dunn index
table(result$dunn.cltr)

## End(Not run)
```

`hgrid` *Construct a Hierarchical Uniform Grid Around a State Space*

Description

This function constructs a uniform grid G(X, w, epsilon) of width w in the k-dimensional Euclidean space R^k. The grid consists of points that are not more than epsilon away from the closest point of the given state space X.

Usage

```
hgrid(
  X,
  w,
  epsilon,
  p.exp = 0.05,
  K = 10,
  n.segments.per.axis = 10,
  n.itrs = 1,
  verbose = TRUE
)
```

Arguments

`X`	A numeric matrix representing the state space for which the grid is to be created.
`w`	The desired edge length of the grid. Points in the grid are spaced at intervals of w.
`epsilon`	The maximum allowed distance from the closest point of X for points in the grid.
`p.exp`	A numeric value representing the expansion factor of the bounding box. Must be between 0 and 1. Default is 0.05.
`K`	The number of nearest neighbors to estimate the denstiy of X at the given point.
`n.segments.per.axis`	An integer number specifying the number of subintervals (of equal length within each axis, but of potentially of different lengths between different axes) of each axis that will be used to construct box tiling of the given box.
`n.itrs`	The number of iterations of box sub-divisions. Default 1.
`verbose`	Set to TRUE for progress messages.

Details

The algorithm first constructs a box tiling of X. For each box, elements of X that are within that box are identified. The order in which the boxes are tested for the presence and identity of elements of X is done starting from the box with the highest density of X elements that is estimated using the distance, dK, to the K-th nearest neighbor. If a box has elements of X in it, it is added to a list of boxes-with-elements-of-X and the corresponding elements of X are deleted from dK. The process is iterated until dK is empty.

This process can be iterated, by performing a second round of rough subdivisions of the boxes in the list of boxes-with-elements-of-X.

When the process is finished a uniform grid of width w is created within each box from the list of boxes-with-elements-of-X.

Value

A numeric matrix containing the coordinates of the points in the hierarchical uniform grid. Each row represents a point in the grid, and the columns correspond to the dimensions of the space.

Examples

```
## Not run:
# Generate a synthetic 2D state space
X <- matrix(runif(200), ncol = 2)
# Construct the hierarchical grid with edge length 0.1 and epsilon 0.2
grid <- hgrid(X, w = 0.1, epsilon = 0.2)
plot(X, col = "red")
points(grid, col = "blue", pch = 3)

## End(Not run)
```

---

hist1 *Histogram with customized default settings*

Description

Creates a histogram with customized default settings for n.breaks, color, and appearance.

Usage

```
hist1(x, main = "", n.breaks = 100, col = "red", ...)
```

Arguments

`x`	A numeric vector of values for which the histogram is plotted.
`main`	The title of the plot. Default is an empty string.
`n.breaks`	The number of breaks (bins) in the histogram. Default is 100.
`col`	The color of the histogram bars. Default is "red".
`...`	Additional parameters passed to `hist`.

Value

Invisibly returns a list of class "histogram" as returned by `hist`.

Examples

```
# Basic usage
data <- rnorm(1000)
hist1(data)

# With custom settings
hist1(data, main = "Normal Distribution", n.breaks = 50, col = "blue")

# Using additional parameters
hist1(data, main = "Custom Histogram", n.breaks = 30, col = "green",
      xlab = "Values", ylab = "Frequency")
```

hist2 *Plot two overlapping histograms*

Description

Creates two overlapping histograms with customizable colors and transparency for comparing two distributions.

Usage

```
hist2(
  x1,
  x2,
  x1.lab,
  x2.lab,
  xlab = "",
  n.x1.breaks = 30,
  n.x2.breaks = 30,
  main = "",
  col.x1 = rgb(173, 216, 230, maxColorValue = 255, alpha = 80),
  col.x2 = rgb(255, 192, 203, maxColorValue = 255, alpha = 80),
  legend.pos = "topright"
)
```

Arguments

`x1`	A numeric vector for the first histogram.
`x2`	A numeric vector for the second histogram.
`x1.lab`	Label for x1 in the legend.
`x2.lab`	Label for x2 in the legend.
`xlab`	Label for the x-axis. Default is an empty string.
`n.x1.breaks`	The number of breaks for the first histogram. Default is 30.
`n.x2.breaks`	The number of breaks for the second histogram. Default is 30.
`main`	The title of the plot. Default is an empty string.
`col.x1`	Color of x1 histogram bars. Default is light blue with transparency.
`col.x2`	Color of x2 histogram bars. Default is light pink with transparency.
`legend.pos`	Position of the legend. Default is "topright". Must be one of "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "center" or "none".

Value

Invisibly returns a list containing:

`xlim`	The x-axis limits used in the plot
`ylim`	The y-axis limits used in the plot

Examples

```
# Basic usage
x1 <- rnorm(1000, mean = 0, sd = 1)
x2 <- rnorm(1000, mean = 2, sd = 1.5)
hist2(x1, x2, x1.lab = "Group A", x2.lab = "Group B")

# With custom settings
hist2(x1, x2, x1.lab = "Control", x2.lab = "Treatment",
      xlab = "Values", main = "Comparison of Distributions",
      n.x1.breaks = 50, n.x2.breaks = 50)

# Custom colors and no legend
hist2(x1, x2, x1.lab = "A", x2.lab = "B",
      col.x1 = "lightgreen", col.x2 = "lightcoral",
      legend.pos = "none")
```

---

hist3 *Plot three overlapping histograms*

Description

Creates three overlapping histograms with customizable colors and transparency for comparing three distributions.

Usage

```
hist3(
  x1,
  x2,
  x3,
  x1.lab,
  x2.lab,
  x3.lab,
  xlab = "",
  n.x1.breaks = 30,
  n.x2.breaks = 30,
  n.x3.breaks = 30,
  main = "",
  ylim.max = NA,
  col.x1 = rgb(255, 0, 0, maxColorValue = 255, alpha = 90),
  col.x2 = rgb(190, 190, 190, maxColorValue = 255, alpha = 90),
  col.x3 = rgb(0, 0, 255, maxColorValue = 255, alpha = 90),
  legend.pos = "topright"
)
```

Arguments

`x1`	A numeric vector for the first histogram.
`x2`	A numeric vector for the second histogram.
`x3`	A numeric vector for the third histogram.

`x1.lab`	Label for x1 in the legend.
`x2.lab`	Label for x2 in the legend.
`x3.lab`	Label for x3 in the legend.
`xlab`	Label for the x-axis. Default is an empty string.
`n.x1.breaks`	The number of breaks for the first histogram. Default is 30.
`n.x2.breaks`	The number of breaks for the second histogram. Default is 30.
`n.x3.breaks`	The number of breaks for the third histogram. Default is 30.
`main`	The title of the plot. Default is an empty string.
`ylim.max`	The upper y-axis limit. If NA (default), automatically determined.
`col.x1`	Color of x1 histogram bars. Default is semi-transparent red.
`col.x2`	Color of x2 histogram bars. Default is semi-transparent gray.
`col.x3`	Color of x3 histogram bars. Default is semi-transparent blue.
`legend.pos`	Position of the legend. Default is "topright". Must be one of "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "center" or "none".

Value

Invisibly returns a list containing:

`xlim`	The x-axis limits used in the plot
`ylim`	The y-axis limits used in the plot

Examples

```
# Basic usage
x1 <- rnorm(1000, mean = 0, sd = 1)
x2 <- rnorm(1000, mean = 2, sd = 1.2)
x3 <- rnorm(1000, mean = 1, sd = 0.8)
hist3(x1, x2, x3,
      x1.lab = "Group A", x2.lab = "Group B", x3.lab = "Group C")

# With custom settings
hist3(x1, x2, x3,
      x1.lab = "Control", x2.lab = "Treatment 1", x3.lab = "Treatment 2",
      xlab = "Measurements", main = "Three-way Comparison",
      n.x1.breaks = 50, n.x2.breaks = 50, n.x3.breaks = 50)

# Fixed y-axis limit and custom colors
hist3(x1, x2, x3,
      x1.lab = "A", x2.lab = "B", x3.lab = "C",
      ylim.max = 0.5,
      col.x1 = "lightgreen", col.x2 = "lightblue", col.x3 = "lightcoral")
```

`hor.error.bar` *Add Horizontal Error Bar to Plot*

Description

Adds a horizontal error bar to an existing 2D plot

Usage

```
hor.error.bar(xmin, xmax, y, dyy = 0.025, lwd = 1, col = "black")
```

Arguments

`xmin`	Left limit of error bar.
`xmax`	Right limit of error bar.
`y`	Y-coordinate for the error bar.
`dyy`	Vertical offset for end caps.
`lwd`	Line width.
`col`	Color of error bar.

Value

Invisibly returns NULL.

Examples

```
plot(rnorm(10), 1:10, xlim = c(-3, 3))
hor.error.bar(-1, 1, 5, col = "blue")
```

`hungarian.frobenius.graph.matching` *Hungarian-Frobenius Graph Matching Algorithm*

Description

This function implements the Hungarian-Frobenius Graph Matching Algorithm to measure the similarity between two graphs. It uses the Hungarian algorithm to find the optimal permutation of vertices that minimizes the cost of matching the vertices between the graphs, and then calculates the Frobenius norm distance between the distance matrices of the graphs after applying the optimal permutation.

Usage

```
hungarian.frobenius.graph.matching(graph1, graph2)
```

Arguments

`graph1`	A list representing the adjacency list of the first graph.
`graph2`	A list representing the adjacency list of the second graph.

Value

The similarity score between the two graphs, ranging from 0 to 1, where 0 indicates perfect similarity.

`identical.vertex.set.weighted.graph.similarity`

Weighted Graph Matching between two graphs with identical vertex sets.

Description

This function computes a similarity measure between two weighted graphs derived from the same dataset. It assumes that both graphs have identical vertex sets and indexing, eliminating the need for vertex permutation.

Usage

```
identical.vertex.set.weighted.graph.similarity(
  graph1.adj.list,
  graph1.weights,
  graph2.adj.list,
  graph2.weights,
  calculate.normalized.deviation = FALSE
)
```

Arguments

`graph1.adj.list`	A list representing the adjacency list of the first graph.
`graph1.weights`	A list representing the weights of the first graph.
`graph2.adj.list`	A list representing the adjacency list of the second graph.
`graph2.weights`	A list representing the weights of the second graph.
`calculate.normalized.deviation`	Logical indicating whether to calculate the normalized deviation (default: FALSE). If TRUE, returns normalized deviation; if FALSE, returns raw distance difference.

Details

The function performs the following steps:

1. Converts the input lists to igraph objects.
2. Calculates the distance matrices for both graphs using edge weights.
3. Computes the L1 norm of the difference between these distance matrices.
4. Normalizes the result using the maximum possible deviation, which is based on the largest distance found in either graph.

Value

A numeric value between 0 and 1 representing the normalized deviation between the two graphs. A value closer to 0 indicates higher similarity between the graphs.

Note

- This function assumes that the graphs are undirected.
- The weights should be stored in the input lists as a vector named 'weight'.
- For disconnected graphs, infinite distances are ignored in calculating the maximum distance.

Examples

```
# Create adjacency lists (as lists, not matrices)
g1.adj.list <- list(c(2), c(3), c(1))  # vertex 1->2, vertex 2->3, vertex 3->1
g1.weights <- list(c(1), c(2), c(3))   # corresponding weights

g2.adj.list <- list(c(2), c(3), c(1))  # same structure
g2.weights <- list(c(1), c(3), c(2))   # different weights

similarity <- identical.vertex.set.weighted.graph.similarity(
  g1.adj.list, g1.weights, g2.adj.list, g2.weights
)
print(similarity)
```

`identify_points_within_box`
Identify Points Within a Box

Description

This function identifies the points within a given box specified by the lower and upper bounds.

Usage

```
identify_points_within_box(X, L, R)
```

Arguments

X	A matrix or data frame containing the points in the space.
L	A numeric vector specifying the lower bounds of the box.
R	A numeric vector specifying the upper bounds of the box.

Value

A matrix or data frame containing only the points that fall within the specified box.

Examples

```
## Not run:
X <- matrix(runif(100), ncol = 2)
L <- c(0.2, 0.2)
R <- c(0.5, 0.5)
points_within_box <- identify_points_within_box(X, L, R)

## End(Not run)
```

---

init.version.nn.distance.ratio.estimator

Estimate Relative Entropy Using Nearest Neighbor Distance Ratio

---

Description

This function estimates the relative entropy (Kullback-Leibler divergence) between two datasets using the nearest neighbor distance ratio method. The intuition behind this approach is that if two distributions are similar, the ratio of distances to nearest neighbors within a set and to the other set should be close to 1. If the distributions differ, this ratio will deviate from 1.

The algorithm works by:

1. Finding the k-nearest neighbors for each point within its own set and in the other set.
2. Computing the ratios of these distances.
3. Using the log of these ratios to estimate the relative entropy.

This method is particularly useful in high-dimensional spaces where traditional density estimation techniques may fail due to the curse of dimensionality.

Usage

```
init.version.nn.distance.ratio.estimator(X, Y, k = 1)
```

Arguments

X	A matrix or data frame representing the first dataset.
Y	A matrix or data frame representing the second dataset.
k	The number of nearest neighbors to consider (default is 1).

Value

A numeric value representing the estimated relative entropy.

References

Wang, Q., Kulkarni, S. R., & Verdú, S. (2009). Divergence estimation for multidimensional densities via k-nearest-neighbor distances. IEEE Transactions on Information Theory, 55(5), 2392-2405.

Examples

```
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- nn.distance.ratio.estimator(X, Y)
print(result)
```

---

instrumented.gds *Instrumented Graph Diffusion Smoother*

Description

Performs graph diffusion smoothing with comprehensive instrumentation and adaptive step sizes. This function collects detailed performance metrics and allows fine-tuned control over the diffusion process.

Usage

```
instrumented.gds(
  graph,
  edge.lengths,
  y,
  y.true,
  n.time.steps,
  base.step.factor,
  use.pure.laplacian = FALSE,
  ikernel = 1,
  kernel.scale = 1,
  dist.normalization.factor = 1.01,
  increase.factor = 1.1,
  decrease.factor = 0.8,
  oscillation.factor = 0.5,
  min.step = 0.01,
  max.step = 2
)
```

Arguments

`graph`	List of integer vectors representing adjacency structure.
`edge.lengths`	List of numeric vectors containing edge lengths.
`y`	Numeric vector of initial vertex values.
`y.true`	Numeric vector of ground truth values for evaluation.
`n.time.steps`	Integer number of diffusion steps.
`base.step.factor`	Initial step size for all vertices.
`use.pure.laplacian`	Logical; if `TRUE`, uses uniform weights. Default is `FALSE`.
`ikernel`	Integer kernel type (1-3). Default is 1.
`kernel.scale`	Scale parameter for kernels. Default is 1.0.

`dist.normalization.factor`	Not used in current implementation.
`increase.factor`	Multiplier for increasing step size. Default is 1.1.
`decrease.factor`	Multiplier for decreasing step size. Default is 0.8.
`oscillation.factor`	Multiplier when oscillating. Default is 0.5.
`min.step`	Minimum allowed step size. Default is 0.01.
`max.step`	Maximum allowed step size. Default is 2.0.

Value

An object of class `"instrumented.gds"` containing detailed metrics and trajectories.

Examples

```
## Not run:
# Example with simple graph
graph <- list(c(2), c(1,3), c(2))
edge.lengths <- list(c(1), c(1,1), c(1))
y.true <- c(0, 1, 0)
y.noisy <- y.true + rnorm(3, 0, 0.1)

result <- instrumented.gds(
  graph = graph,
  edge.lengths = edge.lengths,
  y = y.noisy,
  y.true = y.true,
  n.time.steps = 50,
  base.step.factor = 0.5
)

## End(Not run)
```

`inv.logit` *Inverse logit tranformation x -> 1 / (1 + exp(-x)) from real numbers into the unit interval \((0,1)\).*

Description

Inverse logit tranformation x -> 1 / (1 + exp(-x)) from real numbers into the unit interval \((0,1)\).

Usage

```
inv.logit(x)
```

Arguments

`x`	A numerical vector.

Value

Inverse logit transformed data.

`itriangle.plot` *itriangle.plot*

Description

Custom plot function to draw vertices as inverted triangles in an igraph plot.

Usage

```
itriangle.plot(coords, v = NULL, params)
```

Arguments

`coords`	A matrix of vertex coordinates. Each row should contain the x and y coordinates of a vertex.
`v`	An optional vector of vertex indices to be plotted. If NULL, all vertices will be plotted.
`params`	A list of plotting parameters, typically obtained from the igraph `params` function.

Details

This function is used to plot vertices as inverted equilateral triangles in an igraph graph. The triangles are sized to have approximately the same area as circles of the same vertex size. It is intended to be used with the igraph `add_shape` function to add the "itriangle" (inverted triangle) shape to graph vertices.

The function adjusts the size of the triangles to match the area of circles with the same vertex size. This ensures visual consistency in the plot. The function also handles vertex colors, frame colors, and frame widths.

Examples

```
## Not run:
  library(igraph)

  # Define the inverted triangle plot function
  itriangle.plot <- function(coords, v = NULL, params) {
      vertex.color <- params("vertex", "color")
      if (length(vertex.color) != 1 && !is.null(v)) {
          vertex.color <- vertex.color\[v\]
      }
      vertex.frame.color <- params("vertex", "frame.color")
      if (length(vertex.frame.color) != 1 && !is.null(v)) {
          vertex.frame.color <- vertex.frame.color\[v\]
      }
      vertex.frame.width <- params("vertex", "frame.width")
      if (length(vertex.frame.width) != 1 && !is.null(v)) {
          vertex.frame.width <- vertex.frame.width\[v\]
```

```
        }
        vertex.size <- 1/200 * params("vertex", "size")
        if (length(vertex.size) != 1 && !is.null(v)) {
            vertex.size <- vertex.size\[v\]
        }
        vertex.size <- rep(vertex.size, length.out = nrow(coords))

        # Adjust the size for the inverted triangle
        side.length <- sqrt(4 * pi / sqrt(3)) * vertex.size / 2

        vertex.frame.color\[vertex.frame.width <= 0\] <- NA
        vertex.frame.width\[vertex.frame.width <= 0\] <- 1

        for (i in 1:nrow(coords)) {
            x <- coords\[i, 1\]
            y <- coords\[i, 2\]
            size <- side.length\[i\]
            polygon(x + size * c(cos(3*pi/2), cos(5*pi/6), cos(pi/6)),
                    y + size * c(sin(3*pi/2), sin(5*pi/6), sin(pi/6)),
                    col = vertex.color\[i\],
                    border = vertex.frame.color\[i\],
                    lwd = vertex.frame.width\[i\])
        }
    }

    # Add the inverted triangle shape to igraph
    add_shape("itriangle", clip = shapes(shape = "circle")$clip,
              plot = itriangle.plot)

    # Example graph
    g <- make_ring(10)
    V(g)$shape <- rep(c("circle", "itriangle"), length.out = vcount(g))
    plot(g, vertex.size = 15, vertex.color = "skyblue")

## End(Not run)
```

`jaccard.index` *Calculate Jaccard index between two sets represented as numeric vectors.*

Description

The Jaccard index (also known as Jaccard similarity coefficient) is defined as the ratio of the size of the intersection of the two sets to the size of their union. Formally:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The index ranges from 0 (no overlap) to 1 (identical sets).

Usage

```
jaccard.index(vec.1, vec.2)
```

Arguments

`vec.1`	A numeric vector representing the first set. Duplicate values are automatically removed to form a proper set.
`vec.2`	A numeric vector representing the second set. Duplicate values are automatically removed to form a proper set.

Details

The function first converts the input vectors to sets by removing duplicates using `unique()`, then computes the intersection and union using R's built-in set operations. This ensures that the calculation follows proper set theory principles where each element is counted only once.

Value

A numeric value between 0 and 1 representing the Jaccard index, where:

- 0 indicates no overlap between the sets
- 1 indicates the sets are identical
- Values between 0 and 1 indicate partial overlap

Returns NaN if both sets are empty (0/0 case).

Note

- The function treats the inputs as sets, so duplicate values are ignored
- For comparing multisets (where duplicates matter), consider other similarity measures
- The Jaccard distance can be computed as 1 - Jaccard index
- NA values in the input vectors are preserved and treated as distinct elements

References

Jaccard, P. (1912). The distribution of the flora in the alpine zone. New Phytologist, 11(2), 37-50.

See Also

`intersect` for set intersection, `union` for set union, `setdiff` for set difference

Examples

```
# Example 1: Basic usage with duplicates
samples.1 <- c(1, 2, 5, 5)
samples.2 <- c(2, 2, 5, 7)
jaccard.index(samples.1, samples.2)  # Returns 0.5 (intersection: {2,5}, union: {1,2,5,7})

# Example 2: Identical sets
jaccard.index(c(1, 2, 3), c(3, 2, 1))  # Returns 1

# Example 3: No overlap
jaccard.index(c(1, 2, 3), c(4, 5, 6))  # Returns 0

# Example 4: One empty set
jaccard.index(c(1, 2, 3), numeric(0))  # Returns 0
```

```
# Example 5: Comparing sample indices or cluster memberships
cluster1_samples <- c(1, 3, 5, 7, 9)
cluster2_samples <- c(2, 4, 5, 7, 8)
similarity <- jaccard.index(cluster1_samples, cluster2_samples)
print(paste("Cluster overlap:", round(similarity * 100, 1), "%"))
```

---

jensen.shannon.divergence

Calculate Jensen-Shannon Divergence Between Two Probability Mass Functions

---

Description

Computes the Jensen-Shannon divergence (JSD) between two discrete probability mass functions. The JSD is a symmetrized and smoothed version of the Kullback-Leibler divergence. It is defined as: JSD(P||Q) = 1/2 * D(P||M) + 1/2 * D(Q||M), where M = 1/2 * (P + Q) and D denotes the Kullback-Leibler divergence.

Usage

```
jensen.shannon.divergence(p, q)
```

Arguments

p	Numeric vector representing the first probability mass function. Will be normalized to sum to 1. Must not contain negative values.
q	Numeric vector representing the second probability mass function. Will be normalized to sum to 1. Must not contain negative values.

Details

The function implements the following behavior:

- If input vectors have different lengths, the shorter vector is padded with zeros to match the length of the longer vector
- Input vectors are automatically normalized to sum to 1
- Negative probabilities are not allowed
- Zero probabilities are handled appropriately by only computing the divergence over positive probability values

When vectors of different lengths are provided, the function effectively treats the shorter distribution as having zero probability for the additional categories present in the longer distribution. This allows comparison of distributions with different support sizes while preserving the mathematical properties of the Jensen-Shannon divergence.

Value

A numeric value representing the Jensen-Shannon divergence in bits (using log base 2). The value is always non-negative and bounded above by 1.

References

Lin, J. (1991). Divergence measures based on the Shannon entropy. IEEE Transactions on Information Theory, 37(1), 145-151.

Examples

```
# Calculate JSD between two simple distributions
p <- c(0.4, 0.6)
q <- c(0.5, 0.5)
jensen.shannon.divergence(p, q)

# Calculate JSD between distributions of different lengths
p <- c(0.3, 0.7)
q <- c(0.2, 0.3, 0.5)
jensen.shannon.divergence(p, q)  # p will be padded with 0 to match q's length

# Calculate JSD between two categorical distributions
p <- c(0.2, 0.3, 0.5)
q <- c(0.1, 0.4, 0.5)
jensen.shannon.divergence(p, q)
```

`join.graphs` *Join two graphs at specified vertices*

Description

This function joins two graphs by identifying a vertex from each graph. The resulting graph combines the vertices and edges from both input graphs, with the specified vertices being treated as a single vertex in the joined graph.

Usage

```
join.graphs(graph1, graph2, i1, i2)
```

Arguments

`graph1`	A list representing the adjacency list of the first graph.
`graph2`	A list representing the adjacency list of the second graph.
`i1`	An integer specifying the index of the vertex in the first graph to be joined.
`i2`	An integer specifying the index of the vertex in the second graph to be joined.

Value

A list representing the adjacency list of the joined graph.

Examples

```
graph1 <- list(c(2, 3), c(1), c(1))
graph2 <- list(c(2), c(1, 3), c(2))
joined_graph <- join.graphs(graph1, graph2, 2, 1)
print(joined_graph)
```

`kernel.fn` *A generic interface for evaluating a kernel over a numeric vector*

Description

A generic interface for evaluating a kernel over a numeric vector

Usage

```
kernel.fn(x, kernel.str = "normal")
```

Arguments

`x`	A numeric vector.
`kernel.str`	The name (character string) of a kernel that will be evaluated over x.

`kh.matrix` *Create a kh.matrix object*

Description

Constructor function for creating a kh.matrix object that stores kernel and bandwidth information for model selection or cross-validation procedures.

Usage

```
kh.matrix(kh.mat, existing.k, h.values, id)
```

Arguments

`kh.mat`	Numeric matrix containing kernel values or cross-validation results
`existing.k`	Integer vector specifying which k values were evaluated
`h.values`	Numeric vector of bandwidth (h) values tested
`id`	Character string or identifier for this kh.matrix object

Value

An object of class "kh.matrix" containing the input components

`kNN.cltr.imputation` *kNN-Based Cluster Imputation*

Description

This function utilizes the k-Nearest Neighbors (k-NN) algorithm with a modified majority vote strategy to reassign points in the reference cluster (typically cluster 0, representing noise or outliers) to one of the non-reference clusters. The process iterates until no further reduction in the size of the reference cluster is observed.

Usage

```
kNN.cltr.imputation(X, cltr, ref.cltr = 0, K = 20, use.geodesic.dist = FALSE)
```

Arguments

`X`	Matrix representing the feature space in which the clustering is defined. Each row represents an observation and columns represent features.
`cltr`	Vector containing the initial clustering labels for each point in 'X'. Must have the same length as nrow(X).
`ref.cltr`	Cluster label designated as the reference cluster for reassignment (default is 0). Points with this label will be reassigned to other clusters.
`K`	Number of nearest neighbors used for cluster reassignment (default is 20). Will be automatically adjusted to min(K, nrow(X)-1) if K exceeds available points.
`use.geodesic.dist`	Logical flag indicating whether to use geodesic distance instead of Euclidean distance (default is FALSE). Requires geodesic.knn() function if set to TRUE.

Details

The algorithm works as follows:

1. For each point in the reference cluster, find its K nearest neighbors
2. Among these neighbors, find the first neighbor that belongs to a non-reference cluster
3. Reassign the point to that cluster
4. Repeat until no more points can be reassigned

Note: The current implementation uses a "first valid neighbor" strategy rather than a true majority vote. Points are assigned to the cluster of their first non-reference neighbor, which may lead to suboptimal assignments in some cases.

Value

A vector containing the new clustering labels after iterative imputation. The length and order match the input 'cltr' vector.

Note

- If rownames are provided for X and names for cltr, they must match exactly
- The function requires the FNN package for Euclidean k-NN search
- If use.geodesic.dist = TRUE, the geodesic.knn() function must be available
- The algorithm may not converge if the reference cluster contains isolated points with no non-reference neighbors within K nearest neighbors

See Also

`get.knn` for the k-NN implementation used

Examples

```
## Not run:
# Generate sample data with 3 clusters and some noise
set.seed(123)
X <- rbind(
  matrix(rnorm(100, mean = 0), ncol = 2),
  matrix(rnorm(100, mean = 5), ncol = 2),
  matrix(rnorm(100, mean = c(5, 0)), ncol = 2),
  matrix(runif(20, -2, 7), ncol = 2)  # noise points
)

# Initial clustering with some points labeled as 0 (noise)
cltr <- c(rep(1, 50), rep(2, 50), rep(3, 50), rep(0, 20))
names(cltr) <- rownames(X) <- paste0("point_", 1:nrow(X))

# Apply kNN imputation
new_cltr <- kNN.cltr.imputation(X, cltr, ref.cltr = 0, K = 10)

# Check results
table(Original = cltr, Imputed = new_cltr)

## End(Not run)
```

`kNN.cltr.imputation.enhanced`

Enhanced kNN-Based Cluster Imputation with Majority Vote

Description

An improved version of kNN cluster imputation that uses true majority voting and provides more options for handling ties and isolated points.

Usage

```
kNN.cltr.imputation.enhanced(
  X,
  cltr,
  ref.cltr = 0,
  K = 20,
  method = "majority",
  tie.break = "first",
  min.votes = 1,
  use.geodesic.dist = FALSE,
  verbose = FALSE
)
```

Arguments

`X`	Matrix representing the feature space (observations in rows, features in columns)
`cltr`	Vector of initial cluster labels
`ref.cltr`	Reference cluster label to reassign (default is 0)
`K`	Number of nearest neighbors (default is 20)
`method`	Voting method: "majority" (default), "weighted", or "first"
`tie.break`	How to handle ties: "random", "first", or "none" (keep in ref cluster)
`min.votes`	Minimum number of votes needed for reassignment (default is 1)
`use.geodesic.dist`	Use geodesic distance if TRUE (default is FALSE)
`verbose`	Print progress information (default is FALSE)

Value

Vector of cluster labels after imputation

`knn.weighted.mean` *Calculates K-Nearest Neighbor Weighted Mean*

Description

This function calculates the weighted mean of the response variable `y` for each observation in `X` based on its k-nearest neighbors. The weights are determined by applying a kernel function to the distances between each observation and its neighbors.

Usage

```
knn.weighted.mean(X, y, k, kernel = "epanechnikov", nn.factor = 1.01)
```

Arguments

`X`	A matrix or data frame containing the predictor variables.
`y`	A vector containing the response variable.
`k`	An integer specifying the number of nearest neighbors to consider.
`kernel`	A character string specifying the kernel function to use for weighting the neighbors. Default is "epanechnikov". Other options include "uniform", "triangular", "biweight", "triweight", "cosine", "gaussian", and "rank".
`nn.factor`	A numeric value greater than 1 used to adjust the kernel bandwidth. Default is 1.01.

Details

The function first finds the k-nearest neighbors for each observation in X using the get.knn() function. It then calculates the distances between each observation and its neighbors and applies the specified kernel function to the distances to obtain weights. The weights are normalized to sum to 1 for each observation. Finally, the weighted mean of the response variable is calculated for each observation using its neighbors' weights.

Value

A vector of weighted means of the response variable for each observation in X.

Examples

```
X <- matrix(rnorm(100), ncol = 2)
y <- rnorm(50)
weighted_means <- knn.weighted.mean(X, y, k = 5)
```

---

kullback.leibler.divergence

Calculate Kullback-Leibler Divergence Between Two Probability Mass Functions

---

Description

Computes the Kullback-Leibler divergence (KL divergence) between two discrete probability mass functions. The KL divergence is a measure of the information lost when Q is used to approximate P. Note that KL divergence is not symmetric: $\text{KL}(P||Q) \neq \text{KL}(Q||P)$ in general.

Usage

```
kullback.leibler.divergence(p, q, zero.handling = "pseudo", epsilon = 1e-10)
```

Arguments

p	Numeric vector representing the first probability mass function P. Will be normalized to sum to 1. Must not contain negative values.
q	Numeric vector representing the second probability mass function Q. Will be normalized to sum to 1. Must not contain negative values.
zero.handling	Character string specifying how to handle zeros in Q. Must be one of "strict" (error if Q[i] = 0 where P[i] > 0), "exclude" (compute only where Q > 0), or "pseudo" (add small constant to Q). Defaults to "pseudo".
epsilon	Numeric value specifying the small constant to add when zero.handling = "pseudo". Defaults to 1e-10.

Details

The function implements the following behavior:

- If input vectors have different lengths, the shorter vector is padded with zeros to match the length of the longer vector
- Input vectors are automatically normalized to sum to 1
- Negative probabilities are not allowed

The implementation provides three ways to handle zeros in the `Q` distribution:

- `"strict"`: Throws an error if `Q[i] = 0` where `P[i] > 0`
- `"exclude"`: Computes divergence only over indices where `Q > 0`
- `"pseudo"`: Adds small constant `epsilon` to `Q` (and renormalizes)

When vectors of different lengths are provided, the function treats the shorter distribution as having zero probability for the additional categories present in the longer distribution. This is particularly important when using "strict" zero handling, as padding with zeros could trigger errors if P is the shorter vector.

Value

A numeric value representing the Kullback-Leibler divergence in bits (using log base 2). The value is always non-negative, and equals infinity if there exists an `i` where `P[i] > 0` and `Q[i] = 0` (when `zero.handling = "strict"`).

References

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. The Annals of Mathematical Statistics, 22(1), 79-86.

Examples

```
# Calculate KL divergence between two simple distributions
p <- c(0.4, 0.6)
q <- c(0.5, 0.5)
kullback.leibler.divergence(p, q)

# Using different zero handling methods
p <- c(0.2, 0.8, 0)
q <- c(0.3, 0.6, 0.1)
kullback.leibler.divergence(p, q, zero.handling = "exclude")
kullback.leibler.divergence(p, q, zero.handling = "pseudo")

# Calculate KL divergence between distributions of different lengths
p <- c(0.3, 0.7)
q <- c(0.2, 0.3, 0.5)
kullback.leibler.divergence(p, q)  # p will be padded with 0 to match q's length
```

lcor.1D *1D local weighed Pearson correlation model*

Description

1D local weighed Pearson correlation model

Usage

```
lcor.1D(
  x,
  y1,
  y2,
  grid.size = 400,
  f = NULL,
  bw = NULL,
  smooth = TRUE,
  n.BB = 0,
  get.predictions.CrI = TRUE,
  level = 0.95,
  n.C.itr = 100,
  C = 6,
  stop.C.itr.on.min = TRUE,
  n.cv.folds = 10,
  n.cv.reps = 5,
  min.K = 5,
  nn.kernel = "epanechnikov",
  y1.binary = FALSE,
  cv.nNN = 3,
  verbose = FALSE
)
```

Arguments

`x`	A numeric vector of a predictor variable.
`y1`	A numeric vector of the first outcome variable.
`y2`	A numeric vector of the second outcome variable.
`grid.size`	A number of grid points; was grid.size = 10*length(x), but the results don't seem to be different from 400 which is much faster.
`f`	The proportion of the range of x that is used within a moving window to train the model. If NULL, the optimal value of f will be found using minimum median absolute error optimization algorithm.
`bw`	A bandwidth parameter.
`smooth`	Set to TRUE (default) to smooth the estimates of local weighted correlations.
`n.BB`	The number of Bayesian bootstrap (BB) iterations for estimates of CI's of beta's.
`get.predictions.CrI`	A logical parameter. If TRUE, BB with quantile determined by the value of 'level' will be used to determine the upper and lower limits of CI's.

`level`	A confidence level.
`n.C.itr`	The number of Cleveland's absolute residue based reweighting iterations for a robust estimates of mean y values.
`C`	A scaling of |res| parameter changing |res| to |res|/C before applying ae.kernel to |res|'s.
`stop.C.itr.on.min`	A logical variable, if TRUE, the Cleveland's iterative reweighting stops when the maximum of the absolute values of differences of the old and new predictions estimates are reach a local minimum.
`n.cv.folds`	The number of cross-validation folds. Used only when f = NULL. Default value: 10.
`n.cv.reps`	The number of repetiions of cross-validation. Used only when f = NULL. Default value: 5.
`min.K`	The minimal number of x NN's that must be present in each window.
`nn.kernel`	A kernel.
`y1.binary`	Set to TRUE if y1 is a binary variable.
`cv.nNN`	The number of nearest neighbors in interpolate_gpredictions() used to find predictions given gpredictions in the cv_deg0_ routines.
`verbose`	Prints info about what is being done.

Value

A list of input parameters as well as coefficients and residues of all linear models

---

`left.asymmetric.bump.fn`

Creates Left Asymmetric Bump Function

Description

This function creates an asymmetric bump function, which is a modification of the standard bump function with asymmetric behavior to the left of the offset. It is smooth with compact support, primarily used in mathematical analysis and applications requiring non-symmetric smooth functions.

Usage

```
left.asymmetric.bump.fn(x, offset = 0, h = 1, q = 1)
```

Arguments

`x`	A numeric vector or a single numeric value where the bump function is evaluated.
`offset`	The center (offset) of the bump function (defaults to 0). The function behaves differently to the left and right of this point.
`h`	The radius of the support of the bump function (defaults to 1). The function is zero outside the interval $[offset - h, offset + h]$.
`q`	The power to which the Gaussian function to the right of the offset is raised (defaults to 1).

Details

The asymmetric bump function is defined differently to the left and right of the offset. For `x < offset` and `offset - x < h`, it is defined as exp(-1 / h - (x - offset)^2). For `|x - offset| < h`, it follows the standard bump function exp(-1 / (h - (x - offset)^2)). It is zero outside the specified interval, maintaining smooth transitions and compact support.

Value

A numeric vector or a single numeric value representing the value(s) of the asymmetric bump function at the input `x`. The function smoothly approaches zero at the boundaries of its support and has different behavior to the left of the offset.

Examples

```
## Not run:
x <- seq(-2, 2, length.out = 100)
plot(x, AsymmetricBumpFunction(x), type = "l")

## End(Not run)
```

`llm.1D.beta` *Eestimates the coefficients of linear models in the vicinity of each grid point of radius nn.r.*

Description

The max.K parameter is used to avoid looping over indices where weights are 0.

Usage

```
llm.1D.beta(nn.x, nn.y, nn.w, max.K, degree)
```

Arguments

`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`nn.w`	A matrix of weights over K nearest neighbors of each element of the grid.
`max.K`	A vector of indices indicating the range where weights are not 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.

Value

list with components: beta, gpredictions.

NOTE: Weights must sum up to 1!

`llm.1D.beta.perms` *Estimates the coefficients of linear models in the vicinity of each grid point of radius nn.r on the permuted version*

Description

The max.K parameter is used to avoid looping over indices where weights are 0.

Usage

```
llm.1D.beta.perms(nn.x, nn.i, y, nn.w, max.K, degree, n.perms)
```

Arguments

`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.i`	A matrix of indices of the K nearest neighbors of each element of the grid.
`y`	A vector of response values.
`nn.w`	A matrix of weights over K nearest neighbors of each element of the grid.
`max.K`	A vector of indices indicating the range where weights are not 0.
`degree`	A degree of the polynomial of x in the linear regression. The only allowed values are 1 and 2.
`n.perms`	Number of permutations to perform. Must be a positive integer.

Value

A numeric vector of beta coefficients from all permutations.

NOTE: Weights must sum up to 1!

`llm.1D.fit.and.predict` *Fits 1D rllm model and generates predictions esimates.*

Description

Fits 1D rllm model and generates predictions esimates.

Usage

```
llm.1D.fit.and.predict(nn.i, nn.w, nn.x, nn.y, y.binary, max.K, degree, nx)
```

Arguments

`nn.i`	A matrix of the indices of K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.w`	A matrix of weights.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`y.binary`	Set to TRUE if y's values are within the interval `[0,1]`.
`max.K`	An array of indices indicating the range where weights are not 0. Indices < max.K at i have weights > 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.
`nx`	The number of elements of x.

Value

list with components: beta, predictions.

NOTE: Weights must sum up to 1!

`llm.1D.fit.and.predict.BB.CrI`

Creates BB CI's of a 1D rllm model's predictions esimates.

Description

Creates BB CI's of a 1D rllm model's predictions esimates.

Usage

```
llm.1D.fit.and.predict.BB.CrI(
  nn.i,
  nn.w,
  nn.x,
  nn.y,
  y.binary,
  max.K,
  degree,
  nx,
  predictions,
  n.BB
)
```

Arguments

`nn.i`	A matrix of the indices of K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.w`	A matrix of weights.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid.

`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`y.binary`	Set to TRUE if y values are in the interval `[0,1]`.
`max.K`	An array of indices indicating the range where weights are not 0. Indices < max.K at i have weights > 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.
`nx`	The number of elements of x.
`predictions`	An predictions estimates.
`n.BB`	The number of BB iterations.

Value

predictions.CrI of length nx.

`llm.1D.fit.and.predict.global.BB`
Creates Bayesian bootstrap (BB) estimates of predictions using global reweighting of the elements of x.

Description

Creates Bayesian bootstrap (BB) estimates of predictions using global reweighting of the elements of x.

Usage

```
llm.1D.fit.and.predict.global.BB(
  nn.i,
  nn.w,
  nn.x,
  nn.y,
  max.K,
  degree,
  nx,
  n.BB
)
```

Arguments

`nn.i`	A matrix of the indices of K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.w`	A matrix of weights.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`max.K`	An array of indices indicating the range where weights are not 0. Indices < max.K at i have weights > 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.
`nx`	The number of elements of x.
`n.BB`	The number of BB iterations.

Value

BB predictions matrix of dim nx-by-n.BB.

`llm.1D.fit.and.predict.global.BB.CrI`
Creates BB CI's of a 1D rllm model's predictions esimates using global reweighting of the elements of x.

Description

Creates BB CI's of a 1D rllm model's predictions esimates using global reweighting of the elements of x.

Usage

```
llm.1D.fit.and.predict.global.BB.CrI(
  nn.i,
  nn.w,
  nn.x,
  nn.y,
  max.K,
  degree,
  nx,
  predictions,
  y.binary,
  n.BB
)
```

Arguments

`nn.i`	A matrix of the indices of K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.w`	A matrix of weights.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`max.K`	An array of indices indicating the range where weights are not 0. Indices < max.K at i have weights > 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.
`nx`	The number of elements of x.
`predictions`	An predictions estimates.
`y.binary`	Set to TRUE if y is a binary variable.
`n.BB`	The number of BB iterations.

Value

predictions.CrI of length nx.

llm.1D.fit.and.predict.global.BB.qCrI

Creates BB quantile-based estimates of predictions CI's.

Description

Creates BB quantile-based estimates of predictions CI's.

Usage

```
llm.1D.fit.and.predict.global.BB.qCrI(
  y.binary,
  nn.i,
  nn.w,
  nn.x,
  nn.y,
  max.K,
  degree,
  nx,
  n.BB,
  alpha = 0.05
)
```

Arguments

`y.binary`	A logical variable. If TRUE, predicted predictions will be trimmed to the closed interval `[0, 1]`.
`nn.i`	A matrix of the indices of K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.w`	A matrix of weights.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid.
`nn.y`	A matrix of y values over K nearest neighbors of each element of the grid.
`max.K`	An array of indices indicating the range where weights are not 0. Indices < max.K at i have weights > 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.
`nx`	The number of elements of x.
`n.BB`	The number of BB iterations.
`alpha`	The confidence level.

Value

predictions.qCI matrix with two rows (upper and lower CI) and nx columns.

`llm.predict.1D` *Compute predictions from local linear model coefficients*

Description

Compute predictions from local linear model coefficients

Usage

```
llm.predict.1D(beta, nn.i, nn.d, nn.x, nx, max.K, nn.kernel = "epanechnikov")
```

Arguments

`beta`	Coefficients of the local linear model.
`nn.i`	A matrix of indices of x NN's of xgrid.
`nn.d`	A matrix of NN distances.
`nn.x`	The values of predictor variable over NN's of xgrid.
`nx`	The length of x.
`max.K`	A vector of indices indicating the range where weights are not 0.
`nn.kernel`	A kernel used for generating weights.

Value

A numeric vector of predictions

`lmax.basins` *Identify Basins of Attraction of Local Maxima on a Weighted Graph*

Description

Computes the basins of attraction for local maxima on a weighted graph using a gradient flow algorithm. Each basin consists of vertices that would flow towards a particular local maximum following the steepest ascent path.

Usage

```
lmax.basins(adj.list, weight.list = NULL, y, lmax.list, verbose = FALSE)
```

Arguments

`adj.list`	List where `adj.list[[i]]` contains indices of vertices adjacent to vertex i. Must be a valid adjacency list representation.
`weight.list`	List where `weight.list[[i]]` contains positive weights of edges from vertex i to corresponding vertices in `adj.list[[i]]`. If `NULL`, uniform weights of 1 are used.
`y`	Numeric vector of values at each vertex. Must have the same length as `adj.list`. Can contain `NA` values which are treated as negative infinity.

`lmax.list`	List where each element contains: **`lmax`:** Index of the local maximum vertex; **`vertices`:** Set of vertices forming the initial basin (usually the local maximum and its neighborhood); **`label`:** Character label for the basin
`verbose`	Logical; if `TRUE`, prints progress messages during computation. Default is `FALSE`.

Details

The algorithm expands basins iteratively from local maxima by examining boundary vertices and adding neighboring vertices that satisfy a threshold criterion. The threshold is computed as a weighted average of values at vertices already in the basin.

Edge weights influence both the threshold calculation and the decision to include new vertices. Higher weights indicate stronger connections and give more influence in the weighted averaging process.

The algorithm terminates when no new vertices can be added to any basin, ensuring convergence in finite graphs.

Value

A named list of basins of attraction, where names correspond to labels in `lmax.list`. Each element is an integer vector of vertex indices belonging to that basin, sorted in ascending order.

See Also

`set.boundary`, `lmin.basins`, `compute.graph.gradient.flow`

Examples

```
## Not run:
# Create a weighted graph
adj.list <- list(
  c(2, 3),      # vertex 1 neighbors
  c(1, 3, 4),   # vertex 2 neighbors
  c(1, 2, 5),   # vertex 3 neighbors
  c(2, 5),      # vertex 4 neighbors
  c(3, 4)       # vertex 5 neighbors
)

# Edge weights (optional)
weight.list <- list(
  c(1.0, 0.5),     # weights for edges from vertex 1
  c(1.0, 0.8, 1.2), # weights for edges from vertex 2
  c(0.5, 0.8, 1.0), # weights for edges from vertex 3
  c(1.2, 1.5),      # weights for edges from vertex 4
  c(1.0, 1.5)       # weights for edges from vertex 5
)

# Values at vertices
y <- c(0.5, 0.8, 0.3, 1.2, 1.0)

# Define local maxima
lmax.list <- list(
  list(lmax = 4, vertices = c(4), label = "max1"),
  list(lmax = 5, vertices = c(5), label = "max2")
)

# Compute basins
basins <- lmax.basins(adj.list, weight.list, y, lmax.list, verbose = TRUE)

## End(Not run)
```

`lmin.basins` *Identify Basins of Attraction of Local Minima on a Weighted Graph*

Description

Computes the basins of attraction for local minima on a weighted graph using a gradient flow algorithm. Each basin consists of vertices that would flow towards a particular local minimum following the steepest descent path.

Usage

```
lmin.basins(adj.list, weight.list = NULL, y, lmin.list, verbose = FALSE)
```

Arguments

`adj.list`	List where `adj.list[[i]]` contains indices of vertices adjacent to vertex i. Must be a valid adjacency list representation.
`weight.list`	List where `weight.list[[i]]` contains positive weights of edges from vertex i to corresponding vertices in `adj.list[[i]]`. If `NULL`, uniform weights of 1 are used.
`y`	Numeric vector of values at each vertex. Must have the same length as `adj.list`. Can contain `NA` values which are treated as positive infinity.
`lmin.list`	List where each element contains: `lmin`**:** Index of the local minimum vertex `vertices`**:** Set of vertices forming the initial basin (usually the local minimum and its neighborhood) `label`**:** Character label for the basin
`verbose`	Logical; if `TRUE`, prints progress messages during computation. Default is `FALSE`.

Details

The algorithm expands basins iteratively from local minima by examining boundary vertices and adding neighboring vertices that satisfy a threshold criterion. The threshold is computed as a weighted average of values at vertices already in the basin.

Edge weights influence both the threshold calculation and the decision to include new vertices. Higher weights indicate stronger connections and give more influence in the weighted averaging process.

For local minima, vertices are added if their values are above the threshold, representing flow towards lower values.

Value

A named list of basins of attraction, where names correspond to labels in `lmin.list`. Each element is an integer vector of vertex indices belonging to that basin, sorted in ascending order.

See Also

`set.boundary`, `lmax.basins`, `compute.graph.gradient.flow`

Examples

```
## Not run:
# Create a weighted graph
adj.list <- list(
  c(2, 3),      # vertex 1 neighbors
  c(1, 3, 4),   # vertex 2 neighbors
  c(1, 2, 5),   # vertex 3 neighbors
  c(2, 5),      # vertex 4 neighbors
  c(3, 4)       # vertex 5 neighbors
)

# Values at vertices
y <- c(0.5, 0.8, 0.3, 1.2, 0.2)

# Define local minima
lmin.list <- list(
  list(lmin = 5, vertices = c(5), label = "min1"),
  list(lmin = 3, vertices = c(3), label = "min2")
)

# Compute basins
basins <- lmin.basins(adj.list, weight.list = NULL, y, lmin.list)

## End(Not run)
```

`load.graph.data` *Load Graph Data from RDA Files*

Description

Loads pruned graph data from a series of RDA files for different k values and combines them into a list structure containing adjacency and distance lists.

Usage

```
load.graph.data(
  k.values,
  prefix,
  suffix = "_NN_graph.rda",
  graph.object.name = "S.graph",
  verbose = TRUE
)
```

Arguments

`k.values`	Numeric vector of k values to process. Must be positive integers.
`prefix`	Character string specifying the path and prefix of the RDA files.
`suffix`	Character string specifying the suffix of the RDA files (default: "_NN_graph.rda").
`graph.object.name`	Character string specifying the name of the graph object stored in the RDA files (default: "S.graph").
`verbose`	Logical indicating whether to show progress messages (default: TRUE).

Details

The function expects RDA files to be named according to the pattern: `paste0(prefix, k, suffix)` for each k value.

Each RDA file should contain an object with the name specified in `graph.object.name`, which must have components `pruned_adj_list` and `pruned_dist_list`.

Value

A list containing two components:

`adj.list`	Named list of adjacency lists, one for each k value
`dist.list`	Named list of distance lists, one for each k value

Examples

```
## Not run:
# Load graphs for k = 5, 10, 15
k.values <- c(5, 10, 15)
graph.data <- load.graph.data(
  k.values,
  prefix = "path/to/graphs/graph_k",
  suffix = "_NN_graph.rda",
  graph.object.name = "S.graph"
)

## End(Not run)
```

`loc.const.vertices` *Identifies vertices of a graph at which a numeric function is locally constant*

Description

This function takes an adjacency list representing a graph, a numeric vector of function values at each vertex, and a precision threshold. It identifies the vertices at which the function is considered locally constant, meaning the absolute difference between the function value at the vertex and its neighbors is within the specified precision threshold.

Usage

```
loc.const.vertices(adj.list, y, prec = 1e-08)
```

Arguments

`adj.list`	A list representing the graph adjacency list. Each element of the list should be an integer vector specifying the neighboring vertex indices for a given vertex. The vertex indices should be 1-based.
`y`	A numeric vector representing the function values at each vertex of the graph. The length of `y` should be equal to the number of vertices in the graph.
`prec`	A numeric scalar specifying the precision threshold used to determine local constancy. If the absolute difference between the value at a vertex and any of its neighbors is greater than this threshold, the vertex is not considered locally constant.

Value

An integer vector containing the indices of the vertices at which the function is locally constant. The vertex indices are 1-based.

`logLik.graph.spectral.lowess`
Extract Log-Likelihood from Graph Spectral LOWESS

Description

Computes the log-likelihood assuming normal errors

Usage

```
## S3 method for class 'graph.spectral.lowess'
logLik(object, ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`...`	Additional arguments (currently unused)

Value

Log-likelihood value

`logLik.graph.spectral.ma.lowess`
Extract Log-Likelihood from Graph Spectral MA LOWESS

Description

Computes the log-likelihood for model-averaged predictions assuming normal errors

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
logLik(object, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

Log-likelihood value

`logLik.mabilog` *Extract Log-Likelihood from Mabilog Model*

Description

Computes the log-likelihood for the fitted model

Usage

```
## S3 method for class 'mabilog'
logLik(object, ...)
```

Arguments

`object`	A 'mabilog' object
`...`	Additional arguments (currently unused)

Value

Log-likelihood value with attributes

`logLik.mabilo_plus` *Extract Log-Likelihood from Mabilo Plus Model*

Description

Computes an approximate log-likelihood for model comparison

Usage

```
## S3 method for class 'mabilo_plus'
logLik(object, type = c("ma", "sm"), ...)
```

Arguments

`object`	A 'mabilo_plus' object
`type`	Character string specifying which predictions to use: "ma" (default) or "sm"
`...`	Additional arguments (currently unused)

Value

Log-likelihood value

`loo.llm.1D` *Leave-One-Out (LOO) cross-validation of 1D local linear models*

Description

Leave-One-Out (LOO) cross-validation of 1D local linear models

Usage

```
loo.llm.1D(
  x,
  y,
  grid.size = 400,
  degree = 2,
  f = 0.2,
  bw = NULL,
  min.K = 5,
  nn.kernel = "epanechnikov"
)
```

Arguments

`x`	A numeric vector of a predictor variable.
`y`	A numeric vector of an outcome variable.
`grid.size`	A number of grid points; was grid.size = 10*length(x), but the results don't seem to be different from 400 which is much faster.
`degree`	A degree of the polynomial of x in the linear regresion; 0 means weighted mean, 1 is regular linear model lm(y ~ x), and deg = d is lm(y ~ poly(x, d)). The only allowed values are 1 and 2.
`f`	The proportion of the range of x that is used within moving window to train the model.
`bw`	A bandwidth parameter.
`min.K`	The minimal number of x NN's that must be present in each window.
`nn.kernel`	A kernel.

Value

list of parameters and residues of all linear models

`mabilo` *Model-Averaged Locally Weighted Scatterplot Smoothing (MABILO)*

Description

Implements MABILO algorithm for robust local regression, extending LOWESS by incorporating model averaging and Bayesian bootstrap for uncertainty quantification. The algorithm uses symmetric k-hop neighborhoods and kernel-weighted averaging for predictions.

Usage

```
mabilo(
  x,
  y,
  y.true = NULL,
  k.min = max(3, as.integer(0.05 * length(x))),
  k.max = NULL,
  n.bb = 100,
  p = 0.95,
  kernel.type = 7L,
  dist.normalization.factor = 1.1,
  epsilon = 1e-10,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of x coordinates.
`y`	Numeric vector of y coordinates (response values).
`y.true`	Optional numeric vector of true y values for error calculation.
`k.min`	Minimum number of neighbors on each side (positive integer).
`k.max`	Maximum number of neighbors on each side. If NULL, defaults to min((n-2)/4, max(3*k.min, 10)).
`n.bb`	Number of Bayesian bootstrap iterations (non-negative integer). Set to 0 to skip bootstrap.
`p`	Probability level for credible intervals (between 0 and 1).
`kernel.type`	Integer; kernel type for weight calculation: • 1: Epanechnikov • 2: Triangular • 3: Truncated exponential • 4: Laplace • 5: Normal • 6: Biweight • 7: Tricube (default) • 8: cosine • 9: hyperbolic • 10: constant. Default is 7.
`dist.normalization.factor`	Positive number for distance normalization (default: 1.1).
`epsilon`	Small positive number for numerical stability (default: 1e-10).
`verbose`	Logical; if TRUE, prints progress information.

Details

The function automatically sorts input data by x values. For each point, it uses k-hop neighborhoods (k points on each side when available) rather than k-nearest neighbors, providing more symmetric neighborhoods. The optimal k is selected by minimizing mean LOOCV errors.

The Bayesian bootstrap analysis (when n.bb > 0) provides uncertainty quantification through credible intervals computed at the specified probability level p. Note that setting n.bb > 0 increases computation time proportionally, as the algorithm must be run n.bb times with different weight configurations.

Bayesian Bootstrap:

The Bayesian bootstrap, introduced by Rubin (1981), is a variant of the classical bootstrap that generates smooth posterior distributions. Instead of resampling with replacement (which gives discrete weights n_i/n where n_i is the number of times observation i is selected), the Bayesian bootstrap assigns continuous weights to each observation.

Specifically, for each bootstrap iteration:

1. Generate weights (w_1, ..., w_n) from a Dirichlet(1, 1, ..., 1) distribution
2. These weights sum to 1 and provide a smooth reweighting of the data
3. Compute the MABILO estimate using these weights

This approach has several advantages:

- Provides smooth posterior distributions rather than discrete ones
- Every observation contributes to each bootstrap sample (with varying weights)
- Naturally incorporates uncertainty in a Bayesian framework
- Often produces less variable estimates than classical bootstrap

The credible intervals computed from Bayesian bootstrap samples can be interpreted as Bayesian posterior intervals under a noninformative prior. See Rubin (1981) "The Bayesian Bootstrap" and Lo (1987) "A Large Sample Study of the Bayesian Bootstrap" for theoretical details.

Value

A list of class "mabilo" containing:

- k_values - Vector of tested k values
- opt_k - Optimal k value for model averaging
- opt_k_idx - Index of optimal k value
- k_mean_errors - Mean LOOCV errors for each k
- k_mean_true_errors - Mean true errors if y.true provided
- ma_predictions - Model-averaged predictions using optimal k
- k_predictions - Model-averaged predictions for all k values
- bb_predictions - Central location of bootstrap estimates (if n.bb > 0)
- cri_L - Lower bounds of credible intervals (if n.bb > 0)
- cri_U - Upper bounds of credible intervals (if n.bb > 0)
- x_sorted - Input x values sorted in ascending order
- y_sorted - y values sorted corresponding to x_sorted
- y_true_sorted - y.true values sorted corresponding to x_sorted
- k_min - Minimum neighborhood size used
- k_max - Maximum neighborhood size used

References

Rubin, D.B. (1981). The Bayesian Bootstrap. The Annals of Statistics, 9(1), 130-134.

Lo, A.Y. (1987). A Large Sample Study of the Bayesian Bootstrap. The Annals of Statistics, 15(1), 360-375.

Newton, M.A. & Raftery, A.E. (1994). Approximate Bayesian Inference with the Weighted Likelihood Bootstrap. Journal of the Royal Statistical Society, Series B, 56(1), 3-48.

Examples

```
# Basic usage
x <- seq(0, 10, length.out = 100)
y <- sin(x) + rnorm(100, 0, 0.1)
fit <- mabilo(x, y, k.min = 3, k.max = 10)
plot(x, y)
lines(x, fit$predictions, col = "red")

# With Bayesian bootstrap
fit_bb <- mabilo(x, y, k.min = 3, k.max = 10, n.bb = 100, p = 0.95)
lines(x, fit_bb$cri_L, col = "blue", lty = 2)
lines(x, fit_bb$cri_U, col = "blue", lty = 2)
```

`mabilo.plus`	*Model-Averaged Locally Weighted Scatterplot Smoothing Plus (MA-BILO Plus)*

Description

Performs smoothing using MABILO Plus, which extends the original MABILO algorithm by incorporating flexible model averaging strategies and error filtering approaches without bootstrap resampling.

Usage

```
mabilo.plus(
  x,
  y,
  y.true = NULL,
  k.min = max(3, as.integer(0.05 * length(x))),
  k.max = as.integer((length(x) - 2)/4),
  model.averaging.strategy = "kernel.and.error.weights",
  error.filtering.strategy = "combined.percentile",
  distance.kernel = 7L,
  model.kernel = 7L,
  dist.normalization.factor = 1,
  epsilon = 1e-10,
  verbose = TRUE
)
```

Arguments

`x`	Numeric vector of x coordinates.
`y`	Numeric vector of y coordinates (response values).
`y.true`	Optional numeric vector of true y values for error calculation.
`k.min`	Minimum number of neighbors (positive integer). Default is 5% of data points or 3, whichever is larger.
`k.max`	Maximum number of neighbors (positive integer > k.min). Default corresponds to 2k+1 = n/2.
`model.averaging.strategy`	Character string specifying the model averaging strategy. Must be one of: "kernel.weights.only", "error.weights.only", "kernel.and.error.weights". Default is "kernel.and.error.weights".
`error.filtering.strategy`	Character string specifying the error filtering strategy. Must be one of: "global.percentile", "local.percentile", "combined.percentile", "best.k.models". Default is "combined.percentile".
`distance.kernel`	Integer specifying the kernel type for distance weighting (1-10). Default is 7L.
`model.kernel`	Integer specifying the kernel type for model weighting (1-10). Default is 7L.
`dist.normalization.factor`	Positive numeric value for distance normalization. Default is 1.0.

`epsilon`	Small positive number to prevent division by zero. Default is 1e-10.
`verbose`	Logical indicating whether to print progress information. Default is TRUE.

Value

A list of class "mabilo_plus" containing:

`k_values`	Vector of k values tested
`opt_sm_k`	Optimal k value for simple mean predictions
`opt_ma_k`	Optimal k value for model averaged predictions
`opt_sm_k_idx`	Index of optimal k for simple mean
`opt_ma_k_idx`	Index of optimal k for model averaging
`k_mean_sm_errors`	Mean errors for each k value (simple mean)
`k_mean_ma_errors`	Mean errors for each k value (model averaged)
`k_mean_true_errors`	Mean true errors for each k (if y.true provided)
`sm_predictions`	Simple mean predictions
`ma_predictions`	Model averaged predictions
`sm_errors`	Leave-one-out CV errors (simple mean)
`ma_errors`	Leave-one-out CV errors (model averaged)
`k_predictions`	Matrix of predictions for each k value
`x_sorted`	Sorted x values
`y_sorted`	Sorted y values
`y_true_sorted`	Sorted true y values (if provided)
`k_min`	Minimum k value used
`k_max`	Maximum k value used

Examples

```
# Basic usage
x <- seq(0, 10, length.out = 100)
y <- sin(x) + rnorm(100, 0, 0.1)
fit <- mabilo.plus(x, y)

# With custom parameters
fit2 <- mabilo.plus(x, y, k.min = 5, k.max = 20,
                    model.averaging.strategy = "kernel.weights.only")

# Plot the results
plot(x, y)
lines(fit$x_sorted, fit$ma_predictions, col = "red", lwd = 2)
```

mabilog	*Model-Averaged Binary Locally-Weighted Logistic Smoothing (MABILOG)*

Description

Implements MABILOG algorithm for robust local logistic regression on binary data, extending the MABILO framework by incorporating model averaging with local logistic regression and Bayesian bootstrap for uncertainty quantification. The algorithm uses symmetric k-hop neighborhoods and kernel-weighted averaging for predictions.

Usage

```
mabilog(
  x,
  y,
  y.true = NULL,
  max.iterations = 100,
  ridge.lambda = 0.002,
  max.beta = 100,
  tolerance = 1e-08,
  k.min = max(3, as.integer(0.05 * length(x))),
  k.max = as.integer((length(x) - 1)/2) - 1,
  n.bb = 100,
  p = 0.95,
  distance.kernel = 1,
  dist.normalization.factor = 1.1,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of x coordinates (predictors).
`y`	Numeric vector of binary response values (0 or 1).
`y.true`	Optional numeric vector of true probabilities for error calculation.
`max.iterations`	Maximum number of iterations for logistic regression convergence (default: 100).
`ridge.lambda`	Ridge regression penalty parameter for stabilization (default: 0.002).
`max.beta`	Maximum allowed absolute value for regression coefficients (default: 100.0).
`tolerance`	Convergence tolerance for logistic regression (default: 1e-8).
`k.min`	Minimum number of neighbors on each side (positive integer). Default is 5 percent of data points or 3, whichever is larger.
`k.max`	Maximum number of neighbors on each side. Default is `(n-1)/2 - 1` where n is the number of data points.
`n.bb`	Number of Bayesian bootstrap iterations (non-negative integer). Set to 0 to skip bootstrap (default: 100).
`p`	Probability level for credible intervals (between 0 and 1, default: 0.95).

`distance.kernel`	Integer specifying kernel type for distance weighting (1-10). Common choices: 1 = Uniform, 2 = Triangular, 3 = Epanechnikov, 4 = Quartic, 5 = Tricube, 6 = Gaussian, 7 = Cosine (default: 1).
`dist.normalization.factor`	Positive factor for distance normalization - must be greater than 1 (default: 1.1).
`verbose`	Logical; if TRUE, prints progress information (default: FALSE).

Details

The MABILOG algorithm extends MABILO to binary classification problems by:

- Using local weighted logistic regression instead of linear regression
- Incorporating ridge regularization for numerical stability
- Implementing coefficient constraints to prevent extreme predictions
- Providing probability estimates rather than continuous predictions

The algorithm automatically sorts input data by x values. For each point, it uses k-hop neighborhoods (k points on each side when available) rather than k-nearest neighbors, providing more symmetric neighborhoods. The optimal k is selected by minimizing mean leave-one-out cross-validation error.

The Bayesian bootstrap analysis (when n.bb > 0) provides uncertainty quantification through credible intervals computed at the specified probability level p.

Value

A list of class "mabilog" containing:

`k_values`	Vector of tested k values
`opt_k`	Optimal k value selected by minimizing LOOCV error
`opt_k_idx`	Index of optimal k in k_values vector
`k_mean_errors`	Mean LOOCV errors for each k value
`k_mean_true_errors`	Mean true errors for each k (if y.true provided)
`predictions`	Model-averaged probability predictions using optimal k
`errors`	Leave-one-out cross-validation errors
`k_predictions`	List of predictions for each k value
`bb_predictions`	Bootstrap mean predictions (if n.bb > 0)
`cri_L`	Lower credible interval bounds (if n.bb > 0)
`cri_U`	Upper credible interval bounds (if n.bb > 0)
`x_sorted`	Input x values (sorted)
`y_sorted`	Input y values (sorted by x)
`y_true_sorted`	True probabilities (sorted by x, if provided)
`k_min`	Minimum k value used
`k_max`	Maximum k value used

See Also

`mabilo` for continuous response regression, `predict.mabilog` for predictions on new data, `plot.mabilog` for diagnostic plots

Examples

```
# Generate binary data with smooth probability structure
set.seed(42)
x <- seq(0, 10, length.out = 200)
true_prob <- plogis(sin(x) - 0.5)
y <- rbinom(length(x), 1, true_prob)

# Fit MABILOG model
fit <- mabilog(x, y, y.true = true_prob, k.min = 5, k.max = 20)

# Plot results
plot(x, y, col = c("red", "blue")[y + 1], pch = 19, cex = 0.5)
lines(fit$x_sorted, fit$predictions, lwd = 2)
lines(x, true_prob, col = "green", lty = 2)
legend("topright", c("Data (y=0)", "Data (y=1)", "Fitted", "True"),
       col = c("red", "blue", "black", "green"),
       pch = c(19, 19, NA, NA), lty = c(NA, NA, 1, 2))

# With bootstrap confidence intervals

fit_bb <- mabilog(x, y, k.min = 5, k.max = 20, n.bb = 100)
plot(x, y, col = c("red", "blue")[y + 1], pch = 19, cex = 0.5)
polygon(c(fit_bb$x_sorted, rev(fit_bb$x_sorted)),
        c(fit_bb$cri_L, rev(fit_bb$cri_U)),
        col = "gray80", border = NA)
lines(fit_bb$x_sorted, fit_bb$predictions, lwd = 2)
```

`mae.plot` *Plot Median Absolute Error*

Description

Creates a plot of Median Absolute Error (MAE) with error bars

Usage

```
mae.plot(
  mae.mean,
  mae.mad,
  ylab = "Median Absolute Error",
  xlab = "Number of Nearest Neighbors"
)
```

Arguments

`mae.mean`	Vector of MAE means.
`mae.mad`	Vector of MAE median absolute deviations.
`ylab`	Label for y-axis.
`xlab`	Label for x-axis.

Details

This function creates a line plot with error bars showing the median absolute error across different numbers of nearest neighbors. The error bars represent ±0.5 * MAD (median absolute deviation).

Value

Invisibly returns NULL.

See Also

`vert.error.bar`

Examples

```
## Not run:
mae.mean <- c(NA, 0.5, 0.4, 0.35, 0.33, 0.32)
mae.mad <- c(NA, 0.1, 0.08, 0.07, 0.06, 0.06)
mae.plot(mae.mean, mae.mad)

## End(Not run)
```

`maelog` *Model-Averaged Local Logistic Regression*

Description

Performs model-averaged local logistic regression using kernel-weighted maximum likelihood estimation. The function implements bandwidth selection via cross-validation and supports both linear and quadratic local polynomial models.

Usage

```
maelog(
  x,
  y,
  fit.quadratic = FALSE,
  pilot.bandwidth = -1,
  kernel = 7L,
  min.points = NULL,
  cv.folds = 0L,
  n.bws = 50L,
  min.bw.factor = 0.05,
  max.bw.factor = 0.9,
  max.iterations = 100L,
  ridge.lambda = 1e-06,
  tolerance = 1e-08,
  with.errors = FALSE,
  with.bw.predictions = TRUE
)
```

Arguments

`x`	Numeric vector of predictor variables (explanatory variable).
`y`	Binary response vector containing only 0 or 1 values.
`fit.quadratic`	Logical; if `TRUE`, fits local quadratic models; if `FALSE` (default), fits local linear models.
`pilot.bandwidth`	Numeric; the bandwidth parameter for local fitting. If less than or equal to 0 (default = -1), bandwidth is selected automatically using cross-validation or LOOCV approximation.
`kernel`	Integer specifying the kernel function for weight calculation: **1** Epanechnikov kernel **2** Triangular kernel **4** Laplace (double exponential) kernel **5** Gaussian (normal) kernel **6** Biweight (quartic) kernel **7** Tricube kernel (default)
`min.points`	Integer; minimum number of points required in each local neighborhood for fitting. If `NULL` (default), automatically set to 3 for linear models or 4 for quadratic models. Must be at least 3 for linear and 4 for quadratic models.
`cv.folds`	Integer; number of folds for cross-validation when selecting bandwidth. If 0 (default), uses leave-one-out cross-validation (LOOCV) approximation. Must be between 0 and `length(x)`.
`n.bws`	Integer; number of bandwidth candidates to evaluate during automatic selection. Default is 50. Must be at least 2.
`min.bw.factor`	Numeric between 0 and 1; minimum bandwidth as a fraction of the data range. Default is 0.05. The minimum bandwidth tested is `min.bw.factor * diff(range(x))`.
`max.bw.factor`	Numeric greater than `min.bw.factor`; maximum bandwidth as a fraction of the data range. Default is 0.9. The maximum bandwidth tested is `max.bw.factor * diff(range(x))`.
`max.iterations`	Integer; maximum number of iterations for the local likelihood maximization algorithm. Default is 100. Must be positive.
`ridge.lambda`	Numeric; ridge penalty parameter for numerical stability in the local regression. Default is 1e-6. Must be positive.
`tolerance`	Numeric; convergence tolerance for the iterative fitting algorithm. Default is 1e-8. Must be positive.
`with.errors`	Logical; if `TRUE`, computes and returns standard errors for the predictions. Default is `FALSE`.
`with.bw.predictions`	Logical; if `TRUE` (default), returns the matrix of predictions for all bandwidth values tested during selection. Set to `FALSE` to save memory when only the optimal predictions are needed.

Details

The function implements a local likelihood approach to logistic regression, where a weighted logistic model is fit in a neighborhood around each point. The weights are determined by a kernel function and bandwidth parameter.

Model Specification: At each point x_0, the local log-odds are modeled as:

- Linear: $\log\left(\frac{p(x)}{1-p(x)}\right) = \beta_0 + \beta_1(x - x_0)$
- Quadratic: $\log\left(\frac{p(x)}{1-p(x)}\right) = \beta_0 + \beta_1(x - x_0) + \beta_2(x - x_0)^2$

where $p(x) = P(Y = 1|X = x)$.

Bandwidth Selection: When `pilot.bandwidth` <= `0`, the function automatically selects the bandwidth by minimizing the cross-validation error. The candidate bandwidths are logarithmically spaced between `min.bw.factor` and `max.bw.factor` times the data range.

Kernel Functions: All kernel functions $K(u)$ are defined on the interval $[-1, 1]$ and are zero outside this interval:

- Epanechnikov: $K(u) = \frac{3}{4}(1 - u^2)$
- Triangular: $K(u) = 1 - |u|$
- Laplace: $K(u) = \frac{1}{2}\exp(-|u|)$
- Gaussian: $K(u) = \frac{1}{\sqrt{2\pi}}\exp(-\frac{u^2}{2})$
- Biweight: $K(u) = \frac{15}{16}(1 - u^2)^2$
- Tricube: $K(u) = \frac{70}{81}(1 - |u|^3)^3$

Value

A list of class `"maelog"` containing:

`predictions`	Numeric vector of predicted probabilities at the input points.
`errors`	Numeric vector of standard errors if `with.errors` = `TRUE`; `NULL` otherwise.
`opt.bw`	The selected optimal bandwidth if `pilot.bandwidth` <= `0`; the input bandwidth otherwise.
`candidate.bandwidths`	Numeric vector of bandwidth values tested during selection if `pilot.bandwidth` <= `0`; `NULL` otherwise.
`mean.errors`	Numeric vector of cross-validation errors corresponding to `candidate.bandwidths` if `pilot.bandwidth` <= `0`; `NULL` otherwise.
`bw.predictions`	Matrix where column `i` contains predictions using `candidate.bandwidths[i]` if `with.bw.predictions` = `TRUE` and `pilot.bandwidth` <= `0`; `NULL` otherwise.
`x`	The input predictor values.
`y`	The input response values.
`call`	The matched function call.
`kernel`	The kernel type used.
`fit.quadratic`	Whether quadratic terms were included.

References

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368), 829-836.

Fan, J., & Gijbels, I. (1996). *Local Polynomial Modelling and Its Applications*. Chapman & Hall/CRC.

Loader, C. (1999). *Local Regression and Likelihood*. Springer.

See Also

predict.maelog for predictions at new points, plot.maelog for diagnostic plots

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 200
x <- seq(0, 1, length.out = n)
true.prob <- plogis(10 * (x - 0.5))  # Logistic function
y <- rbinom(n, 1, true.prob)

# Fit model with automatic bandwidth selection
fit1 <- maelog(x, y)

# Plot results
plot(x, y, col = rgb(0, 0, 0, 0.5), pch = 16,
     main = "Local Logistic Regression",
     xlab = "x", ylab = "Probability")
lines(x, fit1$predictions, col = "blue", lwd = 2)
lines(x, true.prob, col = "red", lwd = 2, lty = 2)
legend("topleft", c("Fitted", "True"),
       col = c("blue", "red"), lwd = 2, lty = c(1, 2))

# Fit with quadratic local models
fit2 <- maelog(x, y, fit.quadratic = TRUE, cv.folds = 5)

# Fit with fixed bandwidth
fit3 <- maelog(x, y, pilot.bandwidth = 0.1)

# Compare different kernels
fit.tricube <- maelog(x, y, kernel = 7)
fit.gauss <- maelog(x, y, kernel = 5)

# Larger example with cross-validation
n <- 1000
x <- runif(n, -2, 2)
prob <- plogis(2 * sin(pi * x))
y <- rbinom(n, 1, prob)

# Compare linear vs quadratic with 10-fold CV
fit.linear <- maelog(x, y, fit.quadratic = FALSE, cv.folds = 10)
fit.quad <- maelog(x, y, fit.quadratic = TRUE, cv.folds = 10)

# Plot bandwidth selection results
par(mfrow = c(1, 2))
plot(fit.linear$candidate.bandwidths, fit.linear$mean.errors,
     type = "l", xlab = "Bandwidth", ylab = "CV Error",
     main = "Linear Model")
abline(v = fit.linear$opt.bw, col = "red", lty = 2)

plot(fit.quad$candidate.bandwidths, fit.quad$mean.errors,
     type = "l", xlab = "Bandwidth", ylab = "CV Error",
     main = "Quadratic Model")
abline(v = fit.quad$opt.bw, col = "red", lty = 2)
```

```
## End(Not run)
```

---

magelo *Model Averaged Grid-based Epsilon LOwess (MAGELO)*

Description

A non-parametric regression method that combines local polynomial regression with model averaging using disk-shaped neighborhoods. Unlike traditional LOWESS which uses k-nearest neighbors, MAGELO employs fixed-radius neighborhoods (disks) and averages local polynomial models fitted at uniformly spaced grid points. This approach provides smoother transitions between local models and more stable estimates in regions with varying data density.

Usage

```
magelo(
  x,
  y,
  y.true = NULL,
  grid.size = 400,
  degree = 1,
  min.K = 5,
  f = NULL,
  bw = NULL,
  min.bw.f = 0.025,
  method = ifelse(length(x) < 1000, "LOOCV", "CV"),
  n.bws = 100,
  n.BB = 1000,
  get.predictions.CrI = TRUE,
  get.gpredictions.CrI = TRUE,
  get.BB.predictions = FALSE,
  get.BB.gpredictions = FALSE,
  level = 0.95,
  n.C.itr = 0,
  C = 6,
  stop.C.itr.on.min = TRUE,
  n.cv.folds = 10,
  n.cv.reps = 20,
  nn.kernel = "epanechnikov",
  y.binary = FALSE,
  cv.nNN = 3,
  n.perms = 0,
  n.cores = 1,
  use.binloss = FALSE,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of predictor values
`y`	Numeric vector of response values, must be same length as x
`y.true`	Optional numeric vector of true response values (for validation)
`grid.size`	Integer specifying the number of grid points for model fitting. Default is 400, which provides good balance between accuracy and computation speed
`degree`	Integer specifying polynomial degree (0, 1, or 2): - 0: weighted mean regression - 1: local linear regression (y ~ x) - 2: local quadratic regression (y ~ poly(x, 2))
`min.K`	Integer specifying minimum number of points required in each local neighborhood
`f`	Numeric between 0 and 1, specifying the proportion of x range to use for each local window. If NULL, optimal value is determined by cross-validation
`bw`	Numeric specifying bandwidth (radius of disk neighborhood). If NULL, optimal value is determined by cross-validation
`min.bw.f`	Numeric specifying minimum bandwidth as fraction of x range. min_bandwidth = min.bw.f * range(x). Default: 0.025
`method`	Character string specifying bandwidth optimization method: - "LOOCV": Leave-one-out cross-validation (for n < 1000) - "CV": K-fold cross-validation (for n >= 1000)
`n.bws`	Integer specifying number of bandwidths to try during optimization
`n.BB`	Integer specifying number of Bayesian bootstrap iterations
`get.predictions.CrI`	Logical; if TRUE, compute credible intervals for fitted values at x
`get.gpredictions.CrI`	Logical; if TRUE, compute credible intervals for fitted values at grid points
`get.BB.predictions`	Logical; if TRUE, return matrix of bootstrap estimates at x
`get.BB.gpredictions`	Logical; if TRUE, return matrix of bootstrap estimates at grid points
`level`	Numeric between 0 and 1 specifying credible interval level
`n.C.itr`	Integer specifying number of Cleveland's iterative reweighting steps
`C`	Numeric specifying scaling factor for residuals in robust fitting
`stop.C.itr.on.min`	Logical; if TRUE, stop iterations when improvement plateaus
`n.cv.folds`	Integer specifying number of folds for cross-validation
`n.cv.reps`	Integer specifying number of cross-validation repetitions
`nn.kernel`	Character string specifying kernel function: "epanechnikov", "triangular", "tr.exponential", or "normal"
`y.binary`	Logical; if TRUE, use binary loss function for optimization
`cv.nNN`	Integer specifying number of nearest neighbors for grid interpolation
`n.perms`	Integer specifying number of y permutations for p-value calculation
`n.cores`	Integer specifying number of CPU cores for parallel processing
`use.binloss`	Logical; if TRUE, use binary loss function for optimization
`verbose`	Logical; if TRUE, print progress information

Value

A list with class "magelo" containing:

- Fitted values at grid points (gpredictions) and input locations (predictions)
- Credible intervals if requested
- Bootstrap estimates if requested
- Optimal bandwidth and optimization results
- Model coefficients
- Input parameters

Uncertainty Estimation

The function provides several options for uncertainty quantification through Bayesian bootstrap. Use `n.BB` to specify the number of bootstrap iterations, and control which intervals to compute with `get.predictions.CrI` and `get.gpredictions.CrI`. The credible interval level can be adjusted with the `level` parameter.

Robustness Parameters

MAGELO includes options for robust fitting using Cleveland's iterative reweighting procedure. The `n.C.itr` parameter controls the number of iterations, while `C` sets the scaling factor for residuals. Use `stop.C.itr.on.min` to enable early stopping when improvements plateau.

Permutation Testing

Statistical significance can be assessed through permutation testing. Set `n.perms` for standard permutation tests. If true response values are available, provide them via `y.true` for validation purposes.

Examples

```
x <- seq(0, 10, length.out = 100)
y <- sin(x) + rnorm(100, 0, 0.1)
fit <- magelo(x, y, degree = 1)
plot(x, y)
lines(fit$xgrid, fit$gpredictions, col = "red")
```

`magelo.fit` *Core Fitting Function for MAGELO (Model Averaged Grid-based Epsilon LOwess)*

Description

Internal fitting function for MAGELO that performs local polynomial regression using pre-computed nearest neighbor information. Supports weighted mean (degree 0), local linear (degree 1), and local quadratic (degree 2) regression with optional robust fitting via iterative reweighting and uncertainty estimation via Bayesian bootstrap.

Usage

```
magelo.fit(
  bw,
  degree,
  x,
  y,
  xgrid,
  nn.i,
  nn.d,
  min.K = 15,
  n.C.itr = 0,
  C = 6,
  stop.C.itr.on.min = TRUE,
  n.BB = 0,
  get.BB.gpredictions = FALSE,
  get.BB.predictions = FALSE,
  get.gpredictions.CrI = FALSE,
  get.predictions.CrI = TRUE,
  level = 0.95,
  nn.kernel = "epanechnikov",
  y.binary = FALSE,
  n.perms = 0
)
```

Arguments

`bw`	Numeric specifying bandwidth (radius of disk neighborhood)
`degree`	Integer (0, 1, or 2) specifying polynomial degree: - 0: weighted mean regression - 1: local linear regression (y ~ x) - 2: local quadratic regression (y ~ poly(x, 2))
`x`	Numeric vector of predictor values
`y`	Numeric vector of response values, must be same length as x
`xgrid`	Numeric vector of uniformly spaced grid points spanning range(x)
`nn.i`	Integer matrix (n_grid x K) of indices for x neighbors of each grid point
`nn.d`	Numeric matrix (n_grid x K) of distances to x neighbors of each grid point
`min.K`	Integer specifying minimum number of points required in each local neighborhood
`n.C.itr`	Integer specifying number of Cleveland's iterative reweighting steps
`C`	Numeric scaling factor for residuals in robust fitting. Higher values reduce influence of outliers
`stop.C.itr.on.min`	Logical; if TRUE, stop iterations when improvements in fitted values become negligible
`n.BB`	Integer specifying number of Bayesian bootstrap iterations
`get.BB.gpredictions`	Logical; if TRUE, return matrix of bootstrap estimates at grid points
`get.BB.predictions`	Logical; if TRUE, return matrix of bootstrap estimates at x
`get.gpredictions.CrI`	Logical; if TRUE, compute credible intervals at grid points

`get.predictions.CrI`	Logical; if TRUE, compute credible intervals at x
`level`	Numeric between 0 and 1 specifying credible interval level
`nn.kernel`	Character string specifying kernel function: "epanechnikov", "triangular", "tr.exponential", or "normal"
`y.binary`	Logical; if TRUE, treat y as binary outcome
`n.perms`	Integer specifying number of y permutations for p-value calculation

Value

A list containing:

- x: Input predictor values
- y: Input response values
- xgrid: Grid points
- max.K: Maximum number of neighbors used for each point
- gpredictions: Fitted values at grid points
- predictions: Fitted values at x
- BB.predictions: Bootstrap estimates at x (if requested)
- BB.gpredictions: Bootstrap estimates at grid points (if requested)
- BB.dgpredictions: Bootstrap derivatives at grid points (if requested)
- gpredictions.CrI: Credible intervals at grid points (if requested)
- predictions.CrI: Credible intervals at x (if requested)
- beta: Model coefficients at each grid point

Robust Fitting Parameters

The robust fitting procedure implements Cleveland's iterative reweighting algorithm to reduce the influence of outliers. The `n.C.itr` parameter controls the maximum number of reweighting iterations, while `C` determines how aggressively outliers are downweighted. Setting `stop.C.itr.on.min = TRUE` enables early stopping when the algorithm converges, saving computation time without sacrificing accuracy.

Uncertainty Estimation

Uncertainty quantification is performed using Bayesian bootstrap, which generates posterior distributions for fitted values by resampling with exponential weights. The `n.BB` parameter controls the number of bootstrap samples. Credible intervals can be computed at both the original data points and grid points. The bootstrap estimates themselves can also be returned for custom uncertainty analyses.

Note

This is an internal function called by magelo() after bandwidth optimization. It should not typically be called directly unless you have pre-computed nearest neighbor information and know the appropriate bandwidth.

magelog *Grid-based Model Averaged Bandwidth Logistic Regression*

Description

Performs model-averaged bandwidth logistic regression using local polynomial fitting. The function implements a flexible approach to binary regression by fitting local models at points of a uniform grid spanning the range of predictor values. Multiple local models are combined to produce robust predictions. It supports both linear and quadratic local models, automatic bandwidth selection via cross-validation, and various kernel types for weight calculation.

Usage

```
magelog(
  x,
  y,
  grid.size = 200,
  fit.quadratic = FALSE,
  pilot.bandwidth = -1,
  kernel = 7L,
  min.points = NULL,
  cv.folds = 5L,
  n.bws = 50L,
  min.bw.factor = 0.05,
  max.bw.factor = 0.9,
  max.iterations = 100L,
  ridge.lambda = 1e-06,
  tolerance = 1e-08,
  with.bw.predictions = TRUE
)
```

Arguments

`x`	Numeric vector of predictor variables. Must not contain missing, infinite, or non-numeric values.
`y`	Binary numeric vector (0 or 1) of response variables. Must be the same length as `x` and contain only 0s and 1s.
`grid.size`	Integer; number of points in the uniform grid where local models are centered. Must be at least 2. Larger values provide smoother predictions but increase computation time. Default is 200.
`fit.quadratic`	Logical; whether to include quadratic terms in the local models. If `TRUE`, local quadratic logistic regression is used; if `FALSE`, local linear logistic regression is used. Default is `FALSE`.
`pilot.bandwidth`	Numeric; bandwidth for local fitting. If less than or equal to 0, bandwidth is automatically selected using cross-validation. The bandwidth controls the size of the local neighborhood used for fitting. Default is -1 (automatic selection).
`kernel`	Integer; kernel type for weight calculation: **1** Epanechnikov kernel

2 Triangular kernel
4 Laplace kernel
5 Normal (Gaussian) kernel
6 Biweight kernel
7 Tricube kernel (default)

`min.points`	Integer or `NULL`; minimum number of points required for local fitting. If `NULL`, automatically set to 3 for linear models or 4 for quadratic models. Must be at least 3 for linear models and 4 for quadratic models to ensure identifiability. Default is `NULL`.
`cv.folds`	Integer; number of cross-validation folds for bandwidth selection. Must be at least 3 and cannot exceed the number of observations. Higher values provide more stable bandwidth selection but increase computation time. Default is 5.
`n.bws`	Integer; number of bandwidths to try in automatic selection. Must be at least 2. More bandwidths provide finer resolution but increase computation time. Default is 50.
`min.bw.factor`	Numeric; minimum bandwidth factor relative to data range. Must be positive and less than 1. Controls the smallest bandwidth to consider as a fraction of the x-range. Default is 0.05.
`max.bw.factor`	Numeric; maximum bandwidth factor relative to data range. Must be greater than `min.bw.factor`. Controls the largest bandwidth to consider as a fraction of the x-range. Default is 0.9.
`max.iterations`	Integer; maximum number of iterations for local fitting algorithm. Must be positive. Default is 100.
`ridge.lambda`	Numeric; ridge parameter for numerical stability in local fitting. Must be positive. Larger values provide more stability but may introduce bias. Default is 1e-6.
`tolerance`	Numeric; convergence tolerance for local fitting algorithm. Must be positive. Smaller values provide more precise convergence but may increase iterations. Default is 1e-8.
`with.bw.predictions`	Logical; whether to return predictions for all bandwidth values tried during cross-validation. If `TRUE`, allows examination of how predictions vary with bandwidth. Default is `TRUE`.

Details

The function fits local logistic regression models centered at points of a uniform grid spanning the range of x values. Local models are fit using kernel-weighted maximum likelihood estimation with optional ridge regularization for numerical stability.

The bandwidth parameter controls the size of the local neighborhood used for each local fit. When `pilot.bandwidth <= 0`, the function automatically selects an optimal bandwidth using k-fold cross-validation to minimize the Brier score.

Local models can be either linear or quadratic (controlled by `fit.quadratic`). For numerical stability, each local fit requires a minimum number of points: at least 3 for linear models and 4 for quadratic models.

Predictions at the original x points are obtained by linear interpolation from the grid-based predictions. This grid-based approach provides computational efficiency and naturally smooth prediction curves.

The function uses compiled C++ code for computational efficiency, particularly important when performing cross-validation over multiple bandwidth values.

Value

A list of class "magelog" containing:

`x.grid` Numeric vector of uniform grid points where local models are centered

`predictions` Numeric vector of predicted probabilities at original x points

`bw.grid.predictions` Matrix of predictions at grid points for each bandwidth (only if `with.bw.predictions = TRUE`). Rows correspond to grid points, columns to different bandwidths

`mean.brier.errors` Numeric vector of cross-validation Brier scores for each bandwidth

`opt.brier.bw.idx` Integer index of optimal bandwidth minimizing Brier score

`bws` Numeric vector of bandwidth values tried

`fit.info` List containing fitting parameters used

`x` Original predictor values (sorted if necessary)

`y` Original response values (sorted to match x)

References

Loader, C. (1999). Local Regression and Likelihood. Springer-Verlag.

Fan, J. and Gijbels, I. (1996). Local Polynomial Modelling and Its Applications. Chapman & Hall.

See Also

`predict.magelog` for making predictions on new data,

Examples

```
## Not run:
# Generate example data with logistic relationship
set.seed(123)
n <- 200
x <- seq(0, 1, length.out = n)
# True probability function
p_true <- 1/(1 + exp(-(x - 0.5)*10))
y <- rbinom(n, 1, p_true)

# Fit model with automatic bandwidth selection
fit <- magelog(x, y, grid.size = 100, fit.quadratic = FALSE, cv.folds = 5)

# Plot results
plot(x, y, pch = 19, col = adjustcolor("black", 0.5),
     xlab = "x", ylab = "Probability",
     main = "Local Logistic Regression")
# Add true probability curve
lines(x, p_true, col = "gray", lwd = 2, lty = 2)
# Add fitted curve
lines(fit$x.grid, fit$bw.grid.predictions[, fit$opt.brier.bw.idx],
      col = "red", lwd = 2)
legend("topleft", c("True probability", "Fitted curve", "Data"),
       col = c("gray", "red", "black"), lty = c(2, 1, NA),
       pch = c(NA, NA, 19), lwd = c(2, 2, NA))

# Examine bandwidth selection
plot(fit$bws, fit$mean.brier.errors, type = "b",
     xlab = "Bandwidth", ylab = "Cross-validation Brier Score",
     main = "Bandwidth Selection")
abline(v = fit$bws[fit$opt.brier.bw.idx], col = "red", lty = 2)

## End(Not run)
```

---

map.S.to.X *Map Sample IDs to 3D Space*

Description

Highlights a subset of samples in 3D space with different colors and sizes

Usage

```
map.S.to.X(S, X, radius = 0.075, col = "red", legend.title = NULL)
```

Arguments

S	Vector of sample IDs to highlight.
X	Matrix or data.frame with 3 columns representing 3D coordinates.
radius	Radius of spheres for highlighted samples.
col	Color for highlighted samples.
legend.title	Title for the legend.

Details

This function creates a 3D plot where samples in set S are highlighted with larger spheres in a specified color, while other samples are shown in gray.

Value

Invisibly returns NULL.

Examples

```
## Not run:
X <- matrix(rnorm(300), ncol = 3)
rownames(X) <- paste0("Sample", 1:100)
S <- paste0("Sample", c(1, 5, 10, 15, 20))

map.S.to.X(S, X, radius = 0.075, col = 'red')

## End(Not run)
```

matrix.format *Format Numbers in a Matrix with Specified Decimal Places*

Description

Formats all numeric elements in a matrix to have a specified number of decimal places, ensuring a consistent display with leading and trailing zeros as needed.

Usage

```
matrix.format(x, digits = 2, nsmall = 2, ...)
```

Arguments

`x`	Input matrix (numeric)
`digits`	Number of digits to round to (default = 2)
`nsmall`	Minimum number of digits to display after decimal point (default = 2)
`...`	Additional arguments (currently unused)

Value

A character matrix with all numbers formatted according to specifications

Examples

```
m <- matrix(c(0.7, 1.235, 0.1, 0.8876), nrow = 2)
matrix.format(m)              # default: 2 digits
matrix.format(m, digits = 3)   # 3 digits
matrix.format(m, nsmall = 3)   # at least 3 decimal places
```

meanshift.data.smoother *Mean Shift Data Smoother*

Description

Performs mean shift smoothing on a dataset using various algorithms. Mean shift is a non-parametric feature-space analysis technique for locating the maxima of a density function. This implementation offers multiple variants including adaptive step sizes, gradient field averaging, and momentum-based updates.

Usage

```
meanshift.data.smoother(
  X,
  k,
  density.k = 1,
  n.steps = 10,
  step.size = 0.1,
  ikernel = 1,
  dist.normalization.factor = 1.01,
  method = "precomputed",
  average.direction.only = NULL,
  momentum = 0.9,
  increase.factor = 1.2,
  decrease.factor = 0.5
)
```

Arguments

`X`	A numeric matrix or data frame where rows represent points and columns represent features. Will be coerced to a matrix if necessary.
`k`	An integer specifying the number of nearest neighbors to consider for gradient estimation. Must be positive and less than the number of points.
`density.k`	An integer specifying the number of nearest neighbors to consider for density estimation. Must be positive and less than the number of points. Default is 1.
`n.steps`	An integer specifying the number of smoothing steps to perform. Must be positive. Default is 10.
`step.size`	A numeric value specifying the step size for updating point positions. Should be between 0 and 1. Default is 0.1.
`ikernel`	An integer specifying the kernel function to use: • 1: Epanechnikov kernel • 2: Triangular kernel • 3: Truncated Exponential kernel • 4: Normal (Gaussian) kernel Default is 1.
`dist.normalization.factor`	A numeric value specifying the scaling factor for distance normalization. Should be greater than 1. Default is 1.01.
`method`	A character string or integer specifying the smoothing method: **"basic" (0)** Basic mean shift algorithm **"precomputed" (1)** Mean shift with precomputed nearest neighbors (default) **"grad_field" (2)** Mean shift with gradient field averaging **"grad_field_dir" (3)** Mean shift with gradient field averaging (direction only) **"adaptive" (4)** Adaptive mean shift **"adaptive_dir" (5)** Adaptive mean shift (direction only) **"knn_adaptive" (6)** KNN adaptive mean shift **"knn_adaptive_grad" (7)** KNN adaptive with gradient field averaging **"knn_adaptive_grad_dir" (8)** KNN adaptive with gradient field averaging (direction only)

"adaptive_momentum" (9) Adaptive with gradient field averaging and momentum

"adaptive_momentum_dir" (10) Adaptive with gradient field averaging and momentum (direction only)

Default is "precomputed".

`average.direction.only`
: A logical value indicating whether to average only the directions of gradients (TRUE) or full gradient vectors (FALSE). If NULL (default), this is automatically determined based on the method:

- Methods ending in "_dir" use TRUE
- Other methods use FALSE

`momentum`
: A numeric value specifying the momentum factor for methods "adaptive_momentum" and "adaptive_momentum_dir". Should be between 0 and 1. Default is 0.9. Ignored for other methods.

`increase.factor`
: A numeric value specifying the factor by which to increase the step size in adaptive methods. Should be greater than 1. Default is 1.2. Only used for adaptive methods (4-10).

`decrease.factor`
: A numeric value specifying the factor by which to decrease the step size in adaptive methods. Should be between 0 and 1. Default is 0.5. Only used for adaptive methods (4-10).

Details

The mean shift algorithm iteratively moves each data point towards the mode of the density estimated in its neighborhood. Different variants offer various improvements:

- **Basic/Precomputed**: Standard mean shift with optional nearest neighbor precomputation
- **Gradient Field**: Averages gradients across the dataset for smoother updates
- **Adaptive**: Dynamically adjusts step sizes based on convergence behavior
- **KNN Adaptive**: Uses k-nearest neighbors for adaptive bandwidth selection
- **Momentum**: Incorporates previous update directions for faster convergence

The "direction only" variants normalize gradient vectors before averaging, which can be more robust to outliers but may converge more slowly.

Value

A list of class "MSD" containing:

X The original input dataset

X.traj A list of matrices, each representing the smoothed dataset at each step

median.kdistances A numeric vector of median k-distances for each step

opt.step The step number with minimum median k-distance

dX The smoothed dataset at the optimal step

method The method used (as a string)

params A list of all parameters used

References

Comaniciu, D., & Meer, P. (2002). Mean shift: A robust approach toward feature space analysis. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24(5), 603-619.

See Also

`plot.MSD` for plotting the results

Examples

```
## Not run:
# Generate sample data
set.seed(123)
X <- matrix(rnorm(200), ncol = 2)

# Basic mean shift smoothing
result1 <- meanshift.data.smoother(X, k = 5)

# Adaptive mean shift with gradient field averaging
result2 <- meanshift.data.smoother(X, k = 5, method = "adaptive")

# With momentum
result3 <- meanshift.data.smoother(X, k = 5, method = "adaptive_momentum",
                                  momentum = 0.95)

# Plot the trajectory
plot(result1$median.kdistances, type = 'l',
     xlab = 'Step', ylab = 'Median k-distance')
abline(v = result1$opt.step, col = 'red', lty = 2)

# Compare original and smoothed data
par(mfrow = c(1, 2))
plot(X, main = "Original", pch = 19, cex = 0.5)
plot(result1$dX, main = "Smoothed", pch = 19, cex = 0.5)

## End(Not run)
```

`minh.limit` *Find Minimum Hop Limit for Path Existence*

Description

Determines the minimum hop limit required for a path to exist between two vertices.

Usage

```
minh.limit(x, from, to)
```

Arguments

`x`	A path.graph.series object
`from`	Source vertex index (1-based)
`to`	Target vertex index (1-based)

Value

Integer minimum h value where path exists, or NULL if no path exists for any h value in the series.

Examples

```
## Not run:
pgs <- create.path.graph.series(graph, edge.lengths, h.values = 1:5)
min.h <- minh.limit(pgs, from = 1, to = 5)
if (!is.null(min.h)) {
  cat("Minimum hops needed:", min.h, "\n")
}

## End(Not run)
```

`minmax.normalize` *Min-Max Normalization*

Description

Scales a numeric vector to a specified range [y.min, y.max]. This transformation shifts and rescales the data, so the minimum value corresponds to y.min and the maximum value to y.max.

Usage

```
minmax.normalize(x, y.min = 0, y.max = 1)
```

Arguments

`x`	A numeric vector to be normalized.
`y.min`	The minimum value in the normalized range (default is 0).
`y.max`	The maximum value in the normalized range (default is 1).

Value

A numeric vector where the original values of `x` have been scaled to fall within the range `[y.min, y.max]`.

Examples

```
x <- c(1, 2, 3, 4, 5)
minmax.normalize(x, 0, 1)  # Normalizes x to the range \eqn{[0, 1]}
```

mllm.1D *A local linear 1D model with y being a matrix. It expects bw value.*

Description

A local linear 1D model with y being a matrix. It expects bw value.

Usage

```
mllm.1D(
  x,
  Y,
  bw,
  y.binary = FALSE,
  with.BB = FALSE,
  grid.size = 400,
  min.K = 15,
  nn.kernel = "epanechnikov"
)
```

Arguments

`x`	A numeric vector of a predictor variable.
`Y`	A matrix such that nrow(Y) = length(x) representating many instances of the outcome variable.
`bw`	A bandwidth parameter.
`y.binary`	A logical variable. If TRUE, bw optimization is going to use a binary loss function mean(y(1-p) + (1-y)p).
`with.BB`	A logical parameter. Set to TRUE if Bayesian bootstraps of the column's mean function are to be returned.
`grid.size`	A number of grid points; was grid.size = 10*length(x), but the results don't seem to be different from 400 which is much faster.
`min.K`	The minimal number of x NN's that must be present in each window.
`nn.kernel`	The name of a kernel that will be applied to NN distances of local linear regression models.

Value

EY.grid

mllm.1D.fit.and.predict

Fits 1D rllm model and generates predictions esimates for each column of a matrix Y.

Description

Fits 1D rllm model and generates predictions esimates for each column of a matrix Y.

Usage

```
mllm.1D.fit.and.predict(Y, y.binary, nn.i, nn.w, nn.x, max.K, degree)
```

Arguments

`Y`	A matrix with the number of rows that is the same as the length of x that was used to construct nn.* matrices. The main application of this routine is for the case when the columns of Y are permutations some y.
`y.binary`	Set to TRUE, if the values of all columns of Y are within the interval `[0,1]`.
`nn.i`	A matrix of the indices of K nearest neighbors of each element of the grid, where K is determined in the parent routine.
`nn.w`	A matrix of weights.
`nn.x`	A matrix of x values over K nearest neighbors of each element of the grid.
`max.K`	An array of indices indicating the range where weights are not 0. Indices < max.K at i have weights > 0.
`degree`	A degree of the polynomial of x in the linear regresion. The only allowed values are 1 and 2.

Value

EY.grid

mode.1D

Estimates the mode of a numeric vector

Description

Estimates the mode of a numeric vector

Usage

```
mode.1D(x, min.size = 10, ...)
```

Arguments

`x`	A numeric vector.
`min.size`	The minimal number of finite values within x. If length(x) < min.size, then the result is NA.
`...`	Parameters to be passed to density()

`model.errors` *Create a model.errors object*

Description

Constructor function for creating a model.errors object that stores integrated error measures and their bootstrap distributions for model evaluation.

Usage

```
model.errors(integrals, bb.integrals)
```

Arguments

`integrals`	Numeric vector or matrix of integrated error measures
`bb.integrals`	Numeric matrix containing bootstrap samples of the integrated error measures, used for uncertainty quantification

Value

An object of class "model.errors" containing the error integrals and their bootstrap distributions

`model.frame.graph.spectral.lowess` *Extract Model Frame from Graph Spectral LOWESS*

Description

Constructs a data frame containing the graph structure and response values

Usage

```
## S3 method for class 'graph.spectral.lowess'
model.frame(formula, ...)
```

Arguments

`formula`	A ’graph.spectral.lowess’ object
`...`	Additional arguments (currently unused)

Value

Data frame with vertex information

`model.frame.graph.spectral.ma.lowess`
Extract Model Frame from Graph Spectral MA LOWESS

Description

Constructs a data frame containing the graph structure, response values, and model averaging information

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
model.frame(formula, ...)
```

Arguments

`formula`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

Data frame with vertex and model averaging information

`morse.analysis.plot` *Plot Morse Analysis*

Description

Combined plot showing function contours, critical points, and a gradient trajectory.

Usage

```
morse.analysis.plot(
  grid,
  f.grid,
  trajectory,
  critical.points,
  show.contours = TRUE,
  main = "Morse Analysis",
  ...
)
```

Arguments

`grid`	Grid object from create.grid
`f.grid`	Matrix of function values
`trajectory`	Trajectory object from compute.gradient.trajectory
`critical.points`	List with critical points from find.critical.points
`show.contours`	Logical, whether to show contour lines (default TRUE)
`main`	Plot title
`...`	Additional arguments passed to plotting functions

Value

Invisible NULL

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
critical <- find.critical.points(f.grid)
traj <- compute.gradient.trajectory(15, 20, f.grid)
morse.analysis.plot(grid, f.grid, traj, critical)

## End(Not run)
```

`morse.smale.cells.plot` *Plot Morse-Smale Cells*

Description

Visualizes Morse-Smale cell decomposition with critical points.

Usage

```
morse.smale.cells.plot(grid, morse_smale_complex, f.grid = NULL, ...)
```

Arguments

`grid`	Grid object from create.grid
`morse_smale_complex`	Morse-Smale complex object
`f.grid`	Optional matrix of function values for contours
`...`	Additional arguments passed to plot

Value

Invisible NULL

Examples

```
## Not run:
grid <- create.grid(30)
f <- function(x, y) sin(3*x) * cos(3*y)
f.grid <- evaluate.function.on.grid(f, grid)
complex <- create.morse.smale.complex(f.grid)
morse.smale.cells.plot(grid, complex, f.grid)

## End(Not run)
```

---

morse.smale.complex.lmin.lmax.summary
Process Complete Morse-Smale Complex Analysis Pipeline

Description

Executes a complete analysis pipeline for Morse-Smale complex data by sequentially processing critical points, creating labels, generating indicators, computing frequency tables, and forming contingency relationships.

Usage

```
morse.smale.complex.lmin.lmax.summary(
  MS.res,
  state.space,
  taxonomy,
  rel.condE,
  freq.thld = 100,
  min.relAb.thld = 0.05,
  outcome.name = "outcome"
)
```

Arguments

`MS.res`	List containing Morse-Smale complex results with component MS_cx
`state.space`	Matrix representing the state space (e.g., ASV abundance matrix)
`taxonomy`	Taxonomy information for state space features
`rel.condE`	Numeric vector of relative conditional expectations
`freq.thld`	Frequency threshold for filtering (default: 100)
`min.relAb.thld`	Minimum relative abundance threshold for labels (default: 0.05)
`outcome.name`	Character string describing the outcome variable (default: "outcome")

Details

The function executes the following steps:

1. Creates labels for local minima and maxima based on taxonomic composition
2. Generates indicator vectors and label tables for critical points
3. Computes frequency tables with statistics for critical points
4. Creates contingency table showing relationships between local minima and maxima

Value

A list with four components:

`labels`	Results from create.lmin.lmax.labels including lmin.labels and lmax.labels
`indicators`	Results from create.lmin.lmax.label.indicators including indicator vectors and tables
`frequencies`	Results from create.lmin.lmax.frequency.tables including frequency tables and captions
`contingency`	Results from create.lmin.lmax.contingency.table including contingency table and caption

Examples

```
## Not run:
results <- morse.smale.complex.lmin.lmax.summary(
    MS.res = smoothed.condE.MS.res,
    state.space = asv.matrix,
    taxonomy = asv.taxonomy,
    rel.condE = rel.smoothed.condE,
    outcome.name = "smoothed conditional expectation"
)

## End(Not run)
```

`morse.smale.complex.plot` *Plot Morse-Smale Complex from Function*

Description

Creates comprehensive visualization of Morse-Smale complex.

Creates a comprehensive visualization of the Morse-Smale complex by plotting contour lines and gradient flow trajectories from a grid of sample points. Ascending trajectories (to maxima) are shown in red, descending trajectories (to minima) are shown in blue.

Usage

```
morse.smale.complex.plot(grid, mixture, n_sample_points = 20, ...)

morse.smale.complex.plot(grid, mixture, n_sample_points = 20, ...)
```

Arguments

`grid`	Grid object created by create.grid containing x, y coordinates
`mixture`	Object containing: • f: Function taking (x, y) and returning scalar value • gradient: Function taking (x, y) and returning gradient vector c(dx, dy)
`n_sample_points`	Number of sample points along each axis for trajectory visualization
`...`	Additional arguments passed to function.contours.plot

Value

Invisible NULL

Invisible NULL

Examples

```
## Not run:
# Create a simple function with two critical points
grid <- create.grid(50)
mixture <- list(
  f = function(x, y) sin(3*pi*x) * cos(3*pi*y),
  gradient = function(x, y) {
    c(3*pi*cos(3*pi*x)*cos(3*pi*y),
      -3*pi*sin(3*pi*x)*sin(3*pi*y))
  }
)
morse.smale.complex.plot(grid, mixture, n_sample_points = 10)

## End(Not run)
```

---

morse.smale.complex.plot.from.critical
Plot Morse-Smale Complex from Critical Points

Description

Plots Morse-Smale complex emphasizing critical point structure.

Plots the Morse-Smale complex with emphasis on critical point structure. Shows contour lines, critical points (maxima as red triangles, minima as blue triangles, saddles as green diamonds), and optionally the separatrices emanating from saddle points.

Usage

```
morse.smale.complex.plot.from.critical(
  critical_points,
  mixture,
  grid,
  show_separatrices = TRUE,
  ...
)

morse.smale.complex.plot.from.critical(
  critical_points,
  mixture,
  grid,
  show_separatrices = TRUE,
  ...
)
```

Arguments

`critical_points`	List containing: • maxima: Matrix of maximum coordinates (n x 2) • minima: Matrix of minimum coordinates (n x 2) • saddles: Matrix of saddle point coordinates (n x 2)
`mixture`	Object containing: • f: Function taking (x, y) and returning scalar value • gradient: Function taking (x, y) and returning gradient vector c(dx, dy)
`grid`	Grid object created by create.grid for function evaluation
`show_separatrices`	Logical, whether to compute and show separatrices from saddle points
`...`	Additional arguments passed to function.contours.plot

Value

Invisible NULL

Invisible NULL

Examples

```
## Not run:
# Example with known critical points
grid <- create.grid(50)
mixture <- list(
  f = function(x, y) x^2 - y^2,  # Simple saddle
  gradient = function(x, y) c(2*x, -2*y)
)
critical_points <- list(
  maxima = matrix(c(0.8, 0.2), nrow = 1),
  minima = matrix(c(0.2, 0.8), nrow = 1),
  saddles = matrix(c(0.5, 0.5), nrow = 1)
)
morse.smale.complex.plot.from.critical(critical_points, mixture, grid)

## End(Not run)
```

`morse.smale.graph.persistence`
Morse-Smale Graph Persistence Analysis

Description

This function analyzes the persistence of Morse-Smale graphs over a range of nearest neighbor values (k) for a given data matrix X and response variable y. It computes the nerve graphs and their associated Morse-Smale complex graphs for each k value and determines the isomorphism and deviation from isomorphism between consecutive Morse-Smale graphs.

Usage

```
morse.smale.graph.persistence(
  X,
  y,
  Ks,
  ref.MS.graph = NULL,
  allow.disconnected.graph = TRUE,
  verbose = FALSE
)
```

Arguments

`X`	A data matrix or data frame.
`y`	A numeric vector representing the response variable over X.
`Ks`	A vector of positive integer values specifying the range of k values.
`ref.MS.graph`	A reference Morse-Smale graph. Usually, the Morse-Smale graph of the y over the full graph or of full dataset before subsampling. If ref.MS.graph is not NULL, is.isomorphic.with.ref.MS.graph and dev.from.isomorphism.with.MS.graph are computed.
`allow.disconnected.graph`	Logical. If TRUE (default), the algorithm proceeds even if none of the nerve graphs associated with (X, y) are connected.
`verbose`	If set to TRUE, it prints progress messages.

Value

A list with four components:

- nerve.graph: A list of nerve graphs computed for each k value.
- MS.graph: A list of Morse-Smale complex graphs computed for each k value.
- are.consecutive.graphs.isomorphic: A vector indicating whether consecutive Morse-Smale graphs are isomorphic.
- is.isomorphic.with.ref.MS.graph: A vector indicating whether the computed Morse-Smale graphs are isomorphic with the reference Morse-Smale graph.
- first.index.of.longest.stretch.of.isomorphic.MS.graphs: The first index of the longest stretch of isomorphic MS graphs.
- length.of.longest.stretch.of.isomorphic.MS.graphs: The length of the longest stretch of isomorphic MS graphs.

`mstree` *C implementation of a Minimal Spanning Tree on a matrix X with > 1 column*

Description

C implementation of a Minimal Spanning Tree on a matrix X with > 1 column

Usage

```
mstree(X, i = 1)
```

Arguments

`X`	A numeric matrix or data frame with more than one column.
`i`	An index of the row of X from which the algorithm starts. Default is 1.

Value

A list containing two elements:

`edges`	A matrix of edge indices, where each row represents an edge in the MST.
`edge.lens`	A vector of edge lengths corresponding to the edges in the MST.

`mstree.grid` *Creates a quasi-uniform grid over the edges of the minimal spanning tree of a state space*

Description

For each edge of the the minimal spanning tree of X whose length is greater than eSDf * m a uniform grid with the distance between consecutive points equal to m, where m = mode(edge lengths).

Usage

```
mstree.grid(X, eSDf = 1.5)
```

Arguments

`X`	A set of points for which minimal spanning tree edge subdivision is to be created.
`eSDf`	An edge subdivision factor, such that a uniform grid is created over the given edge of mstree(X) if its length is greater than eSDf * mode(edge.len).

`mutual.information` *Estimate Mutual Information Between Dataset Membership and Features*

Description

This function estimates the mutual information between the dataset membership (X or Y) and the feature values. In the context of comparing two datasets, this mutual information is equivalent to the Jensen-Shannon divergence, which is closely related to relative entropy.

Intuitively, mutual information measures how much knowing the features tells us about which dataset a point came from, and vice versa. If the datasets are very different, knowing the features will give us a lot of information about which dataset a point belongs to, resulting in high mutual information.

The algorithm works by:

1. Combining X and Y into a single dataset with labels.
2. Discretizing the continuous data.
3. Computing the mutual information between each feature and the dataset labels.
4. Summing these mutual information values.

This method is particularly useful when you want to understand which features contribute most to the difference between the datasets, as you can look at the mutual information for each feature separately.

Usage

```
mutual.information(X, Y, num.bins = 10)
```

Arguments

`X`	A matrix or data frame representing the first dataset, where rows are observations and columns are features. Must have the same number of columns as Y.
`Y`	A matrix or data frame representing the second dataset, where rows are observations and columns are features. Must have the same number of columns as X.
`num.bins`	The number of bins to use for discretization (default is 10).

Details

The mutual information I(F;D) between features F and dataset label D is computed as:

$$I(F; D) = \sum_{f,d} p(f,d) \log \frac{p(f,d)}{p(f)p(d)}$$

The function uses the infotheo package's discretization method (equal frequency binning by default) to handle continuous features. The total mutual information is the sum across all features, which assumes feature independence given the dataset label.

Value

A numeric value representing the estimated mutual information in nats (natural units). Higher values indicate greater differences between the datasets. Returns 0 when the datasets have identical distributions.

Note

- Requires the 'infotheo' package to be installed
- The discretization step can lose information, especially for highly continuous data
- Results depend on the discretization method and number of bins used
- For high-dimensional data, consider using only the most informative features
- The assumption of summing MI across features may overestimate total information if features are correlated

References

Cover, T. M., & Thomas, J. A. (2006). Elements of information theory. John Wiley & Sons.

Lin, J. (1991). Divergence measures based on the Shannon entropy. IEEE Transactions on Information Theory, 37(1), 145-151.

See Also

`mutinformation` for the underlying MI computation, `discretize` for the discretization method, `total.variation.distance` for an alternative dataset comparison metric

Examples

```
# Example 1: Datasets with different means
set.seed(123)
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- mutual.information(X, Y)
print(paste("MI between datasets:", round(result, 4), "nats"))

# Example 2: Identical datasets (MI should be ~0)
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000), ncol = 2)
result <- mutual.information(X, Y)
print(paste("MI for identical distributions:", round(result, 4)))

# Example 3: Feature-wise contribution
X <- matrix(rnorm(500), ncol = 5)
Y <- X
Y[,3] <- Y[,3] + 2  # Only change feature 3
# Compute MI for each feature separately
## Not run:
feature_mi <- sapply(1:5, function(i) {
  mutual.information(X[,i,drop=FALSE], Y[,i,drop=FALSE])
})
barplot(feature_mi, names.arg = paste("Feature", 1:5),
        main = "Feature-wise MI Contribution")

## End(Not run)
```

`n.zeros` *Returns the number of times a numeric vector changes the sign between consecutive points*

Description

Returns the number of times a numeric vector changes the sign between consecutive points

Usage

```
n.zeros(x)
```

Arguments

`x`	A numeric vector

`nada.graph.spectral.lowess` *Non-adaptive Local Regression on Graphs Using Spectral Embedding*

Description

Performs local regression on graph data using spectral embeddings with adaptive bandwidth selection.

Usage

```
nada.graph.spectral.lowess(
  adj.list,
  weight.list,
  y,
  n.evectors = 5,
  n.bws = 20,
  log.grid = TRUE,
  min.bw.factor = 0.05,
  max.bw.factor = 0.33,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  precision = 0.001,
  n.cleveland.iterations = 0L,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list\[\[i\]\]` contains the indices of vertices adjacent to vertex i.

`weight.list`	A list of numeric vectors with edge weights corresponding to adjacencies. Each element `weight.list\[\[i\]\]\[j\]` is the weight of the edge from vertex i to `adj.list\[\[i\]\]\[j\]`.
`y`	A numeric vector of response values for each vertex in the graph.
`n.evectors`	Integer specifying the number of eigenvectors to use in the spectral embedding (default: 5).
`n.bws`	Integer specifying the number of candidate bandwidths to evaluate (default: 10).
`log.grid`	Logical indicating whether to use logarithmic spacing for bandwidth grid (default: TRUE).
`min.bw.factor`	Numeric value specifying the minimum bandwidth as a fraction of graph diameter (default: 0.05).
`max.bw.factor`	Numeric value specifying the maximum bandwidth as a fraction of graph diameter (default: 0.25).
`dist.normalization.factor`	Numeric factor for normalizing distances when calculating kernel weights (default: 1.0).
`kernel.type`	Integer specifying the kernel function for weighting vertices: • 1: Epanechnikov • 2: Triangular • 4: Laplace • 5: Normal • 6: Biweight • 7: Tricube (default) Default is 7.
`precision`	Numeric value specifying the precision tolerance for binary search and optimization algorithms (default: 0.001).
`n.cleveland.iterations`	Number of Cleveland's robustness iterations (default: 0)
`verbose`	Logical indicating whether to display progress information (default: FALSE).

Details

This function implements a graph-based extension of LOWESS (Locally Weighted Scatterplot Smoothing) that uses spectral embedding to transform graph distances into a Euclidean space suitable for local linear regression. For each vertex, the function:

- Finds all vertices within the maximum bandwidth radius
- Creates a local spectral embedding using graph Laplacian eigenvectors
- Fits weighted linear models at multiple candidate bandwidths
- Selects the optimal bandwidth based on leave-one-out cross-validation error
- Computes smoothed predictions using the optimal model

Value

A list containing:

- `predictions`: Numeric vector of smoothed values for each vertex
- `errors`: Numeric vector of leave-one-out cross-validation errors
- `scale`: Numeric vector of optimal bandwidths (local scales) for each vertex
- `graph.diameter`: Numeric scalar with computed graph diameter

Examples

```
## Not run:
# Create a simple graph with 100 vertices
n <- 100
set.seed(123)

# Create a ring graph
adj.list <- vector("list", n)
weight.list <- vector("list", n)

for (i in 1:n) {
  neighbors <- c(i-1, i+1)
  # Handle wrap-around for ring structure
  neighbors\[neighbors == 0\] <- n
  neighbors\[neighbors == n+1\] <- 1

  adj.list\[\[i\]\] <- neighbors
  weight.list\[\[i\]\] <- rep(1, length(neighbors))
}

# Generate response values with spatial pattern
y <- sin(2*pi*(1:n)/n) + rnorm(n, 0, 0.2)

# Apply spectral LOWESS
result <- nada.graph.spectral.lowess(
  adj.list = adj.list,
  weight.list = weight.list,
  y = y,
  n.evectors = 5,
  verbose = TRUE
)

# Plot results
plot(y, type="l", col="gray", main="Graph Spectral LOWESS")
lines(result$predictions, col="red", lwd=2)

## End(Not run)
```

`nerve.cx.spectral.filter`

Apply Spectral Filtering to a Nerve Complex

Description

Performs spectral filtering on a signal defined on the vertices of a nerve complex. This method extends graph spectral filtering to incorporate higher-order relationships captured by the simplicial structure of the nerve complex. The filtering uses a full Laplacian that combines information from all dimensions of simplices, weighted according to the specified dimension weights.

Usage

```
nerve.cx.spectral.filter(
    complex,
    y,
    laplacian.type = "STANDARD",
    filter.type = "HEAT",
    laplacian.power = 1,
    dim.weights = NULL,
    kernel.params = list(tau.factor = 0.01, radius.factor = 3, kernel.type = 0),
    n.evectors = 100,
    n.candidates = 100,
    log.grid = TRUE,
    with.t.predictions = FALSE,
    verbose = FALSE
)
```

Arguments

`complex`	A nerve complex object created by `create.nerve.complex`. Must contain a valid complex pointer and dimension information.
`y`	Numeric vector of function values at vertices of the complex. Length must equal the number of vertices in the complex.
`laplacian.type`	Character string specifying the type of Laplacian. One of: `"STANDARD"` Standard combinatorial Laplacian (default) `"NORMALIZED"` Normalized Laplacian `"RANDOM_WALK"` Random walk Laplacian `"SHIFTED"` Shifted Laplacian (I - L) `"REGULARIZED"` Regularized Laplacian (L + epsilon*I)
`filter.type`	Character string specifying the type of spectral filter. One of: `"HEAT"` Heat kernel filter $\exp(-t\lambda)$ (default) `"GAUSSIAN"` Gaussian filter $\exp(-t\lambda^2)$ `"NON_NEGATIVE"` Non-negative heat kernel filter `"CUBIC_SPLINE"` Cubic spline filter $1/(1+t\lambda^2)$ `"EXPONENTIAL"` Exponential filter `"MEXICAN_HAT"` Mexican hat wavelet filter `"IDEAL_LOW_PASS"` Ideal low-pass filter `"BUTTERWORTH"` Butterworth filter `"TIKHONOV"` Tikhonov filter `"POLYNOMIAL"` Polynomial filter `"INVERSE_COSINE"` Inverse cosine filter `"ADAPTIVE"` Adaptive filter
`laplacian.power`	Positive integer specifying the power to which the Laplacian is raised. Default is 1. Higher powers produce smoother results.
`dim.weights`	Numeric vector of non-negative weights for each dimension's contribution to the full Laplacian. Length should be at least max.dimension + 1. Default is a vector of 1s. The i-th element weights the i-1 dimensional simplices.
`kernel.params`	Named list of kernel parameters used by certain Laplacian types: `tau.factor` Factor for kernel bandwidth (0 < tau.factor <= 1) `radius.factor` Factor for neighborhood radius (>= 1)

	`kernel.type` Integer code for kernel type (0-8)
`n.evectors`	Positive integer specifying the number of eigenvectors to compute. Default is 100. Set to 0 to compute all eigenvectors (may be slow for large complexes).
`n.candidates`	Positive integer specifying the number of filter parameter values to evaluate for automatic selection via GCV. Default is 100.
`log.grid`	Logical indicating whether to use logarithmic spacing for the parameter grid (TRUE, default) or linear spacing (FALSE).
`with.t.predictions`	Logical indicating whether to return predictions for all tested parameter values (TRUE) or only the optimal value (FALSE, default).
`verbose`	Logical indicating whether to print progress information during computation. Default is FALSE.

Details

The function implements a comprehensive framework for spectral filtering on nerve complexes. The general process involves:

1. Construction of a weighted Laplacian operator incorporating all simplex dimensions
2. Eigendecomposition of the Laplacian
3. Application of the specified spectral filter in the frequency domain
4. Automatic parameter selection using Generalized Cross-Validation (GCV)

The dimension weights allow flexible control over how much each simplex dimension contributes to the overall smoothing. Setting higher weights for higher dimensions incorporates more global structure into the filtering.

Value

An object of class `nerve_cx_spectral_filter` containing:

`predictions`	Numeric vector of smoothed function values at the optimal parameter
`optimal_parameter`	The selected optimal filter parameter value
`gcv_score`	The GCV score at the optimal parameter
`all_parameters`	Vector of all tested parameter values
`all_gcv_scores`	Vector of GCV scores for each parameter
`compute_time_ms`	Computation time in milliseconds
`method`	List containing the input method parameters
`t_predictions`	(Optional) Matrix of predictions for all parameter values if `with.t.predictions = TRUE`

Parameter Selection

The optimal filter parameter is selected automatically using leave-one-out Generalized Cross-Validation (GCV). This avoids overfitting and typically produces good results without manual tuning.

See Also

`create.nerve.complex` for creating nerve complexes, `set.complex.function.values` for setting function values, `plot.nerve_cx_spectral_filter` for visualization

Examples

```
## Not run:
# Generate 2D points
set.seed(123)
coords <- matrix(runif(200), ncol = 2)

# Create a smooth function with noise
f_true <- function(x) sin(2*pi*x[1]) * cos(2*pi*x[2])
y_true <- apply(coords, 1, f_true)
y_noisy <- y_true + rnorm(length(y_true), 0, 0.2)

# Create nerve complex
complex <- create.nerve.complex(coords, k = 8, max.dim = 2)
complex <- set.complex.function.values(complex, y_noisy)

# Apply spectral filtering with default parameters
result <- nerve.cx.spectral.filter(complex, y_noisy)

# Apply with custom parameters emphasizing higher dimensions
result2 <- nerve.cx.spectral.filter(
  complex, y_noisy,
  laplacian.type = "NORMALIZED",
  filter.type = "GAUSSIAN",
  dim.weights = c(1.0, 0.5, 0.25),
  n.candidates = 150,
  verbose = TRUE
)

# Compare results
mse1 <- mean((result$predictions - y_true)^2)
mse2 <- mean((result2$predictions - y_true)^2)
cat("MSE (default):", mse1, "\n")
cat("MSE (custom):", mse2, "\n")

## End(Not run)
```

`nerve.graph` *Constructs the 1-skeleton (nerve graph) of the nerve simplicial complex associate with a covering of some finite set*

Description

This function constructs the 1-skeleton of the nerve of a given covering of a set. The covering is represented as a list of components, each being an integer index of the elements of the set. The function returns an adjacency list representing the graph. Parallel processing can be utilized by specifying the number of cores.

Usage

```
nerve.graph(covering.list, n.cores = 1)
```

Arguments

`covering.list`	A list where each component is an integer vector representing the indices of elements in a subset of the set being covered. The list represents a covering of the set.
`n.cores`	An integer specifying the number of cores for parallel processing. If NULL, it defaults to `detectCores() - 1`. If set to 1, parallel processing is not used. Based on limited experimental data, the parallel processing becomes a viable option when the length of the list is more than 100.

Details

The nerve of a covering is a simplicial complex where each set in the covering corresponds to a vertex, and vertices are connected by an edge if and only if the corresponding sets have a non-empty intersection. This function constructs the 1-skeleton (graph) of this nerve complex, with edge weights equal to the size of the intersection between sets.

Value

A list containing:

`adjacency.list`	The adjacency list representation of the nerve graph
`weights.list`	The weights (intersection sizes) for each edge
`adjacency.matrix`	The sparse adjacency matrix with intersection sizes as entries

Examples

```
covering.list <- list(c(1, 2), c(2, 3), c(1, 3, 4))
nerve.graph(covering.list, n.cores = 2)
```

---

nn.distance.ratio.estimator

Estimate Relative Entropy Using Nearest Neighbor Distance Ratio

---

Description

This function estimates the relative entropy (Kullback-Leibler divergence) between two datasets using the nearest neighbor distance ratio method. It includes safeguards against zero distances and potential numerical instabilities.

Usage

```
nn.distance.ratio.estimator(X, Y, k = 1, eps = NULL, eps.factor = 1e-08)
```

Arguments

X	A matrix or data frame representing the first dataset.
Y	A matrix or data frame representing the second dataset.
k	The number of nearest neighbors to consider (default is 1).
eps	A small constant to add to distances to avoid numerical issues (default is NULL, which means it will be automatically determined).
eps.factor	A small factor used to compute eps when eps is NULL. eps is set to the smallest non-zero distance multiplied by this factor (default is 1e-8).

Value

A numeric value representing the estimated relative entropy.

References

Wang, Q., Kulkarni, S. R., & Verdú, S. (2009). Divergence estimation for multidimensional densities via k-nearest-neighbor distances. IEEE Transactions on Information Theory, 55(5), 2392-2405.

Examples

```
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- nn.distance.ratio.estimator(X, Y)
print(result)
```

`non.NA.values` *Returns a matrix with two columns. The first column is the index of x where x[i] is not NA and the second column is the largest k such that x[i] == x[i + k].*

Description

Returns a matrix with two columns. The first column is the index of x where x[i] is not NA and the second column is the largest k such that x[i] == x[i + k].

Usage

```
non.NA.values(x)
```

Arguments

x	A numeric vector.

`non.zero.values` *Returns a matrix with two columns. The first column is the index of x where x[i] is not 0 and the second column is the largest k such that x[i] == x[i + k].*

Description

Returns a matrix with two columns. The first column is the index of x where x[i] is not 0 and the second column is the largest k such that x[i] == x[i + k].

Usage

```
non.zero.values(x)
```

Arguments

`x`	A numeric vector.

`normalize.and.inv.logit`
Normalize and Apply Inverse Logit Transformation

Description

First performs Min-Max normalization on a numeric vector to scale it within a specified range, and then applies the inverse logit function to each element.

Usage

```
normalize.and.inv.logit(x, y.min = -3, y.max = 3)
```

Arguments

`x`	A numeric vector to be transformed.
`y.min`	The minimum value of the range for Min-Max normalization (default is -3).
`y.max`	The maximum value of the range for Min-Max normalization (default is 3).

Details

The Min-Max normalization scales the data linearly between `y.min` and `y.max`. The inverse logit function (logistic function) then transforms these normalized values into probabilities, mapping any real-valued number into the range (0, 1).

Value

A numeric vector where each original value in x has been first normalized and then transformed via the inverse logit function.

Examples

```
## Not run:
x <- rnorm(10)
normalize.and.inv.logit(x)  # Normalizes x and applies inverse logit

## End(Not run)
```

`normalize.and.inv.logit.fn`
Transform Continuous Values to Probabilities via Normalization and Logistic Function

---

Description

This function converts continuous-valued input into probabilities through a two-step process:

1. Min-max normalization to a specified range
2. Logistic transformation to convert normalized values to probabilities

Usage

```
normalize.and.inv.logit.fn(y, y.min = -3, y.max = 3)
```

Arguments

`y`	Numeric vector of continuous values to be transformed. Must contain more than one unique finite value.
`y.min`	Numeric scalar indicating the minimum value for the normalization range. Must be less than y.max. Default is -3.
`y.max`	Numeric scalar indicating the maximum value for the normalization range. Must be greater than y.min. Default is 3.

Details

The transformation is useful for converting arbitrary continuous values (like smoothed data or similarity scores) into valid probabilities that can be used for binary sampling or probabilistic modeling.

The function first applies min-max normalization to scale the input values to the range `[y.min, y.max]`. It then applies the inverse logit (logistic) function `1/(1 + exp(-x))` to convert these normalized values to probabilities.

The default range `[-3, 3]` is chosen because these values, when transformed by the logistic function, result in probabilities of approximately 0.05 and 0.95 respectively, providing a good spread of probability values while avoiding extreme probabilities too close to 0 or 1.

Value

A numeric vector of the same length as y, containing probabilities in the range (0,1).

Throws an error if

- Input contains non-finite values
- Input contains fewer than two unique values
- y.min is greater than or equal to y.max

See Also

`rbinom` for generating binary samples from the resulting probabilities

Examples

```
# Transform smoothed values into probabilities
y.smooth <- c(-1, 0, 0.5, 1, 2)
probs <- normalize.and.inv.logit.fn(y.smooth)

# Use custom range for normalization
probs.wide <- normalize.and.inv.logit.fn(y.smooth, y.min = -5, y.max = 5)
```

`normalize.dist` *Normalizes a distance matrix*

Description

normalize.dist(d, min.K, bw) divides the rows of the input distance matrix, d, by 'bw' if the distance of the 'min.K'-th element of the given row, i, is <bw, if it is not, then all elements are divided by the distance of the min.K-st element. The radius, r, is bw in the first case and `d[i,minK]` in the second.

Usage

```
normalize.dist(d, min.K, bw)
```

Arguments

`d`	A numeric matrix.
`min.K`	The mininum _*number* of elements of each row of d after normalization with normalized distance < 1.
`bw`	A normalization non-negative constant.

Value

A list with two components: nd (normalized distances), r (radii).

ALERT: This routine makes sense only in the situation when ncol(d) > minK. Thus, the user has to make sure this condition is satisfied before calling normalize_dist().

`o.inv.fn` *Computes the inverse of a permutation vector generated by order()*

Description

Computes the inverse of a permutation vector generated by order()

Usage

```
o.inv.fn(o)
```

Arguments

`o`	A permutation vector generated by order().

Value

A numeric vector representing the inverse permutation, where element i contains the position of i in the original ordering vector o.

Examples

```
x <- c(3, 1, 4, 2)
o <- order(x)  # Returns c(2, 4, 1, 3)
o.inv <- o.inv.fn(o)  # Returns c(3, 1, 4, 2)
# Verify: x[o][o.inv] equals x
```

`overlap.coefficient` *Calculate the Overlap Coefficient Between Two Numeric Vectors*

Description

Computes the Overlap Coefficient (also known as Szymkiewicz-Simpson coefficient) between two numeric vectors treated as sets. The coefficient is defined as the size of the intersection divided by the size of the smaller set. Duplicates are removed before calculation.

Usage

```
overlap.coefficient(x, y)
```

Arguments

`x`	A numeric vector representing the first set
`y`	A numeric vector representing the second set

Value

A numeric value between 0 and 1, where:

- 1 indicates that one set is a subset of the other
- 0 indicates the sets are disjoint
- Values between 0 and 1 indicate partial overlap

References

Simpson, G.G. (1960) Notes on the measurement of faunal resemblance. American Journal of Science, 258-A: 300-311.

See Also

Other set similarity metrics: dist for general distance measures

Examples

```
overlap.coefficient(c(1, 2, 3), c(1, 2, 3))     # Returns 1
overlap.coefficient(c(1, 2), c(1, 2, 3, 4))     # Returns 1
overlap.coefficient(c(1, 2, 2, 3), c(1, 2, 4))  # Returns 0.67
overlap.coefficient(c(1, 2), c(3, 4))           # Returns 0
overlap.coefficient(numeric(0), c(1, 2))        # Returns 0
```

---

overlap.distribution.plot

Plot the distribution of overlap values

Description

Plot the distribution of overlap values

Usage

```
overlap.distribution.plot(x, radius_idx = NULL)
```

Arguments

x	A vertex_geodesic_stats object
radius_idx	Index of the radius to display. If NULL, uses the radius with the most composite geodesics.

Value

Invisibly returns NULL

`overlap_distance_matrix` *Compute Pairwise Overlap Distance Matrix*

Description

Compute Pairwise Overlap Distance Matrix

Usage

```
overlap_distance_matrix(basin_vertices_list)
```

Arguments

`basin_vertices_list`	A list of integer vectors.

Value

A symmetric matrix where entry $(i, j) = 1 - (|A \cap B| / \min(|A|, |B|))$, measuring the overlap distance between vectors i and j.

`p.cases` *Proportion of cases in a binary variable y*

Description

Proportion of cases in a binary variable y

Usage

```
p.cases(y)
```

Arguments

`y`	A binary variable with 0/1 values.

`parameterize.circular.graph`
Parameterize a Circular Graph Structure for Biological Cycle Analysis

Description

Applies spectral methods to parameterize a graph with a circular structure, calculating the position of each vertex along a circle using eigenvectors of the graph Laplacian. This function was motivated by the discovery that cell cycle genes form a circular structure in high-dimensional expression space, enabling cell cycle position assignment.

Usage

```
parameterize.circular.graph(adj.list, weight.list, use.edge.lengths = TRUE)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency structure. Each element `adj.list[[i]]` contains the indices of vertices adjacent to vertex `i`. Uses 1-based indexing (R convention). In biological applications, vertices might represent genes or cells with similar expression patterns.
`weight.list`	A list of numeric vectors containing edge weights. Each element `weight.list[[i]]` contains the weights for edges from vertex `i` to the vertices listed in `adj.list[[i]]`. Must have the same structure as `adj.list`. Weights can represent similarity or distance metrics between expression profiles.
`use.edge.lengths`	Logical. If `TRUE`, edge weights from `weight.list` are used in constructing the graph Laplacian. If `FALSE`, all edges are treated as having unit weight. Default is `TRUE`.

Details

This function embeds a graph with circular topology onto a circle using the second and third eigenvectors of the graph Laplacian matrix. The algorithm computes an angle for each vertex, positioning it on the unit circle in a way that preserves the graph structure.

The method is particularly useful for biological applications where the underlying process has circular dynamics, such as the cell cycle. In such cases, genes or cells can be ordered along a circle representing progression through the cyclic process. The parameterization allows for synthetic time-within-cycle assignment, where the computed angles represent positions along the circular trajectory.

The graph Laplacian is constructed as L = D - W, where D is the degree matrix and W is the weighted adjacency matrix. When `use.edge.lengths = TRUE`, edge weights are incorporated into the Laplacian construction. The method works best for graphs with approximately circular connectivity patterns and average vertex degree between 3 and 5.

Value

A list of class `"circular_parameterization"` containing:

`angles` Numeric vector of angles in radians (range $[0, 2\pi]$) for each vertex, representing their position on the unit circle. In cell cycle applications, these angles represent the inferred position within the cycle.

`eig_vec2` Numeric vector containing the second eigenvector of the graph Laplacian.

`eig_vec3` Numeric vector containing the third eigenvector of the graph Laplacian.

Note

The function internally converts R's 1-based indexing to 0-based indexing before calling the underlying C++ implementation.

References

Zheng, S. C., Stein-O'Brien, G., Augustin, J. J., Slosberg, J., Carosso, G. A., Winer, B., ... & Hansen, K. D. (2022). Universal prediction of cell-cycle position using transfer learning. Genome Biology, 23(1), 41. doi:10.1186/s1305902102581y

Examples

```
# Example 1: Simple circular graph (like cell cycle progression)
n <- 8
adj.list <- lapply(seq_len(n), function(i) {
  c(if (i == n) 1 else i + 1,  # next vertex
    if (i == 1) n else i - 1)  # previous vertex
})
weight.list <- lapply(adj.list, function(adj) rep(1.0, length(adj)))

# Get circular parameterization
result <- parameterize.circular.graph(adj.list, weight.list, TRUE)

# Display angles (positions along the cycle)
print(round(result$angles, 2))

# Example 2: Biological application - genes with circular expression pattern
# Simulate a gene similarity network where genes are connected if their
# expression patterns are similar (simplified example)
n_genes <- 12
# Create connections based on proximity in the cycle
adj.list <- lapply(seq_len(n_genes), function(i) {
  # Connect to 2 neighbors on each side to simulate local similarity
  neighbors <- c((i - 2):(i - 1), (i + 1):(i + 2))
  neighbors <- ((neighbors - 1) %% n_genes) + 1
  neighbors[neighbors != i]  # Remove self-connections
})
# Weights decrease with distance in the cycle
weight.list <- lapply(seq_len(n_genes), function(i) {
  neighbors <- adj.list[[i]]
  weights <- sapply(neighbors, function(j) {
    dist <- min(abs(i - j), n_genes - abs(i - j))
    exp(-dist/2)  # Exponential decay
  })
  weights
})

result <- parameterize.circular.graph(adj.list, weight.list)
```

```
# Plot genes positioned by their inferred cycle position
plot(cos(result$angles), sin(result$angles),
     xlim = c(-1.2, 1.2), ylim = c(-1.2, 1.2),
     pch = 19, xlab = "x", ylab = "y", asp = 1,
     main = "Gene Positions in Expression Cycle")

# Add gene labels
text(1.1 * cos(result$angles), 1.1 * sin(result$angles),
     labels = paste0("G", seq_len(n_genes)), cex = 0.8)

# The angles can be interpreted as positions in the biological cycle
# For cell cycle: 0 = G1/S, pi/2 = S, pi = G2, 3*pi/2 = M phase
```

partition.of.unity.1D *Creates a Partition of Unity in 1D*

Description

This function creates a partition of unity in one dimension using bump functions. It is designed to cover a specified interval with a set of overlapping bump functions, ensuring that the sum of these functions at any point in the interval equals one.

Usage

```
partition.of.unity.1D(
  x.center,
  x.min = 0,
  x.max = 10,
  n.grid = 400,
  C = 2,
  q = 3,
  compact.supp = FALSE
)
```

Arguments

`x.center`	A numeric vector specifying the center points of the bump functions.
`x.min`	The minimum value of the interval over which the partition is created (defaults to 0).
`x.max`	The maximum value of the interval over which the partition is created (defaults to 10).
`n.grid`	The number of points in the grid over the interval (defaults to 400).
`C`	A scaling factor to control the spread of the bump functions. Defaults to 2.
`q`	A q parameter in q-Gaussian
`compact.supp`	Set to TRUE to use bump functions with compact support.

Details

The function uses asymmetric bump functions at the ends of the interval and symmetric bump functions for internal points. The spread of each bump function is determined by the distance to its neighboring centers, scaled by `C`. The function checks for valid numeric input and requires at least two center points.

Value

A list with the following components

- `U`: A matrix with `n.grid` rows and `length(x.center)` columns, representing the values of the bump functions at each grid point. The sum of the column values at each row should be normalized to 1 to ensure a true partition of unity.
- `U.before`: U before normalization.
- `x.grid`: A uniform grid over the inverval $[x.min, x.max]$.
- `x.centers`: Sorted in the ascending order x.centers.
- `breaks`: Mid points between x.centers.

Examples

```
## Not run:
x.center <- seq(1, 9, by = 2)
partition <- partition.of.unity.1D(x.center)
plot(seq(0, 10, length.out = 400), partition[,1], type = "l")

## End(Not run)
```

`partition.of.unity.xD` *Creates a Partition of Unity in dimensions greater than 1*

Description

This function creates a partition of unity (PoU) over a subset S of \(R^d)\), where d is the number of columns of S. The components of the partition of unity are constructed from q-exponential Gaussian radial functions. If a matrix or data frame, X.centers, of the centers of the components of the PoU, the centers of the compoenents will be places at these points. If X.centers is NULL, then the n.comp centers will be selected from the points of S.

Usage

```
partition.of.unity.xD(S, X.centers = NULL, n.comp = 3, C = 2, q = 3)
```

Arguments

`S`	A matrix or data frame of points over which the partition of unity will be evaluated.
`X.centers`	A matrix or data frame of the centers of the components of the PoU.
`n.comp`	The number of components of the PoU. Used when X.centers is NULL. In this case a random set of n.comp rows of S is taken to be X.centers.
`C`	A scaling factor to control the spread of the bump functions. Defaults to 2.
`q`	A q parameter in radial q-exponential Gaussian functions. Controls the steepness of the radial Gaussian function.

Value

A list with three components:

- `U`: A matrix of the values of each component of the PoU over the points of S. The return matrix has nrow(S) rows and n.comp columns.
- `nn.d`: A vector of distances to the nearest neighbor center.
- `nn.i`: A vector of indices of the nearest neighbor.

`path.dist` *Distances from the first vertex of a path to each consecutive vertex along the given path normalized so that the total distance is 1.*

Description

Distances from the first vertex of a path to each consecutive vertex along the given path normalized so that the total distance is 1.

Usage

```
path.dist(s, V, edge.col = "gray")
```

Arguments

`s`	A sequences of vertex indices of the graph - the path.
`V`	A vertex positions matrix.
`edge.col`	A color of the edges.

Details

This function calculates the cumulative Euclidean distances along a path defined by a sequence of vertex indices. The distances are then normalized by the total path length so that the result represents relative positions along the path.

Value

A numeric vector of length equal to the path length, containing the cumulative distances from the first vertex to each vertex along the path, normalized to the range $[0, 1]$. The first element is always 0 and the last element is always 1.

Note

The parameter edge.col is currently not used in the function implementation.

Examples

```
# Create a simple vertex matrix
V <- matrix(c(0,0, 1,0, 1,1, 0,1), ncol=2, byrow=TRUE)
path <- c(1, 2, 3, 4)
distances <- path.dist(path, V)
# distances will be c(0.0, 0.333..., 0.666..., 1.0)
```

path.fn — *Animate Path Along Trajectory*

Description

Function for animation of a smoothed trajectory

Usage

```
path.fn(
  i,
  Epath,
  radius = 8e-04,
  sphere.col = "red",
  path.col = "blue",
  path.lwd = 5
)
```

Arguments

i	Index in the Epath matrix.
Epath	Matrix of smoothed trajectory positions.
radius	Radius of the animated sphere.
sphere.col	Color of the animated sphere.
path.col	Color of the path trail.
path.lwd	Width of the path trail.

Details

This function is designed to be used in animation loops. It draws a sphere at the current position and adds a line segment from the previous position. Note: This function assumes 'sphere.id' exists in the calling environment.

Value

Invisibly returns NULL.

Examples

```
## Not run:
# This function is typically used within an animation loop
Epath <- matrix(rnorm(30), ncol = 3)
sphere.id <- NULL
for (i in 2:nrow(Epath)) {
  path.fn(i, Epath)
  Sys.sleep(0.1)
}

## End(Not run)
```

`path.length` *Computes the length of a path specified by a matrix X.*

Description

Computes the length of a path specified by a matrix X.

Usage

```
path.length(X)
```

Arguments

`X`	A matrix of points along a path.

Details

This function calculates the total path length by summing the Euclidean distances between consecutive points (rows) in the matrix X. Each row of X represents a point in the path, and columns represent dimensions.

Value

A numeric value representing the total Euclidean length of the path defined by consecutive points in X.

Examples

```
# Create a simple path in 2D
path_points <- matrix(c(0,0, 1,0, 1,1, 0,1), ncol=2, byrow=TRUE)
length <- path.length(path_points)
# length will be 3 (three unit-length segments)

# 3D path example
path_3d <- matrix(c(0,0,0, 1,0,0, 1,1,0, 1,1,1), ncol=3, byrow=TRUE)
length_3d <- path.length(path_3d)
```

`pca.variance.analysis` *Perform PCA Variance Analysis*

Description

This function performs Principal Component Analysis (PCA) on the input data and analyzes the variance explained by different numbers of principal components.

Usage

```
pca.variance.analysis(
  X,
  pc.90 = 0.9,
  pc.95 = 0.95,
  pc.97.5 = 0.975,
  pc.99 = 0.99,
  max.components = 300
)
```

Arguments

`X`	A numeric matrix or data frame with at least two columns.
`pc.90`	Threshold for explaining 90% of variance (default: 0.90).
`pc.95`	Threshold for explaining 95% of variance (default: 0.95).
`pc.97.5`	Threshold for explaining 97.5% of variance (default: 0.975).
`pc.99`	Threshold for explaining 99% of variance (default: 0.99).
`max.components`	Maximum number of components to analyze (default: 300).

Value

A list containing:

`variance_explained`	Vector of cumulative variance explained
`pc_90`	Number of PCs explaining >90% variance
`pc_95`	Number of PCs explaining >95% variance
`pc_97_5`	Number of PCs explaining >97.5% variance
`pc_99`	Number of PCs explaining >99% variance
`X_pca90`	PCA projection for 90% variance threshold
`X_pca95`	PCA projection for 95% variance threshold
`X_pca97_5`	PCA projection for 97.5% variance threshold
`X_pca99`	PCA projection for 99% variance threshold
`auc`	Area under the curve of variance explained
`pca_result`	Full PCA result object

Examples

```
data <- matrix(rnorm(1000*50), ncol=50)
result <- pca.variance.analysis(data)
print(result$pc.95)
```

`pearson.wcor` *Pearson Weighted Correlation Coefficient*

Description

Computes the Pearson correlation coefficient between two numeric vectors with weights.

Usage

```
pearson.wcor(x, y, w)
```

Arguments

x	A numeric vector.
y	A numeric vector of the same length as x.
w	A numeric vector of non-negative weights of the same length as x.

Details

This function computes the weighted Pearson correlation coefficient using C code for efficiency. The weights must be non-negative and the three vectors must have the same length.

Value

A single numeric value representing the weighted Pearson correlation coefficient.

Examples

```
x <- rnorm(100)
y <- rnorm(100)
w <- runif(100)
pearson.wcor(x, y, w)
```

`perform.harmonic.smoothing` *Perform Harmonic Smoothing on Graph Function Values*

Description

Applies harmonic smoothing to function values defined on vertices of a graph, preserving values at the boundary of a specified region while smoothly interpolating interior values. This function implements a discrete Laplace equation solution using weighted averaging.

Usage

```
perform.harmonic.smoothing(
  adj.list,
  weight.list,
  values,
  region.vertices,
  max.iterations = 100,
  tolerance = 1e-06
)
```

Arguments

`adj.list`	A list of integer vectors, where each vector contains indices of vertices adjacent to the corresponding vertex. Indices must be 1-based.
`weight.list`	A list of numeric vectors containing weights of edges corresponding to adjacencies in `adj.list`.
`values`	A numeric vector of function values defined at each vertex.
`region.vertices`	An integer vector of vertex indices (1-based) defining the region to be smoothed. Boundary vertices will have fixed values.
`max.iterations`	Integer scalar, the maximum number of relaxation iterations to perform. Default is 100.
`tolerance`	Numeric scalar, the convergence threshold for value changes. Default is 1e-6.

Details

Harmonic smoothing preserves the overall shape of a function defined on a graph while removing local fluctuations. It works by iteratively updating interior vertex values as weighted averages of their neighbors until convergence, while keeping boundary values fixed.

The algorithm:

1. Identifies boundary vertices (vertices with neighbors outside the region or degree 1 vertices)
2. Iteratively updates interior vertex values using edge-weighted averaging
3. Continues until convergence or maximum iterations reached

Edge weights are incorporated by using their inverse as weighting factors, respecting the geometric structure of the graph.

Value

A numeric vector of the same length as `values`, with smoothed values within the specified region.

See Also

`harmonic.smoother` for smoothing with topology tracking, `get.region.boundary` for boundary vertex identification

Examples

```
## Not run:
# Create a simple grid graph
grid.graph <- create.graph.from.grid(10, 10)

# Create noisy function values
values <- sin(0.1 * seq_len(100)) + rnorm(100, 0, 0.1)

# Define a region for smoothing (center of the grid)
region <- 35:65

# Apply harmonic smoothing
smoothed.values <- perform.harmonic.smoothing(
  grid.graph$adj.list,
  grid.graph$weight.list,
  values,
  region,
  max.iterations = 200,
  tolerance = 1e-8
)

# Plot original vs smoothed values
plot(values, type = "l", col = "gray")
lines(smoothed.values, col = "red")

## End(Not run)
```

persistent.values *Returns a matrix with two columns. The first column is the index of x where* `x[i] = x[i+1]` *and the second column is the largest k such that* `x[i] = x[i + k]`.

Description

Returns a matrix with two columns. The first column is the index of x where `x[i] = x[i+1]` and the second column is the largest k such that `x[i] = x[i + k]`.

Usage

```
persistent.values(x)
```

Arguments

x	A numeric vector.

`pgmalo` *Path Graph Model Averaging Local Linear Model*

Description

Fits a path graph model averaging local linear model using cross-validation to select optimal neighborhood sizes. The function implements adaptive bandwidth selection for graph-structured data with optional bootstrap confidence intervals.

Usage

```
pgmalo(
  neighbors,
  edge_lengths,
  y,
  y.true = NULL,
  use.median = TRUE,
  h.min = 4L,
  h.max = min(30L, length(y) - 2L),
  p = 0.95,
  n.bb = 50L,
  bb.max.distance.deviation = 1L,
  n.CVs = 100L,
  n.CV.folds = 10L,
  seed = 0L,
  kernel.type = 7L,
  dist.normalization.factor = 1.1,
  epsilon = 1e-15,
  verbose = FALSE
)
```

Arguments

`neighbors`	List of integer vectors containing adjacency lists for each vertex. Each element should contain valid vertex indices (1 to n).
`edge_lengths`	List of numeric vectors containing positive edge lengths corresponding to each adjacency. Must have same structure as neighbors.
`y`	Numeric vector of response variables for each vertex.
`y.true`	Optional numeric vector of true response values for computing prediction errors. Must have same length as y.
`use.median`	Logical; if TRUE uses median instead of mean for bootstrap interval estimation (default: TRUE).
`h.min`	Integer; minimum neighborhood size to consider. Must be even and at least 4 (default: 4).
`h.max`	Integer; maximum neighborhood size to consider. Must be even and less than n-2 where n is the number of vertices (default: min(30, n-2)).
`p`	Numeric; confidence level for bootstrap intervals. Must be between 0 and 1 (default: 0.95).
`n.bb`	Integer; number of bootstrap iterations. Set to 0 to skip bootstrap (default: 50).

`bb.max.distance.deviation`	Integer; maximum distance deviation allowed in bootstrap samples. Must be >= -1 (default: 1).
`n.CVs`	Integer; number of cross-validation iterations (default: 100).
`n.CV.folds`	Integer; number of cross-validation folds. Must be between 2 and n (default: 10).
`seed`	Integer; random seed for reproducibility (default: 0).
`kernel.type`	Integer between 1 and 10 specifying the kernel function: • 1 = Gaussian • 2 = Epanechnikov • 3 = Quartic • 4 = Triweight • 5 = Tricubic • 6 = Cosine • 7 = Uniform (default) • 8-10 = Additional kernels (see documentation)
`dist.normalization.factor`	Numeric; factor for normalizing distances. Must be greater than 1 (default: 1.1).
`epsilon`	Numeric; numerical stability threshold. Must be positive (default: 1e-15).
`verbose`	Logical; if TRUE prints progress messages (default: FALSE).

Details

The Path Graph Model Averaging Local Linear (PGMALO) method performs local linear regression on graph-structured data. For each vertex, it considers neighborhoods of different sizes and uses cross-validation to select the optimal neighborhood size globally or locally.

The method accounts for the graph structure through path distances and applies kernel weighting based on these distances. Bootstrap sampling can be used to quantify prediction uncertainty.

Value

An S3 object of class "pgmalo" containing:

h_values Integer vector of tested neighborhood sizes

opt_h Optimal neighborhood size selected by cross-validation

opt_h_idx Index of optimal h in h_values vector

h_cv_errors Numeric vector of cross-validation errors for each h

true_errors Numeric vector of true prediction errors (if y.true provided)

predictions Numeric vector of predictions using optimal h

local_predictions Numeric vector of locally adaptive predictions

h_predictions List of prediction vectors for each h value

bb_predictions Matrix of bootstrap predictions (n.bb x n)

ci_lower Numeric vector of lower confidence bounds

ci_upper Numeric vector of upper confidence bounds

has_bootstrap Logical indicating if bootstrap was performed

call The matched call

References

Goldstein, L., Chu, B., Nayak, S., and Minsker, S. (2024). "Path Graph Model Averaging Local Linear Estimation." *Journal of Statistical Software* (forthcoming).

See Also

upgmalo for univariate version, plot.pgmalo for visualization

Examples

```
# Create a simple chain graph
n <- 50
neighbors <- vector("list", n)
edge_lengths <- vector("list", n)

# Build chain structure
for(i in 1:n) {
  if(i == 1) {
    neighbors[[i]] <- 2L
    edge_lengths[[i]] <- 1.0
  } else if(i == n) {
    neighbors[[i]] <- (n-1L)
    edge_lengths[[i]] <- 1.0
  } else {
    neighbors[[i]] <- c(i-1L, i+1L)
    edge_lengths[[i]] <- c(1.0, 1.0)
  }
}

# Generate smooth response with noise
x_pos <- seq(0, 1, length.out = n)
y <- sin(2 * pi * x_pos) + rnorm(n, 0, 0.1)

# Fit model with small h range for speed
  fit <- pgmalo(neighbors, edge_lengths, y, h.max = 10, n.CVs = 10)
  summary(fit)
  print(fit)
```

---

plaplace *Laplace Distribution Function*

Description

Calculates the cumulative distribution function for the Laplace distribution.

Usage

```
plaplace(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`q`	Vector of quantiles.
`location`	The location parameter μ. Default is 0.
`scale`	The scale parameter b. Must be positive. Default is 1.
`lower.tail`	Logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
`log.p`	Logical; if TRUE, probabilities p are given as $\log(p)$. Default is FALSE.

Value

A vector of cumulative probabilities.

Examples

```
x <- seq(-5, 5, by = 0.1)
y <- plaplace(x, location = 0, scale = 1)
plot(x, y, type = "l", main = "Laplace CDF")
```

`plot.assoc0` *Plot Method for assoc0 Objects*

Description

Creates diagnostic plots for objects of class "assoc0" produced by fassoc0.test().

Usage

```
## S3 method for class 'assoc0'
plot(
  x,
  plot = c("Exy", "d1hist", "bc.d1hist"),
  xlab = "x",
  ylab = "y",
  Eyg.col = "blue",
  Eyg.lwd = 2,
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.line.col = "gray70",
  null.CrI.col = "gray90",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  dEy.CrI.col = "cornflowerblue",
  d1hist.x = NULL,
  d1hist.y = NULL,
  title = "",
  ylim = NULL,
  ...
)
```

Arguments

`x`	An object of class "assoc0" from fassoc0.test().
`plot`	Character string specifying plot type. Options are: **"Exy"** Conditional mean function with credible intervals **"d1hist"** Histogram of null distribution with observed value **"bc.d1hist"** Histogram of Box-Cox transformed null distribution
`xlab`	Character string for x-axis label. Default is "x".
`ylab`	Character string for y-axis label. Default is "y".
`Eyg.col`	Color specification for conditional mean curve. Default is "blue".
`Eyg.lwd`	Line width for conditional mean curve. Default is 2.
`pt.col`	Color specification for data points. Default is "black".
`pt.pch`	Integer or character specifying point type. Default is 1.
`CrI.as.polygon`	Logical indicating whether to draw credible intervals as polygons (TRUE) or lines (FALSE). Default is TRUE.
`CrI.polygon.col`	Color specification for credible interval polygon. Default is "gray90".
`null.line.col`	Color specification for null hypothesis line. Default is "gray70".
`null.CrI.col`	Color specification for null distribution credible intervals. Default is "gray90".
`CrI.line.col`	Color specification for credible interval lines. Default is "gray".
`CrI.line.lty`	Line type for credible interval lines. Default is 2.
`dEy.CrI.col`	Color specification for conditional mean credible intervals. Default is "cornflowerblue".
`d1hist.x`	Numeric x-coordinate for p-value label in histogram plots. If NULL, automatically positioned.
`d1hist.y`	Numeric y-coordinate for p-value label in histogram plots. If NULL, automatically positioned.
`title`	Character string for plot title. Default is "".
`ylim`	Numeric vector of length 2 giving y-axis limits. If NULL, automatically determined.
`...`	Additional arguments (currently ignored).

Value

Invisible NULL. Called for side effect of creating plot.

Examples

```
## Not run:
# Generate example data and run test
set.seed(123)
n <- 200
x <- runif(n)
y <- sin(2*pi*x) + rnorm(n, sd = 0.3)
result <- fassoc0.test(x, y, n.cores = 2, n.perms = 1000, plot.it = FALSE)

# Create different plots
plot(result, plot = "Exy")
plot(result, plot = "d1hist")
plot(result, plot = "bc.d1hist")

## End(Not run)
```

`plot.assoc1` *Plot Method for assoc1 Objects*

Description

Creates diagnostic plots for objects of class "assoc1" produced by fassoc1.test().

Usage

```
## S3 method for class 'assoc1'
plot(
  x,
  plot = c("Exy", "dExy", "d1hist", "bc.d1hist"),
  xlab = "x",
  ylab = "y",
  x1.lab = "null.fassoc1",
  x2.lab = "BB.fassoc1",
  show.pval = TRUE,
  show.vlines = FALSE,
  Eyg.col = "blue",
  Eyg.lwd = 2,
  pt.col = "black",
  pt.pch = 1,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray90",
  null.line.col = "gray70",
  null.CrI.col = "gray90",
  CrI.line.col = "gray",
  CrI.line.lty = 2,
  dEy.CrI.col = "cornflowerblue",
  d1hist.x = NULL,
  d1hist.y = NULL,
  title = "",
  ylim = NULL,
  ...
)
```

Arguments

`x`	An object of class "assoc1" from fassoc1.test().
`plot`	Character string specifying plot type. Options are:
	"Exy" Conditional mean function with credible intervals
	"dExy" Derivative of conditional mean with credible intervals
	"d1hist" Histogram of null distribution with observed value

	"bc.d1hist" Histogram of Box-Cox transformed null distribution
`xlab`	Character string for x-axis label. Default is "x".
`ylab`	Character string for y-axis label. Default is "y".
`x1.lab`	Character string for first histogram label. Default is "null.fassoc1".
`x2.lab`	Character string for second histogram label. Default is "BB.fassoc1".
`show.pval`	Logical indicating whether to show p-value. Default is TRUE.
`show.vlines`	Logical indicating whether to show vertical reference lines. Default is FALSE.
`Eyg.col`	Color specification for conditional mean curve. Default is "blue".
`Eyg.lwd`	Line width for conditional mean curve. Default is 2.
`pt.col`	Color specification for data points. Default is "black".
`pt.pch`	Integer or character specifying point type. Default is 1.
`CrI.as.polygon`	Logical indicating whether to draw credible intervals as polygons (TRUE) or lines (FALSE). Default is TRUE.
`CrI.polygon.col`	Color specification for credible interval polygon. Default is "gray90".
`null.line.col`	Color specification for null hypothesis line. Default is "gray70".
`null.CrI.col`	Color specification for null distribution credible intervals. Default is "gray90".
`CrI.line.col`	Color specification for credible interval lines. Default is "gray".
`CrI.line.lty`	Line type for credible interval lines. Default is 2.
`dEy.CrI.col`	Color specification for derivative credible intervals. Default is "cornflowerblue".
`d1hist.x`	Numeric x-coordinate for p-value label in histogram plots. If NULL, automatically positioned.
`d1hist.y`	Numeric y-coordinate for p-value label in histogram plots. If NULL, automatically positioned.
`title`	Character string for plot title. Default is "".
`ylim`	Numeric vector of length 2 giving y-axis limits. If NULL, automatically determined.
`...`	Additional arguments (currently ignored).

Value

Invisible NULL. Called for side effect of creating plot.

Examples

```
## Not run:
# Generate example data
set.seed(123)
n <- 200
x <- runif(n)
y <- sin(4*pi*x) + rnorm(n, sd = 0.2)
result <- fassoc1.test(x, y, n.cores = 2, n.perms = 500, plot.it = FALSE)

# Create different plots
plot(result, plot = "Exy")
plot(result, plot = "dExy")
plot(result, plot = "d1hist")

## End(Not run)
```

`plot.basin_cx` *Plot method for basin_cx objects*

Description

Plot method for basin_cx objects

Usage

```
## S3 method for class 'basin_cx'
plot(x, y, ..., type = c("graph", "basins", "cells", "comparison"))
```

Arguments

`x`	A basin_cx object
`y`	Not used
`...`	Additional arguments passed to specific plotting functions
`type`	Type of plot: "graph" (default), "basins", "cells", or "comparison"

`plot.box.tiling` *Plots boxes of a box tiling of a 2D state space.*

Description

This function creates a plot showing the edges of boxes of the given box tiling and their indices in the center of each box. It is designed to work with the output of the `box.tiling` function.

Usage

```
## S3 method for class 'box.tiling'
plot(
  x,
  L = NULL,
  R = NULL,
  with.box.ids = TRUE,
  BBox.col = "red",
  col = "gray",
  xlab = "",
  ylab = "",
  sleep = 0,
  ...
)
```

Arguments

`x`	A list of sub-boxes, as returned by `create.ED.boxes`.
`L`	The lower left vertex of the bounding box whose grid is plotted. Default is NULL.
`R`	The upper right vertex of the bounding box whose grid is plotted. Default is NULL.
`with.box.ids`	Set to TRUE if box IDs are to be displayed.
`BBox.col`	A color to be used for the edges of bounding box whose grid is plotted. Default is "red".
`col`	A color to be used for the edges of sub-boxes. Default is "gray".
`xlab`	Label for x-axis. Default is "".
`ylab`	Label for y-axis. Default is "".
`sleep`	The number of seconds to sleep before drawing the next box.
`...`	Additional graphical parameters to be passed to the plotting functions.

Examples

```
## Not run:
w <- 1
L <- c(0, 0)
R <- c(2, 3)
grid_boxes <- create.ED.boxes(w, L, R)
plot.boxes(grid_boxes, L, R)

## End(Not run)
```

`plot.chain.with.path` *Plot a Chain Graph with Optional Highlighted Paths*

Description

Creates a visualization of a chain graph where vertices are arranged horizontally. The function can highlight specific paths and vertices based on path data, and displays edge weights when they differ from 1.

Usage

```
## S3 method for class 'chain.with.path'
plot(
  x,
  vertex.size = 0.5,
  margin = 0.5,
  title = "Chain Graph with Highlighted Path",
  y.offset = 0.1,
  ...
)
```

Arguments

`x`	An object of class "chain.with.path" containing the following elements: • `adj.list`: List of adjacency lists for each vertex, where each element contains the indices of neighboring vertices • `weight.list`: List of corresponding weights for each adjacency, matching the structure of adj.list • `gpd.obj`: List of path data objects, typically output from a get.path.data function. Each object should contain a 'vertices' element listing vertex indices in the path. The first object should have a 'ref_vertex' element indicating a reference vertex.
`vertex.size`	Numeric value for the size of vertex points (default: 0.5)
`margin`	Numeric value for the margin around the plot (default: 0.5)
`title`	Character string for the plot title (default: "Chain Graph with Highlighted Path")
`y.offset`	Numeric value for vertical offset of vertex labels (default: 0.1)
`...`	Additional arguments (currently unused)

Details

The graph is plotted with vertices arranged horizontally at equal intervals. Edge weights are displayed above the edges when they differ from 1. Vertices in the paths specified by gpd.obj are highlighted with red circles, and the reference vertex is highlighted with a larger blue circle.

Value

No return value, called for side effects (plotting)

Examples

```
# Create a simple chain graph
adj_list <- list(
  c(2),    # Vertex 1 connected to 2
  c(1, 3), # Vertex 2 connected to 1 and 3
  c(2)     # Vertex 3 connected to 2
)
weight_list <- list(
  c(1),    # Weight for edge 1-2
  c(1, 2), # Weights for edges 2-1 and 2-3
  c(2)     # Weight for edge 3-2
)
path_data <- list(
  list(vertices = c(1, 2, 3), ref_vertex = 1)
)

chain.with.path.obj <- chain.with.path(adj_list, weight_list, path_data)
plot(chain.with.path.obj)
```

plot.circular_parameterization

Plot Method for circular_parameterization Objects

Description

Creates a visualization of vertices positioned on a circle according to their computed angles.

Usage

```
## S3 method for class 'circular_parameterization'
plot(
  x,
  adj.list = NULL,
  vertex.labels = seq_len(length(x$angles)),
  vertex.cex = 1.5,
  label.cex = 0.8,
  edge.col = "gray70",
  vertex.col = "black",
  ...
)
```

Arguments

`x`	An object of class `"circular_parameterization"` as returned by `parameterize.circular.graph`
`adj.list`	Optional adjacency list to draw edges. If provided, edges will be drawn between connected vertices.
`vertex.labels`	Labels for vertices. Default is `seq_len(n)` where `n` is the number of vertices. Use `NA` to suppress labels.
`vertex.cex`	Character expansion factor for vertex points. Default is 1.5.
`label.cex`	Character expansion factor for vertex labels. Default is 0.8.
`edge.col`	Color for edges. Default is "gray70".
`vertex.col`	Color for vertex points. Default is "black".
`...`	Further graphical parameters passed to `plot`.

Value

Invisibly returns the input object `x`.

Examples

```
# Create example graph
n <- 8
adj.list <- lapply(seq_len(n), function(i) {
  c(if (i == n) 1 else i + 1, if (i == 1) n else i - 1)
})
weight.list <- lapply(adj.list, function(adj) rep(1.0, length(adj)))
result <- parameterize.circular.graph(adj.list, weight.list)

# Basic plot
```

```
plot(result)

# Plot with edges
plot(result, adj.list = adj.list,
     main = "Circular Graph Visualization")
```

---

plot.gaussian_mixture *Plot method for gaussian_mixture objects*

Description

Creates a 2x2 layout with contour plot, 3D surface, heat map, and filled contours.

Usage

```
## S3 method for class 'gaussian_mixture'
plot(x, n.grid = 50, main = "2D Gaussian Mixture", ...)
```

Arguments

`x`	A gaussian_mixture object
`n.grid`	Number of grid points in each dimension. Default is 50.
`main`	Main title for the plots. Default is "2D Gaussian Mixture"
`...`	Additional arguments (currently unused)

Value

Invisible NULL

---

plot.gaussian_mixture_data *Plot Method for Gaussian Mixture Data Objects*

Description

Provides various visualization options for data sampled from Gaussian mixture models. Designed to work with objects created by `sample.gaussian_mixture`.

Usage

```
## S3 method for class 'gaussian_mixture_data'
plot(
  x,
 type = c("scatter2d", "scatter3d", "surface", "surface3d", "contour", "heatmap",
    "components", "nerve", "compare"),
  nerve.complex = NULL,
  y.estimate = NULL,
  grid_size = 50,
  knn_edges = FALSE,
  k_neighbors = 5,
  ...
)
```

Arguments

`x`	A gaussian_mixture_data object created by `sample.gaussian_mixture`
`type`	Character string specifying the type of plot: `"scatter2d"` 2D scatter plot colored by function values (default) `"scatter3d"` 3D scatter plot showing (x, y, z) points `"surface"` Interactive 3D surface with data points (requires `rgl`) `"surface3d"` Static 3D surface using base graphics `"contour"` Contour plot of the underlying mixture function `"heatmap"` Heatmap visualization of the mixture function `"components"` Individual Gaussian components and full mixture `"nerve"` 2D scatter with nerve complex overlay (requires `nerve.complex`) `"compare"` Compare true values, noisy data, and estimates
`nerve.complex`	Optional nerve complex object. Required only for `type = "nerve"`.
`y.estimate`	Optional vector of estimated `y` values for comparison plots. Must have same length as data points. Used with `type = "compare"`.
`grid_size`	Integer specifying grid resolution for function plots (contour, heatmap, surface). Default is 50.
`knn_edges`	Logical. For `type = "scatter3d"`, whether to create k-nearest neighbor edges between points. If `TRUE`, connects each point to its k nearest neighbors, creating a graph structure that can help visualize local data relationships. If `FALSE` (default), points are displayed without any connecting edges. Requires the `FNN` package when `TRUE`.
`k_neighbors`	Integer. For `type = "scatter3d"` with `knn_edges = TRUE`, specifies the number of nearest neighbors to connect for each point. Default is 5. The actual number used will be the minimum of this value and n-1, where n is the number of data points. Larger values create denser graphs that may obscure the 3D structure, while smaller values show only the most local connections.
`...`	Additional arguments passed to plotting functions

Details

This method visualizes data sampled from Gaussian mixture models. The input object must contain:

- `X`: Matrix of sampled points (n x 2)
- `y`: Vector of function values (possibly with noise)
- `y.true`: Vector of true function values (optional)
- `mixture`: The underlying gaussian_mixture object

The plotting domain is automatically determined from the mixture object's `x.range` and `y.range`.

Value

Invisibly returns the gaussian_mixture_data object

Note

The "components" plot type requires that the mixture object contains component information (centers, amplitudes, covariances). The "compare" plot type works best when y.true is available in the data object.

See Also

sample.gaussian_mixture for creating data objects, get.gaussian.mixture.2d for creating 2D mixtures

Examples

```
## Not run:
# Create a 2D Gaussian mixture
gm <- get.gaussian.mixture.2d(n.components = 3, sd.value = 0.1)

# Sample points from the mixture
data <- sample(gm, n = 500, sampling.method = "random", noise.sd = 0.05)

# Basic 2D scatter plot
plot(data)

# Contour plot of underlying function
plot(data, type = "contour", nlevels = 20)

# View individual components
plot(data, type = "components")

# Compare true vs noisy values
plot(data, type = "compare")

# 3D visualization (requires rgl)
plot(data, type = "surface", grid_size = 75)

## End(Not run)
```

---

plot.geodesic_stats *Plot Geodesic Statistics*

Description

Creates visualizations of geodesic statistics to help understand the relationship between radius and number of geodesics.

Usage

```
## S3 method for class 'geodesic_stats'
plot(
  x,
  plot.type = c("summary", "heatmap", "vertex", "all"),
  selected.vertices = NULL,
  max.vertices = 10,
    ...
)
```

Arguments

`x`	List. Output from compute.geodesic.stats().
`plot.type`	Character. Type of plot to generate: "summary" (default), "heatmap", "vertex", or "all".
`selected.vertices`	Integer vector. Specific vertices to highlight in "vertex" plots. If NULL, a random sample will be used.
`max.vertices`	Integer. Maximum number of vertices to show in vertex plot.
`...`	Additional arguments passed to summary

Value

Invisibly returns NULL.

Examples

```
## Not run:
stats <- compute.geodesic.stats(adj.list, weight.list)
plot.geodesic.stats(stats, "all")

## End(Not run)
```

`plot.ggraph` *Plot method for ggraph objects*

Description

Plot method for ggraph objects

Usage

```
## S3 method for class 'ggraph'
plot(
  x,
  y = NULL,
  dim = 2,
  vertex.size = 1,
  vertex.radius = 0.1,
  vertex.label = NA,
  layout = "kk",
  vertex.color = NULL,
  draw.edges = TRUE,
  legend.cex = 1,
  legend.position = "topleft",
  quantize.method = "uniform",
  label.cex = 1.2,
  label.adj = c(0.5, 0.5),
  use.saved.layout = NULL,
  ...
)
```

Arguments

x	A ggraph object created by ggraph()
y	A function over vertices of the graph (optional)
dim	The embedding dimension. Possible values 2 or 3. Default 2.
vertex.size	Numeric scalar specifying the size of the vertices in the plot
vertex.radius	Numeric scalar specifying the radius of spheres in 3D
vertex.label	Character vector specifying the labels for vertices
layout	Layout algorithm name (see details)
vertex.color	Controls the color of vertices
draw.edges	Set to TRUE to draw edges in 3d
legend.cex	Size of legend window when y is not NULL
legend.position	Character string specifying legend position
quantize.method	Method to quantize a variable: "uniform" or "quantile"
label.cex	Size of vertex labels in 3D plots
label.adj	Horizontal and vertical adjustment of labels
use.saved.layout	Layout matrix from previous call
...	Additional parameters passed to igraph plot

Value

Invisibly returns a list containing graph, layout, and color information

Examples

```
## Not run:
# Create a ggraph object
adj_list <- list(c(2, 3), c(1, 3), c(1, 2))
weight_list <- list(c(1, 1), c(1, 1), c(1, 1))
g <- ggraph(adj_list, weight_list)

# Plot with default settings
plot.res <- plot(g)

# Plot with custom vertex size and labels
plot(g, vertex.size = 5, vertex.label = c("A", "B", "C"))

# Plot with a different layout
plot(g, layout = "in_circle")

# Recreate the same layout
plot(g, vertex.size = 2, use.saved.layout = plot.res$layout)

## End(Not run)
```

`plot.graph.3d` *Plot a graph with a function z in 3D with basin extrema points*

Description

Creates a 3D visualization using rgl where the graph structure is displayed in the x-y plane and function values (z) are represented as points or spheres above the graph. The function supports multiple input formats and can highlight basin extrema (local minima and maxima).

Usage

```
## S3 method for class 'graph.3d'
plot(
  x,
  z = NULL,
  layout = "kk",
  conn.points = TRUE,
  use.spheres = TRUE,
  graph.alpha = 0.7,
  z.point.size = 0.5,
  z.color = NULL,
  z.alpha = 1,
  edge.color = "gray70",
  edge.width = 1,
  base.plane = TRUE,
  base.vertex.size = 0.5,
  z.scale = 1,
  show.axes = TRUE,
  vertical.lines = TRUE,
  vertical.line.style = "dashed",
  dash.length = 0.05,
  gap.length = 0.05,
  vertical.line.color = "darkgray",
  vertical.line.alpha = 0.5,
  vertical.line.width = 0.5,
  basins.df = NULL,
  evertex.sphere.radius = 0.2,
  evertex.min.color = "blue",
  evertex.max.color = "red",
  evertex.cex = 1,
  evertex.adj = c(0.5, 0.5),
  evertex.label.offset = 0.3,
  ...
)
```

Arguments

`x` One of the following:

- A `graph.3d` object created by `graph.3d()`
- A `ggraph` object created by `ggraph()`

- A list returned by `plot.ggraph()` containing graph and layout elements

`z`	A numeric vector containing z values for each vertex in the graph. Required unless x is a graph.3d object that already contains z values.
`layout`	Character string or matrix. Used only when x is a ggraph object: • Character options: "kk" (Kamada-Kawai), "fr" (Fruchterman-Reingold), "nicely", "circle", "tree", "lgl", "graphopt", "drl", "dh", "mds" • Matrix: Pre-computed 2D layout coordinates (n x 2 matrix) Default is "kk". Ignored when x already contains layout information.
`conn.points`	Logical, whether to connect z-values with lines if their vertices are connected in the original graph. Default is TRUE.
`use.spheres`	Logical, whether to use spheres (TRUE) or points (FALSE) to represent z values. Default is TRUE.
`graph.alpha`	Numeric between 0 and 1, opacity of the original graph in the x-y plane. Default is 0.7.
`z.point.size`	Numeric, size of the points or spheres representing z values. Default is 0.5.
`z.color`	Color specification for z-values. Can be: • NULL: Uses rainbow colors based on z values (default) • Single color: Applied to all vertices • Vector of colors: Must match the number of vertices
`z.alpha`	Numeric between 0 and 1, opacity of the z points/spheres. Default is 1.
`edge.color`	Color for the edges connecting z values. Default is "gray70".
`edge.width`	Numeric, width of the edges connecting z values. Default is 1.
`base.plane`	Logical, whether to draw the original graph in the x-y plane. Default is TRUE.
`base.vertex.size`	Numeric, size of vertices in the base graph. Default is 0.5.
`z.scale`	Numeric, scaling factor for z values. Default is 1.
`show.axes`	Logical, whether to show 3D axes. Default is TRUE.
`vertical.lines`	Logical, whether to draw vertical lines connecting base vertices to z points. Default is TRUE.
`vertical.line.style`	Character, style of vertical lines: "solid" or "dashed". Default is "dashed".
`dash.length`	Numeric, length of each dash when using dashed lines. Default is 0.05.
`gap.length`	Numeric, length of gaps between dashes. Default is 0.05.
`vertical.line.color`	Color for vertical lines. Default is "darkgray".
`vertical.line.alpha`	Numeric between 0 and 1, opacity of vertical lines. Default is 0.5.
`vertical.line.width`	Numeric, width of vertical lines. Default is 0.5.
`basins.df`	Data frame containing basin extrema information with columns: • `evertex`: Vertex indices (required) • `is_max`: 0 for minima, 1 for maxima (required) • `label`: Text labels for extrema (required) Default is NULL (no extrema highlighted).

`evertex.sphere.radius`	Numeric, radius of spheres representing extrema. Default is 0.2.
`evertex.min.color`	Color for local minima. Default is "blue".
`evertex.max.color`	Color for local maxima. Default is "red".
`evertex.cex`	Numeric, size of extrema labels. Default is 1.
`evertex.adj`	Numeric vector of length 2, horizontal and vertical adjustment of extrema labels. Default is c(0.5, 0.5).
`evertex.label.offset`	Numeric, offset of labels from extrema points. Default is 0.3.
`...`	Additional arguments (currently unused).

Details

The function provides three workflows:

Workflow 1: Direct from ggraph

```
g <- ggraph(adj.list, weight.list)
plot.graph.3d(g, z.values)
```

Workflow 2: Using plot result

```
g <- ggraph(adj.list, weight.list)
plot.result <- plot(g)
plot.graph.3d(plot.result, z.values)
```

Workflow 3: Using graph.3d object

```
g3d <- graph.3d(plot.result, z.values)
plot(g3d)
```

Value

NULL invisibly. The function creates an rgl 3D plot as a side effect.

See Also

`ggraph` for creating graph objects, `plot.ggraph` for 2D graph visualization, `graph.3d` for creating graph.3d objects

Examples

```
## Not run:
# Create sample graph
adj.list <- list(c(2, 3), c(1, 3, 4), c(1, 2), c(2))
weight.list <- list(c(1, 1), c(1, 1, 1), c(1, 1), c(1))
z.values <- c(0.5, 1.2, 0.8, 1.5)

# Workflow 1: Direct from ggraph
g <- ggraph(adj.list, weight.list)
plot.graph.3d(g, z.values)
```

```
# Workflow 2: Using plot result with custom layout
plot.result <- plot(g, dim = 2, layout = "fr")
plot.graph.3d(plot.result, z.values, vertical.line.style = "solid")

# Workflow 3: Using graph.3d object
g3d <- graph.3d(plot.result, z.values)
plot(g3d, z.color = "red", base.plane = FALSE)

# With basin extrema highlighting
basins.df <- data.frame(
  evertex = c(1, 3),
  is_max = c(0, 1),
  label = c("Local Min", "Local Max")
)
plot.graph.3d(g, z.values, basins.df = basins.df)

# Custom styling
plot.graph.3d(g, z.values,
  z.color = heat.colors(4),
  vertical.line.style = "dashed",
  base.plane = TRUE,
  show.axes = TRUE
)

## End(Not run)
```

`plot.graphMScx` *Plot Method for graphMScx Objects*

Description

Visualizes various aspects of a Morse-Smale complex computed on a graph, including the complex itself, cell graphs, branches of attraction, and the original graph with cell assignments. The graphMScx object is typically created by the `graph.MS.cx` function.

Usage

```
## S3 method for class 'graphMScx'
plot(
  x,
 type = c("MS_cx_graph", "MS_cx_nerve_graph", "lmin_BoA", "lmax_BoA", "MS_graphs",
   "MS_cell", "Ey"),
  i = 1,
  vertex.size = 2,
  vertex.label = NULL,
  show.only.lextr.labels = TRUE,
  edge.label.cex = 2,
  legend.cex = 1,
  lmin.color = "blue",
  lmax.color = "red",
  label.dist = 1.5,
  legend.position = "topleft",
  lext.sfactor = 5,
  ...
)
```

Arguments

`x`	A graphMScx object containing the results of Morse-Smale complex construction, as returned by `graph.MS.cx`
`type`	Character string specifying the type of plot to generate: `"MS_cx_graph"` Plot the Morse-Smale complex graph with vertices colored by type (local minimum/maximum) using a bipartite layout `"MS_cx_nerve_graph"` Plot the Čech (nerve) graph of the Morse-Smale complex with edges weighted by the number of shared vertices between cells `"lmin_BoA"` Plot the graph with vertices colored by their local minima branches of attraction `"lmax_BoA"` Plot the graph with vertices colored by their local maxima branches of attraction `"MS_graphs"` Plot a specific Morse-Smale cell subgraph (specify cell with parameter `i`) `"MS_cell"` Plot the entire graph highlighting a specific cell with vertices colored by function values `"Ey"` Plot the entire graph with vertices colored by function values (Ey)
`i`	Integer specifying the index of the Morse-Smale cell to plot when `type = "MS_graphs"` or `type = "MS_cell"`. Default is 1.
`vertex.size`	Numeric scalar or vector specifying vertex sizes. For local extrema, this is scaled by `lext.sfactor`. Default is 2.
`vertex.label`	Character vector of vertex labels. If NULL (default), labels are determined based on `show.only.lextr.labels`.
`show.only.lextr.labels`	Logical. If TRUE (default), only show labels for local extrema vertices.
`edge.label.cex`	Numeric scalar controlling the size of edge labels in the Morse-Smale complex graph. Default is 2.
`legend.cex`	Numeric scalar controlling legend text size. Default is 1.
`lmin.color`	Color for local minima vertices. Default is "blue".
`lmax.color`	Color for local maxima vertices. Default is "red".
`label.dist`	Numeric scalar controlling the distance of vertex labels from vertices. Default is 1.5.
`legend.position`	Character string specifying legend position. Options include "topleft", "topright", "bottomleft", "bottomright". Default is "topleft".
`lext.sfactor`	Numeric scalar for scaling the size of local extrema vertices relative to regular vertices. Default is 5.
`...`	Additional arguments passed to plotting functions

Details

The function provides multiple visualization options for understanding the structure of the Morse-Smale complex:

- The complex graph shows connections between critical points
- Branch visualizations show which extrema influence each vertex
- Cell visualizations show the decomposition of the graph

Local minima are displayed as downward-pointing triangles (blue by default), while local maxima are shown as upward-pointing triangles (red by default).

Value

Invisible NULL. The function is called for its side effect of creating a plot.

Note

This function requires the igraph package for graph visualization. The input object must be properly formatted with all required components from the Morse-Smale complex computation.

See Also

graph.MS.cx for computing the Morse-Smale complex, ggraph for creating graph objects

Examples

```
## Not run:
# Create a simple graph
adj_list <- list(c(2, 3), c(1, 3, 4), c(1, 2, 4), c(2, 3))
weight_list <- list(c(1, 1), c(1, 1, 1), c(1, 1, 1), c(1, 1))
g <- ggraph(adj_list, weight_list)

# Define function values on vertices
Ey <- c(0.5, 0.3, 0.8, 0.2)

# Compute Morse-Smale complex
mscx <- graph.MS.cx(g, Ey)

# Plot the Morse-Smale complex graph
plot(mscx)

# Show branches of attraction for local minima
plot(mscx, type = "lmin_BoA")

# Plot a specific cell
plot(mscx, type = "MS_graphs", i = 2)

# Show the entire graph with first cell highlighted
plot(mscx, type = "MS_cell", i = 1)

## End(Not run)
```

plot.graph_kernel_smoother

Plot Method for graph_kernel_smoother Objects

Description

Creates diagnostic plots for `graph_kernel_smoother` objects.

Usage

```
## S3 method for class 'graph_kernel_smoother'
plot(x, which = "cv", main = NULL, ...)
```

Arguments

`x`	A `graph_kernel_smoother` object.
`which`	Character string or integer specifying the plot type: • `"cv"` or 1: Cross-validation error vs bandwidth • `"predictions"` or 2: Predictions vs vertex index • `"both"` or 3: Both plots in a 2x1 layout
`main`	Optional main title for the plot(s). If `NULL`, default titles are used.
`...`	Additional graphical parameters passed to plotting functions.

Value

Invisibly returns the input object.

Examples

```
# See examples in graph.kernel.smoother()
```

plot.graph_spectral_filter

Plot Method for Graph Spectral Filter Results

Description

Plot Method for Graph Spectral Filter Results

Usage

```
## S3 method for class 'graph_spectral_filter'
plot(x, which = c(1L, 4L), main = NULL, ...)
```

Arguments

`x`	An object of class `"graph_spectral_filter"`
`which`	Integer or character vector specifying which plots to produce: 1 = "gcv" (GCV scores), 2 = "spectrum" (eigenvalue spectrum), 3 = "filter" (filter response), 4 = "signal" (original vs filtered signal)
`main`	Main title for the plots
`...`	Further graphical parameters

Value

Invisibly returns the input object

`plot.gridX` *Plots the grid of a set around which the grid was created*

Description

Plots the grid of a set around which the grid was created

Usage

```
## S3 method for class 'gridX'
plot(x, with.bounding.box.pts = TRUE, ...)
```

Arguments

`x`	An output from v1.create.X.grid.xD() or create.X.grid.xD().
`with.bounding.box.pts`	Set to TRUE to show the bounding box.
`...`	Additional arguments passed to the plotting functions.

`plot.harmonic_smoother` *Plot Method for Harmonic Smoother Results*

Description

Creates various plots to visualize the results of harmonic smoothing with topology tracking.

Usage

```
## S3 method for class 'harmonic_smoother'
plot(x, y = NULL, ..., type = c("topology", "extrema", "values"))
```

Arguments

`x`	An object of class `"harmonic_smoother"`.
`y`	Ignored (included for S3 method consistency).
`...`	Additional graphical parameters passed to `plot()`.
`type`	Character string specifying the type of plot. Options are: **"topology"** Evolution of topology differences (default) **"extrema"** Evolution of extrema counts **"values"** Original vs smoothed values

Value

Invisibly returns the input object.

See Also

`harmonic.smoother`

Examples

```
## Not run:
# After running harmonic.smoother()
result <- harmonic.smoother(adj.list, weight.list, values, region)

# Plot topology evolution
plot(result, type = "topology")

# Plot extrema counts
plot(result, type = "extrema")

# Plot smoothed values
plot(result, type = "values")

## End(Not run)
```

`plot.IkNNgraphs` *Plot Diagnostics for Intersection k-NN Graph Analysis*

Description

Creates diagnostic plots for analyzing the properties of intersection k-NN graphs across different k values. Can display edit distances, edge counts, and Jensen-Shannon divergence metrics.

Usage

```
## S3 method for class 'IkNNgraphs'
plot(
  x,
  type = "diag",
  diags = c("edist", "edge", "deg"),
  with.pwlm = TRUE,
  with.lmin = FALSE,
  breakpoint.col = "blue",
  lmin.col = "gray",
  k = NA,
  mar = c(2.5, 2.5, 0.5, 0.5),
  mgp = c(2.5, 0.5, 0),
  tcl = -0.3,
  xline = 2.4,
  yline = 3.15,
  ...
)
```

Arguments

`x`	A list containing analysis results for intersection k-NN graphs, typically with components like 'k.values', 'edit.distances', 'n.edges.in.pruned.graph', and 'js.div'.
`type`	Character string specifying the type of plot. Either "diag" for diagnostic plots (default) or "graph" for network visualization.
`diags`	Character vector specifying which diagnostics to show. Options include "edist" (edit distances), "edge" (edge counts), and "deg" (degree distribution). Default is c("edist", "edge", "deg").
`with.pwlm`	Logical. If TRUE, overlays piecewise linear model fits on the plots. Default is TRUE.
`with.lmin`	Logical. If TRUE, shows vertical lines at local minimum points. Default is FALSE.
`breakpoint.col`	Color for breakpoint vertical lines. Default is "blue".
`lmin.col`	Color for local minima vertical lines. Default is "gray".
`k`	Optional numeric value to highlight a specific k value on the plots.
`mar`	Numeric vector of plot margins (bottom, left, top, right). Default is c(2.5, 2.5, 0.5, 0.5).
`mgp`	Numeric vector for axis title, labels, and line positions. Default is c(2.5, 0.5, 0).
`tcl`	Numeric value for tick mark length. Default is -0.3.
`xline`	Numeric value for position of x-axis label. Default is 2.4.
`yline`	Numeric value for position of y-axis label. Default is 3.15.
`...`	Additional arguments passed to plot().

Value

Invisibly returns NULL. The function is called for its side effect of creating plots.

Examples

```
## Not run:
# Create sample data for IkNNgraphs analysis
set.seed(123)
res <- list(
  k.values = 3:10,
  edit.distances = c(25, 18, 14, 12, 10, 9, 8, 7),
  n.edges.in.pruned.graph = c(50, 80, 100, 115, 125, 132, 138, 142),
  js.div = c(0.5, 0.3, 0.2, 0.15, 0.12, 0.1, 0.09, 0.08),
  edit.distances.breakpoint = 5,
  n.edges.in.pruned.graph.breakpoint = 6,
  js.div.breakpoint = 5.5
)

# Plot diagnostics with default settings
  plot.IkNNgraphs(res)

  # Plot only edit distances and edge counts
  plot.IkNNgraphs(res, diags = c("edist", "edge"))

## End(Not run)
```

`plot.instrumented.gds` *Plot Instrumented GDS Results*

Description

Creates diagnostic plots for instrumented graph diffusion smoother results using base R graphics. Supports both multi-panel and single-panel layouts.

Usage

```
## S3 method for class 'instrumented.gds'
plot(
  x,
  metrics = c("snr", "mad", "energy.ratio"),
  layout = c("multi", "single"),
  normalize = FALSE,
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  col = NULL,
  lty = NULL,
  lwd = 2,
  legend.pos = "topright",
  cex.main = 1.2,
  cex.lab = 1,
  cex.axis = 0.9,
  mar.panel = c(4, 4, 2, 1),
  oma = c(0, 0, 3, 0),
  ...
)
```

Arguments

`x`	An object of class `"instrumented.gds"`.
`metrics`	Character vector of metrics to plot. Options include "snr", "mad", "energy.ratio". Default is c("snr", "mad", "energy.ratio").
`layout`	Character string specifying the layout type:

- "multi": Creates separate panels for each metric (default)
- "single": Plots all metrics in a single panel

`normalize`	Logical; if TRUE and layout="single", normalizes all metrics to $[0, 1]$ range for comparison. Default is FALSE.
`main`	Main title for the plot. Default depends on layout.
`xlab`	Label for x-axis. Default is "Iteration" for single layout.
`ylab`	Label for y-axis. Default depends on layout and normalization.
`col`	Colors for each metric. Can be a single color or a vector of colors. Default uses a color palette based on the number of metrics.
`lty`	Line types for each metric. Default is 1 (solid) for multi-panel, or different line types for single-panel.
`lwd`	Line width. Default is 2.
`legend.pos`	Position of legend for single layout. Options include "topright", "topleft", "bottomright", "bottomleft", "top", "bottom", "left", "right", "center", or "none". Default is "topright".
`cex.main`	Character expansion for main title. Default is 1.2.
`cex.lab`	Character expansion for axis labels. Default is 1.
`cex.axis`	Character expansion for axis text. Default is 0.9.
`mar.panel`	Margins for each panel (multi layout only). Default is c(4, 4, 2, 1).
`oma`	Outer margins (multi layout only). Default is c(0, 0, 3, 0).
`...`	Additional arguments passed to plotting functions.

Details

This function supports two layout modes:

- Multi-panel layout: Creates vertically stacked panels with one panel per metric, each with its own y-axis scale.
- Single-panel layout: Plots all metrics on the same axes, optionally normalized to facilitate comparison.

The choice of layout depends on your analysis needs:

- Use "multi" when metrics have very different scales or when you want to see detailed patterns in each metric.
- Use "single" when you want to compare the timing and relative changes across metrics.

Value

Invisibly returns NULL. Called for side effect of creating plots.

See Also

`instrumented.gds`

Examples

```
## Not run:
# Create example data
graph <- list(c(2), c(1,3), c(2))
edge.lengths <- list(c(1), c(1,1), c(1))
y.true <- c(0, 1, 0)
y.noisy <- y.true + rnorm(3, 0, 0.1)

# Run instrumented GDS
result <- instrumented.gds(
  graph = graph,
  edge.lengths = edge.lengths,
  y = y.noisy,
  y.true = y.true,
  n.time.steps = 50,
  base.step.factor = 0.5
)

# Default multi-panel plot
plot(result)

# Single panel with all metrics
plot(result, layout = "single")

# Single panel with normalization
plot(result, layout = "single", normalize = TRUE)

# Custom colors for multi-panel
plot(result, col = c("blue", "red", "darkgreen"), lwd = 3)

# Plot only specific metrics in single panel
plot(result, metrics = c("snr", "mad"), layout = "single",
     col = c("navy", "darkred"), lty = c(1, 2))

## End(Not run)
```

`plot.kh.matrix` *Plot a k-h Matrix*

Description

This function creates a heatmap-style plot of a k-h matrix, which represents the presence or absence of a local maximum for different combinations of k (number of nearest neighbors) and h (hop size) values.

Usage

```
## S3 method for class 'kh.matrix'
plot(
  x,
  existing.k = NULL,
  h.values = NULL,
  id = NULL,
  color.palette = c("white", "black"),
  xlab = "k (number of nearest neighbors)",
  ylab = "h (hop size)",
  main = NULL,
  with.legend = FALSE,
  ...
)
```

Arguments

`x`	Either a kh.matrix object (list with components kh.mat, existing.k, h.values, id) or a matrix where rows represent k values and columns represent h values. Values should be 0 (absent) or 1 (present).
`existing.k`	A vector of k values corresponding to the rows (used when x is a matrix).
`h.values`	A vector of h values corresponding to the columns (used when x is a matrix).
`id`	A string identifier for the local maximum being plotted (used when x is a matrix).
`color.palette`	A vector of two colors for the plot. Default is c("white", "black").
`xlab`	Label for the x-axis. Default is "k (number of nearest neighbors)".
`ylab`	Label for the y-axis. Default is "h (hop size)".
`main`	Title of the plot. Default is "Presence of Local Maximum" followed by id.
`with.legend`	Logical. If TRUE, a legend is added to the plot. Default is FALSE.
`...`	Additional arguments passed to image()

Details

The function creates a heatmap where each cell represents a combination of k and h values. Black cells indicate the presence of a local maximum, while white cells indicate its absence.

Value

A plot is created on the current graphics device.

Examples

```
## Not run:
# Create a kh.matrix object
kh.mat <- matrix(c(0,1,1,0,1,0,1,1,0), nrow = 3)
existing.k <- c(10, 20, 30)
h.values <- c(1, 2, 3)
khm <- kh.matrix(kh.mat, existing.k, h.values, id = "LM1")

# Basic plot
plot(khm)

# With custom colors and a legend
plot(khm,
     color.palette = c("lightblue", "darkred"),
     with.legend = TRUE)

# You can also override parameters from the object
```

```
plot(khm, main = "Custom Title", xlab = "k values")

## End(Not run)
```

`plot.local_extrema` *Plot Local Extrema Detection Results*

Description

Creates visualizations showing the distribution of function values and neighborhood characteristics of detected extrema.

Usage

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

Arguments

x	An object of class "local_extrema".
...	Additional graphical parameters passed to plotting functions.

Details

This function creates a two-panel plot:

- Left panel: Function values at extrema, ordered by rank
- Right panel: Neighborhood radii for each extremum

Point sizes in both panels are proportional to neighborhood sizes.

Value

Invisibly returns x.

Examples

```
# Create example data
adj.list <- list(c(2), c(1,3), c(2,4), c(3,5), c(4))
weight.list <- list(c(1), c(1,1), c(1,1), c(1,1), c(1))
y <- c(1, 3, 2, 5, 1)

# Detect and plot maxima
maxima <- detect.local.extrema(adj.list, weight.list, y, 2, 2)
plot(maxima)
```

plot.mabilo *Plot Method for Mabilo Objects*

Description

Generates diagnostic and visualization plots for mabilo objects. The function supports different types of plots including fitted values with optional credible intervals, error diagnostics, and residual analyses.

Usage

```
## S3 method for class 'mabilo'
plot(
  x,
  type = "fit",
  title = "",
  xlab = "",
  ylab = "",
  with.y.true = TRUE,
  with.pts = FALSE,
  with.CrI = TRUE,
  true.col = "red",
  ma.col = "blue",
  pts.col = "gray60",
  with.predictions.pts = FALSE,
  predictions.pts.col = "blue",
  predictions.pts.pch = 20,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray85",
  CrI.line.col = "gray10",
  CrI.line.lty = 2,
  ylim = NULL,
  legend.cex = 0.8,
  k = NULL,
  with.legend = TRUE,
  legend.pos = "topright",
  ...
)
```

Arguments

`x`	An object of class "mabilo"
`type`	Character string specifying the type of plot to create (partial matching supported): • "fit": Plot fitted values against x values (e.g., "f" or "fi") • "diagnostic": Plot error diagnostics for different k values (e.g., "d" or "dia") • "residuals": Plot residuals against x values (e.g., "res") • "residuals.hist": Plot histogram of residuals (e.g., "rh" or "residuals.h")
`title`	Plot title (default: "")

`xlab`	Label for x-axis (default: "")
`ylab`	Label for y-axis (default: "")
`with.y.true`	Logical; if TRUE and true values are available, adds them to the plot (default: TRUE)
`with.pts`	Logical; if TRUE, adds original data points to the plot (default: FALSE)
`with.CrI`	Logical; if TRUE and bootstrap results available, adds credible intervals (default: TRUE)
`true.col`	Color for true values line (default: "red")
`ma.col`	Color for predictions (default: "blue")
`pts.col`	Color for data points (default: "gray60")
`with.predictions.pts`	Logical; if TRUE, adds points at prediction values (default: FALSE)
`predictions.pts.col`	Color for prediction points (default: "blue")
`predictions.pts.pch`	Point character for prediction points (default: 20)
`CrI.as.polygon`	Logical; if TRUE, draws credible intervals as a polygon (default: TRUE)
`CrI.polygon.col`	Color for credible interval polygon (default: "gray85")
`CrI.line.col`	Color for credible interval lines (default: "gray10")
`CrI.line.lty`	Line type for credible interval lines (default: 2)
`ylim`	Numeric vector of length 2 giving y-axis limits (default: NULL)
`legend.cex`	Character expansion factor for legend (default: 0.8)
`k`	Specific k value to plot predictions for (default: NULL, uses optimal k)
`with.legend`	Logical; if TRUE, displays legend on the plot (default: TRUE)
`legend.pos`	Character string specifying legend position (default: "topright")
`...`	Additional arguments passed to plotting functions

Details

The function produces different types of plots:

For type = "fit":

- Plots the fitted values against x values
- Optionally includes true values and data points
- Shows credible intervals if bootstrap results available
- Supports customization of colors and styles

For type = "diagnostic":

- Shows LOOCV error diagnostics for different k values
- Indicates optimal k value with vertical line
- Shows true errors if available

For type = "residuals" and "residuals.hist":

- Visualizes model residuals in various ways
- Includes reference lines and density curves

Value

Invisibly returns NULL. The function is called for its side effect of producing plots.

See Also

mabilo for fitting the mabilo model

Examples

```
# Generate example data
x <- seq(0, 10, length.out = 100)
y <- sin(x) + rnorm(100, 0, 0.1)

# Fit mabilo model
fit <- mabilo(x, y, n.bb = 100)

# Basic fit plot with credible intervals
plot(fit)

# Diagnostic plot showing errors
plot(fit, type = "diagnostic")

# Residual plot
plot(fit, type = "residuals")
```

---

plot.mabilog *Complete Utility Functions for mabilog()*

Description

Generates diagnostic and visualization plots for mabilog objects. The function supports different types of plots including fitted values with optional credible intervals, error diagnostics, and residual analyses.

Usage

```
## S3 method for class 'mabilog'
plot(
  x,
  type = "fit",
  title = "",
  xlab = "",
  ylab = "",
  with.y.true = TRUE,
  with.pts = FALSE,
  with.CrI = TRUE,
  true.col = "red",
  ma.col = "blue",
  pts.col = c("red", "darkgreen"),
  with.predictions.pts = FALSE,
  predictions.pts.col = "blue",
  predictions.pts.pch = 20,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray85",
  CrI.line.col = "gray10",
  CrI.line.lty = 2,
  ylim = NULL,
  legend.cex = 0.8,
  k = NULL,
  with.legend = TRUE,
  legend.pos = "topright",
  ...
)
```

Arguments

`x`	An object of class "mabilog"
`type`	Character string specifying the type of plot to create (partial matching supported): • "fit": Plot fitted values against x values (e.g., "f" or "fi") • "diagnostic": Plot error diagnostics for different k values (e.g., "d" or "dia") • "residuals": Plot residuals against x values (e.g., "res") • "residuals.hist": Plot histogram of residuals (e.g., "rh" or "residuals.h") • "roc": ROC curve for binary classification (e.g., "ro")
`title`	Plot title (default: "")
`xlab`	Label for x-axis (default: "")
`ylab`	Label for y-axis (default: "")
`with.y.true`	Logical; if TRUE and true values are available, adds them to the plot (default: TRUE)
`with.pts`	Logical; if TRUE, adds original data points to the plot (default: FALSE)
`with.CrI`	Logical; if TRUE and bootstrap results available, adds credible intervals (default: TRUE)
`true.col`	Color for true values line (default: "red")
`ma.col`	Color for predictions (default: "blue")
`pts.col`	Color for data points when y=0 and y=1 (default: c("red", "darkgreen"))
`with.predictions.pts`	Logical; if TRUE, adds points at prediction values (default: FALSE)
`predictions.pts.col`	Color for prediction points (default: "blue")
`predictions.pts.pch`	Point character for prediction points (default: 20)
`CrI.as.polygon`	Logical; if TRUE, draws credible intervals as a polygon (default: TRUE)
`CrI.polygon.col`	Color for credible interval polygon (default: "gray85")
`CrI.line.col`	Color for credible interval lines (default: "gray10")
`CrI.line.lty`	Line type for credible interval lines (default: 2)
`ylim`	Numeric vector of length 2 giving y-axis limits (default: NULL)

`legend.cex`	Character expansion factor for legend (default: 0.8)
`k`	Specific k value to plot predictions for (default: NULL, uses optimal k)
`with.legend`	Logical; if TRUE, displays legend on the plot (default: TRUE)
`legend.pos`	Character string specifying legend position (default: "topright")
`...`	Additional arguments passed to plotting functions

Details

This file contains all utility functions adapted for the mabilog() function, including plot, print, summary, predict, fitted, residuals, coef, and other S3 methods. Plot Method for Mabilog Objects

Value

Invisibly returns NULL. The function is called for its side effect of producing plots.

`plot.mabilo_plus` *Plot Method for Mabilo Plus Objects*

Description

Generates various diagnostic and visualization plots for mabilo.plus objects. The function supports different types of plots including fitted values, diagnostic plots, and residual analyses. For each plot type, various customization options are available through the function parameters.

Usage

```
## S3 method for class 'mabilo_plus'
plot(
  x,
  type = "fit",
  title = "",
  xlab = "",
  ylab = "",
  predictions.type = "both",
  diagnostic.type = "both",
  with.y.true = TRUE,
  with.pts = FALSE,
  true.lwd = 2,
  true.col = "red",
  sm.col = "#00FF00",
  ma.col = "blue",
  pts.col = "gray60",
  with.predictions.pts = FALSE,
  predictions.pts.col = "blue",
  predictions.pts.pch = 20,
  ylim = NULL,
  legend.cex = 0.8,
  k = NULL,
  with.legend = TRUE,
  ...
)
```

Arguments

`x`	An object of class "mabilo_plus"
`type`	Character string specifying the type of plot to create (partial matching supported): • "fit": Plot fitted values against x values (e.g., "f" or "fi") • "diagnostic": Plot error diagnostics for different k values (e.g., "d" or "dia") • "residuals": Plot residuals against x values (e.g., "res") • "residuals.hist": Plot histogram of residuals (e.g., "rh" or "residuals.h")
`title`	Plot title (default: "")
`xlab`	Label for x-axis (default: "")
`ylab`	Label for y-axis (default: "")
`predictions.type`	Type of predictions to plot: • "both": plot both sm_predictions and ma_predictions • "sm": sm_predictions only • "ma": ma_predictions only
`diagnostic.type`	Type of errors to show in diagnostic plot: • "both": show both sm and ma errors • "sm": sm errors only • "ma": ma errors only
`with.y.true`	Logical; if TRUE and true values are available, adds them to the plot (default: TRUE)
`with.pts`	Logical; if TRUE, adds original data points to the plot (default: FALSE)
`true.lwd`	Line width for true values (default: 2)
`true.col`	Color for true values line (default: "red")
`sm.col`	Color for sm prediction and error lines (default: "#00FF00")
`ma.col`	Color for ma prediction and error lines (default: "blue")
`pts.col`	Color for points (default: "gray60")
`with.predictions.pts`	Logical; if TRUE, adds points at prediction values (default: FALSE)
`predictions.pts.col`	Color for prediction points (default: "blue")
`predictions.pts.pch`	Point character for prediction points (default: 20)
`ylim`	Numeric vector of length 2 giving y-axis limits; if NULL, computed from data (default: NULL)
`legend.cex`	Character expansion factor for legend (default: 0.8)
`k`	Specific k value to plot predictions for (default: NULL, uses optimal k)
`with.legend`	Logical; if TRUE, displays legend on the plot (default: TRUE)
`...`	Additional arguments passed to plotting functions

Value

Invisibly returns NULL. The function is called for its side effect of producing plots.

Examples

```
## Not run:
# Generate example data
x <- seq(0, 10, length.out = 100)
y <- sin(x) + rnorm(100, 0, 0.1)

# Fit mabilo.plus model
fit <- mabilo.plus(x, y)

# Basic fit plot
plot(fit)

# Plot both SM and MA predictions
plot(fit, type = "fit", predictions.type = "both")

# Diagnostic plot
plot(fit, type = "diagnostic", diagnostic.type = "both")

# Residual plots
plot(fit, type = "residuals", predictions.type = "ma")
plot(fit, type = "residuals.hist", predictions.type = "both")

## End(Not run)
```

`plot.maelog` *Plot Method for maelog Objects*

Description

Produces diagnostic plots for fitted `maelog` objects.

Usage

```
## S3 method for class 'maelog'
plot(x, which = 1L, main = "", ...)
```

Arguments

`x`	A fitted object of class "`maelog`".
`which`	Integer specifying which plot to produce: **1** Fitted values vs predictor with data points (default) **2** Cross-validation error vs bandwidth (if bandwidth was selected) **3** Residuals vs fitted values **4** QQ-plot of Pearson residuals
`main`	Character string for plot title. Default title is used if `""`.
`...`	Additional graphical parameters passed to the plotting functions.

Value

Invisibly returns the input object.

Note

Plot 2 (bandwidth selection) is only available when the model was fitted with automatic bandwidth selection (i.e., `pilot.bandwidth` <= `0`).

To display multiple plots, use `par(mfrow)` or call `plot` multiple times:

```
par(mfrow = c(2, 2))
for (i in 1:4) plot(fit, which = i)
```

See Also

`maelog`, `summary.maelog`

Examples

```
# Generate example data
set.seed(123)
n <- 200
x <- runif(n)
y <- rbinom(n, 1, plogis(10 * (x - 0.5)))

# Fit with fixed bandwidth
fit1 <- maelog(x, y, pilot.bandwidth = 0.1)

# Plot fitted values
plot(fit1)

## Not run:
# Plot residuals
plot(fit1, which = 3)

# Plot QQ plot
plot(fit1, which = 4)

# Fit with automatic bandwidth selection
fit2 <- maelog(x, y, pilot.bandwidth = -1, n.bws = 30)

# Show all plots
par(mfrow = c(2, 2))
plot(fit2, which = 1)  # Fitted values
plot(fit2, which = 2)  # Bandwidth selection
plot(fit2, which = 3)  # Residuals
plot(fit2, which = 4)  # QQ plot
par(mfrow = c(1, 1))

## End(Not run)
```

`plot.magelo` *Plot method for magelo objects*

Description

Plot method for magelo objects

Usage

```
## S3 method for class 'magelo'
plot(
  x,
  type = "fit",
  title = "",
  xlab = "",
  ylab = "",
  with.y.true = TRUE,
  with.pts = FALSE,
  with.CrI = TRUE,
  true.col = "red",
  ma.col = "blue",
  pts.col = "gray60",
  with.predictions.pts = FALSE,
  predictions.pts.col = "blue",
  predictions.pts.pch = 20,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray85",
  CrI.line.col = "gray10",
  CrI.line.lty = 2,
  ylim = NULL,
  legend.cex = 0.8,
  bw = NULL,
  with.legend = TRUE,
  legend.position = "topright",
  legend.inset = 0.05,
  ...
)
```

Arguments

`x`	An output from magelo()
`type`	Type of plot: "fit", "diagnostic", "residuals" or "residuals.hist"
`title`	Plot title
`xlab`	X-axis label
`ylab`	Y-axis label
`with.y.true`	Logical, whether to plot true values if available
`with.pts`	Logical, whether to plot data points
`with.CrI`	Logical, whether to plot credible intervals
`true.col`	Color for true values
`ma.col`	Color for magelo fit
`pts.col`	Color for data points
`with.predictions.pts`	Logical, whether to show prediction points
`predictions.pts.col`	Color for prediction points
`predictions.pts.pch`	Point character for predictions

`CrI.as.polygon`	Logical, whether to show CrI as polygon
`CrI.polygon.col`	Color for CrI polygon
`CrI.line.col`	Color for CrI lines
`CrI.line.lty`	Line type for CrI lines
`ylim`	Y-axis limits
`legend.cex`	Legend text size
`bw`	Bandwidth value to use (analogous to k in malowess)
`with.legend`	Logical, whether to show legend
`legend.position`	Character string indicating legend position (default: "topright"). One of "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", or "center"
`legend.inset`	Numeric value for inset distance from the margins as a fraction of the plot region (default: 0.05)
`...`	Additional parameters passed to plot methods

`plot.model.errors` *Create Error Plot for Model Comparisons*

Description

Creates a plot showing total absolute true errors for different models, with error bars representing the 95% credible intervals based on Bayesian bootstrap samples. Supports both bar plot and point plot visualizations.

Usage

```
## S3 method for class 'model.errors'
plot(
  x,
  method = c("bars", "points"),
  title = NULL,
  y.lab = "Total Absolute True Error",
  point.col = "black",
  bar.col = "white",
  border.col = "black",
  error.bar.col = "black",
  error.bar.width = 0.1,
  bar.width = 0.7,
  point.size = 1.2,
  x.margin = 0.5,
  line1 = 1,
  line2 = 2.8,
  line3 = 1,
  with.vertical.lines = TRUE,
  vertical.line.col = "gray",
  vertical.line.lty = 1,
  margin.bottom = 8,
  ...
)
```

Arguments

`x`	An object of class "model.errors" containing the following elements: • `integrals`: Named numeric vector of integral values • `bb.integrals`: List of Bayesian bootstrap integral values for each model
`method`	Visualization method: "bars" or "points" (default: "bars")
`title`	Plot title (optional)
`y.lab`	Y-axis label (default: "Total Absolute True Error")
`point.col`	Color of points when method="points" (default: "black")
`bar.col`	Color of bars when method="bars" (default: "white")
`border.col`	Color of bar borders (default: "black")
`error.bar.col`	Color of error bars (default: "black")
`error.bar.width`	Width of error bar ends (default: 0.1)
`bar.width`	Width of the bars (default: 0.7)
`point.size`	Size of points when method="points" (default: 1.2)
`x.margin`	Margin to add on both sides of x-axis range (default: 0.5)
`line1`	Distance of x-axis labels from axis (default: 1)
`line2`	Distance of y-axis label from axis (default: 2.8)
`line3`	Distance of title from plot (default: 1)
`with.vertical.lines`	Logical, whether to add vertical grid lines (default: TRUE)
`vertical.line.col`	Color of vertical grid lines (default: "gray")
`vertical.line.lty`	Line type for vertical grid lines (default: 1)
`margin.bottom`	Extra space for x-axis labels (default: 8)
`...`	Additional graphical parameters passed to plotting functions

---

`plot.morse.smale.trajectories.from.point`
Plot Morse-Smale Trajectories from Point

Description

Visualizes gradient flow trajectories (both ascending and descending) from a specific starting point. Shows the trajectories on a contour plot background, marks the starting point and endpoints (critical points), and optionally adds directional arrows along the trajectories.

Usage

```
## S3 method for class 'morse.smale.trajectories.from.point'
plot(
  x,
  y,
  mixture,
  grid,
  step = 0.01,
  show.arrows = TRUE,
  use.custom.legend = FALSE,
  ...
)
```

Arguments

`x`	Starting x coordinate
`y`	Starting y coordinate
`mixture`	Object containing: • f: Function taking (x, y) and returning scalar value • gradient: Function taking (x, y) and returning gradient vector c(dx, dy)
`grid`	Grid object created by create.grid for contour plot background
`step`	Step size for gradient flow integration (default 0.01)
`show.arrows`	Logical, whether to show directional arrows on trajectories
`use.custom.legend`	Logical, whether to use custom legend.
`...`	Additional arguments passed to function.contours.plot

Value

Invisible list containing:

`ascending`	Matrix of points along ascending trajectory
`descending`	Matrix of points along descending trajectory

Examples

```
## Not run:
grid <- create.grid(50)
mixture <- list(
  f = function(x, y) -((x-0.3)^2 + (y-0.7)^2) - 2*((x-0.7)^2 + (y-0.3)^2),
  gradient = function(x, y) {
    c(-2*(x-0.3) - 4*(x-0.7), -2*(y-0.7) - 4*(y-0.3))
  }
)
# Plot trajectories from a point
result <- plot.morse.smale.trajectories.from.point(0.5, 0.5, mixture, grid)

## End(Not run)
```

`plot.MSD` *Plot Results of Mean Shift Data Denoising*

Description

This function plots the results of the Mean Shift Data Denoising function `meanshift.data.smoother`. It can produce three types of plots based on the ’type’ parameter.

Usage

```
## S3 method for class 'MSD'
plot(
  x,
  type = "dX",
  i = 2,
  rg = NULL,
  mar = c(2.5, 2.5, 2, 0.5),
  mgp = c(2.5, 0.5, 0),
  tcl = -0.3,
  ...
)
```

Arguments

`x`	A list of class "MSD" containing the results from meanshift.data.smoother function.
`type`	A character string specifying the type of plot. Must be one of "dX" (default), "dXi", or "kdists".
`i`	An integer specifying which trajectory step to plot when type = "dXi". Default is 2.
`rg`	A numeric value specifying the range for x and y axes in "dX" and "dXi" plots. If NULL (default), ranges are computed from the data.
`mar`	Numeric vector of length 4 specifying plot margins.
`mgp`	Numeric vector of length 3 for axis label positions.
`tcl`	Numeric value for tick mark length.
`...`	Additional arguments to be passed to the plot function.

Details

The function produces different plots based on the ’type’ parameter:

- "dX": Plots original data (X) and denoised data (dX) side by side.
- "dXi": Plots original data (X) and a specific trajectory step (X.traj\[\[i\]\]) side by side.
- "kdists": Plots median k-distances with a vertical line at the optimal step.

Value

This function does not return a value; it produces a plot as a side effect.

See Also

meanshift.data.smoother

Examples

```
## Not run: X1™
# Generate sample data
X <- matrix(rnorm(200), ncol = 2)

# Run mean shift smoothing
res <- meanshift.data.smoother(X, k = 5)

# Plot original vs denoised data
plot(res, type = "dX")

# Plot trajectory at step 3
plot(res, type = "dXi", i = 3)

# Plot median k-distances
plot(res, type = "kdists")

## End(Not run)
```

---

plot.nerve_cx_spectral_filter

Plot Nerve Complex Spectral Filter Results

Description

Creates visualizations for the results of nerve.cx.spectral.filter. Supports multiple plot types including parameter selection curves, prediction comparisons, and residual analysis.

Usage

```
## S3 method for class 'nerve_cx_spectral_filter'
plot(
  x,
  type = c("parameters", "predictions", "residuals"),
  y = NULL,
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  col = "black",
  pch = 1,
  ...
)
```

Arguments

x	A nerve_cx_spectral_filter object returned by nerve.cx.spectral.filter.
type	Character string specifying the plot type. One of:

	`"parameters"` GCV scores vs. filter parameter values (default) `"predictions"` Original vs. smoothed values scatter plot `"residuals"` Residuals vs. original values
`y`	Original function values. Required for `"predictions"` and `"residuals"` plot types. Must have the same length as the predictions.
`main`	Character string giving the main title for the plot. If `NULL` (default), an appropriate title is chosen based on the plot type.
`xlab`	Character string giving the x-axis label. If `NULL` (default), an appropriate label is chosen based on the plot type.
`ylab`	Character string giving the y-axis label. If `NULL` (default), an appropriate label is chosen based on the plot type.
`col`	Color specification for the plot. Default is "black" for most plots, with the optimal parameter highlighted in red for the parameters plot.
`pch`	Plotting character. Default is 1 (open circles).
`...`	Additional graphical parameters passed to the underlying plot functions.

Details

The `"parameters"` plot shows the GCV score as a function of the filter parameter on a log-log scale, with the optimal parameter marked in red. This helps visualize the parameter selection process and assess whether the minimum is well-defined.

The `"predictions"` plot shows a scatter plot of original vs. smoothed values with a diagonal reference line. Points close to the diagonal indicate good agreement.

The `"residuals"` plot shows residuals (original - smoothed) vs. original values with a horizontal reference line at zero. This helps identify systematic patterns in the filtering errors.

Value

No return value. Function is called for its side effect of creating a plot.

See Also

`nerve.cx.spectral.filter` for the main filtering function, `summary.nerve_cx_spectral_filter` for numerical summaries

Examples

```
## Not run:
# Create example data and apply filtering
coords <- matrix(runif(200), ncol = 2)
complex <- create.nerve.complex(coords, k = 5, max.dim = 2)
y <- sin(coords[,1] * 2*pi) * cos(coords[,2] * 2*pi) + rnorm(100, 0, 0.1)
complex <- set.complex.function.values(complex, y)
result <- nerve.cx.spectral.filter(complex, y)

# Create different plot types
par(mfrow = c(1, 3))
plot(result, type = "parameters")
plot(result, type = "predictions", y = y)
plot(result, type = "residuals", y = y)

## End(Not run)
```

`plot.pgmalo`	*Plot Method for pgmalo Objects*

Description

Creates diagnostic and visualization plots for pgmalo model fits.

Usage

```
## S3 method for class 'pgmalo'
plot(x, type = "diagnostic", ...)
```

Arguments

`x`	An object of class "pgmalo".
`type`	Character string specifying plot type (default: "diagnostic").
`...`	Additional arguments passed to plotting functions.

Details

Currently only implements diagnostic plot showing cross-validation errors across different h values. For more detailed plotting options, use `plot.upgmalo` with upgmalo objects.

Value

NULL (invisibly). Called for side effect of creating plots.

Examples

```
# See examples in pgmalo()
```

`plot.prediction.errors`	*Plot Prediction Errors for Multiple Smoothing Models*

Description

Creates a line plot comparing prediction errors across different smoothing models, with customizable visual parameters. Each model's errors are represented by a line with distinct color, line type, and point markers.

Usage

```
## S3 method for class 'prediction.errors'
plot(
  x,
  xvals = NULL,
  cols = NULL,
  ltys = NULL,
  main = "",
  xlab = "Gaussian Mean",
  ylab = "Mean Absolute True Error",
  legend.pos = "topleft",
  legend.cex = 0.7,
  legend.ncol = 2,
  point.cex = 0.8,
  ylab.line = 2.8,
  legend.inset = 0.02,
  offset.factor = NULL,
  n.points = 10,
  ...
)
```

Arguments

`x`	A prediction.errors object or a named list, data frame, or matrix containing prediction errors for different models. If a list, each element should be a numeric vector of errors and the names of the list elements are used in the plot legend. If a data frame or matrix, each column represents errors for a different model and column names are used in the legend.
`xvals`	Numeric vector of x-axis values (typically independent variable values). Can be NULL if x contains an 'xvals' attribute or component. Default is NULL.
`cols`	Character vector of colors for the lines. Defaults to a preset palette of 15 colors. Must be at least as long as the number of models.
`ltys`	Numeric vector of line types. Defaults to a preset pattern of 15 line types. Must be at least as long as the number of models.
`main`	Character string for the plot title (default: "")
`xlab`	Character string for x-axis label (default: "Gaussian Mean")
`ylab`	Character string for y-axis label (default: "Mean Absolute True Error")
`legend.pos`	Character string specifying legend position (default: "topleft")
`legend.cex`	Numeric scaling factor for legend text size (default: 0.7)
`legend.ncol`	Integer number of columns in legend (default: 2)
`point.cex`	Numeric scaling factor for point size (default: 0.8)
`ylab.line`	Numeric value specifying the distance of the y-axis label from the plot (in margin line units, default: 2.8)
`legend.inset`	Numeric value specifying the inward shift of the legend from the plot border as a fraction of the plot region (default: 0.02)
`offset.factor`	Numeric value controlling the spacing between points on different lines (default: floor(length(x) / 140)). Larger values create more separation between points across different models. Set to 0 to align points vertically across all lines.
`n.points`	Integer number of points to plot per line (default: 10)
`...`	Additional arguments (currently unused)

Details

The function plots multiple lines representing prediction errors for different models. Each line is distinguished by color, line type, and point markers. Points are added at regular intervals along each line with slight offsets to prevent overlapping. If more than 15 models are provided, the function will throw an error unless custom color and line type vectors of sufficient length are supplied.

Value

Invisibly returns a list containing the colors and line types used in the plot

Examples

```
## Not run:
# Using a named list
errors_list <- list(
  model1 = c(1,2,3,4,5),
  model2 = c(2,3,4,5,6)
)
xvals <- c(1,2,3,4,5)

# Create prediction.errors object
pe <- prediction.errors(errors_list, xvals)

# Basic plot
plot(pe)

# Using a data frame
errors_df <- data.frame(
  model1 = c(1,2,3,4,5),
  model2 = c(2,3,4,5,6)
)

# Create prediction.errors object from data frame
pe_df <- prediction.errors(errors_df, xvals = 1:5)

# Plot with custom parameters
plot(pe_df,
     cols = c("red", "blue"),
     ltys = c(1, 2),
     main = "Prediction Errors",
     legend.pos = "topright")

## End(Not run)
```

`plot.pwlm` *Plot a Piecewise Linear Model*

Description

Creates a visualization of a piecewise linear model fit, showing the data points, fitted lines, and breakpoints. This function handles both segmented models with breakpoints and simple linear models.

Usage

```
## S3 method for class 'pwlm'
plot(
  x,
  main = "Piecewise Linear Regression",
  xlab = "X",
  ylab = "Y",
  point_color = "black",
  line_color = "blue",
  breakpoint_color = "red",
  ...
)
```

Arguments

`x`	An object of class "pwlm", typically the output of fit.pwlm().
`main`	Character string for plot title. Default is "Piecewise Linear Regression".
`xlab`	Character string for x-axis label. Default is "X".
`ylab`	Character string for y-axis label. Default is "Y".
`point_color`	Color for data points. Default is "black".
`line_color`	Color for fitted lines. Default is "blue".
`breakpoint_color`	Color for breakpoint vertical lines. Default is "red".
`...`	Additional arguments passed to plot().

Value

Invisibly returns NULL. The function is called for its side effect of creating a plot.

Examples

```
# Generate sample data
set.seed(123)
x <- 1:20
y <- c(1:10, 20:11) + rnorm(20, 0, 2)

# Fit piecewise linear model with one breakpoint
pwlm_fit <- fit.pwlm(x, y)

# Plot the model
## Not run:
  plot(pwlm_fit)

  # Customize plot appearance
  plot(pwlm_fit, main = "My Custom PWLM Plot",
       point_color = "blue", line_color = "red")

## End(Not run)
```

`plot.star_object` *Visualize 2D Star Graph with Smooth Function Values*

Description

Creates a visualization of a 2D star graph where vertices are colored according to the values of the smooth function. The visualization includes a reference circle, edges connecting vertices, and optional path highlighting. Vertices can be labeled and the function center (mu) can be displayed.

Usage

```
## S3 method for class 'star_object'
plot(
  x,
  y = NULL,
  point.size = 1.5,
  edge.col = "gray70",
  edge.lwd = 1,
  title = "",
  color.palette = c("blue", "yellow", "red"),
  color.edges = FALSE,
  circle.color = "gray",
  pt.lab.adj = c(1, 1),
  pt.lab.cex = 0.75,
  grid.pt.lab.adj = c(1, 1),
  grid.pt.lab.cex = 0.75,
  grid.pt.color = "cyan",
  grid.pt.text.color = "cyan",
  grid.pt.cex.factor = 1.5,
  show.vertex.labs = FALSE,
  zero.based = FALSE,
  show.arm.labs = TRUE,
  arm.cex = 2,
  arms.adj = c(-0.5, 1),
  arm.label.offset = 0,
  show.mu = TRUE,
  mu.cex = 3,
  epsilon = 0.1,
  legend.inset = 0.05,
  legend.bty = "n",
  legend.title = "Function Value",
  axes = FALSE,
  xlab = "",
  ylab = "",
  gpd.obj = NULL,
  gpd.path.index = NULL,
  ugg.obj = NULL,
  ...
)
```

Arguments

`x`	A star_object (output from generate.star.dataset function)
`y`	Optional numeric vector. If not NULL, it has have the same length as graph.data$y.smooth.
`point.size`	Numeric. Size of vertex points (default = 1.5)
`edge.col`	Character. Color of edges when not using color.edges (default = "gray70")
`edge.lwd`	Numeric. Line width for edges (default = 1)
`title`	Character. Plot title (default = "")
`color.palette`	Character vector. Colors for the gradient (default = c("blue", "yellow", "red"))
`color.edges`	Logical. If TRUE, edges are colored based on average function values of connected vertices (default = FALSE)
`circle.color`	Character. Color of the reference circle (default = "gray")
`pt.lab.adj`	Numeric vector of length 2. Text adjustment for vertex labels (default = c(1.0, 1.0))
`pt.lab.cex`	Numeric. Character expansion factor for vertex labels (default = 0.75)
`grid.pt.lab.adj`	Numeric vector of length 2. Text adjustment for grid point labels (default = c(1.0, 1.0))
`grid.pt.lab.cex`	Numeric. Character expansion factor for grid point labels (default = 0.75)
`grid.pt.color`	Character. Color for grid points (default = "cyan")
`grid.pt.text.color`	Character. Color for grid point text labels (default = "cyan")
`grid.pt.cex.factor`	Numeric. Size factor for grid points (default = 1.5)
`show.vertex.labs`	Logical. Whether to show vertex labels (default = FALSE)
`zero.based`	Logical. If TRUE, print vertex labels in 0-based format; otherwise 1-based (default = FALSE)
`show.arm.labs`	Logical. Whether to show arm labels (default = TRUE)
`arm.cex`	Numeric. Character expansion factor for arm labels (default = 2)
`arms.adj`	Numeric vector of length 2. Text adjustment for arm labels (default = c(-0.5, 1))
`arm.label.offset`	Numeric. It controls label distance from arm end.
`show.mu`	Logical. Whether to show the function center point (default = TRUE)
`mu.cex`	Numeric. Character expansion factor for function center point (default = 3)
`epsilon`	Numeric. Margin factor for plot boundaries (default = 0.1)
`legend.inset`	Numeric. Inset factor for legend position (default = 0.05)
`legend.bty`	Character. Box type for legend ("o" for box, "n" for no box) (default = "n")
`legend.title`	Character. Title for the color scale legend (default = "Function Value")
`axes`	Logical. Whether to show plot axes (default = FALSE)
`xlab`	Character. Label for x-axis (default = "")
`ylab`	Character. Label for y-axis (default = "")

`gpd.obj`	List. Optional path data for highlighting vertices. Each element should be a list containing $vertices (vector of vertex indices) and optionally $ref_vertex (index of reference vertex) (default = NULL)
`gpd.path.index`	Integer. Index of the path to highlight from gpd.obj (default = NULL)
`ugg.obj`	Object. Uniform grid graph object for additional visualization (default = NULL)
`...`	Additional graphical parameters (currently unused)

Details

The function creates a visualization where vertices are arranged according to the star graph structure. Vertices are colored based on their function values using the specified color palette. A reference circle is drawn to show the maximum arm length. When gpd.obj is provided, vertices in paths are highlighted with red circles and the reference vertex (if specified) is highlighted with a larger blue circle.

Value

NULL (generates a plot as a side effect)

Examples

```
# Generate basic star graph data
data <- generate.star.dataset(
  n.points = 20,
  n.arms = 5,
  min.arm.length = 1,
  max.arm.length = 3,
  dim = 2,
  fn = "exp",
  noise = "norm",
  noise.sd = 0.1
)

# Basic visualization
plot(data)

# Visualization with custom styling and path highlighting
path_data <- list(
  list(vertices = c(1, 2, 3), ref_vertex = 1)
)
plot(data,
  point.size = 2,
  color.edges = TRUE,
  show.vertex.labs = TRUE,
  gpd.obj = path_data
)
```

`plot.ugkmm` *Plot ugkmm Objects*

Description

Visualizes results from adaptive neighborhood size graph k-means regression, including fitted values, credible intervals, residuals, and diagnostics.

Usage

```
## S3 method for class 'ugkmm'
plot(
  x,
  type = c("fit", "diagnostic", "residuals", "residuals_hist"),
  main = "",
  xlab = "",
  ylab = "",
  ...
)
```

Arguments

`x`	An object of class "ugkmm".
`type`	Plot type: "fit" (default), "diagnostic", "residuals", or "residuals_hist".
`main`	Plot title.
`xlab`	X-axis label.
`ylab`	Y-axis label.
`...`	Additional graphical parameters.

Details

Plot types:

- "fit": Shows fitted curve with optional credible intervals
- "diagnostic": Cross-validation error vs neighborhood size
- "residuals": Residuals vs fitted values
- "residuals_hist": Histogram of residuals

Value

NULL (invisibly)

Examples

```
# See examples in univariate.gkmm
```

`plot.upgmalo` *Plot Method for upgmalo Objects*

Description

Creates comprehensive visualizations for univariate PGMALO results including fitted values with confidence bands, diagnostic plots, and residual analysis.

Usage

```
## S3 method for class 'upgmalo'
plot(
  x,
  type = c("fit", "diagnostic", "residuals", "residuals_hist"),
  title = "",
  xlab = "",
  ylab = "",
  predictions.type = c("predictions", "bb.predictions", "local.predictions"),
  with.y.true = TRUE,
  with.pts = FALSE,
  with.CrI = TRUE,
  y.true.col = "red",
  predictions.col = "blue",
  with.predictions.pts = FALSE,
  predictions.pts.col = "blue",
  predictions.pts.pch = 20,
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray85",
  CrI.line.col = "gray10",
  CrI.line.lty = 2,
  ylim = NULL,
  ...
)
```

Arguments

`x`	An object of class "upgmalo" returned by `upgmalo`.
`type`	Character string specifying the plot type: **"fit"** Fitted values with optional confidence bands (default) **"diagnostic"** Cross-validation error by neighborhood size **"residuals"** Residual scatter plot **"residuals_hist"** Histogram of residuals
`title`	Main title for the plot (default: "").
`xlab`	X-axis label (default: "x" for most plots).
`ylab`	Y-axis label (default: "y" for fit plot, appropriate label for others).
`predictions.type`	Character string specifying which predictions to plot: **"predictions"** Global optimal h predictions (default)

	"bb.predictions" Bootstrap-based predictions
	"local.predictions" Locally adaptive predictions
`with.y.true`	Logical; include true values if available (default: TRUE).
`with.pts`	Logical; show original data points (default: FALSE).
`with.CrI`	Logical; show credible/confidence intervals if available (default: TRUE).
`y.true.col`	Color for true values line (default: "red").
`predictions.col`	Color for predictions line (default: "blue").
`with.predictions.pts`	Logical; show prediction points (default: FALSE).
`predictions.pts.col`	Color for prediction points (default: "blue").
`predictions.pts.pch`	Point character for predictions (default: 20).
`CrI.as.polygon`	Logical; show confidence band as polygon (default: TRUE).
`CrI.polygon.col`	Color for confidence band polygon (default: "gray85").
`CrI.line.col`	Color for confidence band lines if not polygon (default: "gray10").
`CrI.line.lty`	Line type for confidence bands if not polygon (default: 2).
`ylim`	Y-axis limits (default: NULL for automatic).
`...`	Additional graphical parameters passed to base plotting functions.

Details

The function provides multiple visualization options:

- Fit plots show the estimated function with optional confidence bands
- Diagnostic plots help assess the cross-validation process
- Residual plots aid in model checking

Confidence intervals are only shown for "predictions" and "bb.predictions" types when bootstrap was performed (n.bb > 0 in the original call).

Value

NULL (invisibly). Called for side effect of creating plots.

See Also

`upgmalo` for model fitting, `summary.upgmalo` for numerical summaries

Examples

```
## Not run:
# Generate example data
n <- 100
x <- seq(0, 2*pi, length.out = n)
y <- sin(x) + rnorm(n, 0, 0.2)

# Fit model
```

```
fit <- upgmalo(x, y, n.bb = 100)

# Create various plots
plot(fit, type = "fit", with.pts = TRUE)
plot(fit, type = "diagnostic")
plot(fit, type = "residuals")

## End(Not run)
```

plot.vertex_geodesic_stats *Plot method for vertex_geodesic_stats objects*

Description

Plot method for vertex_geodesic_stats objects

Usage

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

Arguments

x	A vertex_geodesic_stats object
...	Additional arguments passed to plot

Value

Invisibly returns NULL

plot2D.cltr.profiles *Plot Cluster Profile Lines*

Description

Creates a line plot showing profiles of all members in a cluster

Usage

```
plot2D.cltr.profiles(X, cltr, id, xlab = "", ylab = "")
```

Arguments

X	Matrix or data frame where rows are observations and columns are variables.
cltr	Vector of cluster assignments.
id	Cluster ID to plot.
xlab	Label for x-axis.
ylab	Label for y-axis.

Details

This function creates a profile plot showing all members of a specified cluster as gray lines, with the median profile highlighted in red. Missing values are replaced with zeros.

Value

Invisibly returns the median profile.

Examples

```
## Not run:
X <- matrix(rnorm(100), ncol = 10)
cltr <- sample(1:3, 10, replace = TRUE)
plot2D.cltr.profiles(X, cltr, id = 2, xlab = "Variables", ylab = "Values")

## End(Not run)
```

`plot2D.colored.graph` *Plot a Graph with Colored Vertices*

Description

Creates a visualization of a graph where vertices are colored according to a numeric function value, using base R graphics. The function displays the graph structure with edges as lines and vertices as colored points, with an optional color scale legend.

Usage

```
plot2D.colored.graph(
  embedding,
  adj.list,
  vertex.colors,
  vertex.size = 1,
  edge.alpha = 0.2,
  color.palette = NULL,
  main = "",
  add.legend = TRUE
)
```

Arguments

`embedding`	A numeric matrix with dimensions n x 2, where n is the number of vertices. Contains the 2D coordinates for each vertex, typically generated by a graph embedding algorithm.
`adj.list`	A list of length n, where each element i contains a numeric vector of indices representing the vertices adjacent to vertex i.
`vertex.colors`	A numeric vector of length n containing the function values used to color the vertices.
`vertex.size`	Numeric scalar controlling the size of vertices (default: 1). Uses the same scale as base R's cex parameter.

`edge.alpha`	Numeric scalar between 0 and 1 controlling edge transparency (default: 0.2). Higher values make edges more opaque.
`color.palette`	Optional vector of colors defining the color gradient for vertices. If NULL (default), uses a blue-white-red gradient.
`main`	Character string for the plot title (default: "").
`add.legend`	Logical indicating whether to add a color scale legend (default: TRUE).

Details

The function creates a 2D visualization of a graph structure where: - Edges are drawn as semi-transparent lines - Vertices are drawn as colored points - Colors are mapped to vertex_colors values using a continuous gradient - The aspect ratio is maintained at 1:1 - Axes and tick marks are suppressed An optional color scale legend can be added to interpret vertex colors.

Value

No return value; produces a plot as a side effect.

Note

The function temporarily modifies graphical parameters using par() but restores them before exiting.

Examples

```
## Not run:
# Basic usage with simulated data
n <- 100  # number of vertices
embedding <- matrix(rnorm(2*n), ncol=2)
adj.list <- lapply(1:n, function(i) sample(1:n, 5))
vertex.colors <- rnorm(n)
plot.colored.graph(embedding, adj.list, vertex.colors)

# Custom styling
plot2D.colored.graph(
    embedding,
    adj.list,
    vertex.colors,
    vertex.size = 1.5,
    edge.alpha = 0.3,
    color.palette = colorRampPalette(c("navy", "white", "darkred"))(100),
    main = "My Graph",
    add.legend = TRUE
)

## End(Not run)
```

`plot2D.cont` *Create 2D Plot with Continuous Color Coding*

Description

Creates a 2D plot of points color coded by the values of a continuous variable

Usage

```
plot2D.cont(
  X,
  y,
  legend.title = "",
  legend.cex = 0.7,
  legend.loc = NULL,
  legend.line = 0.5,
  legend.bty = "o",
  use.layout = TRUE,
  layout.widths = c(7, 1.5),
  quantize.method = "uniform",
  quantize.wins.p = 0.02,
  quantize.round = FALSE,
  quantize.dig.lab = 2,
  start = 1/6,
  end = 0,
  n.levels = 10,
  axes = FALSE,
  pch = 20,
  ...
)
```

Arguments

`X`	A matrix or data.frame with at least 2 columns. Only the first 2 columns are used.
`y`	A numeric vector of values for color coding.
`legend.title`	Title for the legend.
`legend.cex`	Character expansion factor for legend text.
`legend.loc`	Location of the legend. If NULL, defaults to "topleft".
`legend.line`	Line parameter for the legend.
`legend.bty`	Box type for the legend.
`use.layout`	Logical. If TRUE, uses layout() to position the legend separately.
`layout.widths`	Numeric vector of length 2 specifying relative widths when use.layout is TRUE.
`quantize.method`	Method for quantizing y values: "uniform" or "quantile".
`quantize.wins.p`	Winsorization parameter for the "uniform" method.
`quantize.round`	Logical. Whether to round the quantization endpoints.

`quantize.dig.lab`	Number of digits for quantization labels.
`start`	Start value for color range.
`end`	End value for color range.
`n.levels`	Number of color levels.
`axes`	Logical. Whether to draw axes.
`pch`	Plotting character.
`...`	Additional graphical parameters passed to `plot`.

Details

This function creates a 2D scatter plot where points are colored according to a continuous variable. The continuous variable is discretized into categories for visualization. The legend can be positioned using standard graphics layout or a custom layout approach.

Value

Invisibly returns a list containing:

`y.cols`	Vector of colors assigned to each data point
`y.col.tbl`	Color table mapping categories to colors
`legend.labs`	Labels for the legend
`legend.out`	Output from legend() function

Examples

```
## Not run:
# Generate sample data
X <- matrix(rnorm(200), ncol = 2)
y <- runif(100)

# Basic plot
plot2D.cont(X, y)

# Plot with custom legend position and no axes
plot2D.cont(X, y, legend.title = "Values", axes = FALSE)

## End(Not run)
```

`plot2D.node.level.props` *Plot Node Level Proportions on Graph*

Description

Visualizes proportions of factor levels within graph nodes

Usage

```
plot2D.node.level.props(
  adj.mat,
  props,
  color.tbl = NULL,
  layout = "auto",
  epsilon = 0.1,
  legend.pos = "topleft"
)
```

Arguments

`adj.mat`	Adjacency matrix of a graph.
`props`	Matrix of proportions where rows are nodes and columns are factor levels.
`color.tbl`	Vector of colors for factor levels. If NULL, colors are assigned automatically.
`layout`	Character string specifying layout algorithm or a matrix of coordinates. Options include "auto", "circle", "sphere", "dh", "fr", "kk", "lgl", "mds", "sugiyama", "star", "tree", "grid", "random". Default is "auto".
`epsilon`	Scaling factor for the radius of proportion circles within nodes.
`legend.pos`	Position of the legend (e.g., "topleft", "topright").

Details

This function creates a network visualization where each node displays a pie chart showing the proportions of different factor levels. The network structure is determined by the adjacency matrix.

Value

Invisibly returns the graph layout.

Examples

```
## Not run:
# Create example adjacency matrix
adj.mat <- matrix(0, 5, 5)
adj.mat\code{[1,2]} <- adj.mat\code{[2,1]} <- 1
adj.mat\code{[2,3]} <- adj.mat\code{[3,2]} <- 1

# Create proportions matrix
props <- matrix(runif(15), ncol = 3)
props <- props / rowSums(props)
colnames(props) <- c("Type A", "Type B", "Type C")

plot2D.node.level.props(adj.mat, props)

## End(Not run)
```

`plot2D.tree` *Add Minimal Spanning Tree to 2D Plot*

Description

Adds edges of a minimal spanning tree to an existing 2D plot

Usage

```
plot2D.tree(X, T.edges, col = "gray")
```

Arguments

`X`	Matrix with at least 2 columns. Only first 2 columns are used.
`T.edges`	Matrix with 2 columns containing start and end indices of edges.
`col`	Color of the edges.

Value

Invisibly returns NULL.

Examples

```
## Not run:
X <- matrix(rnorm(20), ncol = 2)
T.edges <- matrix(c(1,2, 2,3, 3,4, 4,5), ncol = 2, byrow = TRUE)

plot(X, pch = 19)
plot2D.tree(X, T.edges)

## End(Not run)
```

`plot3d.box.tiling` *Plots boxes of a box tiling of a 3D state space.*

Description

This function creates a 3D visualization of box tilings using the rgl package.

Usage

```
## S3 method for class 'box.tiling'
plot3d(x, open3d = TRUE, col = "gray", ...)
```

Arguments

`x`	A list of sub-boxes, as returned by `create.ED.boxes`.
`open3d`	Logical. Whether to open a new 3D plotting device (default = TRUE).
`col`	Character. Color of the box edges (default = "gray").
`...`	Additional arguments passed to rgl plotting functions.

`plot3D.cl` *Plot a Specific Cluster with Custom Colors*

Description

Creates a 3D plot highlighting one or more specific clusters

Usage

```
plot3D.cl(cl, cltr, X, ...)
```

Arguments

`cl`	Cluster ID(s) to highlight. Can be a single value or vector.
`cltr`	Vector of cluster IDs.
`X`	A matrix or data.frame with 3 columns representing 3D coordinates.
`...`	Additional arguments passed to `plot3D.cltrs`.

Details

This function creates a 3D plot where specified clusters are highlighted with distinct colors while other clusters are shown in gray shades. If multiple clusters are specified, they are colored using the mclust color scheme.

Value

Invisibly returns the output from plot3D.cltrs.

Examples

```
## Not run:
X <- matrix(rnorm(300), ncol = 3)
cltr <- sample(1:5, 100, replace = TRUE)

# Highlight single cluster
plot3D.cl(2, cltr, X)

# Highlight multiple clusters
plot3D.cl(c(2, 4), cltr, X)

## End(Not run)
```

`plot3D.cltrs` *Create 3D Plot with Cluster Visualization*

Description

Creates a 3D plot of points with cluster assignments shown by different colors

Usage

```
plot3D.cltrs(
  X,
  cltr = NULL,
  cltr.col.tbl = NULL,
  ref.cltr = NULL,
  ref.cltr.color = "gray",
  show.ref.cltr = TRUE,
  show.cltr.labels = FALSE,
  add = FALSE,
  title = "",
  cex.labs = 2,
  pal.type = "numeric",
  brewer.pal.n = 3,
  brewer.pal = "Set1",
  open.3d = FALSE,
  show.legend = TRUE,
  sort.legend.labs.by.freq = FALSE,
  sort.legend.labs.by.name = FALSE,
  filter.out.freq.0.cltrs = TRUE,
  legend.title = NULL,
  radius = NA,
  axes = FALSE,
  xlab = "",
  ylab = "",
  zlab = "",
  ...
)
```

Arguments

`X`	A 3D matrix or data.frame with exactly 3 columns.
`cltr`	A vector of cluster IDs (numeric or character) of length nrow(X).
`cltr.col.tbl`	A named vector mapping cluster IDs to colors. If NULL, colors are assigned automatically.
`ref.cltr`	A reference cluster ID (usually ’0’) that can be colored differently.
`ref.cltr.color`	The color for the reference cluster.
`show.ref.cltr`	Logical. Whether to display the reference cluster.
`show.cltr.labels`	Logical. Whether to show cluster labels in the plot.
`add`	Logical. Whether to add to an existing plot.

`title`	Title of the plot.
`cex.labs`	Size scaling parameter for cluster labels.
`pal.type`	Palette type: "numeric", "brewer", or "mclust".
`brewer.pal.n`	Number of colors in the RColorBrewer palette.
`brewer.pal`	Name of the RColorBrewer palette.
`open.3d`	Logical. Whether to open a new 3D window.
`show.legend`	Logical. Whether to show the legend.
`sort.legend.labs.by.freq`	Logical. Sort legend labels by frequency.
`sort.legend.labs.by.name`	Logical. Sort legend labels alphabetically.
`filter.out.freq.0.cltrs`	Logical. Remove clusters with zero frequency from legend.
`legend.title`	Title for the legend.
`radius`	Numeric. Size of spheres at data points.
`axes`	Logical. Whether to show axes.
`xlab, ylab, zlab`	Axis labels.
`...`	Additional arguments passed to `plot3d`.

Details

This function provides comprehensive cluster visualization in 3D space with automatic color assignment, legend generation, and various customization options. It supports highlighting specific clusters and different color palette options.

Value

Invisibly returns a list containing:

`ids`	RGL object IDs
`cltr.col.tbl`	Color table used for clusters
`cltr.labs`	Cluster labels
`legend.cltr.labs`	Legend labels
`cltr.centers`	Matrix of cluster centers (if show.cltr.labels = TRUE)

Examples

```
## Not run:
# Generate sample data
set.seed(123)
X <- matrix(rnorm(300), ncol = 3)
cltr <- sample(c("A", "B", "C"), 100, replace = TRUE)

# Basic cluster plot
plot3D.cltrs(X, cltr)

# Plot with reference cluster
cltr[1:10] <- "0"
plot3D.cltrs(X, cltr, ref.cltr = "0", show.cltr.labels = TRUE)

## End(Not run)
```

`plot3D.cont` *Create 3D Plot with Continuous Color Coding*

Description

Creates a 3D plot of a set of points defined by X color coded by the values of y

Usage

```
plot3D.cont(
  X,
  y,
  subset = NULL,
  non.highlight.type = "sphere",
  non.highlight.color = "gray",
  point.size = 3,
  legend.title = "",
  legend.cex = 2,
  legend.side = 3,
  legend.line = 0.5,
  radius = NULL,
  quantize.method = "uniform",
  quantize.wins.p = 0.02,
  quantize.round = FALSE,
  quantize.dig.lab = 2,
  start = 1/6,
  end = 0,
  n.levels = 10
)
```

Arguments

`X`	A matrix or data frame with 3 columns representing 3D coordinates.
`y`	A numeric vector of values to be used for color coding.
`subset`	A logical vector indicating which points to highlight (default is NULL).
`non.highlight.type`	Character, either "sphere" or "point" for non-highlighted points (default is "sphere").
`non.highlight.color`	Character, indicating the color of non-highlighted points (default is "gray").
`point.size`	Numeric, size of points when non.highlight.type is "point" (default is 3).
`legend.title`	A string for the legend title (default is "").
`legend.cex`	A numeric value for the legend text size (default is 2).
`legend.side`	The side of the plot for the legend (default is 3).
`legend.line`	The line position for the legend (default is 0.5).
`radius`	The radius of spheres for 3D plotting (default is NULL).
`quantize.method`	Method for quantizing y values: "uniform" or "quantile" (default is "uniform").

`quantize.wins.p`	Winsorization parameter for the "uniform" method (default is 0.02).
`quantize.round`	Logical, whether to round the quantization endpoints (default is FALSE).
`quantize.dig.lab`	Number of digits for quantization labels (default is 2).
`start`	Start value for color range (default is 1/6).
`end`	End value for color range (default is 0).
`n.levels`	Number of color levels (default is 10).

Details

This function creates a 3D scatter plot where points are colored according to a continuous variable. The continuous variable is discretized into categories for visualization. When a subset is specified, highlighted points are shown as spheres while non-highlighted points can be shown as either smaller spheres or points.

Value

Invisibly returns a list containing:

`y.cols`	Vector of colors assigned to each data point
`y.col.tbl`	Color table mapping categories to colors
`legend.labs`	Labels for the legend

See Also

`plot3D.plain`, `plot3d`, `spheres3d`

Examples

```
## Not run:
# Generate sample data
set.seed(123)
X <- matrix(rnorm(300), ncol = 3)
y <- runif(100)
some.logical.vector <- y > median(y)

# Plot all points as spheres (original behavior)
plot3D.cont(X, y)

# Plot highlighted points as spheres, non-highlighted as points
plot3D.cont(X, y, subset = some.logical.vector, non.highlight.type = "point")

## End(Not run)
```

plot3D.diskEmbdg *Plot 3d Disk Embedding Object*

Description

Visualizes a disk embedding object in 3D space

Usage

```
plot3D.diskEmbdg(
  ebdg.obj,
  col = "blue",
  radius = 0.005,
  vertex.radius = 0.02,
  vertex.col = "red",
  vertex.cex = 1.2,
  edge.col = "gray",
  adj.df = NULL
)
```

Arguments

`ebdg.obj`	A diskEmbdg object containing embedding information.
`col`	Color of spheres representing data points.
`radius`	Radius of data point spheres.
`vertex.radius`	Radius of vertex spheres.
`vertex.col`	Color of vertex spheres.
`vertex.cex`	Character expansion factor for vertex labels.
`edge.col`	Color of edges.
`adj.df`	Matrix of text adjustment values for vertex labels.

Details

This function visualizes a disk embedding where data is projected onto a sphere with axes represented as points on the sphere and one at the center.

Value

Invisibly returns NULL.

Examples

```
## Not run:
# Assuming ebdg.obj is a properly formatted disk embedding object
plot3D.diskEmbdg(ebdg.obj, col = "blue", vertex.col = "red")

## End(Not run)
```

plot3d.gaussian_mixture *Create 3D interactive plot of gaussian_mixture*

Description

Create 3D interactive plot of gaussian_mixture

Usage

```
## S3 method for class 'gaussian_mixture'
plot3d(x, n.grid = 50, col = "skyblue", alpha = 0.7, add.contours = TRUE, ...)
```

Arguments

`x`	A gaussian_mixture object
`n.grid`	Number of grid points in each dimension
`col`	Surface color
`alpha`	Surface transparency
`add.contours`	Whether to add contour lines at the base
`...`	Additional arguments passed to rgl functions

Value

Invisible surface object

plot3D.geodesic *Plot Geodesic Path in 3D*

Description

Plots the shortest path between two vertices in a graph

Usage

```
plot3D.geodesic(S.3d, i1, i2, G, X, edge.col = "gray")
```

Arguments

`S.3d`	Matrix of 3D vertex positions.
`i1`	Index of the starting vertex.
`i2`	Index of the ending vertex.
`G`	An igraph object representing the graph structure.
`X`	Original state space (currently unused but kept for compatibility).
`edge.col`	Color of the geodesic path.

Details

This function finds the shortest path between two vertices using igraph's shortest_paths function and visualizes it in 3D space.

Value

Invisibly returns the sequence of vertex indices in the shortest path.

Examples

```
## Not run:
library(igraph)
# Create a simple graph
G <- make_ring(10)
S.3d <- matrix(rnorm(30), ncol = 3)

plot3D.plain(S.3d)
plot3D.geodesic(S.3d, 1, 5, G, S.3d)

## End(Not run)
```

`plot3D.path` *Plot Path in 3D Graph*

Description

Plots a sequence of edges forming a path in a graph with 3D vertex positions

Usage

```
plot3D.path(s, V, edge.col = "gray")
```

Arguments

`s`	Vector of vertex indices defining the path.
`V`	Matrix of vertex positions with 3 columns.
`edge.col`	Color of the path edges.

Details

This function draws line segments connecting consecutive vertices in the specified path sequence.

Value

Invisibly returns NULL.

Examples

```
## Not run:
V <- matrix(rnorm(15), ncol = 3)
s <- c(1, 3, 2, 5, 4)

plot3D.plain(V)
plot3D.path(s, V, edge.col = "red")

## End(Not run)
```

---

plot3D.plain *Plot 3D Points or Spheres Without Axes*

Description

This function creates a 3D plot of points or spheres without axes labels using the rgl package. It automatically switches between spheres and points based on whether a radius is specified.

Usage

```
plot3D.plain(X, radius = NULL, col = "gray", size = 3, ...)
```

Arguments

`X`	A numeric matrix or data frame with exactly 3 columns representing x, y, and z coordinates.
`radius`	Numeric value for the radius of spheres. If NULL (default), points will be plotted instead of spheres.
`col`	Color for the points or spheres. Default is "gray".
`size`	Numeric value for the size of points when plotting points (radius = NULL). Default is 3.
`...`	Additional arguments passed to `plot3d`.

Details

The function creates a 3D plot without axes or labels. If radius is specified, it plots spheres using type="s"; otherwise, it plots points using type="p".

Value

Invisibly returns NULL. The function is called for its side effect of creating a plot.

Examples

```
## Not run:
# Create sample 3D data
X <- matrix(rnorm(30), ncol = 3)

# Plot as points
plot3D.plain(X)
```

```
# Plot as spheres
plot3D.plain(X, radius = 0.2, col = "blue")

## End(Not run)
```

`plot3D.tree` *Add Minimal Spanning Tree to 3D Plot*

Description

Adds edges of a minimal spanning tree to an existing 3D plot

Usage

```
plot3D.tree(X, T.edges, col = "gray", lwd = 1)
```

Arguments

`X`	Matrix with 3 columns representing 3D coordinates.
`T.edges`	Matrix with 2 columns containing start and end indices of edges.
`col`	Color of the edges.
`lwd`	Line width of the edges.

Value

Invisibly returns NULL.

Examples

```
## Not run:
X <- matrix(rnorm(30), ncol = 3)
# Assuming T.edges is computed from a minimal spanning tree algorithm
T.edges <- matrix(c(1,2, 2,3, 3,4, 4,5), ncol = 2, byrow = TRUE)

plot3D.plain(X)
plot3D.tree(X, T.edges)

## End(Not run)
```

plotlcor.1D *Plot Output from lcor.1D()*

Description

Creates a visualization of local correlation results from the lcor.1D() function

Usage

```
plotlcor.1D(
  r,
  with.CrI = TRUE,
  use.smoothed.lcors = TRUE,
  title = "",
  xlab = "",
  ylab = "Local Pearson correlation",
  lcor.col = "blue",
  CrI.as.polygon = TRUE,
  CrI.polygon.col = "gray95",
  CrI.line.col = "gray",
  add.box = FALSE,
  add.x.axis = FALSE,
  add.y.axis = FALSE,
  ...
)
```

Arguments

`r`	An output from lcor.1D() function.
`with.CrI`	A logical parameter. If TRUE, credible intervals are plotted.
`use.smoothed.lcors`	A logical parameter. If TRUE, smoothed local correlations will be used (if available) instead of raw local correlations.
`title`	The title of the plot.
`xlab`	The x-axis label.
`ylab`	The y-axis label.
`lcor.col`	The color of local correlation line.
`CrI.as.polygon`	A logical value parameter. If TRUE, credible interval region is drawn as a polygon. Otherwise, credible intervals are drawn as contour lines.
`CrI.polygon.col`	The color of the credible interval polygon region.
`CrI.line.col`	The color of the credible interval upper and lower limit lines.
`add.box`	Set to TRUE if box() is to be called after plot(); useful when axes is FALSE.
`add.x.axis`	Set to TRUE if axis(1) is to be called after plot().
`add.y.axis`	Set to TRUE if axis(2) is to be called after plot().
`...`	Graphics parameters passed to `plot`.

Details

This function visualizes the output from lcor.1D(), which typically contains local correlation estimates along a one-dimensional variable. The function can display either raw or smoothed local correlations, with optional credible intervals shown either as a shaded polygon or as contour lines.

Value

Invisibly returns NULL. The function is called for its side effect of creating a plot.

Examples

```
## Not run:
# Assuming 'result' is output from lcor.1D()
plotlcor.1D(result, with.CrI = TRUE, title = "Local Correlations")

# Plot with credible intervals as lines instead of polygon
plotlcor.1D(result, with.CrI = TRUE, CrI.as.polygon = FALSE)

## End(Not run)
```

`points3d.select` *Select Points in 3D Space Using RGL*

Description

A modified version of `rgl::selectpoints3d()` that provides more stable point selection in 3D visualizations without crashing.

Usage

```
points3d.select(
  objects = rgl::ids3d()$id,
  value = TRUE,
  closest = TRUE,
  multiple = TRUE,
  ...
)
```

Arguments

`objects`	A vector of rgl object ID values to search within. Defaults to all objects in the current rgl scene.
`value`	Logical. If `TRUE` (default), returns the coordinates of selected points. If `FALSE`, returns their indices.
`closest`	Logical. If `TRUE` (default), returns the points closest to the selection region when no points fall exactly within it.
`multiple`	Logical or function. If `TRUE`, allows multiple selections. If a function, it should accept the current selection and return `TRUE` to continue selecting or `FALSE` to stop.
`...`	Additional parameters passed to `rgl::select3d()`.

Details

This function provides an interactive method for selecting points in 3D space. It improves upon the original `selectpoints3d()` by adding better error handling and preventing crashes that could occur with certain object configurations.

Value

If `value` = `TRUE`, returns a matrix with columns 'x', 'y', 'z' containing the coordinates of selected points. If `value` = `FALSE`, returns a matrix with columns 'id' and 'index' identifying the selected points within their respective objects.

Examples

```
## Not run:
library(rgl)
# Create a simple 3D scatter plot
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
plot3d(x, y, z)

# Select points interactively
selected <- points3d.select()

# Highlight selected points
points3d(selected, col = "red", size = 10)

## End(Not run)
```

`predict.eigen.ulogit` *Predict Method for eigen.ulogit Objects*

Description

Obtain predictions from a fitted eigen.ulogit model

Usage

```
## S3 method for class 'eigen.ulogit'
predict(object, newdata = NULL, type = c("response", "link"), ...)
```

Arguments

`object`	An object of class `"eigen.ulogit"`
`newdata`	Optional numeric vector of new x values for prediction. If omitted, fitted values are returned.
`type`	Character string specifying the type of prediction. Either `"response"` for probabilities or `"link"` for linear predictors.
`...`	Additional arguments (currently ignored)

Value

Numeric vector of predictions

---

predict.graph.spectral.lowess
Predict Method for Graph Spectral LOWESS

Description

Makes predictions from a graph spectral LOWESS fit. For new data, this would require extending the graph structure.

Usage

```
## S3 method for class 'graph.spectral.lowess'
predict(object, newdata, se.fit = FALSE, ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`newdata`	Optional new data (currently not implemented)
`se.fit`	Logical; should standard errors be returned? (currently not implemented)
`...`	Additional arguments (currently unused)

Value

If newdata is missing, returns fitted values. Otherwise, not yet implemented.

---

predict.graph.spectral.ma.lowess
Predict Method for Graph Spectral MA LOWESS

Description

Makes predictions from a graph spectral model-averaged LOWESS fit. For new data, this would require extending the graph structure and model averaging.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
predict(object, newdata, se.fit = FALSE, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`newdata`	Optional new data (currently not implemented)
`se.fit`	Logical; should standard errors be returned? (currently not implemented)
`...`	Additional arguments (currently unused)

Value

If newdata is missing, returns fitted values. Otherwise, not yet implemented.

`predict.graph_spectral_filter` *Predict Method for Graph Spectral Filter*

Description

Predict Method for Graph Spectral Filter

Usage

```
## S3 method for class 'graph_spectral_filter'
predict(object, newdata = NULL, t = NULL, ...)
```

Arguments

`object`	An object of class "`graph_spectral_filter`"
`newdata`	Optional numeric vector of new signal values. If NULL, returns the filtered signal from the original fit
`t`	Optional filter parameter value. If NULL, uses the optimal value from the original fit
`...`	Further arguments (currently ignored)

Value

Numeric vector of filtered signal values

`predict.mabilog` *Predict Method for Mabilog Objects*

Description

Predict probability values for new data using a fitted mabilog model

Usage

```
## S3 method for class 'mabilog'
predict(object, newdata, type = c("response", "link"), k = NULL, ...)
```

Arguments

`object`	A 'mabilog' object
`newdata`	Numeric vector of new x values for prediction
`type`	Character string specifying the type of prediction: "response" for probabilities (default), "link" for logit scale
`k`	Optional specific k value to use for predictions. If NULL, uses optimal k value
`...`	Additional arguments (currently unused)

Details

This function performs local weighted logistic regression predictions for new data points. For each new x value, it finds the k nearest neighbors from the training data and performs a weighted logistic regression.

Note: This is a simplified implementation. Full local logistic regression would require iterative fitting at each prediction point.

Value

Numeric vector of predictions

---

predict.mabilo_plus *Predict Method for Mabilo Plus Objects*

Description

Predict response values for new data using a fitted mabilo_plus model

Usage

```
## S3 method for class 'mabilo_plus'
predict(object, newdata, type = c("ma", "sm", "both"), k = NULL, ...)
```

Arguments

`object`	A 'mabilo_plus' object
`newdata`	Numeric vector of new x values for prediction
`type`	Character string specifying which predictions to return: "ma" (default), "sm", or "both"
`k`	Optional specific k value to use for predictions. If NULL, uses optimal k values
`...`	Additional arguments (currently unused)

Details

This function performs local weighted regression predictions for new data points. For each new x value, it finds the k nearest neighbors from the training data and performs a weighted local regression.

Value

Numeric vector or matrix of predictions

predict.maelog *Predict Method for maelog Objects*

Description

Obtains predictions from a fitted `maelog` object at new predictor values.

Usage

```
## S3 method for class 'maelog'
predict(object, newdata, type = c("response", "link"), se.fit = FALSE, ...)
```

Arguments

`object`	A fitted object of class `"maelog"`.
`newdata`	Numeric vector of new predictor values at which to make predictions. If missing, predictions at the original data points are returned.
`type`	Character string specifying the type of prediction: `"response"` for predicted probabilities (default) or `"link"` for linear predictors on the logit scale.
`se.fit`	Logical; if `TRUE`, returns standard errors of predictions (only available if the model was fit with `with.errors = TRUE`).
`...`	Additional arguments (currently ignored).

Value

If `se.fit = FALSE`, a numeric vector of predictions. If `se.fit = TRUE`, a list with components:

`fit`	Vector of predictions.
`se.fit`	Vector of standard errors (if available).

See Also

`maelog`

Examples

```
## Not run:
# Fit model
set.seed(123)
x <- seq(0, 1, length.out = 100)
y <- rbinom(100, 1, plogis(10 * (x - 0.5)))
fit <- maelog(x, y, with.errors = TRUE)

# Predictions at new points
x.new <- seq(0.2, 0.8, by = 0.1)
pred <- predict(fit, x.new)

# Predictions with standard errors
pred.se <- predict(fit, x.new, se.fit = TRUE)

## End(Not run)
```

`predict.magelo` *Predicting values from a magelo model*

Description

Predicting values from a magelo model

Usage

```
## S3 method for class 'magelo'
predict(object, newdata, ...)
```

Arguments

`object`	an object of class "magelo", typically the result of a call to `magelo`
`newdata`	a numeric vector of values at which predictions are required
`...`	additional arguments (currently unused)

Value

numeric vector of predicted values

Examples

```
## Not run:
model <- magelo(x, y)
predictions <- predict(model, newdata = c(1, 2, 3))

## End(Not run)
```

`predict.magelog` *Predict Method for magelog Objects*

Description

Obtains predictions from a fitted magelog model. Can predict at the original data points or at new x values.

Usage

```
## S3 method for class 'magelog'
predict(object, newdata = NULL, type = c("response", "logit"), ...)
```

Arguments

`object`	A fitted model object of class `"magelog"`
`newdata`	Numeric vector of new x values for prediction. If `NULL` (default), predictions are returned for the original x values.
`type`	Character string specifying the type of prediction: `"response"` Predicted probabilities (default) `"logit"` Predicted values on the logit scale
`...`	Additional arguments (currently ignored)

Details

Predictions are obtained by linear interpolation from the grid-based predictions computed during model fitting. For x values outside the range of the original data, predictions are extrapolated using the nearest grid point values (i.e., constant extrapolation).

Value

A numeric vector of predictions

Examples

```
## Not run:
# Fit model
fit <- magelog(x, y)

# Predictions at original points
pred <- predict(fit)

# Predictions at new points
x_new <- seq(min(x), max(x), length.out = 50)
pred_new <- predict(fit, newdata = x_new)

## End(Not run)
```

`prediction.errors` *Create a prediction.errors object*

Description

Constructor function for creating a prediction.errors object that stores model prediction errors along with optional x-values for plotting or analysis.

Usage

```
prediction.errors(errors, xvals = NULL)
```

Arguments

`errors`	Numeric vector of prediction errors
`xvals`	Optional numeric vector of x-values corresponding to the errors. Default is NULL.

Value

An object of class "prediction.errors" containing the errors with xvals stored as an attribute

print.amagelo *Print method for amagelo objects*

Description

Print method for amagelo objects

Usage

```
## S3 method for class 'amagelo'
print(x, digits = 4, ...)
```

Arguments

x	An object of class ’amagelo’
digits	Number of digits to display (default: 4)
...	Additional arguments passed to print

print.assoc0 *Print Method for assoc0 Objects*

Description

Print Method for assoc0 Objects

Usage

```
## S3 method for class 'assoc0'
print(x, digits = 4, ...)
```

Arguments

x	An object of class "assoc0".
digits	Number of significant digits to display.
...	Additional arguments (currently ignored).

Value

Invisible x.

`print.assoc1` *Print Method for assoc1 Objects*

Description

Print Method for assoc1 Objects

Usage

```
## S3 method for class 'assoc1'
print(x, digits = 4, ...)
```

Arguments

`x`	An object of class "assoc1".
`digits`	Number of significant digits to display.
`...`	Additional arguments (currently ignored).

Value

Invisible x.

`print.basin_cx` *Print Method for Basin Complex Objects*

Description

Prints a concise summary of a gradient flow basin complex object, displaying key statistics about merged minima/maxima, function value ranges, and cell complex structure.

Usage

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

Arguments

`x`	An object of class "basin_cx" as created by `create.basin.cx`.
`...`	Additional arguments passed to print methods (currently unused).

Details

The print method displays:

- Number of merged minima (ascending basins)
- Number of merged maxima (descending basins)
- Range of original function values
- Cell complex summary including total number of cells (if available)

For more detailed information about the basin complex, use `summary()`.

Value

Invisibly returns the input object `x`. This function is called for its side effect of printing a summary to the console.

See Also

`create.basin.cx` for creating basin complex objects, `summary.basin_cx` for detailed basin complex information

Examples

```
## Not run:
# Create a basin complex
adj_list <- list(c(2,3), c(1,3,4), c(1,2,5), c(2,5), c(3,4))
weight_list <- list(c(1,2), c(1,1,3), c(2,1,2), c(3,1), c(2,1))
y <- c(2.5, 1.8, 3.2, 0.7, 2.1)
basin_cx <- create.basin.cx(adj_list, weight_list, y)

# Print basic information
print(basin_cx)
# or simply:
basin_cx

## End(Not run)
```

`print.circular_parameterization` *Print Method for circular_parameterization Objects*

Description

Prints a summary of a circular parameterization result.

Usage

```
## S3 method for class 'circular_parameterization'
print(x, digits = 4, ...)
```

Arguments

`x`	An object of class `"circular_parameterization"` as returned by `parameterize.circular.graph`
`digits`	Number of digits to display for numeric values. Default is 4.
`...`	Further arguments passed to or from other methods.

Value

Invisibly returns the input object `x`.

Examples

```
# Create example graph
n <- 6
adj.list <- lapply(seq_len(n), function(i) {
  c(if (i == n) 1 else i + 1, if (i == 1) n else i - 1)
})
weight.list <- lapply(adj.list, function(adj) rep(1.0, length(adj)))
result <- parameterize.circular.graph(adj.list, weight.list)
print(result)
```

`print.eigen.ulogit` *Print Method for eigen.ulogit Objects*

Description

Print Method for eigen.ulogit Objects

Usage

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

Arguments

`x`	An object of class "`eigen.ulogit`"
`...`	Additional arguments (currently ignored)

Value

Invisibly returns the input object

`print.fassoc` *Print Method for Functional Association Tests*

Description

Print Method for Functional Association Tests

Usage

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

Arguments

`x`	An object of class "assoc0" or "assoc1".
`...`	Additional arguments passed to specific print methods.

Value

Invisible x.

`print.gaussian_mixture` *Print method for gaussian_mixture objects*

Description

Print method for gaussian_mixture objects

Usage

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

Arguments

`x`	A gaussian_mixture object
`...`	Additional arguments (unused)

Value

Invisible x

`print.geodesic_stats` *Print method for geodesic_stats objects*

Description

Prints a brief overview of the geodesic statistics.

Usage

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

Arguments

`x`	A geodesic_stats object from compute.geodesic.stats().
`...`	Additional arguments passed to print.

Value

The object invisibly.

print.gflow_cx *Print Method for gflow_cx Objects*

Description

Provides a concise summary of a graph flow complex object, including information about the smoother used, extrema counts, and spurious extrema identification.

Usage

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

Arguments

`x`	A `gflow_cx` object returned by `create.gflow.cx`
`...`	Additional arguments passed to internal print functions (currently unused)

Value

Invisibly returns the input object `x`

Examples

```
## Not run:
# Create simple example
adj.list <- list(c(2,3), c(1,3), c(1,2))
weight.list <- list(c(1,1), c(1,1), c(1,1))
y <- c(0.5, 1.0, 0.7)

result <- create.gflow.cx(adj.list, weight.list, y, verbose = FALSE)
print(result)

## End(Not run)
```

print.graph.spectral.lowess *Print Graph Spectral LOWESS Object*

Description

Provides a concise summary of a graph spectral LOWESS object

Usage

```
## S3 method for class 'graph.spectral.lowess'
print(x, ...)
```

Arguments

`x`	A 'graph.spectral.lowess' object
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.graph.spectral.ma.lowess` *Print Graph Spectral MA LOWESS Object*

Description

Provides a concise summary of a graph spectral model-averaged LOWESS object

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
print(x, ...)
```

Arguments

`x`	A 'graph.spectral.ma.lowess' object
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.graph_kernel_smoother` *Print Method for graph_kernel_smoother Objects*

Description

Prints a concise summary of a `graph_kernel_smoother` object.

Usage

```
## S3 method for class 'graph_kernel_smoother'
print(x, digits = 4, ...)
```

Arguments

`x`	A `graph_kernel_smoother` object.
`digits`	Number of significant digits to display. Default is 4.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns the input object.

`print.graph_spectral_filter`
Print Method for Graph Spectral Filter Results

Description

Print Method for Graph Spectral Filter Results

Usage

```
## S3 method for class 'graph_spectral_filter'
print(x, digits = 4L, ...)
```

Arguments

`x`	An object of class `"graph_spectral_filter"`
`digits`	Integer; number of significant digits to print (default: 4)
`...`	Further arguments passed to or from other methods

Value

Invisibly returns the input object

`print.harmonic_smoother`
Print Method for Harmonic Smoother Results

Description

Prints a concise summary of harmonic smoother results.

Usage

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

Arguments

`x`	An object of class `"harmonic_smoother"`.
`...`	Further arguments passed to or from other methods.

Value

Invisibly returns the input object.

See Also

`harmonic.smoother`, `summary.harmonic_smoother`

`print.knn.outliers` *Print Method for knn.outliers Objects*

Description

Prints a summary of the outlier detection results from `remove.knn.outliers`.

Usage

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

Arguments

`x`	An object of class "knn.outliers" as returned by `remove.knn.outliers`.
`...`	Additional arguments passed to `print` methods.

Value

Invisibly returns the input object.

Examples

```
# Using the example from remove.knn.outliers
set.seed(123)
S <- rbind(matrix(rnorm(2000), ncol = 2),
           cbind(rnorm(10, 10), rnorm(10, 10)))
result <- remove.knn.outliers(S, K = 10)
print(result)
```

`print.mabilog` *Print Method for Mabilog Objects*

Description

Prints a concise summary of a mabilog object

Usage

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

Arguments

`x`	A 'mabilog' object
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.mabilo_plus` *Print Method for Mabilo Plus Objects*

Description

Prints a concise summary of a mabilo_plus object

Usage

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

Arguments

`x`	A 'mabilo_plus' object
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.maelog` *Print Method for maelog Objects*

Description

Prints a brief summary of a fitted `maelog` object.

Usage

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

Arguments

`x`	A fitted object of class `"maelog"`.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns the input object.

See Also

`maelog`, `summary.maelog`

print.magelog *Print Method for magelog Objects*

Description

Prints a summary of a fitted magelog model.

Usage

```
## S3 method for class 'magelog'
print(x, digits = 4, ...)
```

Arguments

`x`	A fitted model object of class `"magelog"`
`digits`	Integer; number of digits to display for numeric values
`...`	Additional arguments (currently ignored)

Value

Invisibly returns the input object

print.maximal_packing *Print Method for Maximal Packing Results*

Description

Print Method for Maximal Packing Results

Usage

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

Arguments

`x`	An object of class "maximal_packing"
`...`	Additional arguments (currently ignored)

Value

Invisible copy of x

print.mknn_graph *Print Method for mknn_graph Objects*

Description

Prints a concise summary of a mutual k-nearest neighbor graph object.

Usage

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

Arguments

x	An object of class "mknn_graph".
...	Additional arguments (currently unused).

Value

Invisibly returns the input object.

Examples

```
# Create a simple graph
X <- matrix(rnorm(50 * 2), ncol = 2)
graph <- create.mknn.graph(X, k = 5)
print(graph)
```

print.mknn_graphs *Print Method for mknn_graphs Objects*

Description

Prints a concise summary of a collection of mutual k-nearest neighbor graphs.

Usage

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

Arguments

x	An object of class "mknn_graphs".
...	Additional arguments (currently unused).

Value

Invisibly returns the input object.

Examples

```
# Create multiple graphs
X <- matrix(rnorm(100 * 3), ncol = 3)
graphs <- create.mknn.graphs(X, kmin = 5, kmax = 10)
print(graphs)
```

---

print.mst_completion_graph
Print Method for MST Completion Graph Objects

Description

Displays a concise summary of an MST completion graph object, including basic graph statistics and parameter information.

Usage

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

Arguments

x	An object of class mst_completion_graph, as returned by create.cmst.graph.
...	Additional arguments (currently ignored).

Details

This method provides a human-readable overview of the graph structure without displaying the full adjacency lists. For detailed inspection of graph components, use direct indexing (e.g., x$mst_adj_list) or the summary method.

Value

Invisibly returns the input object x.

See Also

summary.mst_completion_graph for more detailed summaries, create.cmst.graph for creating MST completion graphs

Examples

```
## Not run:
# Create and print a graph
X <- matrix(rnorm(50 * 3), nrow = 50, ncol = 3)
graph <- create.cmst.graph(X, q.thld = 0.8)
print(graph)

## End(Not run)
```

`print.nerve_complex` *Print Method for Nerve Complex Objects*

Description

Print Method for Nerve Complex Objects

Usage

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

Arguments

`x`	A nerve complex object
`...`	Additional arguments (not used)

Value

Invisibly returns the object

`print.nerve_cx_spectral_filter` *Print Method for nerve_cx_spectral_filter Objects*

Description

Prints a concise summary of nerve complex spectral filtering results.

Usage

```
## S3 method for class 'nerve_cx_spectral_filter'
print(x, digits = 4, ...)
```

Arguments

`x`	A `nerve_cx_spectral_filter` object returned by `nerve.cx.spectral.filter`.
`digits`	Integer indicating the number of significant digits to print for numeric values. Default is 4.
`...`	Additional arguments (currently ignored).

Value

The object `x` is returned invisibly.

See Also

`summary.nerve_cx_spectral_filter` for more detailed summary, `nerve.cx.spectral.filter` for the main function

`print.packing_validation` *Print Method for Packing Validation Results*

Description

Print Method for Packing Validation Results

Usage

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

Arguments

`x`	An object of class "packing_validation"
`...`	Additional arguments (currently ignored)

Value

Invisible copy of x

`print.path.graph` *Print Method for path.graph Objects*

Description

Print a concise summary of a path.graph object.

Usage

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

Arguments

`x`	A path.graph object
`...`	Additional arguments (currently ignored)

Value

Invisible copy of x

Examples

```
## Not run:
pg <- create.path.graph(graph, edge.lengths, h = 2)
print(pg)

## End(Not run)
```

print.path.graph.plm *Print Method for path.graph.plm Objects*

Description

Print a summary of a PLM path graph object.

Usage

```
## S3 method for class 'path.graph.plm'
print(x, ...)
```

Arguments

x	A path.graph.plm object
...	Additional arguments (currently ignored)

Value

Invisible copy of x

print.path.graph.series *Print Method for path.graph.series Objects*

Description

Print a summary of a series of path graphs.

Usage

```
## S3 method for class 'path.graph.series'
print(x, ...)
```

Arguments

x	A path.graph.series object
...	Additional arguments passed to print.path.graph

Value

Invisible copy of x

`print.pgmalo` *Print Method for pgmalo Objects*

Description

Displays a concise summary of a pgmalo model fit.

Usage

```
## S3 method for class 'pgmalo'
print(x, digits = 3, ...)
```

Arguments

`x`	An object of class "pgmalo".
`digits`	Number of significant digits for numeric output (default: 3).
`...`	Additional arguments (currently unused).

Value

The input object invisibly.

`print.pwlm` *Print Method for PWLM Objects*

Description

Print Method for PWLM Objects

Usage

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

Arguments

`x`	An object of class "pwlm"
`...`	Additional arguments passed to print

print.spectral.lowess.result *Print method for spectral.lowess.result objects*

Description

Print method for spectral.lowess.result objects

Usage

```
## S3 method for class 'spectral.lowess.result'
print(x, ...)
```

Arguments

x	An object of class "spectral.lowess.result"
...	Additional arguments (ignored)

Value

Invisibly returns the input object

print.summary.amagelo *Print method for summary.amagelo objects*

Description

Print method for summary.amagelo objects

Usage

```
## S3 method for class 'summary.amagelo'
print(x, digits = 4, ...)
```

Arguments

x	An object of class ’summary.amagelo’
digits	Number of digits to display (default: 4)
...	Additional arguments passed to print

`print.summary.basin_cx` *Print Method for Basin Complex Summary Objects*

Description

Prints a formatted, comprehensive summary of a gradient flow basin complex, including detailed statistics about extrema, basins, basin overlaps, and cell complex structure.

Usage

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

Arguments

`x`	An object of class "summary.basin_cx" as created by `summary.basin_cx`.
`...`	Additional arguments passed to print methods (currently unused).

Details

The print method displays the following sections:

Extrema: Total counts of local extrema, broken down by minima and maxima.

Local Minima/Maxima Information: Tables showing each extremum with its vertex index, label, basin size, and function value. Minima are sorted by increasing function value, maxima by decreasing function value.

Basins: Counts of merged ascending (minimum) and descending (maximum) basins, along with size statistics.

Basin Size Statistics: Summary statistics (min, quartiles, mean, max) for the number of vertices in each type of basin.

Cell Complex Summary: Total number of cells and breakdown by type (ascending-descending, ascending-ascending, descending-descending).

Jaccard Index Matrices: Matrices showing pairwise Jaccard indices (overlap measures) between basins of the same type, rounded to 3 decimal places.

Messages: Any informational messages generated during basin complex creation.

Value

Invisibly returns the input summary object `x`. This function is called for its side effect of printing detailed information to the console.

See Also

`summary.basin_cx` for creating summary objects, `create.basin.cx` for creating basin complex objects, `print.basin_cx` for basic basin complex printing

Examples

```
## Not run:
# Create a basin complex
adj_list <- list(c(2,3), c(1,3,4), c(1,2,5), c(2,5), c(3,4))
weight_list <- list(c(1,2), c(1,1,3), c(2,1,2), c(3,1), c(2,1))
y <- c(2.5, 1.8, 3.2, 0.7, 2.1)
basin_cx <- create.basin.cx(adj_list, weight_list, y)

# Get and print detailed summary
basin_summary <- summary(basin_cx)
print(basin_summary)
# or simply:
basin_summary

## End(Not run)
```

`print.summary.gaussian_mixture`
Print summary of gaussian_mixture

Description

Print summary of gaussian_mixture

Usage

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

Arguments

`x`	A summary.gaussian_mixture object
`...`	Additional arguments (unused)

Value

Invisible x

`print.summary.gflow_basins`
Print Summary of Gradient-Flow Basin Results

Description

Print Summary of Gradient-Flow Basin Results

Usage

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

Arguments

`x`	An object of class `"summary.gflow_basins"`
`...`	Additional arguments (currently ignored)

`print.summary.ggflow` *Print Gradient Flow Structure Summary*

Description

Print method for objects of class `"summary.ggflow"`. Displays a formatted summary of a gradient flow structure.

Usage

```
## S3 method for class 'summary.ggflow'
print(x, digits = 3L, ...)
```

Arguments

`x`	An object of class `"summary.ggflow"`.
`digits`	Number of significant digits for numeric output.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns `x`.

Examples

```
## Not run:
flow <- construct.graph.gradient.flow(adj.list, weight.list, y, scale)
summary(flow)

## End(Not run)
```

`print.summary.graph.spectral.lowess`
Print Summary Statistics for Graph Spectral LOWESS

Description

Formats and displays comprehensive summary statistics for graph spectral LOWESS fits.

Usage

```
## S3 method for class 'summary.graph.spectral.lowess'
print(x, digits = 4, ...)
```

Arguments

`x`	A 'summary.graph.spectral.lowess' object
`digits`	Number of significant digits for numerical output (default: 4)
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.summary.graph.spectral.ma.lowess`
Print Summary Statistics for Graph Spectral MA LOWESS

Description

Formats and displays comprehensive summary statistics for graph spectral model-averaged LOWESS fits, including model averaging details.

Usage

```
## S3 method for class 'summary.graph.spectral.ma.lowess'
print(x, digits = 4, ...)
```

Arguments

`x`	A 'summary.graph.spectral.ma.lowess' object
`digits`	Number of significant digits for numerical output (default: 4)
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.summary.graph_spectral_filter`
Print Method for Summary of Graph Spectral Filter Results

Description

Print Method for Summary of Graph Spectral Filter Results

Usage

```
## S3 method for class 'summary.graph_spectral_filter'
print(x, digits = 4L, ...)
```

Arguments

`x`	An object of class `"summary.graph_spectral_filter"`
`digits`	Integer; number of significant digits to print (default: 4)
`...`	Further arguments passed to or from other methods

Value

Invisibly returns the input object

`print.summary.harmonic_smoother` *Print Method for Summary of Harmonic Smoother Results*

Description

Prints the summary of harmonic smoother results in a formatted manner.

Usage

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

Arguments

`x`	An object of class `"summary.harmonic_smoother"`.
`...`	Further arguments passed to or from other methods.

Value

Invisibly returns the input object.

See Also

`summary.harmonic_smoother`

`print.summary.knn.outliers` *Print Method for summary.knn.outliers Objects*

Description

Prints the summary of outlier detection results.

Usage

```
## S3 method for class 'summary.knn.outliers'
print(x, ...)
```

Arguments

`x`	An object of class "summary.knn.outliers".
`...`	Additional arguments passed to `print` methods.

Value

Invisibly returns the input object.

`print.summary.local_extrema`
Print Summary of Local Extrema Detection Results

Description

Prints a formatted summary of local extrema detection results.

Usage

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

Arguments

`x`	An object of class `"summary.local_extrema"`.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns `x`.

`print.summary.mabilo` *Print Summary Statistics for Mabilo Fits*

Description

Formats and displays comprehensive summary statistics for mabilo fits. Includes model parameters, fit statistics, error analysis, diagnostic information, and Bayesian bootstrap results when available.

Usage

```
## S3 method for class 'summary.mabilo'
print(x, digits = 4, ...)
```

Arguments

`x`	A 'summary.mabilo' object from summary.mabilo
`digits`	Number of significant digits for numerical output (default: 4)
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.summary.mabilog` *Print Summary Statistics for Mabilog Fits*

Description

Formats and displays comprehensive summary statistics for mabilog fits. Includes model parameters, fit statistics, error analysis, diagnostic information, and Bayesian bootstrap results when available.

Usage

```
## S3 method for class 'summary.mabilog'
print(x, digits = 4, ...)
```

Arguments

`x`	A 'summary.mabilog' object from summary.mabilog
`digits`	Number of significant digits for numerical output (default: 4)
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.summary.mabilo_plus` *Print Summary Statistics for Mabilo Plus Fits*

Description

Formats and displays summary statistics for mabilo.plus fits in a structured, easy-to-read format. Output includes model parameters, fit statistics, error analysis, and diagnostic information for both simple mean (SM) and model averaging (MA) predictions.

Usage

```
## S3 method for class 'summary.mabilo_plus'
print(x, digits = 4, ...)
```

Arguments

`x`	A 'summary.mabilo_plus' object from summary.mabilo_plus
`digits`	Number of significant digits for numerical output (default: 4)
`...`	Additional arguments (currently unused)

Value

Returns x invisibly

`print.summary.magelog` *Print Method for summary.magelog Objects*

Description

Prints the summary of a magelog model.

Usage

```
## S3 method for class 'summary.magelog'
print(x, digits = 4, ...)
```

Arguments

`x`	A summary object of class `"summary.magelog"`
`digits`	Integer; number of digits to display for numeric values
`...`	Additional arguments (currently ignored)

Value

Invisibly returns the input object

`print.summary.mst_completion_graph` *Print Summary of MST Completion Graph*

Description

Formats and displays the summary information for an MST completion graph in a readable format.

Usage

```
## S3 method for class 'summary.mst_completion_graph'
print(x, digits = 3L, ...)
```

Arguments

`x`	An object of class `summary.mst_completion_graph`, as returned by `summary.mst_completion_gra`
`digits`	Integer; number of significant digits for numeric output. Default is 3.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns the input summary object.

Examples

```
## Not run:
X <- matrix(rnorm(75 * 4), nrow = 75, ncol = 4)
graph <- create.cmst.graph(X)
graph_summary <- summary(graph)
print(graph_summary, digits = 4)

## End(Not run)
```

`print.summary.nerve_cx_spectral_filter` *Print Method for summary.nerve_cx_spectral_filter Objects*

Description

Prints a formatted summary of nerve complex spectral filtering results.

Usage

```
## S3 method for class 'summary.nerve_cx_spectral_filter'
print(x, digits = 3, ...)
```

Arguments

`x`	A `summary.nerve_cx_spectral_filter` object.
`digits`	Integer indicating the number of significant digits to print. Default is 3.
`...`	Additional arguments (currently ignored).

Value

The object `x` is returned invisibly.

`print.summary.pgmalo` *Print Method for summary.pgmalo Objects*

Description

Displays formatted summary of pgmalo model fits with organized sections for model information, cross-validation results, and diagnostic statistics.

Usage

```
## S3 method for class 'summary.pgmalo'
print(x, digits = 4, show_cv_details = FALSE, ...)
```

Arguments

`x`	An object of class "summary.pgmalo" from `summary.pgmalo`.
`digits`	Number of significant digits for numeric output (default: 4).
`show_cv_details`	Logical; if TRUE shows detailed CV results (default: FALSE).
`...`	Additional arguments (currently unused).

Value

The input object invisibly.

Examples

```
## Not run:
fit <- pgmalo(neighbors, edge_lengths, y)
summary(fit)

# Show more CV details
print(summary(fit), show_cv_details = TRUE)

## End(Not run)
```

`print.summary.pwlm` *Print Method for PWLM Summary Objects*

Description

Print Method for PWLM Summary Objects

Usage

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

Arguments

`x`	An object of class "summary.pwlm"
`...`	Additional arguments passed to print

`print.summary.spectral.lowess.result` *Print method for summary.spectral.lowess.result objects*

Description

Print method for summary.spectral.lowess.result objects

Usage

```
## S3 method for class 'summary.spectral.lowess.result'
print(x, ...)
```

Arguments

`x`	An object of class "summary.spectral.lowess.result"
`...`	Additional arguments (ignored)

Value

Invisibly returns the input object

`print.summary.uggmalo` *Print Method for UGGMALO Summary*

Description

Print Method for UGGMALO Summary

Usage

```
## S3 method for class 'summary.uggmalo'
print(x, digits = 4, ...)
```

Arguments

`x`	An object of class "summary.uggmalo"
`digits`	Integer. Number of significant digits to print
`...`	Additional arguments (currently ignored)

Value

Invisibly returns the input object

`print.summary.uggmalog` *Print method for summary.uggmalog objects*

Description

Print method for summary.uggmalog objects

Usage

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

Arguments

`x`	An object of class "`summary.uggmalog`"
`...`	Additional arguments (currently ignored)

Value

Invisibly returns the input object

`print.summary.ugkmm` *Print summary.ugkmm Objects*

Description

Print summary.ugkmm Objects

Usage

```
## S3 method for class 'summary.ugkmm'
print(x, digits = 4, ...)
```

Arguments

`x`	An object of class "summary.ugkmm".
`digits`	Number of significant digits.
`...`	Additional arguments (unused).

print.summary.upgmalo *Print Method for summary.upgmalo Objects*

Description

Formats and displays UPGMALO model summaries in a clean, organized format.

Usage

```
## S3 method for class 'summary.upgmalo'
print(x, digits = 4, ...)
```

Arguments

x	An object of class "summary.upgmalo".
digits	Number of significant digits for numeric output (default: 4).
...	Additional arguments (currently unused).

Value

The input object invisibly.

Examples

```
# See examples in upgmalo() and summary.upgmalo()
```

print.uggmalo *Print Method for UGGMALO Results*

Description

Print Method for UGGMALO Results

Usage

```
## S3 method for class 'uggmalo'
print(x, digits = 4, ...)
```

Arguments

x	An object of class "uggmalo"
digits	Integer. Number of significant digits to print
...	Additional arguments (currently ignored)

Value

Invisibly returns the input object

print.uggmalog *Print method for uggmalog objects*

Description

Print method for uggmalog objects

Usage

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

Arguments

x	An object of class "uggmalog"
...	Additional arguments (currently ignored)

Value

Invisibly returns the input object

print.ulogit *Print Method for ulogit Objects*

Description

Print Method for ulogit Objects

Usage

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

Arguments

x	An object of class "ulogit"
...	Additional arguments (currently ignored)

Value

Invisibly returns the input object

print.upgmalo *Print Method for upgmalo Objects*

Description

Displays a concise summary of a upgmalo model fit.

Usage

```
## S3 method for class 'upgmalo'
print(x, digits = 3, ...)
```

Arguments

`x`	An object of class "upgmalo".
`digits`	Number of significant digits for numeric output (default: 3).
`...`	Additional arguments (currently unused).

Value

The input object invisibly.

print.weighted.p.summary *Print method for weighted.p.summary objects*

Description

Print method for weighted.p.summary objects

Usage

```
## S3 method for class 'weighted.p.summary'
print(x, ...)
```

Arguments

`x`	A weighted.p.summary object
`...`	Additional arguments (ignored)

Value

Invisible copy of x

prof.fn — *Extract Most Abundant ASVs with Taxonomy Information*

Description

Extracts the n most abundant Amplicon Sequence Variants (ASVs) from a sample or group of samples, optionally including their taxonomic classification.

Usage

```
prof.fn(id, S, bm.tx = NULL, n.prof = 5, k.neighbors = 1, verbose = FALSE)
```

Arguments

`id`	Character string or numeric index identifying the sample in the abundance matrix S. If character, must match a row name in S.
`S`	Numeric matrix of ASV abundances with samples in rows and ASVs in columns. Row names should contain sample identifiers and column names should contain ASV identifiers.
`bm.tx`	Optional named vector containing taxonomic classifications for ASVs. Names must match column names in S. If NULL, only abundances are returned.
`n.prof`	Integer specifying the number of most abundant ASVs to report. Default is 5.
`k.neighbors`	Integer specifying the number of nearest neighbors to include in abundance calculations. Default is 1 (only target sample). If greater than 1, returns mean abundances across the k nearest neighbors of the target sample (including the target itself).
`verbose`	Logical indicating whether to print the resulting profile matrix. Default is FALSE.

Details

When k.neighbors > 1, the function uses k-nearest neighbors based on Euclidean distance in the abundance space to compute mean abundances. This can help smooth profiles in sparse data.

Value

A matrix with one column containing abundance values (rounded to 2 significant figures). If bm.tx is provided, additional columns contain taxonomic information. Row names are ASV identifiers.

Examples

```
# Create example data
S <- matrix(runif(100), nrow = 10, ncol = 10)
rownames(S) <- paste0("Sample", 1:10)
colnames(S) <- paste0("ASV", 1:10)

# Extract profile for first sample
prof <- prof.fn("Sample1", S, n.prof = 3)

# With taxonomy
taxonomy <- paste0("Taxon", 1:10)
names(taxonomy) <- colnames(S)
```

```
prof_with_tax <- prof.fn(1, S, bm.tx = taxonomy, n.prof = 3, verbose = TRUE)
```

`project.onto.subspace` *Projects a Vector onto a Subspace in R^n*

Description

This function projects a given vector in R^n onto a subspace spanned by unit vectors. The subspace is defined by a matrix where each column is a unit vector spanning the subspace.

Usage

```
project.onto.subspace(x, U)
```

Arguments

x	A numeric vector of length n representing the vector to be projected.
U	A matrix of dimensions n x m, where each column is a unit vector spanning the subspace. The matrix should have full column rank.

Value

A numeric vector representing the projection of x onto the subspace spanned by the columns of U.

Examples

```
u1 <- c(1, 0, 0)
u2 <- c(0, 1, 0)
u3 <- c(0, 0, 1)
U <- matrix(c(u1, u2, u3), nrow=3)
x <- c(2, 3, 4)
project.onto.subspace(x, U)
```

`pts3d.select` *Interactive 3D Point Selection with Profile Display*

Description

Allows interactive selection of points in a 3D visualization and optionally displays profiles of the selected points based on associated data.

Usage

```
pts3d.select(
  X,
  Z = NULL,
  n.comp = 5,
  show.profiles = FALSE,
  use.pts = FALSE,
  color = "red",
  alpha = 0.3,
  size = 0.1,
  radius = 0.1
)
```

Arguments

`X`	Numeric matrix with 3 columns representing x, y, z coordinates of points in 3D space.
`Z`	Optional matrix or data frame where rows correspond to points in `X` and columns represent features (e.g., species abundances).
`n.comp`	Integer. Number of top features to show in profiles (default: 5).
`show.profiles`	Logical. Whether to print profiles to console (default: FALSE).
`use.pts`	Logical. If TRUE, displays selected points as points; if FALSE, displays as spheres (default: FALSE).
`color`	Character string specifying the color for highlighting selected points (default: "red").
`alpha`	Numeric between 0 and 1. Transparency of spheres when `use.pts = FALSE` (default: 0.3).
`size`	Numeric. Size of points when `use.pts = TRUE` (default: 0.1).
`radius`	Numeric. Radius of spheres when `use.pts = FALSE` (default: 0.1).

Details

This function provides an interactive interface for selecting points in an existing 3D rgl plot. Selected points are highlighted and their profiles can be computed from associated data.

Value

Invisibly returns a list with two components:

`idx`	Integer vector of indices of selected points in `X`
`prof`	Matrix showing the mean profile of selected points' features, or NA if `Z` is not provided

Examples

```
## Not run:
library(rgl)
# Create example 3D data
set.seed(123)
n <- 100
X <- matrix(rnorm(n * 3), ncol = 3)
```

```
Z <- matrix(rpois(n * 10, lambda = 5), ncol = 10)
colnames(Z) <- paste0("Species", 1:10)

# Create 3D plot
plot3d(X, col = "blue")

# Interactively select points
selection <- pts3d.select(X, Z, show.profiles = TRUE)

# Access results
selected_indices <- selection$idx
profile_data <- selection$prof

## End(Not run)
```

---

qexp.gaussian *q-Exponential Gaussian Function*

Description

This function computes the values of a modified Gaussian function, known as a q-Gaussian, which raises the standard Gaussian to the power of q. This modification can be used to adjust the "flatness" or "peakiness" of the Gaussian curve.

Usage

```
qexp.gaussian(x, offset = 0, h = 1, q = 4)
```

Arguments

x	A numeric vector or a single numeric value where the q-Gaussian function is evaluated.
offset	The center of the q-Gaussian function (defaults to 0).
h	The standard deviation of the Gaussian part of the function (defaults to 1).
q	The power to which the Gaussian function is raised (defaults to 4).

Details

The q-Gaussian function is defined as (exp(-((x - offset)^2) / h2))^q. The parameter q allows for adjusting the shape of the Gaussian curve: higher values of q make the curve peakier, while lower values make it flatter. This function can be useful in various statistical and signal processing applications where a modified Gaussian shape is required.

Value

A numeric vector or a single numeric value representing the value(s) of the q-Gaussian function at the input x. The function alters the shape of a standard Gaussian curve based on the value of q.

Examples

```
## Not run:
x <- seq(-5, 5, length.out = 100)
plot(x, qgaussian(x, q = 4), type = "l")

## End(Not run)
```

---

qlaplace *Laplace Distribution Quantile Function*

Description

Calculates the quantile function for the Laplace distribution.

Usage

```
qlaplace(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

p	Vector of probabilities.
location	The location parameter μ. Default is 0.
scale	The scale parameter b. Must be positive. Default is 1.
lower.tail	Logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
log.p	Logical; if TRUE, probabilities p are given as $\log(p)$. Default is FALSE.

Value

A vector of quantiles.

Examples

```
p <- seq(0, 1, by = 0.1)
q <- qlaplace(p, location = 0, scale = 1)
plot(p, q, type = "l", main = "Laplace Quantile Function")
```

---

quantize.cont.var *Quantize a continuous variable into categories*

Description

Quantize a continuous variable into categories

Usage

```
quantize.cont.var(
  x,
  method = "uniform",
  wins.p = 0.01,
  round = TRUE,
  dig.lab = 2,
  breaks = NULL,
  include.lowest = TRUE,
  start = 1/6,
  end = 0,
  n.levels = 10,
  use.viridis = FALSE,
  brewer.palette = NULL,
  use.unique = FALSE
)
```

Arguments

`x`	Numeric vector.
`method`	"uniform" or "quantile".
`wins.p`	Winsorization proportion (uniform method).
`round`	Round $[0, 1]$ endpoints to one decimal (uniform method).
`dig.lab`	Digits for labels.
`breaks`	Custom breaks.
`include.lowest`	Include lowest value in first interval.
`start`, `end`	Start/end for rainbow().
`n.levels`	Number of levels.
`use.viridis`	Use viridis palette if available.
`brewer.palette`	Name of RColorBrewer palette to use (optional).
`use.unique`	Use only unique x for break calc.

Value

list(x.cat, x.col.tbl)

`quantize.for.legend` *Quantize Continuous Variable for Legend Display*

Description

Quantizes a continuous variable and generates legend colors and labels for visualization

Usage

```
quantize.for.legend(
  y,
  quantize.method = "uniform",
  quantize.wins.p = 0.02,
  quantize.round = FALSE,
  quantize.dig.lab = 2,
  start = 1/6,
  end = 0,
  n.levels = 10
)
```

Arguments

`y`	A numeric vector to be quantized.
`quantize.method`	Method to use for quantization. Options are "uniform" (equal-width bins) or "quantile" (equal-frequency bins).
`quantize.wins.p`	Winsorization parameter for the "uniform" method. Values beyond the p and 1-p quantiles are grouped together. Default is 0.02.
`quantize.round`	Logical. If TRUE, rounds the endpoints of the range.
`quantize.dig.lab`	Number of digits for labels in the cut() function.
`start`	Start value for the color range in HSV color space (0-1).
`end`	End value for the color range in HSV color space (0-1).
`n.levels`	Number of discrete levels for quantization.

Details

This function is primarily used internally by plotting functions to convert continuous variables into discrete categories with associated colors for visualization. The winsorization parameter helps handle outliers in the uniform method.

Value

A list containing:

`y.cat`	Factor of quantized values
`y.col.tbl`	Named vector mapping categories to colors
`y.cols`	Vector of colors for each input value
`legend.labs`	Formatted labels for legend display

Examples

```
## Not run:
y <- rnorm(100)
q <- quantize.for.legend(y, n.levels = 5)
plot(1:100, y, col = q$y.cols, pch = 19)
legend("topright", legend = q$legend.labs, fill = q$y.col.tbl)
```

```
## End(Not run)
```

R.density.distance *Calculates a Symmetric Density-Associated Distance*

Description

This function computes a custom, symmetric distance metric between two points p and q based on the provided density function density.func. The Euclidean distance between p and q is normalized by the average of the densities at these points, making the distance symmetric.

Usage

```
R.density.distance(p, q, p.density, q.density, density.func = NULL)
```

Arguments

p	A numeric vector representing coordinates of the first point.
q	A numeric vector representing coordinates of the second point.
p.density	The density value at point p.
q.density	The density value at point q.
density.func	A function representing the density at a given point in the space (optional, default = NULL).

Details

Any function that is non-negative over all points of the given set can be used as a density function in this function. This allows for scaling of the "density function" so that it affects the Euclidean distance more and more. The function transform.ED() can be used to scale any "density function".

Value

The calculated density-associated distance between p and q.

Examples

```
density.func <- function(x) { return(dnorm(x, mean=0, sd=1)) }
p <- c(0, 0)
q <- c(1, 1)
p.density <- density.func(p)
q.density <- density.func(q)
R.density.distance(p, q, p.density, q.density, density.func)
```

radial.Lp.transform *Perform Lp Radial Transformation*

Description

This function applies the Lp radial transformation to each row of the input matrix or data frame. The transformation is defined as x -> r^p * x / r, where r is the L2 norm of x.

Usage

```
radial.Lp.transform(X, p = 0.5)
```

Arguments

X	A matrix or data frame containing numeric values to be transformed.
p	The Lp exponent. Must be greater than 0. Default is 0.5.

Value

A matrix or data frame of the same dimensions as the input, with the Lp radial transformation applied to each row.

Examples

```
X <- matrix(runif(20), ncol = 2)
transformed_X <- radial.Lp.transform(X, p = 0.5)
```

radial.qexp.gaussian *Evaluates radial q-exponential Gaussian on set of points*

Description

Evaluates radial q-exponential Gaussian cetnered at 'center' with standard deviation sigma and power q on a set of points X.

Usage

```
radial.qexp.gaussian(X, center, sigma = 1, q = 1)
```

Arguments

X	A matrix of data frame of points at which the radia q-exponential Gaussian is to be evaluated.
center	The center of the radial q-exponential Gaussian.
sigma	The standard deviation of the radial q-exponential Gaussian.
q	The power to which the radial q-exponential Gaussian is raised.

Value

Returns a vector of values of the radia q-exponential Gaussian at the points of X.

`remove.close.neighbors` *Removes Close Neighbors*

Description

This function takes a vector of positions (`x`) and a minimum distance (`min.dist`), and returns a modified vector of knots where no two knots are closer than the specified minimum distance.

Usage

```
remove.close.neighbors(x, min.dist)
```

Arguments

`x`	A numeric vector.
`min.dist`	A positive numeric value specifying the minimum allowable distance between any two consecutive elements of x.

Details

The function first sorts the input vector of knots. It then iterates through the sorted knots, keeping only those that are at least `min.dist` away from the previously accepted knot. This ensures that the output vector of knots satisfies the distance requirement.

Value

A modified version of x such that no two consecutive elements are closer than `min.dist`.

Examples

```
## Not run:
x <- c(0.1, 0.2, 0.4, 0.7, 1.0)
min.dist <- 0.25
remove.close.neighbors(x, min.dist)

## End(Not run)
```

`remove.knn.outliers` *Remove Outliers from a State Space Using k-Nearest Neighbors*

Description

Identifies and removes outliers from a multivariate state space based on k-nearest neighbor distances. This function implements several strategies for outlier detection, all based on the principle that outliers tend to be isolated from the main data clusters and thus have larger distances to their nearest neighbors.

Usage

```
remove.knn.outliers(
  S,
  y = NULL,
  p = 0.98,
  dist.factor = 100,
  K = 30,
  method = "diff.dist.factor"
)
```

Arguments

`S`	A numeric matrix or data frame representing the state space, where each row is an observation and each column is a dimension or feature. Must contain at least `K+1` observations.
`y`	An optional numeric vector containing values of a variable defined over the state space. If provided, must have length equal to `nrow(S)`. The function will filter this vector to match the filtered state space.
`p`	A numeric value between 0 and 1 (exclusive) indicating the percentile threshold for outlier removal. The default value of 0.98 removes points whose distance to the nearest neighbor exceeds the 98th percentile of all nearest neighbor distances.
`dist.factor`	A positive numeric value used as a threshold factor in certain outlier detection methods. In "diff.dist.factor", points with relative jumps greater than this factor are considered outliers. Default is 100.
`K`	A positive integer specifying the number of nearest neighbors to compute for each point. Must be less than the number of observations in `S`. Default is 30.
`method`	A character string specifying the outlier detection method to use when K > 1. Must be one of "dist.factor", "diff.dist.factor" (default), or "default" for the unnamed method.

Details

The function determines outliers by analyzing the distances between points and their K nearest neighbors. Several detection methods are available:

When K = 1: Points are flagged as outliers if their distance to the nearest neighbor exceeds the p-th percentile threshold of all nearest neighbor distances.

"dist.factor": Points are considered outliers if the ratio of the K-th nearest neighbor distance to the 1st nearest neighbor distance exceeds `dist.factor`.

"diff.dist.factor" (default): For each point, finds the maximum jump in distance between consecutive neighbors, normalizes by the median jump across all points, and flags points as outliers if their relative jump exceeds `dist.factor`. This method is particularly robust as it adapts to the overall distribution of distances.

Default unnamed method: Iteratively checks if the first nearest neighbor distance exceeds the threshold or if any consecutive difference between neighbor distances exceeds the threshold.

Value

A list of class "knn.outliers" containing the following components:

S.q The filtered state space matrix with outliers removed.

y.q The filtered dependent variable (if `y` was provided), otherwise `NULL`.

nn.d A matrix of dimensions `nrow(S)` x `K` containing distances to the K nearest neighbors for each point.

d.thld The distance threshold used for outlier detection.

idx A logical vector indicating which points were kept (`TRUE`) and which were identified as outliers (`FALSE`).

n.outliers The number of outliers detected.

method The outlier detection method used.

See Also

`get.knn` for the k-nearest neighbor calculation, `quantile` for percentile calculations

Examples

```
# Create a sample dataset with outliers
set.seed(123)
n_normal <- 1000
n_outliers <- 10

# Generate normal data
normal_data <- matrix(rnorm(n_normal * 2), ncol = 2)

# Generate outliers far from the main cluster
outliers <- cbind(rnorm(n_outliers, mean = 10, sd = 0.5),
                  rnorm(n_outliers, mean = 10, sd = 0.5))

# Combine data
S <- rbind(normal_data, outliers)
y <- c(rnorm(n_normal), rnorm(n_outliers, mean = 5))

# Remove outliers using the default method
result <- remove.knn.outliers(S, y, K = 10)

# Print summary
cat("Number of outliers detected:", result$n.outliers, "\n")
cat("Percentage of data retained:",
    round(100 * nrow(result$S.q) / nrow(S), 2), "%\n")

# Use a different method with more conservative threshold
result2 <- remove.knn.outliers(S, y, p = 0.95, method = "dist.factor",
                               dist.factor = 5, K = 10)

## Not run:
# Plot the original and filtered state space
par(mfrow = c(1, 2))
plot(S, col = "gray", main = "Original State Space",
     xlab = "Dimension 1", ylab = "Dimension 2")
points(S[!result$idx, ], col = "red", pch = 16, cex = 1.2)
```

```
plot(result$S.q, col = "blue", pch = 16,
     main = "Filtered State Space",
     xlab = "Dimension 1", ylab = "Dimension 2")
legend("topright", legend = c("Retained", "Removed"),
       col = c("blue", "red"), pch = 16)

## End(Not run)
```

replace.basin.label *Replace a Basin Label in Gradient Flow Object*

Description

Changes a specific basin label throughout a gradient flow object, ensuring consistency across extrema table, basins, cells, and trajectories.

Usage

```
replace.basin.label(flow, old.label, new.label)
```

Arguments

`flow`	An object of class "ggflow".
`old.label`	Character string specifying the label to be replaced.
`new.label`	Character string specifying the new label.

Details

This function is useful for customizing basin labels after computation, for example, to use more meaningful names based on domain knowledge.

Value

A modified gradient flow object with the updated label.

Examples

```
## Not run:
# Replace a generic label with a meaningful one
flow_updated <- replace.basin.label(flow, "M1", "HighState")

## End(Not run)
```

`replace.basin.label.basin_cx`
Replace a Basin Label in a Basin Complex Object

Description

Replaces a specified label for a local extremum in a basin_cx object. Updates all references to this label in extrema, basins, and cells.

Usage

```
replace.basin.label.basin_cx(bcx, old.label, new.label)
```

Arguments

`bcx`	An object of class "basin_cx"
`old.label`	The existing label to be replaced
`new.label`	The new label to replace with

Value

A modified object of class "basin_cx" with updated labels

`residuals.graph.spectral.lowess`
Extract Residuals from Graph Spectral LOWESS

Description

Computes and returns residuals from a graph spectral LOWESS fit

Usage

```
## S3 method for class 'graph.spectral.lowess'
residuals(object, type = c("response", "pearson", "deviance"), ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`type`	Character string specifying residual type: "response" (default), "pearson", or "deviance"
`...`	Additional arguments (currently unused)

Value

Numeric vector of residuals

`residuals.graph.spectral.ma.lowess`
Extract Residuals from Graph Spectral MA LOWESS

Description

Computes and returns residuals from a graph spectral model-averaged LOWESS fit

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
residuals(object, type = c("response", "pearson", "deviance"), ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`type`	Character string specifying residual type: "response" (default), "pearson", or "deviance"
`...`	Additional arguments (currently unused)

Value

Numeric vector of residuals

`residuals.mabilog` *Extract Residuals from Mabilog Model*

Description

Extracts residuals from a mabilog object

Usage

```
## S3 method for class 'mabilog'
residuals(object, type = c("deviance", "pearson", "response", "working"), ...)
```

Arguments

`object`	A 'mabilog' object
`type`	Character string specifying residual type: "deviance" (default), "pearson", "response", or "working"
`...`	Additional arguments (currently unused)

Value

Numeric vector of residuals

`residuals.mabilo_plus` *Extract Residuals from Mabilo Plus Model*

Description

Extracts residuals from a mabilo_plus object

Usage

```
## S3 method for class 'mabilo_plus'
residuals(object, type = c("ma", "sm", "both"), ...)
```

Arguments

`object`	A 'mabilo_plus' object
`type`	Character string specifying which residuals to return: "ma" (default), "sm", or "both"
`...`	Additional arguments (currently unused)

Value

Numeric vector or matrix of residuals

`restric.to.Linf.unit.sphere` *Restrict Points to L-infinity Unit Sphere*

Description

This function restricts the points represented by rows in the input matrix or data frame to the L-infinity unit sphere. If the maximum absolute value of a row exceeds 1, the row is scaled down to have a maximum absolute value of 1.

Usage

```
restric.to.Linf.unit.sphere(X)
```

Arguments

`X`	A matrix or data frame containing numeric values to be restricted.

Value

A matrix or data frame of the same dimensions as the input, with each row restricted to the L-infinity unit sphere.

Examples

```
X <- matrix(runif(20, -2, 2), ncol = 2)
restricted_X <- restric.to.Linf.unit.sphere(X)
```

`right.asymmetric.bump.fn` *Creates Right Asymmetric Bump Function*

Description

This function creates an asymmetric bump function, which is a modification of the standard bump function with asymmetric behavior to the right of the offset. It is smooth with compact support, primarily used in mathematical analysis and applications requiring non-symmetric smooth functions.

Usage

```
right.asymmetric.bump.fn(x, offset = 0, h = 1, q = 1)
```

Arguments

`x`	A numeric vector or a single numeric value at which the bump function is evaluated.
`offset`	The center (offset) of the bump function (defaults to 0). The function behaves differently to the right and right of this point.
`h`	The radius of the support of the bump function (defaults to 1). The function is zero outside the interval $[offset - h, offset + h]$.
`q`	The power to which the Gaussian function to the left of the offset is raised (defaults to 1).

Details

The asymmetric bump function is defined differently to the right and right of the offset. For `x < offset` and `offset - x < h`, it is defined as exp(-1 / h - (x - offset)^2). For `|x - offset| < h`, it follows the standard bump function exp(-1 / (h - (x - offset)^2)). It is zero outside the specified interval, maintaining smooth transitions and compact support.

Value

A numeric vector or a single numeric value representing the value(s) of the asymmetric bump function at the input `x`. The function smoothly approaches zero at the boundaries of its support and has different behavior to the right of the offset.

Examples

```
## Not run:
x <- seq(-2, 2, length.out = 100)
plot(x, AsymmetricBumpFunction(x), type = "l")

## End(Not run)
```

`right.winsorize`	*Right Winsorize a numeric vector*

Description

This function applies right-winsorization to a numeric vector, replacing extreme values on the right tail.

Usage

```
right.winsorize(y, p = 0.01)
```

Arguments

`y`	A numeric vector to be right-winsorized
`p`	A numeric value between 0 and 0.25 indicating the proportion of data to be replaced on the right tail (default is 0.01)

Value

A numeric vector with extreme values on the right tail replaced

Examples

```
y <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
right.winsorize(y, p = 0.1)
```

`rlaplace`	*Generate Random Variates from the Laplace Distribution*

Description

This function generates random variates from the Laplace distribution (also known as the double exponential distribution) with specified location and scale parameters.

Usage

```
rlaplace(n, location = 0, scale = 1, seed = NULL)
```

Arguments

`n`	An integer specifying the number of observations to generate. Must be a positive integer.
`location`	A numeric value specifying the location parameter μ of the distribution. Default is 0.
`scale`	A numeric value specifying the scale parameter b of the distribution. Must be positive. Default is 1.
`seed`	An optional integer specifying the random seed. If NULL (the default), a random seed will be used.

Details

The probability density function of the Laplace distribution is:

$$f(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x-\mu|}{b}\right)$$

where μ is the location parameter and b > 0 is the scale parameter.

Value

A numeric vector of length n containing the generated random variates.

Note

This function uses a C++ implementation for efficient random number generation. The underlying algorithm uses the inverse transform sampling method.

Examples

```
# Generate 1000 random variates from the standard Laplace distribution
x <- rlaplace(1000)

# Generate 500 random variates with location 2 and scale 3, using a specific seed
y <- rlaplace(500, location = 2, scale = 3, seed = 12345)
```

`rllm.2os.1D` *1D local linear model of two outcomes*

Description

1D local linear model of two outcomes

Usage

```
rllm.2os.1D(
  x,
  y1,
  y2,
  grid.size = 400,
  degree = 2,
  f = NULL,
  bw = NULL,
  n.BB = 1000,
  get.predictions.CrI = TRUE,
  level = 0.95,
  n.C.itr = 100,
  C = 6,
  stop.C.itr.on.min = TRUE,
  n.cv.folds = 10,
  n.cv.reps = 5,
  min.K = 5,
  nn.kernel = "epanechnikov",
  y1.binary = FALSE,
  cv.nNN = 3,
  verbose = FALSE
)
```

Arguments

`x`	A numeric vector of a predictor variable.
`y1`	A numeric vector of the first outcome variable.
`y2`	A numeric vector of the second outcome variable.
`grid.size`	A number of grid points; was grid.size = 10*length(x), but the results don't seem to be different from 400 which is much faster.
`degree`	A degree of the polynomial of x in the linear regresion; 0 means weighted mean, 1 is regular linear model lm(y ~ x), and deg = d is lm(y ~ poly(x, d)). The only allowed values are 1 and 2.
`f`	The proportion of the range of x that is used within a moving window to train the model. If NULL, the optimal value of f will be found using minimum median absolute error optimization algorithm.
`bw`	A bandwidth parameter.
`n.BB`	The number of Bayesian bootstrap (BB) iterations for estimates of CI's of beta's.
`get.predictions.CrI`	A logical parameter. If TRUE, BB with quantile determined by the value of 'level' will be used to determine the upper and lower limits of CI's.
`level`	A confidence level.
`n.C.itr`	The number of Cleveland's absolute residue based reweighting iterations for a robust estimates of mean y values.
`C`	A scaling of |res| parameter changing |res| to |res|/C before applying ae.kernel to |res|'s.
`stop.C.itr.on.min`	A logical variable, if TRUE, the Cleveland's iterative reweighting stops when the maximum of the absolute values of differences of the old and new predictions estimates are reach a local minimum.
`n.cv.folds`	The number of cross-validation folds. Used only when f = NULL. Default value: 10.
`n.cv.reps`	The number of repetiions of cross-validation. Used only when f = NULL. Default value: 5.
`min.K`	The minimal number of x NN's that must be present in each window.
`nn.kernel`	A kernel.
`y1.binary`	Set to TRUE if y1 is binary.
`cv.nNN`	The number of nearest neighbors in interpolate_gpredictions() used to find predictions given gpredictions in the cv_deg0_ routines.
`verbose`	Prints info about what is being done.

Value

A list of input parameters as well as coefficients and residues of all linear models

rllmf.1D *A helper 1D local linear model routine for smoothing predictions.CI's*

Description

A helper 1D local linear model routine for smoothing predictions.CI's

Usage

```
rllmf.1D(
  x,
  y,
  grid.size = 400,
  degree = 2,
  f = 0.3,
  bw = NULL,
  n.C.itr = 0,
  C = 6,
  stop.C.itr.on.min = TRUE,
  n.cv.folds = 10,
  n.cv.reps = 1,
  n.BB = 0,
  min.K = 5,
  nn.kernel = "epanechnikov"
)
```

Arguments

`x`	A numeric vector of a predictor variable.
`y`	A numeric vector of an outcome variable.
`grid.size`	A number of grid points; was grid.size = 10*length(x), but the results don't seem to be different from 400 which is much faster.
`degree`	A degree of the polynomial of x in the linear regresion; 0 means weighted mean, 1 is regular linear model lm(y ~ x), and deg = d is lm(y ~ poly(x, d)). The only allowed values are 1 and 2.
`f`	The proportion of the range of x that is used within moving window to train the model. It can have NULL value in which case the optimal value of f will be found using minimum median absolute error optimization algorithm.
`bw`	A bandwidth parameter.
`n.C.itr`	The number of Cleveland's absolute residue based reweighting iterations for a robust estimates of mean y values.
`C`	A scaling of |res| parameter changing |res| to |res|/C before applying ae.kernel to |res|'s.
`stop.C.itr.on.min`	A logical variable, if TRUE, the Cleveland's iterative reweighting stops when the maximum of the absolute values of differences of the old and new predictions estimates are reach a local minimum.
`n.cv.folds`	The number of cross-validation folds. Used only when f = NULL. Default value: 10.

`n.cv.reps`	The number of repetiions of cross-validation. Used only when f = NULL. Default value: 5.
`n.BB`	The number of Bayesian bootstrap iterations for estimates of CI's of beta's.
`min.K`	The minimal number of x NN's that must be present in each window.
`nn.kernel`	A kernel.

Value

list of parameters and residues of all linear models

`rm.self.loops` *Removes self-loops from a graph*

Description

This function takes an adjacency list representation of a graph and removes all self-loops from it. A self-loop is an edge that connects a vertex to itself.

Usage

```
rm.self.loops(adj.list)
```

Arguments

`adj.list`	A named list representing the adjacency list of the graph. Each element of the list is a numeric vector containing the indices of the adjacent vertices for a given vertex. The names of the list elements correspond to the vertex names.

Value

A named list representing the adjacency list of the graph with self-loops removed. The format is the same as the input adjacency list, but without any self-loops.

Examples

```
graph <- list(
  "A" = c(1, 2),
  "B" = c(2, 3),
  "C" = c(1, 3),
  "D" = c(4)
)

graph.no.self.loops <- rm.self.loops(graph)
print(graph.no.self.loops)
```

rm.SS.outliers *Remove outliers from a state space*

Description

Removes outliers from a state space (SS) and optionally from a variable defined over it using k-nearest neighbor distances.

Usage

```
rm.SS.outliers(
  S,
  y = NULL,
  p = 0.98,
  dist.factor = 100,
  K = 30,
  method = "diff.dist.factor"
)
```

Arguments

`S`	A numeric matrix or data frame representing the state space, where rows are observations and columns are dimensions.
`y`	An optional numeric vector of length `nrow(S)` representing a variable defined over the state space. Default is NULL.
`p`	Numeric between 0 and 1. The percentile threshold for outlier removal. Default value of 0.98 removes samples whose distance to the nearest neighbor is greater than the 98th percentile of all nearest neighbor distances.
`dist.factor`	A positive numeric constant used for outlier selection. For method "dist.factor", points with ratio of furthest to nearest neighbor distance exceeding this value are considered outliers. For method "diff.dist.factor", points with relative jump in neighbor distances exceeding this value are considered outliers. Default is 100.
`K`	Integer specifying the number of nearest neighbors to use for outlier identification. Must be positive. Default is 30.
`method`	Character string specifying the outlier detection method when K > 1. Options are: • "diff.dist.factor" (default): Uses the maximum jump in consecutive neighbor distances relative to the median jump. • "dist.factor": Uses the ratio of the K-th to 1st neighbor distance. • Any other value: Uses a threshold-based approach on all neighbor distances.

Value

A list containing:

`S.q`	The filtered state space with outliers removed
`y.q`	The filtered y variable (NULL if y was NULL)
`nn.d`	The K-nearest neighbor distance matrix from the original data

`d.thld`	The distance threshold used (p-th percentile)
`idx`	Logical vector indicating which observations were kept (TRUE) or removed as outliers (FALSE)

See Also

`get.knn` for K-nearest neighbor computation

Examples

```
## Not run:
# Create example data
set.seed(123)
S <- matrix(rnorm(200), ncol = 2)
y <- rnorm(100)

# Remove outliers
result <- rm.SS.outliers(S, y, p = 0.95)

# Check how many points were removed
sum(!result$idx)

## End(Not run)
```

`robust.log.transform` *Returns robust log transform of a non-negative vector*

Description

This function applies a log transformation to positive values and normalizes them by their mean, while keeping zero values as zero. This creates a scale-invariant transformation that preserves the relative differences between positive values.

Usage

```
robust.log.transform(x)
```

Arguments

`x`	A numeric vector of non-negative numbers.

Details

The transformation process:

1. Positive values are log-transformed: `log(x[i])` for `x[i] > 0`
2. The mean of log-transformed values is calculated: `M = mean(log(x[x > 0]))`
3. Log-transformed values are divided by M for normalization
4. Zero values remain unchanged as 0

Value

A numeric vector of the same length as x, where positive values are log-transformed and normalized by the mean of log-transformed values, and zero values remain zero.

Note

This function assumes all input values are non-negative. Negative values will cause unexpected behavior as they are not checked.

Examples

```
# Example 1: Basic usage
x <- c(0, 1, 10, 100, 1000)
robust.log.transform(x)

# Example 2: Scale invariance property
x1 <- c(0, 1, 2, 5, 10)
x2 <- c(0, 10, 20, 50, 100)  # x1 * 10
# The relative differences are preserved
robust.log.transform(x1)
robust.log.transform(x2)
```

`robust.transform` *Returns robust transform of a non-negative vector*

Description

This function normalizes positive values by their geometric mean while keeping zero values as zero. The geometric mean is more robust to outliers than the arithmetic mean, making this transformation useful for data with skewed distributions.

Usage

```
robust.transform(x)
```

Arguments

`x`	A numeric vector of non-negative numbers.

Details

The transformation process:

1. The geometric mean of positive values is calculated: `D = exp(mean(log(x[x > 0])))`
2. Each positive value is divided by `D`: `x[i] / D` for `x[i] > 0`
3. Zero values remain unchanged as 0

The geometric mean of the transformed positive values will be 1.

Value

A numeric vector of the same length as x, where positive values are divided by the geometric mean of all positive values, and zero values remain zero.

Note

This function assumes all input values are non-negative. Negative values will cause unexpected behavior as they are not checked.

Examples

```
# Example 1: Basic usage
x <- c(0, 1, 4, 16, 64)
robust.transform(x)
# Geometric mean of positive transformed values equals 1

# Example 2: Comparison with arithmetic mean normalization
x <- c(0, 1, 2, 100)  # Outlier present
# Robust (geometric mean) normalization
robust.transform(x)
# Compare to arithmetic mean normalization
x_arith <- x
x_arith[x > 0] <- x[x > 0] / mean(x[x > 0])
x_arith  # More affected by the outlier
```

`robust.z.normalize` *Perform Robust Z-Score Normalization*

Description

Normalizes data using robust statistics by subtracting the median and dividing by the Median Absolute Deviation (MAD). This method is more resistant to outliers compared to traditional z-score normalization that uses mean and standard deviation.

Usage

```
robust.z.normalize(x)
```

Arguments

`x`	A numeric vector to be normalized

Details

The function implements robust z-score normalization using the formula: z = (x - median(x)) / MAD(x)

The MAD is scaled by a factor of 1.4826 to make it consistent with the standard deviation when the data is normally distributed.

Value

A numeric vector of the same length as the input containing robust z-scores. Missing values (NA) in the input will result in NA in the output at the same positions.

Note

If the MAD is zero, the function will return a vector of zeros and issue a warning.

Examples

```
# Basic usage
data <- c(1, 2, 3, 100, 4, 5, 6)  # Note the outlier
robust.z.normalize(data)

# Handling missing values
data.with.na <- c(1, NA, 3, 100, 4, NA, 6)
robust.z.normalize(data.with.na)
```

`robust.zscore` *Robust Z-score normalization using median and MAD*

Description

Robust Z-score normalization using median and MAD

Usage

```
robust.zscore(data, scale.factor = 1.4826)
```

Arguments

`data`	A numeric matrix or data frame where rows are samples and columns are features
`scale.factor`	Scaling factor for MAD (default: 1.4826 to make MAD consistent with standard deviation for normal distributions)

Value

A matrix of robust Z-scored values

Examples

```
# Example with random data
set.seed(123)
example.data <- matrix(rnorm(100, 5, 2), ncol=5)
# Add some outliers
example.data[1,1] <- 25
example.data[2,3] <- -15
normalized.data <- robust.zscore(example.data)
```

`row.eval` *Row-wise evaluates x at nn.i.*

Description

Row-wise evaluates x at nn.i.

Usage

```
row.eval(nn.i, x)
```

Arguments

`nn.i`	An array of indices of K nearest neighbors of the i-th element of x.
`x`	An array of nx elements.

Value

A matrix obtained from x by evaluating it at the indices of nn.i.

`row.TS.norm` *Row-wise total sum normalization*

Description

Row-wise total sum normalization

Usage

```
row.TS.norm(x)
```

Arguments

`x`	A matrix.

Value

A matrix obtained from x by dividing the rows of x by the sum of the given row’s elements, if the sum is not 0.

row.weighting *Generates kernel defined wieghts of the rows of an input numeric metrix*

Description

Generates kernel defined wieghts of the rows of an input numeric metrix

Usage

```
row.weighting(x, bws, kernel.str = "epanechnikov")
```

Arguments

`x`	A numeric matrix.
`bws`	A numeric vector of bandwidths.
`kernel.str`	A character string indicating what kernel to apply to rows of x. ALERT: Values of max.K can be 0 if there is a row of x, where the weight of the first elements is 0. Thus `max.K[i] = 0` indicates that there are no non-zero elements in that row.

runif.simplex *Produces a random sample from the uniform distribution over the (K-1)-dimensional simplex.*

Description

Produces a random sample from the uniform distribution over the (K-1)-dimensional simplex.

Usage

```
runif.simplex(K)
```

Arguments

`K`	The size of vector of non-negative numbers that sum up to 1.

Value

A numeric vector of non-negative numbers that sum to 1.

`runif.sphere` *Uniformly Sample Points from a Unit Sphere Surface*

Description

Generates uniform random samples from the surface of a unit sphere in (dim+1)-dimensional space using the Gaussian normalization method. For example, dim=2 generates points on the surface of a 3D unit ball (2-sphere).

Usage

```
runif.sphere(n.samples, dim)
```

Arguments

`n.samples`	Integer > 0 indicating the number of samples to generate.
`dim`	Integer >= 0 indicating the intrinsic dimension of the sphere. The sphere will be embedded in (dim+1)-dimensional space. E.g., dim=2 for points on a regular 3D sphere surface.

Details

The function uses the fact that normalizing a vector of independent standard normal variables results in a uniform distribution on the sphere surface.

Value

A matrix of dimension `n.samples x (dim+1)` where each row represents a point uniformly sampled from the surface of the unit sphere. All points have unit Euclidean norm.

References

Muller, M. E. (1959). A note on a method for generating points uniformly on n-dimensional spheres. Communications of the ACM, 2(4), 19-20.

Examples

```
# Generate 10 points on a 2D circle (1-sphere)
points_circle <- runif.sphere(10, 1)

# Generate 100 points on a 3D sphere surface (2-sphere)
points_sphere <- runif.sphere(100, 2)
```

`runif.torus` *Generate Uniform Random Sample from n-dimensional Torus*

Description

Generates points uniformly distributed on an n-dimensional torus. Each point on the torus is represented by its embedding in 2n-dimensional Euclidean space, where each pair of coordinates represents a circle in a different dimension.

Usage

```
runif.torus(n.pts, dim = 1)
```

Arguments

`n.pts`	numeric; Number of points to generate
`dim`	numeric; Dimension of the torus (default = 1, which gives points on a circle)

Details

The n-dimensional torus is the product of n circles. For each dimension, a random angle is generated uniformly in `[0, 2pi]`, and its sine and cosine give the coordinates of the point's projection onto that circular component. The resulting points are uniformly distributed with respect to the natural (product) measure on the torus.

Value

A matrix with n.pts rows and 2*dim columns. Each row represents a point on the torus, with pairs of columns representing the (x,y) coordinates of each circular component.

Examples

```
# Generate 100 points on a circle (1-dimensional torus)
circle_points <- runif.torus(100, dim = 1)

# Generate 100 points on a 2-dimensional torus
torus_points <- runif.torus(100, dim = 2)
```

`sample.from.1d.density` *Sample points from an empirical probability distribution*

Description

This function generates random samples from a probability distribution defined by discrete density values over a grid. It uses inverse transform sampling:

1. Normalizes the input density values to create a proper PDF
2. Computes the cumulative distribution function (CDF)
3. Generates uniform random numbers
4. Uses inverse CDF method to transform uniform samples to the target distribution

Usage

```
sample.from.1d.density(n, y, x = NULL)
```

Arguments

`n`	Integer. Number of sample points to generate.
`y`	Numeric vector. Non-negative values representing the density/histogram heights at each grid point. Does not need to be normalized.
`x`	Optional numeric vector. Grid points corresponding to y values. Must be strictly increasing. If not provided, a uniform grid on `[0,1]` is created with length(y) points.

Details

The function performs the following steps:

1. If x is not provided, creates a uniform grid on `[0,1]`
2. Normalizes y to create proper PDF by ensuring y*dx integrates to 1
3. Computes CDF through cumulative trapezoidal integration
4. Generates n uniform random numbers
5. Uses binary search to find inverse CDF values

Value

```
 Numeric vector of length n containing samples from the distribution
        defined by the empirical density y over grid x.
```

`sample.from.empirical.distribution`
Sample points from empirical distribution with optional linear approximation

---

Description

This function implements the inverse transform sampling method to generate random samples from an empirical distribution. The algorithm consists of the following steps:

Usage

```
sample.from.empirical.distribution(
  n,
  y,
  nbins = 100,
  use.linear.approximation = FALSE
)
```

Arguments

`n`	Integer. Number of sample points to generate.
`y`	Numeric vector. Raw data points from which to estimate distribution.
`nbins`	Integer. Number of bins to use when estimating density. More bins provide finer resolution but may introduce noise.
`use.linear.approximation`	Logical. If TRUE, uses linear interpolation between CDF points. If FALSE, uses step function approach.

Details

1. Density Estimation:
   - Constructs a histogram of the input data using nbins equally spaced bins
   - The histogram counts represent the empirical density
2. PDF Normalization:
   - Normalizes the density values to create a proper PDF that integrates to 1
   - For histogram with equal bin widths dx: pdf = counts / (sum(counts) * dx)
3. CDF Computation:
   - Computes the cumulative distribution function (CDF) from the normalized PDF
   - For histogram with equal bin widths: `cdf[i] = sum(pdf[1:i]) * dx`
   - The CDF is a step function by nature, jumping at bin midpoints
4. Inverse Transform Sampling:
   - Generates n uniform random numbers u ~ U(0,1)
   - For each u, finds x such that F(x) = u, where F is the CDF
   - Two methods available for finding x: a) Step Function (use.linear.approximation=FALSE):
     - Returns the first x value where CDF $\geq$ u
     - Preserves the discrete nature of the histogram b) Linear Interpolation (use.linear.approximation=TRUE):
     - Linearly interpolates between surrounding x values
     - Smooths the distribution but may not perfectly represent histogram

Value

```
Numeric vector of length n containing samples from the distribution
        estimated from y.
```

Examples

```
# Generate bimodal test data
y <- c(rnorm(1000, 0.3, 0.1), rnorm(1000, 0.7, 0.1))

# Sample using step function (default)
samples1 <- sample.from.empirical.distribution(1000, y)

# Sample using linear approximation
samples2 <- sample.from.empirical.distribution(1000, y, use.linear.approximation=TRUE)

# Compare distributions
par(mfrow=c(1,3))
hist(y, main="Original Data")
hist(samples1, main="Step Function Sampling")
hist(samples2, main="Linear Interpolation Sampling")
```

`sample.gaussian_mixture` *Sample and evaluate points from a Gaussian mixture*

Description

Generates a dataset by sampling points from a domain and evaluating a Gaussian mixture function at those points, with optional noise.

Usage

```
sample.gaussian_mixture(
  mixture,
  n = 500,
  sampling.method = c("random", "grid", "stratified"),
  noise.sd = 0.1,
  seed = NULL,
  ...
)
```

Arguments

`mixture`	A gaussian_mixture object from get.gaussian.mixture.2d()
`n`	Number of points to sample
`sampling.method`	Method for sampling: "random", "grid", or "stratified"
`noise.sd`	Standard deviation of additive Gaussian noise
`seed`	Random seed for reproducibility (default: NULL)
`...`	Additional parameters

Value

A list of class "gaussian_mixture_data" containing:

`X`	Matrix of sampled coordinates (n x 2)
`y`	Vector of noisy function values
`y.true`	Vector of true function values
`mixture`	The original gaussian_mixture object

`scaled.log.tan` *Applies a Scaled Log Transformation*

Description

This function first applies a scaled log transformation, C*log(x), to the input variable 'x'. The scaling is performed such that the range of the transformed variable 'x' lies within the interval `[-pi/2, pi/2]`. Subsequently, a tangent function is applied to the result of the transformation.

Usage

```
scaled.log.tan(x, scale = 0.1)
```

Arguments

`x`	The input variable to undergo transformation.
`scale`	A value representing the desired offset of the transformed variable range from -pi/2 and pi/2.

Value

Returns the tangent of the scaled log of x.

Examples

```
## Not run:
x <- 0.1
scaled.log.tan(x)

## End(Not run)
```

`scale_to_range` *Linear transformation of 'x' values so that the min(x) is sent to ymin and max(x) to ymax*

Description

Linear transformation of 'x' values so that the min(x) is sent to ymin and max(x) to ymax

Usage

```
scale_to_range(x, ymin, ymax, xrange = NULL)
```

Arguments

`x`	Input values
`ymin`	The value to which min(x) is sent.
`ymax`	The value to which max(x) is sent.
`xrange`	The minimum of x and the maximum of x that is to be mapped to ymin and ymax, respectively.

Value

A numeric vector of the same length as 'x' with values linearly transformed to the range `[ymin, ymax]`.

`separatrices.plot` *Plot Separatrices*

Description

Plots separatrices emanating from saddle points.

Usage

```
separatrices.plot(
  separatrices,
  M1_pos,
  M2_pos,
  line.lwd = 2,
  line.lty = 2,
  line.col = "black",
  add = TRUE,
  ...
)
```

Arguments

`separatrices`	List of separatrix trajectories
`M1_pos`	Position of first maximum
`M2_pos`	Position of second maximum
`line.lwd`	Line width (default 2)
`line.lty`	Line type (default 2, dashed)
`line.col`	Line color (default "black")
`add`	Logical, whether to add to existing plot
`...`	Additional arguments passed to plot

Value

Invisible NULL

separatrices.with.cells.plot

Plot Separatrices with Cells

Description

Plots Morse-Smale cells defined by separatrices using filled polygons. This function visualizes the decomposition of the domain into cells based on gradient flow separatrices emanating from saddle points.

Usage

```
separatrices.with.cells.plot(
  separatrices,
  M1.pos,
  M2.pos,
  saddle_pos,
  add = TRUE,
 cell_colors = list(M1_m1 = rgb(0, 0.5, 0, alpha = 0.1), M1_m2 = rgb(1, 1, 0, alpha =
   0.1), M1_m3 = rgb(1, 0, 0, alpha = 0.1), M2_m2 = rgb(0, 0, 1, alpha = 0.1), M2_m3 =
    rgb(0.6, 0, 1, alpha = 0.1), M2_m4 = rgb(0, 1, 1, alpha = 0.1)),
  ...
)
```

Arguments

`separatrices`	List of separatrix trajectories containing: • boundary_left: Boundary critical point on left edge • boundary_right: Boundary critical point on right edge • boundary_top: Boundary critical point on top edge • boundary_bottom: Boundary critical point on bottom edge • saddle_ascending1: First ascending trajectory from saddle • saddle_ascending2: Second ascending trajectory from saddle • saddle_descending1: First descending trajectory from saddle • saddle_descending2: Second descending trajectory from saddle
`M1.pos`	Position of first maximum as c(x, y)
`M2.pos`	Position of second maximum as c(x, y)
`saddle_pos`	Position of saddle point as c(x, y)
`add`	Logical, whether to add to existing plot (default TRUE)
`cell_colors`	List of colors for each cell with names: • M1_m1: Cell connecting M1 to minimum m1 • M1_m2: Cell connecting M1 to minimum m2 • M1_m3: Cell connecting M1 to minimum m3 • M2_m2: Cell connecting M2 to minimum m2 • M2_m3: Cell connecting M2 to minimum m3 • M2_m4: Cell connecting M2 to minimum m4
`...`	Additional arguments passed to plot if add = FALSE

Value

Invisible NULL

Examples

```
## Not run:
# Typically used with pre-computed separatrices
# Example with a simple two-maximum system
separatrices <- list(
  boundary_left = c(0, 0.5),
  boundary_right = c(1, 0.5),
  boundary_top = c(0.5, 1),
  boundary_bottom = c(0.5, 0),
  saddle_descending1 = matrix(c(0.5, 0.5, 0.3, 0.3), nrow = 2, byrow = TRUE),
  saddle_descending2 = matrix(c(0.5, 0.5, 0.7, 0.7), nrow = 2, byrow = TRUE)
)
separatrices.with.cells.plot(separatrices,
                            M1.pos = c(0.3, 0.7),
                            M2.pos = c(0.7, 0.3),
                            saddle_pos = c(0.5, 0.5))

## End(Not run)
```

set.boundary *Compute Boundary Vertices of a Subset in a Graph*

Description

Computes the boundary of a subset U of vertices in a graph. A vertex v is in the boundary of U if and only if v is in U and its expanded neighborhood (itself and its adjacent vertices) intersects both U and the complement of U.

Usage

```
set.boundary(U, adj.list)
```

Arguments

`U`	Integer vector of vertex indices representing a subset of the graph vertices. Must contain valid indices between 1 and the number of vertices in the graph.
`adj.list`	List where each element contains the adjacency list for a vertex. Element `adj.list[[i]]` contains indices of vertices adjacent to vertex i. Self-loops are ignored.

Details

The boundary of a subset U consists of those vertices in U that are adjacent to at least one vertex outside of U. Mathematically, v is in the boundary of U if and only if v is in U and there exists a vertex w not in U such that w is adjacent to v.

This definition is consistent with the topological notion of boundary adapted to the discrete setting of graphs.

Value

Integer vector containing the indices of vertices in the boundary of U, sorted in ascending order. Returns an empty vector if U has no boundary vertices.

See Also

lmax.basins, lmin.basins

Examples

```
# Create a simple graph adjacency list
adj.list <- list(c(2, 3), c(1, 3, 4), c(1, 2), c(2, 5), c(4))

# Define a subset of vertices
U <- c(1, 2, 3)

# Compute the boundary
boundary <- set.boundary(U, adj.list)
print(boundary)  # Returns c(2) since only vertex 2 is adjacent to vertex 4 (outside U)
```

`set.complex.function.values`
Set Function Values on Nerve Complex Vertices

Description

This function sets function values at the vertices of the nerve complex.

Usage

```
set.complex.function.values(complex, values)
```

Arguments

complex	A nerve complex object
values	Vector of function values (one per vertex)

Value

The updated nerve complex object (invisibly)

`set.complex.weight.scheme` *Set Weight Scheme for Nerve Complex*

Description

This function sets the weighting scheme for simplices in the nerve complex.

Usage

```
set.complex.weight.scheme(complex, weight.type, params = numeric(0))
```

Arguments

`complex`	A nerve complex object
`weight.type`	Type of weight scheme to use
`params`	Parameters for the weight scheme (if needed)

Details

Available weight types:

- "uniform": All simplices have weight 1.0
- "inverse_distance": Weights are inversely proportional to average squared distance
- "gaussian": Weights use Gaussian kernel based on distances (param: sigma)
- "volume": Weights based on approximate simplex volume (param: alpha)
- "gradient": Weights based on function gradient over simplex (param: gamma)

Value

The updated nerve complex object (invisibly)

`shifted.tan` *Applies a Linear Transformation and Tangent Function*

Description

This function applies a linear transformation to the input variable 'x', scaling it so that the range of the transformed 'x' falls within the interval `[-pi/2 + offset, pi/2 - offset]`. The function then applies a tangent function to the result of the transformation.

Usage

```
shifted.tan(x, offset = 0.1)
```

Arguments

`x`	The input variable to undergo transformation.
`offset`	A value representing the desired offset of the transformed variable range from -pi/2 and pi/2.

Value

Returns the tangent of the linearly transformed 'x'.

Examples

```
## Not run:
x <- 0.001
shifted.tan(x)

## End(Not run)
```

`shortest.path` *Computes Shortest Path Distances for a Selected Set of Vertices of a Graph*

Description

This function computes the shortest paths between specified vertices in a weighted graph. It serves as an R interface to a C++ implementation of the shortest path algorithm.

Usage

```
shortest.path(graph, edge.lengths, vertices)
```

Arguments

`graph`	A list of numeric vectors. Each vector represents a vertex and contains the indices of its neighboring vertices.
`edge.lengths`	A list of numeric vectors. Each vector corresponds to a vertex in `graph` and contains the lengths of the edges to its neighbors.
`vertices`	A numeric vector of vertex indices for which to compute the shortest paths.

Details

The function first performs input validation, ensuring that `graph` and `edge.lengths` are properly formatted and consistent. It then converts the graph to 0-based indexing (as required by the C++ function) before calling the C++ implementation.

Value

A matrix where the element at position `(i,j)` represents the shortest path distance from `vertices[i]` to `vertices[j]`.

Note

The C++ function assumes 0-based indexing, but this R function accepts 1-based index inputs as is standard in R. The conversion is handled internally.

Examples

```
graph <- list(c(2,3), c(1,3,4), c(1,2,4), c(2,3))
edge.lengths <- list(c(1,4), c(1,2,5), c(4,2,1), c(5,1))
vertices <- c(1,3,4)
result <- shortest.path(graph, edge.lengths, vertices)
print(result)
```

`show.3d.cl` *Highlight a Specific Cluster in 3D Space*

Description

Shows a specified cluster with emphasis while displaying other clusters in gray

Usage

```
show.3d.cl(
  cl,
  cltr,
  X,
  cl.radius = 1e-04,
  show.ref.cltr = TRUE,
  show.labels = FALSE,
  adj = c(1.3, 0),
  ...
)
```

Arguments

`cl`	Cluster ID to highlight.
`cltr`	Vector of cluster IDs of length nrow(X).
`X`	A matrix or data.frame with 3 columns representing 3D coordinates.
`cl.radius`	Radius of spheres for the highlighted cluster.
`show.ref.cltr`	Logical. Whether to show non-highlighted clusters.
`show.labels`	Logical. Whether to show row names of highlighted points.
`adj`	Adjustment parameter for text positioning.
`...`	Additional arguments.

Details

This function creates a 3D plot where one specific cluster is highlighted in red while all other clusters are shown in gray. Optionally, labels can be shown for the highlighted cluster points.

Value

Invisibly returns NULL. The function is called for its side effect.

Examples

```
## Not run:
X <- matrix(rnorm(300), ncol = 3)
cltr <- sample(1:5, 100, replace = TRUE)

# Highlight cluster 3
show.3d.cl(3, cltr, X)

# Show with labels
rownames(X) <- paste0("Point", 1:100)
show.3d.cl(3, cltr, X, show.labels = TRUE)

## End(Not run)
```

---

show.basin *Show Basin Details from a Basin Complex*

Description

Retrieves and returns a specific basin from a basin complex object by its label. The function searches through both ascending (minima) and descending (maxima) basins.

Usage

```
show.basin(x, ...)
```

Arguments

x	A basin complex object returned by create.basin.cx().
...	Additional arguments passed to methods.

---

show.basin.basin_cx *Show Basin Details from a Basin Complex*

Description

Retrieves and returns a specific basin from a basin complex object by its label. The function searches through both ascending (minima) and descending (maxima) basins.

Usage

```
## S3 method for class 'basin_cx'
show.basin(x, basin.label, ...)
```

Arguments

x	A basin complex object returned by create.basin.cx().
basin.label	Character string specifying the basin label to retrieve (e.g., "M1" or "m2").
...	Additional arguments (not used).

Details

Basin labels follow the naming convention of the basin complex: "Mx" for maxima (descending) and "mx" for minima (ascending), where x is a sequential number assigned during the basin complex construction.

The returned basin list typically contains:

- vertex: 1-based index of the extremum vertex
- value: Function value at the extremum
- label: The basin label matching the requested label
- basin: Matrix with vertex indices and distances from the extremum
- basin_bd: Matrix of boundary vertices and their monotonicity spans

Value

A basin list corresponding to the requested label. If no basin with the specified label is found, the function returns NULL and prints a message.

`show.cltrs` *Display Cluster Labels in 3D Space*

Description

Shows cluster labels at the median center of each cluster in 3D space

Usage

```
show.cltrs(X, cltr, cex = 1, adj = c(0.5, 1))
```

Arguments

`X`	A matrix or data.frame representing a set of points in 3D space.
`cltr`	A vector of cluster assignments for the rows of X.
`cex`	Character expansion factor for cluster labels.
`adj`	Adjustment values for label positioning.

Details

This function calculates the median center of each cluster and displays the cluster label at that position. For clusters with only one member, the label is placed at the point itself.

Value

Invisibly returns a matrix of cluster centers with row names as cluster labels.

Examples

```
## Not run:
# Generate sample data
X <- matrix(rnorm(300), ncol = 3)
cltr <- sample(1:5, 100, replace = TRUE)

# Plot points first
plot3D.plain(X)

# Add cluster labels
show.cltrs(X, cltr)

## End(Not run)
```

`show.profile` *Display Profile of Top Components*

Description

Shows a profile of the columns with the largest mean values from a selected row of a matrix, typically used for analyzing biomarker abundances.

Usage

```
show.profile(i, Z, n.comp = 5)
```

Arguments

`i`	Integer. The row index at which to extract the profile.
`Z`	Matrix. A matrix of values (e.g., biomarker abundances) where rows represent samples and columns represent features.
`n.comp`	Integer. The number of top components to include in the profile (default: 5).

Details

This function extracts a single row from the input matrix and returns the features with the highest mean values. If the extracted row is a vector, it's converted to a single-row matrix for consistent processing.

Value

A named numeric vector containing the top `n.comp` features sorted by their mean values in descending order.

Examples

```
# Create example data
set.seed(123)
Z <- matrix(rnorm(100, mean = 10, sd = 2), nrow = 10, ncol = 10)
colnames(Z) <- paste0("Feature", 1:10)
```

```
# Get profile for row 1
profile <- show.profile(1, Z, n.comp = 3)
print(profile)
```

`show.tx` *Display and Cluster Taxon-Specific Embeddings*

Description

Reads taxon-specific 3D embedding data and performs or loads clustering analysis. Optionally reorders clusters by size and creates 3D visualization.

Usage

```
show.tx(
  taxon,
  data.dir = NULL,
  cltr.from.scratch = TRUE,
  min.pts = seq(5, 50, by = 5),
  reorder.cltrs = TRUE,
  show.plot = TRUE,
  n.cores = 10,
  verbose = TRUE
)
```

Arguments

`taxon`	Character string specifying the taxon name to process. Used to construct input/output file names.
`data.dir`	Character string specifying the directory containing data files. Should include trailing slash. If NULL, current directory is used.
`cltr.from.scratch`	Logical indicating whether to perform new clustering (TRUE) or load existing cluster assignments (FALSE). Default is TRUE.
`min.pts`	Numeric vector of minimum points parameters for HDBSCAN clustering. If length > 1, performs parameter selection. Default is seq(5, 50, by = 5).
`reorder.cltrs`	Logical indicating whether to reorder clusters by decreasing size. Default is TRUE.
`show.plot`	Logical indicating whether to display 3D visualization of clusters. Default is TRUE.
`n.cores`	Integer specifying number of cores for parallel processing. Default is 10.
`verbose`	Logical indicating whether to print progress messages. Default is TRUE.

Details

The function expects the following file naming convention:

- PaCMAP embedding: `<data.dir><taxon>_pacmap.csv`
- Clustering: `<data.dir><taxon>_pacmap_cltr.csv`
- Extended clustering: `<data.dir><taxon>_pacmap_cltr_ext.csv`

When cltr.from.scratch = FALSE and extended clustering file doesn't exist, it will be created using k-NN imputation.

Value

Invisibly returns a list containing:

`taxon`	The taxon name
`cltr.ext`	Extended cluster assignments (including imputed clusters)
`cltr`	Original cluster assignments
`X`	The 3D embedding matrix
`pacmap.file`	Path to the PaCMAP embedding file
`cltr.ext.file`	Path to the extended clustering file

Examples

```
## Not run:
# Process clustering for a specific taxon
result <- show.tx(taxon = "Bacteroides",
                  data.dir = "./data/",
                  cltr.from.scratch = TRUE)

## End(Not run)
```

`simple.pts3d.select` *Simple 3D Point Selection*

Description

Provides a simplified interface for selecting points in 3D space without visual feedback during selection.

Usage

```
simple.pts3d.select(X)
```

Arguments

`X`	Numeric matrix with 3 columns representing x, y, z coordinates of points in 3D space.

Details

This function provides a minimal interface for point selection in 3D space. Unlike `pts3d.select`, it doesn't provide visual feedback during selection. It assumes the points in `X` correspond to vertices in the current rgl scene.

Value

Integer vector containing the indices of selected points.

Examples

```
## Not run:
library(rgl)
# Create example 3D data
X <- matrix(rnorm(300), ncol = 3)

# Plot points
plot3d(X, col = "blue")

# Select points
selected_indices <- simple.pts3d.select(X)

# Highlight selected points
points3d(X[selected_indices, ], col = "red", size = 10)

## End(Not run)
```

`skewed.gaussian` *Compute Value of a Single Skewed Gaussian*

Description

The skewed Gaussian distribution is defined by the following formula:

f(x | xi, omega, alpha) = (2 / omega) * phi((x - xi) / omega) * Phi(alpha * ((x - xi) / omega))

where phi is the standard normal density and Phi is the standard normal cumulative distribution function.

Usage

```
skewed.gaussian(x, mu, sigma, alpha)
```

Arguments

`x`	Numeric. The point at which to evaluate the skewed Gaussian.
`mu`	Numeric. The location parameter (equivalent to xi in the formula).
`sigma`	Numeric. The scale parameter (equivalent to omega in the formula).
`alpha`	Numeric. The shape parameter controlling the skewness.

Details

This function calculates the value of a skewed Gaussian distribution at a given point.

The parameters of the distribution are:

- xi (equivalent to mu): location parameter
- omega (equivalent to sigma): scale parameter
- alpha: shape parameter controlling the skewness

Value

Numeric. The value of the skewed Gaussian distribution at point x.

Examples

```
skewed.gaussian(0, mu = 0, sigma = 1, alpha = 2)
```

---

smooth.PL.geodesic *Smooths a piece-wise linear (PL) geodesic path*

Description

Smooths a piece-wise linear (PL) geodesic path

Usage

```
smooth.PL.geodesic(
  S,
  path.vs,
  min.vs = 10,
  adj.n.vs = 50,
  grid.size = 200,
  bw = grid.size * 3/200,
  fb.C = 3
)
```

Arguments

`S`	A state space.
`path.vs`	A vector of indices of S constituting an geodesic path.
`min.vs`	The minimum number of vertices in path.vs for which a smoothing can be done. If |path.vs| < min.vs, then approx() is used to generate adj.n.vs vertices.
`adj.n.vs`	The number of vertices that is being created as the input for fb.magelo() in case |path.vs| < min.vs.
`grid.size`	The number of grid points; A parameter passed to magelo().
`bw`	The bandwidth of fb.magelo().
`fb.C`	The number of bw's away from the boundary points of the domain of x that the adjusting of Eyg will take place, so that Eyg satisfies the boundary condition.

Details

This function takes a piecewise linear geodesic path defined by vertex indices and produces a smooth version using local linear regression. If the input path has fewer than min.vs vertices, it first interpolates to create adj.n.vs points before smoothing. The smoothing is performed separately for each dimension of the state space.

Value

A matrix representing the smoothed geodesic path with grid.size rows and the same number of columns as the state space S. Each row contains the coordinates of a point along the smoothed path.

`spectral.lowess.graph.smoothing`
Iterative Spectral LOWESS Graph Smoothing

Description

Applies iterative spectral LOWESS (Locally Weighted Scatterplot Smoothing) to graph-structured data. The algorithm iteratively smooths features by estimating conditional expectations using spectral embeddings of local neighborhoods, then reconstructs the graph from the smoothed data.

Usage

```
spectral.lowess.graph.smoothing(
  adj.list,
  weight.list,
  X,
  max.iterations = 10,
  convergence.threshold = 1e-04,
  convergence.type = 1,
  k = 10,
  pruning.thld = 0.1,
  n.evectors = 8,
  n.bws = 10,
  log.grid = TRUE,
  min.bw.factor = 0.05,
  max.bw.factor = 0.5,
  dist.normalization.factor = 1.1,
  kernel.type = 7L,
  n.cleveland.iterations = 1,
  compute.errors = TRUE,
  compute.scales = TRUE,
  switch.to.residuals.after = NULL,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `adj.list[[i]]` contains the indices of vertices adjacent to vertex `i`. Indices must be between 1 and `length(adj.list)`.

`weight.list`	A list of numeric vectors containing edge weights corresponding to the adjacencies. Each element `weight.list[[i]][j]` is the weight of the edge from vertex `i` to vertex `adj.list[[i]][j]`. Weights must be non-negative. Must have the same structure as `adj.list`.
`X`	A numeric matrix where rows represent samples/vertices and columns represent features. Must have `nrow(X) == length(adj.list)`. Missing values are not allowed.
`max.iterations`	Maximum number of iterations to perform. Must be a positive integer. Default: 10.
`convergence.threshold`	Threshold for convergence. The algorithm stops when the convergence metric falls below this value. Must be positive. Default: 1e-4.
`convergence.type`	Type of convergence criterion to use: • 1: Maximum absolute difference between successive iterations • 2: Mean absolute difference between successive iterations • 3: Maximum relative change between successive iterations Default: 1.
`k`	Number of nearest neighbors for k-NN graph construction. Must be a positive integer less than the number of vertices. Default: 10.
`pruning.thld`	Threshold for pruning edges in graph construction. Edges with weights below this threshold are removed. Must be non-negative. Default: 0.1.
`n.evectors`	Number of eigenvectors to use for spectral embedding. Must be a positive integer. Larger values capture more graph structure but increase computation time. Default: 8.
`n.bws`	Number of candidate bandwidths to consider for LOWESS smoothing. Must be a positive integer. Default: 10.
`log.grid`	Logical. If `TRUE`, use logarithmic spacing for the bandwidth grid; if `FALSE`, use linear spacing. Default: `TRUE`.
`min.bw.factor`	Factor for minimum bandwidth, multiplied by the graph diameter to determine the actual minimum bandwidth. Must be positive and less than `max.bw.factor`. Default: 0.05.
`max.bw.factor`	Factor for maximum bandwidth, multiplied by the graph diameter to determine the actual maximum bandwidth. Must be positive and greater than `min.bw.factor`. Default: 0.5.
`dist.normalization.factor`	Factor for normalizing distances in kernel weight calculation. Must be at least 1.1. Higher values result in more uniform weights. Default: 1.1.
`kernel.type`	Integer specifying the kernel function for weighting: • 7: Gaussian kernel (default) • Other values may be supported depending on the C++ implementation Default: 7L.
`n.cleveland.iterations`	Number of robustness iterations for Cleveland's LOWESS algorithm. Must be a non-negative integer. Default: 1.
`compute.errors`	Logical. If `TRUE`, compute prediction errors for each iteration. May increase computation time. Default: `TRUE`.

`compute.scales`	Logical. If `TRUE`, compute bandwidth/scale information for each iteration. May increase computation time. Default: `TRUE`.
`switch.to.residuals.after`	Number of iterations to perform direct smoothing before switching to residual smoothing (boosting mode). Must be a non-negative integer. Set to 0 to use residual smoothing from the start. If `NULL`, defaults to `max.iterations` (never switch). Default: `NULL`.
`verbose`	Logical. If `TRUE`, print progress information during iterations. Default: `FALSE`.

Details

The algorithm performs the following steps iteratively:

1. Compute spectral embedding of the graph using eigenvectors of the normalized Laplacian
2. For each vertex, estimate conditional expectations of features using LOWESS on the spectral embedding coordinates
3. Reconstruct the graph from the smoothed features using k-NN with specified pruning
4. Check convergence criterion and stop if threshold is met

The boosting mode (residual smoothing) can be activated by setting `switch.to.residuals.after` to a value less than `max.iterations`. In this mode, the algorithm smooths residuals from previous iterations rather than the data directly, which can improve convergence in some cases.

Value

A list of class `"spectral.lowess.result"` containing:

`smoothed.graphs`	A list of length `iterations.performed`, where each element is a graph (represented as adjacency and weight lists) after the corresponding iteration
`smoothed.X`	A list of length `iterations.performed`, where each element is the smoothed data matrix after the corresponding iteration
`convergence.metrics`	A numeric vector of length `iterations.performed` containing the convergence metric at each iteration
`iterations.performed`	Integer indicating the number of iterations actually performed before convergence or reaching `max.iterations`
`used.boosting`	Logical indicating whether boosting (residual smoothing) was used in any iteration
`call`	The matched function call
`parameters`	A list containing all input parameters for reproducibility

Note

- The function requires a C++ implementation accessed via `.Call`
- Large graphs or high-dimensional data may require substantial memory
- The choice of convergence type affects both speed and quality of results
- Edge weights should typically be similarity measures (larger = more similar)

References

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368), 829-836.

Examples

```
## Not run:
# Generate synthetic data
set.seed(123)
n <- 100
p <- 10
X <- matrix(rnorm(n * p), nrow = n, ncol = p)

# Create a k-NN graph
library(FNN)
knn.result <- get.knn(X, k = 10)
adj.list <- lapply(1:n, function(i) knn.result$nn.index[i, ])
weight.list <- lapply(1:n, function(i) 1 / (1 + knn.result$nn.dist[i, ]))

# Apply spectral LOWESS smoothing
result <- spectral.lowess.graph.smoothing(
  adj.list = adj.list,
  weight.list = weight.list,
  X = X,
  max.iterations = 5,
  convergence.threshold = 1e-3,
  verbose = TRUE
)

# Examine results
cat("Iterations performed:", result$iterations.performed, "\n")
cat("Final convergence metric:",
    result$convergence.metrics[result$iterations.performed], "\n")

# Compare original and smoothed data
X.smoothed <- result$smoothed.X[[result$iterations.performed]]
par(mfrow = c(1, 2))
image(X, main = "Original Data", xlab = "Sample", ylab = "Feature")
image(X.smoothed, main = "Smoothed Data", xlab = "Sample", ylab = "Feature")

# Example with boosting
result.boost <- spectral.lowess.graph.smoothing(
  adj.list = adj.list,
  weight.list = weight.list,
  X = X,
  max.iterations = 10,
  switch.to.residuals.after = 3,
  verbose = TRUE
)

# Compare convergence
plot(result$convergence.metrics, type = "b", col = "blue",
     xlab = "Iteration", ylab = "Convergence Metric",
     main = "Convergence Comparison")
lines(result.boost$convergence.metrics, type = "b", col = "red")
legend("topright", legend = c("Standard", "Boosting"),
        col = c("blue", "red"), lty = 1, pch = 1)

## End(Not run)
```

`standardize.string` *Standardize String for File Names*

Description

Removes special characters from a string and replaces spaces with underscores to create valid file names or standardized identifiers.

Usage

```
standardize.string(str)
```

Arguments

`str`	Character string to be standardized. Typically a name that needs to be converted to a valid file name or identifier.

Details

This function is useful for converting names (such as metabolite names, sample identifiers, or feature names) into standardized formats suitable for use as file names or database keys. The function performs literal character replacement, not regular expression matching.

Value

Character string with special characters removed or replaced:

- Commas, spaces, and forward slashes are replaced with underscores
- Parentheses, colons, asterisks, apostrophes, and plus signs are removed

Examples

```
# Standardize a metabolite name
standardize.string("Glucose (6-phosphate)")
# Returns: "Glucose_6-phosphate"

# Standardize a complex name
standardize.string("Compound A/B + C's metabolite*")
# Returns: "Compound_A_B__Cs_metabolite"

# Use for creating file names
sample_name <- "Sample 1: Day 3 (replicate)"
file_name <- paste0(standardize.string(sample_name), ".csv")
# Results in: "Sample_1_Day_3_replicate.csv"
```

`subdivide.path` *Subdivides a path of points in $R\hat{}n$ into a uniform grid of points along the path.*

Description

Subdivides a path of points in R^n into a uniform grid of points along the path.

Usage

```
subdivide.path(path, n.subdivision.pts)
```

Arguments

`path`	A matrix of consecutive path points in some state space.
`n.subdivision.pts`	The number of points in the grid.

Details

This function takes a path defined by consecutive points and creates a new set of points that are uniformly distributed along the path's arc length. The subdivision is based on the cumulative Euclidean distances between consecutive points, ensuring that the output points are evenly spaced along the actual path length rather than just parametrically.

The first point of the output will be the first point of the input path, and the last point of the output will be the last point of the input path.

Value

A matrix with n.subdivision.pts rows and the same number of columns as the input path. Each row represents a point along the subdivided path, with points evenly spaced according to the path's arc length.

Examples

```
# Create a simple 2D path
path <- matrix(c(0,0, 1,0, 2,1, 3,1), ncol=2, byrow=TRUE)
# Subdivide into 10 evenly spaced points
subdivided <- subdivide.path(path, 10)
```

`subdivide.path.v1` *Subdivides a path of points in R^n into a uniform grid of points along the path.*

Description

Subdivides a path of points in R^n into a uniform grid of points along the path.

Usage

```
subdivide.path.v1(path, n.subdivision.pts)
```

Arguments

`path`	A matrix of consecutive path points in some state space.
`n.subdivision.pts`	The number of points in the grid.

Details

This is version 1 of the path subdivision algorithm. Like subdivide.path, it creates a set of points uniformly distributed along the path's arc length. The implementation differs slightly in the algorithm used but produces similar results.

Value

A matrix with n.subdivision.pts rows and the same number of columns as the input path. Each row represents a point along the subdivided path, with points evenly spaced according to the path's arc length.

Note

This appears to be an alternative implementation of subdivide.path. Consider using subdivide.path unless there's a specific reason to use this version.

See Also

`subdivide.path` for the primary implementation

`subj.D.risk` *Estimates subject risk of a disease or adverse outcome, D, given the risk D over some state space.*

Description

Estimates subject risk of a disease or adverse outcome, D, given the risk D over some state space.

Usage

```
subj.D.risk(sID, S, G.S, ED, subjID, visit)
```

Arguments

`sID`	A subject ID.
`S`	A state space.
`G.S`	A graph associated with S.
`ED`	An estimate of the risk of a disease or adverse outcome, D, over S.
`subjID`	A vector of subject IDs.
`visit`	A vector of visits.

Details

This function calculates disease risk for a specific subject by:

1. Finding the subject's visit states in the state space
2. Computing geodesic paths between consecutive visits
3. Interpolating risk values along these paths
4. Extracting maximum risk values and predicted risks

The function assumes the subject has at least 3 visits and uses geodesic paths to model the subject's trajectory through the state space between visits.

Value

A list containing risk estimates for the subject:

`subj.ids`	Character vector of subject visit IDs.
`subj.sVisit`	Numeric vector of subject visits.
`E.pred.D`	Predicted risk value interpolated along the geodesic path.
`E.D2`	Risk value at the second visit.
`E.D3`	Risk value at the third visit.
`subj.path12.ED.max`	Maximum risk value along the path from visit 1 to 2.
`subj.path23.ED.max`	Maximum risk value along the path from visit 2 to 3.

`subj.factor.ppmEy.profs.df` *Generates a data frame of proportion plus/minus dE.geodesic.F profiles.*

Description

Generates a data frame of proportion plus/minus dE.geodesic.F profiles.

Usage

```
subj.factor.ppmEy.profs.df(
  subj.factor.signEy.profs,
  rowInd,
  rlabs,
  save.file.prefix,
  N = 10
)
```

Arguments

`subj.factor.signEy.profs`	An output from subj.factor.signEy.profs().
`rowInd`	Row index permutation vector as returned by 'order.dendrogram'.
`rlabs`	Labels of the subject factors with levels.
`save.file.prefix`	The prefix of the save() files.
`N`	The number of +/0/- symbols in each profile.

Details

This function creates a visual representation of how factor expected values change along the geodesic by discretizing the derivative signs into N bins. Each row shows whether the factor is increasing (+), decreasing (-), or constant (0) in each bin along the geodesic path.

Value

A data frame with rows corresponding to factors (reordered by rowInd) and columns containing:

`Columns 1 to N`	Character columns containing "+", "-", or "0" symbols representing the discretized derivative profile of each factor.
`pval`	Numeric column containing p-values loaded from the saved files.
`lab`	Character column containing the factor labels.

See Also

`subj.factor.signEy.profs` for generating the input derivative signs `E.geodesic.F` for generating the data files containing p-values

---

`subj.factor.signEy.profs`   *Sings of the derivatives of the E.geodesic.F porfiles.*

Description

Sings of the derivatives of the E.geodesic.F porfiles.

Usage

```
subj.factor.signEy.profs(save.file.prefix, rlabs, grid.size = 400)
```

Arguments

`save.file.prefix`	The prefix of the save() files.
`rlabs`	Labels of the subject factors with levels.
`grid.size`	Grid size in E.geodesic.F estimates.

Details

This function computes the signs of the derivatives (differences) of the expected value profiles generated by E.geodesic.F. For each factor, it loads the saved results and calculates whether the expected values are increasing (sign = 1) or decreasing (sign = -1) between consecutive grid points along the geodesic.

Value

A matrix with dimensions length(rlabs) x (grid.size-1) containing the signs of the derivatives of the E.geodesic.F profiles. Each row corresponds to a factor in rlabs, and each column represents the sign of the derivative at consecutive grid points. Values are 1 for positive derivatives, -1 for negative derivatives, and NA for factors without corresponding data files.

See Also

`E.geodesic.F` for generating the data files used by this function

`summarize.and.test.cst` *Summarize and Test CST Patterns with Binary Outcomes*

Description

Analyzes the relationship between Community State Types (CSTs) and binary outcomes in longitudinal data where subjects have multiple CST measurements but a single, constant outcome. Provides two methods for summarizing repeated CST measurements before testing.

Usage

```
summarize.and.test.cst(
  x,
  y,
  subj.ids,
  pos.label = "sPTB",
  neg.label = "TB",
  summary.method = c("most_frequent", "proportions")
)
```

Arguments

`x`	factor or character vector of CST measurements, converted to factor if necessary
`y`	binary outcome vector that must be constant within each subject. Can be logical, numeric (0/1), or factor
`subj.ids`	vector of subject identifiers corresponding to CST measurements. Must be same length as `x`
`pos.label`	character, label for positive/case outcome (default: "sPTB")
`neg.label`	character, label for negative/control outcome (default: "TB")
`summary.method`	character, method for summarizing repeated CSTs. Must be either "most_frequent" (use each subject's modal CST) or "proportions" (calculate time spent in each CST). Default: "most_frequent"

Details

This function addresses the challenge of analyzing longitudinal microbiome data where the outcome of interest (e.g., preterm birth) is measured once per subject but microbiome composition (CST) is measured multiple times.

The "most_frequent" method identifies each subject's predominant CST and performs standard proportion tests. The "proportions" method calculates how much time each subject spends in each CST and uses Wilcoxon tests to compare these proportions between outcome groups.

Value

A list whose structure depends on `summary.method`:

If `summary.method = "most_frequent"`:

prop.test.mat matrix of proportion test results for modal CSTs

contingency contingency table of modal CSTs vs outcome

overall.prop overall proportion of positive outcome

latex.caption LaTeX caption for results table

If `summary.method = "proportions"`:

proportions matrix where rows are subjects and columns are CSTs, values are percentage of measurements in each CST

tests list of Wilcoxon test results comparing CST proportions between outcome groups

summary data frame with median proportions by outcome group and p-values for each CST

Note

This function assumes that the outcome is constant within subjects. If outcomes vary within subjects, consider alternative approaches or time-to-event analyses.

See Also

`analyze.weighted.cst` for time-weighted CST analysis

Examples

```
## Not run:
# Simulated longitudinal CST data
set.seed(123)
n_subjects <- 50
n_timepoints <- 5

# Generate subject IDs and CSTs
subj_ids <- rep(1:n_subjects, each = n_timepoints)
csts <- sample(c("I", "II", "III", "IV"), n_subjects * n_timepoints,
               replace = TRUE, prob = c(0.4, 0.3, 0.2, 0.1))

# Binary outcome (constant per subject)
outcomes <- rep(rbinom(n_subjects, 1, 0.3), each = n_timepoints)

# Analyze using most frequent CST
result_mode <- summarize.and.test.cst(
  x = csts,
  y = outcomes,
  subj.ids = subj_ids,
  summary.method = "most_frequent"
)

# Analyze using CST proportions
result_prop <- summarize.and.test.cst(
  x = csts,
  y = outcomes,
  subj.ids = subj_ids,
  summary.method = "proportions"
)

print(result_prop$summary)

## End(Not run)
```

`summary.amagelo` *Summary method for amagelo objects*

Description

Summary method for amagelo objects

Usage

```
## S3 method for class 'amagelo'
summary(object, digits = 4, ...)
```

Arguments

`object`	An object of class 'amagelo'
`digits`	Number of digits to display (default: 4)
`...`	Additional arguments passed to summary

`summary.assoc0` *Summary Method for assoc0 Objects*

Description

Summary Method for assoc0 Objects

Usage

```
## S3 method for class 'assoc0'
summary(object, ...)
```

Arguments

`object`	An object of class "assoc0".
`...`	Additional arguments (currently ignored).

Value

A list of class "summary.assoc0" with summary statistics.

`summary.assoc1` *Summary Method for assoc1 Objects*

Description

Summary Method for assoc1 Objects

Usage

```
## S3 method for class 'assoc1'
summary(object, ...)
```

Arguments

`object`	An object of class "assoc1".
`...`	Additional arguments (currently ignored).

Value

A list of class "summary.assoc1" with summary statistics.

summary.basin_cx *Summary Method for Basin Complex Objects*

Description

Generates a comprehensive summary of a gradient flow basin complex, including detailed information about extrema, basins, basin overlaps, and cell complex structure.

Usage

```
## S3 method for class 'basin_cx'
summary(object, ...)
```

Arguments

`object`	An object of class "basin_cx" as created by `create.basin.cx`.
`...`	Additional arguments (currently unused).

Details

The summary method computes several key statistics and relationships:

Extrema Analysis: Counts and details of all local minima and maxima in the original function.

Basin Statistics: For both ascending (minimum) and descending (maximum) basins, provides counts, size distributions, and detailed information including the representative vertex, label, size (number of vertices), and function value.

Basin Overlap: Computes Jaccard indices between all pairs of basins of the same type. The Jaccard index measures the overlap between two basins as the ratio of their intersection size to their union size, ranging from 0 (no overlap) to 1 (complete overlap).

Cell Complex: Summarizes the structure of cells representing intersections between basins.

Value

An object of class "summary.basin_cx" containing:

n_extrema Total number of local extrema

n_minima Number of local minima

n_maxima Number of local maxima

minima Data frame of local minima information

maxima Data frame of local maxima information

n_ascending_basins Number of merged ascending basins (minima)

n_descending_basins Number of merged descending basins (maxima)

ascending_basin_sizes Summary statistics of ascending basin sizes

descending_basin_sizes Summary statistics of descending basin sizes

minima_info Detailed information about each minimum basin including vertex, label, size, and function value

maxima_info Detailed information about each maximum basin including vertex, label, size, and function value

ascending_basins_jaccard_index Matrix of Jaccard indices between pairs of ascending basins

descending_basins_jaccard_index Matrix of Jaccard indices between pairs of descending basins

cells_summary Summary of cell complex structure

messages Any messages generated during basin complex creation

See Also

`create.basin.cx` for creating basin complex objects, `print.basin_cx` for basic basin complex information, `print.summary.basin_cx` for printing summary objects

Examples

```
## Not run:
# Create a basin complex
adj_list <- list(c(2,3), c(1,3,4), c(1,2,5), c(2,5), c(3,4))
weight_list <- list(c(1,2), c(1,1,3), c(2,1,2), c(3,1), c(2,1))
y <- c(2.5, 1.8, 3.2, 0.7, 2.1)
basin_cx <- create.basin.cx(adj_list, weight_list, y)

# Get detailed summary
basin_summary <- summary(basin_cx)

# Access specific components
basin_summary$minima_info
basin_summary$ascending_basins_jaccard_index

## End(Not run)
```

`summary.fassoc` *Summary Method for Functional Association Tests*

Description

Summary Method for Functional Association Tests

Usage

```
## S3 method for class 'fassoc'
summary(object, ...)
```

Arguments

`object`	An object of class "assoc0" or "assoc1".
`...`	Additional arguments passed to specific summary methods.

Value

A summary object.

summary.gaussian_mixture
Summary method for gaussian_mixture objects

Description

Summary method for gaussian_mixture objects

Usage

```
## S3 method for class 'gaussian_mixture'
summary(object, ...)
```

Arguments

object	A gaussian_mixture object
...	Additional arguments (unused)

Value

A summary list

summary.geodesic_stats
Summary method for geodesic_stats objects

Description

Provides a summary table of geodesic statistics with percentages and counts for different metric types.

Usage

```
## S3 method for class 'geodesic_stats'
summary(object, type = c("rays", "composite", "overlap"), ...)
```

Arguments

object	A geodesic_stats object from compute.geodesic.stats().
type	Character. Type of summary to generate: "rays" (default), "composite", or "overlap".
...	Additional arguments passed to summary.

Value

A matrix with row for each radius and columns for each unique count value. Each cell contains a string with "percent (count)" format.

Examples

```
## Not run:
stats <- compute.geodesic.stats(adj.list, weight.list)
summary(stats)  # Default type = "rays"
summary(stats, type = "composite")
summary(stats, type = "overlap")

## End(Not run)
```

---

summary.gflow_basins *Summarize Gradient-Flow Basin Results*

Description

Summarize Gradient-Flow Basin Results

Usage

```
## S3 method for class 'gflow_basins'
summary(object, ...)
```

Arguments

object	An object of class "gflow_basins"
...	Additional arguments (currently ignored)

Value

An object of class "summary.gflow_basins" containing:

n_extrema Total number of extrema detected

n_minima Number of local minima

n_maxima Number of local maxima

n_ascending_basins Number of ascending basins

n_descending_basins Number of descending basins

ascending_basin_sizes Summary statistics for ascending basin sizes

descending_basin_sizes Summary statistics for descending basin sizes

summary.ggflow *Summarize Gradient Flow Structure*

Description

Provides a comprehensive summary of a gradient flow structure including information about local extrema, basins, trajectories, cells, and Jaccard indices between basins.

Usage

```
## S3 method for class 'ggflow'
summary(object, ...)
```

Arguments

`object`	An object of class `"ggflow"`.
`...`	Additional arguments (currently ignored).

Value

An object of class `"summary.ggflow"` containing:

n_extrema Total number of extrema

n_maxima Number of local maxima

n_minima Number of local minima

maxima Data frame with information about local maxima

minima Data frame with information about local minima

n_trajectories Number of trajectories (if available)

trajectory_types Table of trajectory type counts

avg_trajectory_length Average trajectory length

connectivity Matrix showing min-max connections

n_cells Number of gradient flow cells

cell_sizes Summary statistics of cell sizes

n_ascending_basins Number of ascending basins

n_descending_basins Number of descending basins

ascending_basins_jaccard_index Jaccard similarity matrix for ascending basins

descending_basins_jaccard_index Jaccard similarity matrix for descending basins

Examples

```
## Not run:
flow <- construct.graph.gradient.flow(adj.list, weight.list, y, scale)
summary(flow)

## End(Not run)
```

`summary.graph.spectral.lowess`
Utility functions for graph.spectral.lowess

Description

This file contains utility functions for working with graph.spectral.lowess objects, including summary statistics, predictions, residuals, and other methods. Summary Statistics for Graph Spectral LOWESS

Provides comprehensive summary statistics for graph spectral LOWESS fits, including model information, fit statistics, residuals analysis, and bandwidth selection details.

Usage

```
## S3 method for class 'graph.spectral.lowess'
summary(object, quantiles = c(0, 0.25, 0.5, 0.75, 1), ...)
```

Arguments

`object`	A 'graph.spectral.lowess' object
`quantiles`	Numeric vector of quantiles for residual statistics (default: c(0, 0.25, 0.5, 0.75, 1))
`...`	Additional arguments (currently unused)

Value

A 'summary.graph.spectral.lowess' object containing:

- model_info: Basic information about the model fit
- fit_stats: Prediction accuracy metrics
- bandwidth_stats: Bandwidth selection statistics
- residual_stats: Residual analysis results
- component_info: Graph component information

`summary.graph.spectral.ma.lowess`
Utility functions for graph.spectral.ma.lowess

Description

This file contains utility functions for working with graph.spectral.ma.lowess objects, including summary statistics, predictions, residuals, and other methods. These functions are designed to handle the model averaging aspects of the MA version. Summary Statistics for Graph Spectral MA LOWESS

Provides comprehensive summary statistics for graph spectral model-averaged LOWESS fits, including model information, fit statistics, residuals analysis, bandwidth selection details, and model averaging information.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
summary(object, quantiles = c(0, 0.25, 0.5, 0.75, 1), ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`quantiles`	Numeric vector of quantiles for residual statistics (default: c(0, 0.25, 0.5, 0.75, 1))
`...`	Additional arguments (currently unused)

Value

A 'summary.graph.spectral.ma.lowess' object containing:

- model_info: Basic information about the model fit
- fit_stats: Prediction accuracy metrics
- bandwidth_stats: Bandwidth selection statistics
- model_averaging_info: Information about model averaging
- residual_stats: Residual analysis results
- component_info: Graph component information

`summary.graph_kernel_smoother` *Summary Method for graph_kernel_smoother Objects*

Description

Computes and displays detailed summary statistics for a `graph_kernel_smoother` object.

Usage

```
## S3 method for class 'graph_kernel_smoother'
summary(object, digits = 4, ...)
```

Arguments

`object`	A `graph_kernel_smoother` object.
`digits`	Number of significant digits to display. Default is 4.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns a list containing summary statistics with components:

`n_vertices`	Number of vertices in the graph
`n_bandwidths`	Number of bandwidths tested
`opt_bw`	Optimal bandwidth value
`opt_bw_idx`	Index of optimal bandwidth
`buffer_hops_used`	Buffer hop distance used
`min_error`	Minimum cross-validation error
`bw_errors`	Vector of all cross-validation errors
`pred_mean`	Mean of predictions
`pred_sd`	Standard deviation of predictions
`pred_range`	Range of predictions
`pred_quantiles`	Quantiles of predictions

---

summary.graph_spectral_filter *Summary Method for Graph Spectral Filter Results*

Description

Summary Method for Graph Spectral Filter Results

Usage

```
## S3 method for class 'graph_spectral_filter'
summary(object, ...)
```

Arguments

`object`	An object of class `"graph_spectral_filter"`
`...`	Further arguments passed to or from other methods

Value

An object of class `"summary.graph_spectral_filter"` containing key summary statistics `summary.harmonic_smoother` *Summary Method for Harmonic Smoother Results*

Description

Provides detailed summary statistics for harmonic smoother results, including the evolution of extrema counts and basin structure differences.

Usage

```
## S3 method for class 'harmonic_smoother'
summary(object, ...)
```

Arguments

`object`	An object of class `"harmonic_smoother"`.
`...`	Further arguments passed to or from other methods.

Value

An object of class `"summary.harmonic_smoother"` containing:

`stable_iteration`	The iteration at which topology stabilized
`iterations_recorded`	Total number of recorded iterations
`initial_extrema`	Number of extrema at the first iteration
`initial_maxima`	Number of maxima at the first iteration
`initial_minima`	Number of minima at the first iteration
`final_extrema`	Number of extrema at the final iteration
`final_maxima`	Number of maxima at the final iteration
`final_minima`	Number of minima at the final iteration
`extrema_reduction`	Reduction in number of extrema
`topology_diff_summary`	Summary statistics of topology differences

See Also

`harmonic.smoother`, `print.summary.harmonic_smoother` `summary.IkNN` *Summarize IkNN Graph Object*

Description

Prints a summary of an IkNN graph object, including the number of vertices, edges, and pruning statistics.

Usage

```
## S3 method for class 'IkNN'
summary(object, ...)
```

Arguments

`object`	An object of class "IkNN", typically output from create.single.iknn.graph()
`...`	Additional arguments passed to summary().

Value

Invisibly returns NULL while printing summary information to the console

Examples

```
# Generate sample data
set.seed(123)
X <- matrix(rnorm(100 * 5), ncol = 5)
result <- create.single.iknn.graph(X, k = 3)
summary(result)
```

`summary.iknn_graphs` *Summarize an iknn_graphs Object*

Description

Provides a detailed summary of an iknn_graphs object created by the create.iknn.graphs() function. The summary includes statistics for each intersection kNN graph in the sequence, displaying information about the connectivity and structure of the graphs for different k values.

Usage

```
## S3 method for class 'iknn_graphs'
summary(object, use_isize_pruned = FALSE, ...)
```

Arguments

`object`	An object of class 'iknn_graphs', typically the output of create.iknn.graphs().
`use_isize_pruned`	Logical. If TRUE, computes and displays statistics for the intersection-size pruned graphs (isize_pruned_graphs). If FALSE (default), computes statistics for the geometrically pruned graphs (geom_pruned_graphs).
`...`	Additional arguments passed to or from other methods (not currently used).

Details

The summary function extracts and presents key statistics about the structure of each intersection kNN graph in the iknn_graphs object. All graphs share the same number of vertices (equal to the number of rows in the input data matrix), but they differ in the number of edges due to varying k values and the pruning methods applied during graph creation.

The function displays general information about the graph sequence, including the number of vertices, the range of k values, and the pruning parameters used. It then presents a tabular summary of statistics for each individual graph, showing how the graph structure changes as k increases.

For each intersection kNN graph, the following metrics are calculated:

- Number of edges: Total number of connections in the graph
- Mean degree: Average number of connections per vertex
- Min/Max degree: Range of vertex connectivity
- Density: Proportion of potential connections that are actually present
- Sparsity: 1 - density, measuring the proportion of missing connections

Value

Invisibly returns a data frame containing statistics for each graph. The data frame has the following columns:

`idx`	The index of the given k value
`k`	The k value for the intersection kNN graph
`n_ccomp`	Number of connected components of the graph
`edges`	Number of edges in the graph
`mean_degree`	Average number of connections per vertex
`min_degree`	Minimum vertex degree (least connected vertex)
`max_degree`	Maximum vertex degree (most connected vertex)
`sparsity`	Graph sparsity, calculated as 1 - density. It measures how many (proportion) potential connections are missing.

See Also

`create.iknn.graphs` for creating intersection kNN graphs.

Examples

```
# Create sample data
set.seed(123)
x <- matrix(rnorm(1000), ncol = 5)

# Generate intersection kNN graphs
iknn.res <- create.iknn.graphs(x, kmin = 3, kmax = 10)

# Summarize the geometrically pruned graphs
summary(iknn.res)

# Summarize the intersection-size pruned graphs
summary(iknn.res, use_isize_pruned = TRUE)
```

`summary.knn.outliers` *Summary Method for knn.outliers Objects*

Description

Provides a detailed summary of the outlier detection results from `remove.knn.outliers`.

Usage

```
## S3 method for class 'knn.outliers'
summary(object, ...)
```

Arguments

`object`	An object of class "knn.outliers" as returned by `remove.knn.outliers`.
`...`	Additional arguments (currently unused).

Value

A list of class "summary.knn.outliers" containing summary statistics.

Examples

```
# Using the example from remove.knn.outliers
set.seed(123)
S <- rbind(matrix(rnorm(2000), ncol = 2),
           cbind(rnorm(10, 10), rnorm(10, 10)))
result <- remove.knn.outliers(S, K = 10)
summary(result)
```

summary.local_extrema *Summarize Local Extrema Detection Results*

Description

Provides a comprehensive summary of local extrema detection results, including statistics on the number of extrema found, their function values, neighborhood sizes, and radii.

Usage

```
## S3 method for class 'local_extrema'
summary(object, ...)
```

Arguments

`object`	An object of class `"local_extrema"`, as returned by `detect.local.extrema`.
`...`	Additional arguments (currently ignored).

Value

An object of class `"summary.local_extrema"`, which is a list containing:

n_extrema Integer; total number of extrema found

extrema_type Character; type of extrema ("Maximum" or "Minimum")

fn_values_summary Named numeric vector; summary statistics of function values at extrema

neighborhood_sizes_summary Named numeric vector; summary statistics of neighborhood sizes

radius_summary Named numeric vector; summary statistics of neighborhood radii

extrema_details Data frame with detailed information about each extremum

Examples

```
# Create example data
adj.list <- list(c(2), c(1,3), c(2,4), c(3,5), c(4))
weight.list <- list(c(1), c(1,1), c(1,1), c(1,1), c(1))
y <- c(1, 3, 2, 5, 1)

# Detect and summarize maxima
maxima <- detect.local.extrema(adj.list, weight.list, y, 2, 2)
summary(maxima)
```

`summary.mabilo`	*Compute Summary Statistics for Mabilo Objects*

Description

Computes comprehensive summary statistics for mabilo fits including model parameters, fit statistics, error analysis, and diagnostic information for model-averaged predictions. Also includes Bayesian bootstrap statistics when available.

Usage

```
## S3 method for class 'mabilo'
summary(object, quantiles = c(0, 0.25, 0.5, 0.75, 1), ...)
```

Arguments

`object`	A 'mabilo' object
`quantiles`	Numeric vector of probabilities for quantile computations
`...`	Additional arguments (currently unused)

Value

A 'summary.mabilo' object containing:

- model_info: Basic information about the model fit
- fit_stats: Prediction accuracy metrics
- k_error_stats: Error statistics for different k values
- residual_stats: Residual analysis results
- true_error_stats: Statistics comparing to true values (if available)
- bootstrap_stats: Bayesian bootstrap statistics (if available)

`summary.mabilog`	*Compute Summary Statistics for Mabilog Objects*

Description

Computes comprehensive summary statistics for mabilog fits including model parameters, fit statistics, error analysis, and diagnostic information for logistic regression. Also includes Bayesian bootstrap statistics when available.

Usage

```
## S3 method for class 'mabilog'
summary(object, quantiles = c(0, 0.25, 0.5, 0.75, 1), ...)
```

Arguments

`object`	A 'mabilog' object
`quantiles`	Numeric vector of probabilities for quantile computations
`...`	Additional arguments (currently unused)

Value

A 'summary.mabilog' object containing:

- `model_info`: Basic information about the model fit
- `fit_stats`: Prediction accuracy metrics
- `k_error_stats`: Error statistics for different k values
- `residual_stats`: Residual analysis results
- `classification_stats`: Classification performance metrics
- `true_error_stats`: Statistics comparing to true values (if available)
- `bootstrap_stats`: Bayesian bootstrap statistics (if available)

`summary.mabilo_plus` *Compute Summary Statistics for Mabilo Plus Objects*

Description

Computes summary statistics for mabilo.plus fits including model parameters, fit statistics, error analysis, and diagnostic information for both simple mean (SM) and model averaging (MA) predictions.

Usage

```
## S3 method for class 'mabilo_plus'
summary(object, quantiles = c(0, 0.25, 0.5, 0.75, 1), ...)
```

Arguments

`object`	A 'mabilo_plus' object
`quantiles`	Numeric vector of probabilities for quantile computations
`...`	Additional arguments (currently unused)

Value

A 'summary.mabilo_plus' object containing:

- model_info: Basic information about the model fit
- fit_stats: Prediction accuracy metrics for both SM and MA
- k_error_stats: Error statistics for different k values
- residual_stats: Residual analysis results for both SM and MA
- true_error_stats: Statistics comparing to true values (if available)

summary.maelog *Summary Method for maelog Objects*

Description

Produces a summary of a fitted `maelog` object.

Usage

```
## S3 method for class 'maelog'
summary(object, ...)
```

Arguments

`object`	A fitted object of class `"maelog"`.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns a list containing summary statistics. The function is called primarily for its side effect of printing the summary.

See Also

`maelog`

summary.magelog *Summary Method for magelog Objects*

Description

Produces a summary of a fitted magelog model including goodness-of-fit statistics.

Usage

```
## S3 method for class 'magelog'
summary(object, ...)
```

Arguments

`object`	A fitted model object of class `"magelog"`
`...`	Additional arguments (currently ignored)

Value

An object of class `"summary.magelog"` containing summary statistics

`summary.mknn_graphs` *Summary Method for mknn_graphs Objects*

Description

Provides a comprehensive summary of mutual k-nearest neighbor graphs created by `create.mknn.graphs`, including structural statistics and connectivity information for each k value.

Usage

```
## S3 method for class 'mknn_graphs'
summary(object, ...)
```

Arguments

`object`	An object of class "mknn_graphs", typically output from `create.mknn.graphs`.
`...`	Additional arguments (currently unused).

Details

The summary provides insights into how graph structure changes with k:

- **Connected Components**: Indicates graph fragmentation. A value of 1 means the graph is fully connected.
- **Degree Statistics**: Shows the distribution of vertex connectivity. Large differences between min and max degree may indicate hub vertices.
- **Density/Sparsity**: Measures how many of the possible edges are present. MkNN graphs are typically very sparse.

For pruned graphs, the statistics reflect the structure after geometric pruning has been applied.

Value

Invisibly returns a data frame containing graph statistics with columns:

idx Index of the k value (1 to number of graphs)

k The k value used for the graph

n_components Number of connected components

n_edges Number of edges in the graph

mean_degree Average vertex degree

min_degree Minimum vertex degree

max_degree Maximum vertex degree

density Graph density (proportion of possible edges present)

sparsity Graph sparsity (1 - density)

Examples

```
# Create sample data
set.seed(123)
X <- matrix(rnorm(200 * 5), ncol = 5)

# Generate MkNN graphs
mknn_result <- create.mknn.graphs(X, kmin = 5, kmax = 15, compute.full = TRUE)

# Display summary
summary(mknn_result)

# Store summary statistics for plotting
stats <- summary(mknn_result)
plot(stats$k, stats$mean_degree, type = "b",
     xlab = "k", ylab = "Mean Degree",
     main = "Mean Vertex Degree vs k")
```

---

summary.mst_completion_graph
Summary Method for MST Completion Graph Objects

Description

Computes and returns a comprehensive summary of an MST completion graph, including graph statistics, edge weight distributions, and PCA information if applicable.

Usage

```
## S3 method for class 'mst_completion_graph'
summary(object, ...)
```

Arguments

`object`	An object of class `mst_completion_graph`.
`...`	Additional arguments (currently ignored).

Value

An object of class `summary.mst_completion_graph` containing:

`n_vertices` Number of vertices in the graph

`n_mst_edges` Number of edges in the minimal spanning tree

`n_cmst_edges` Number of edges in the completed graph

`q_thld` Quantile threshold used for completion

`edge_weight_summary` Five-number summary of MST edge weights

`completion_threshold` The actual distance threshold used

`pca_applied` Logical indicating if PCA was applied

`pca_info` PCA details (if applicable)

See Also

print.summary.mst_completion_graph for printing summaries, create.cmst.graph for creating graphs

Examples

```
## Not run:
X <- matrix(rnorm(100 * 5), nrow = 100, ncol = 5)
graph <- create.cmst.graph(X, q.thld = 0.85)
summary(graph)

## End(Not run)
```

---

summary.nerve_cx_spectral_filter

Summary Method for nerve_cx_spectral_filter Objects

Description

Provides a comprehensive summary of nerve complex spectral filtering results, including statistics about the predictions, parameter optimization, and computational details.

Usage

```
## S3 method for class 'nerve_cx_spectral_filter'
summary(object, ...)
```

Arguments

`object`	A `nerve_cx_spectral_filter` object returned by nerve.cx.spectral.filter.
`...`	Additional arguments (currently ignored).

Value

An object of class `summary.nerve_cx_spectral_filter` containing:

`method`	List of method parameters used
`optimal_parameter`	The selected optimal filter parameter
`gcv_score`	The GCV score at the optimal parameter
`compute_time_ms`	Computation time in milliseconds
`predictions`	Summary statistics of the smoothed values
`parameters`	Summary of the parameter search
`improvement`	GCV improvement from worst to best parameter

See Also

print.summary.nerve_cx_spectral_filter for printing, nerve.cx.spectral.filter for the main function

summary.path.graph *Summary Method for path.graph Objects*

Description

Provides a detailed summary of a path.graph object including statistics about paths and connectivity.

Usage

```
## S3 method for class 'path.graph'
summary(object, ...)
```

Arguments

object	A path.graph object
...	Additional arguments (currently ignored)

Value

A list containing summary statistics (invisibly)

Examples

```
## Not run:
pg <- create.path.graph(graph, edge.lengths, h = 3)
summary(pg)

## End(Not run)
```

summary.path.graph.series *Summary Method for path.graph.series Objects*

Description

Provides detailed statistics about a series of path graphs.

Usage

```
## S3 method for class 'path.graph.series'
summary(object, ...)
```

Arguments

object	A path.graph.series object
...	Additional arguments (currently ignored)

Value

A data.frame with summary statistics (invisibly)

summary.pgmalo *Summary Method for pgmalo Objects*

Description

Produces summary statistics for pgmalo model fits including model parameters, fit quality measures, and cross-validation results.

Usage

```
## S3 method for class 'pgmalo'
summary(object, ...)
```

Arguments

`object`	An object of class "pgmalo" from `pgmalo`.
`...`	Additional arguments (currently unused).

Details

The summary includes basic model information, cross-validation results, and bootstrap information if available. When true values are provided, additional error metrics are computed.

Value

An object of class "summary.pgmalo" containing:

call The matched call

n_vertices Number of vertices in the graph

optimal_h Optimal neighborhood size selected by CV

cv_info Cross-validation statistics including range and errors

bootstrap_info Bootstrap information if available

true_error_info True error statistics if y.true was provided

prediction_summary Summary statistics of predictions

See Also

`pgmalo` for model fitting, `print.summary.pgmalo` for formatted output

Examples

```
## Not run:
# Create example data
n <- 50
neighbors <- vector("list", n)
edge_lengths <- vector("list", n)
for(i in 1:n) {
  if(i == 1) {
    neighbors[[i]] <- 2L
    edge_lengths[[i]] <- 1.0
  } else if(i == n) {
    neighbors[[i]] <- (n-1L)
    edge_lengths[[i]] <- 1.0
  } else {
    neighbors[[i]] <- c(i-1L, i+1L)
    edge_lengths[[i]] <- c(1.0, 1.0)
  }
}
y <- rnorm(n)

# Fit model and get summary
fit <- pgmalo(neighbors, edge_lengths, y)
summary(fit)

## End(Not run)
```

`summary.pwlm` *Summary Method for PWLM Objects*

Description

Summary Method for PWLM Objects

Usage

```
## S3 method for class 'pwlm'
summary(object, ...)
```

Arguments

`object`	An object of class "pwlm"
`...`	Additional arguments passed to summary

`summary.spectral.lowess.result`
Summary method for spectral.lowess.result objects

Description

Summary method for spectral.lowess.result objects

Usage

```
## S3 method for class 'spectral.lowess.result'
summary(object, ...)
```

Arguments

`object`	An object of class "spectral.lowess.result"
`...`	Additional arguments (ignored)

Value

A summary object containing key information

`summary.uggmalo` *Summary Method for UGGMALO Results*

Description

Summary Method for UGGMALO Results

Usage

```
## S3 method for class 'uggmalo'
summary(object, ...)
```

Arguments

`object`	An object of class "uggmalo"
`...`	Additional arguments (currently ignored)

Value

A list of class "summary.uggmalo" containing summary statistics

---

`summary.uggmalog` *Summary method for uggmalog objects*

Description

Summary method for uggmalog objects

Usage

```
## S3 method for class 'uggmalog'
summary(object, ...)
```

Arguments

`object`	An object of class `"uggmalog"`
`...`	Additional arguments (currently ignored)

Value

A list with summary information, invisibly

`summary.ugkmm` *Summary Method for ugkmm Objects*

Description

Provides comprehensive summary statistics for adaptive neighborhood size graph k-means regression results.

Usage

```
## S3 method for class 'ugkmm'
summary(object, digits = 4, ...)
```

Arguments

`object`	An object of class "ugkmm".
`digits`	Number of significant digits for output. Default is 4.
`...`	Additional arguments (currently unused).

Value

An object of class "summary.ugkmm" containing model statistics, cross-validation results, and fit diagnostics.

`summary.upgmalo` *Summary Method for upgmalo Objects*

Description

Produces comprehensive summary statistics for UPGMALO model fits including fit quality measures, cross-validation results, residual diagnostics, and optional comparison with true values.

Usage

```
## S3 method for class 'upgmalo'
summary(object, quantiles = c(0, 0.25, 0.5, 0.75, 1), ...)
```

Arguments

`object`	An object of class "upgmalo" from `upgmalo`.
`quantiles`	Numeric vector of probabilities for residual quantiles (default: c(0, 0.25, 0.5, 0.75, 1)).
`...`	Additional arguments (currently unused).

Details

When true values are available, the summary includes additional diagnostics:

- Equivalent normal standard deviation
- Shapiro-Wilk test for normality of standardized errors
- Relative efficiency compared to optimal linear estimator

Value

An object of class "summary.upgmalo" containing:

call The matched call

model_info Basic model information

fit_stats Fit quality statistics (MSE, RMSE, MAE)

cv_stats Cross-validation summary

residual_stats Residual distribution summary

true_error_stats True error analysis (if y.true was provided)

bootstrap_info Bootstrap summary (if n.bb > 0)

See Also

`upgmalo` for model fitting, `print.summary.upgmalo` for formatted output

Examples

```
# See examples in upgmalo()
```

`summary.vertex_geodesic_stats` *Summary method for vertex_geodesic_stats objects*

Description

Summary method for vertex_geodesic_stats objects

Usage

```
## S3 method for class 'vertex_geodesic_stats'
summary(object, ...)
```

Arguments

`object`	A vertex_geodesic_stats object
`...`	Additional arguments passed to summary

Value

A summary of the vertex geodesic statistics `synthetic.1D.spline` *Generates the values of a synthetic 1d spline function over a uniform grid in the pre-specified range.*

Description

This function generates a synthetic 1D spline with a specified number of local maxima. The synthetic function is evaluated on a uniform grid over a specified x range. The local minima are strategically placed to control the shape of the function.

Usage

```
synthetic.1D.spline(
  n.lmax,
  x.min = 0,
  x.max = 10,
  y.min = 1,
  y.max = 5,
  n.grid = 400,
  p.offset = 0.1,
  alpha = 0.5,
  method = "natural"
)
```

Arguments

`n.lmax`	Number of local maxima
`x.min`	Minimum x value. Default 0.
`x.max`	Maximum x value. Default 10.
`y.min`	Minimum y value. Default 1.
`y.max`	Maximum y value. Default 5.
`n.grid`	Number of grid points. Default 400.
`p.offset`	Fraction of interval between local maxima for local minima. Default 0.1.
`alpha`	Shape parameter for beta distribution. Default 0.5.
`method`	Spline method. Default 'natural'.

Value

List with components:

- `x`: Locations of grid points
- `y`: Function values at grid points
- `x.lmax`: Locations of local maxima
- `y.lmax`: Values of local maxima

Examples

```
## Not run:
synth <- synthetic.1D.spline(n.lmax = 5)
plot(synth$x, synth$y, type = "l")
points(synth$x.lmax, synth$y.lmax, col = "red")

## End(Not run)
```

`synthetic.mixture.of.gaussians`
Generate a Synthetic Smooth Function in One Dimension

Description

This function constructs a synthetic smooth function using a series of Gaussian distributions in one-dimensional space. It is centered at given points with standard deviations determined by the distances to neighboring points. The function can handle both positive and negative Gaussian distributions.

Usage

```
synthetic.mixture.of.gaussians(x, x.knot, y.knot = NULL, sd.knot = NULL)
```

Arguments

`x`	Values at which the function is to be evaluated.
`x.knot`	A numeric vector of the centers of the Gaussian functions.
`y.knot`	An optional numeric vector of response values at the locations specified in `x.knot`. If `y.knot` is not provided, random values will be generated.
`sd.knot`	Standard deviations of the Gaussians.

Details

For each point in `x`, the function calculates the standard deviation of the Gaussian distribution as half of the minimum distance to its neighbors. It then forms a synthetic function by combining these Gaussian distributions. The function can generate smooth functions that take also negative values by multiplying Gaussian components by a vector of ± 1 values.

Value

A function representing the synthetic smooth function. This function takes a numeric value as input and returns a numeric value, representing the function's output at that point.

Examples

```
## Not run:
# Example usage
x <- seq(0, 1, length.out = 10)
synthetic.fn <- generate.synthetic.function(x)
point <- 0.5 # A point in 1D space
value <- synthetic.fn(point) # Evaluate the function at the given point

## End(Not run)
```

`synthetic.xD.spline` *Generate a Spline Function*

Description

This function creates a spline function based on a given set of points. It interpolates these points using cubic splines, providing a smooth curve through the data.

Usage

```
synthetic.xD.spline(X, y = NULL)
```

Arguments

`X`	A numeric matrix containing the coordinates of the points.
`y`	A numeric vector containing the y-coordinates of the points. If not provided, random values will be generated.

Details

The function first checks that the lengths of `x` and `y` are equal and then sorts these vectors based on the `x` values. It then creates a cubic spline function using the `splinefun` function in R, with the method set to "fmm" for flexible monotone splines. If `y` is not provided, random values are generated to create the spline.

Value

A spline function that interpolates the given points. The function takes a numeric value (x-coordinate) as input and returns the interpolated y-coordinate.

Examples

```
## Not run:
# Example usage
x <- seq(0, 10, length.out = 10)
y <- sin(x)  # Example y values
spline.fn <- synthetic.spline(x, y)
point <- 5  # An x-coordinate
value <- spline.fn(point)  # Evaluate the spline function at this point

## End(Not run)
```

test.bidirectional.dijkstra

Test Bidirectional Dijkstra and Composite Path Validation

Description

Validates the correctness of bidirectional Dijkstra algorithm and composite path geodesic verification using igraph.

Usage

```
test.bidirectional.dijkstra(
  grid.vertex,
  debug.dir,
  full.report = FALSE,
  graph = NULL
)
```

Arguments

`grid.vertex`	Numeric index of the grid vertex to validate
`debug.dir`	Character string specifying the directory containing debugging data
`full.report`	Logical; if TRUE, returns detailed validation information
`graph`	List; An output of graph_from_data_frame()

Value

A list containing test results and validation summaries

test.graph.MS.cx.on.synth.cloud.data

Tests graph Morse-Smale complex construct (its persistence and reconstruction) on synthetic datasets

Description

Tests graph Morse-Smale complex construct (its persistence and reconstruction) on synthetic datasets

Usage

```
test.graph.MS.cx.on.synth.cloud.data(
  n.datasets,
  n.pts,
  n.side = NULL,
  dim = 2,
  type = "grid",
  Ks,
  n.lmax = 10,
  y.min = 1,
  y.max = 15,
  out.dir,
  verbose = TRUE
)
```

Arguments

`n.datasets`	The number of datasets to generate.
`n.pts`	The number of points the synthetic cloud dataset suppose to have.
`n.side`	If the data is to be a sample from some grid, then n.side is the number of points on the axis of each dimension, thus the total number of grid points is n.size^dim.
`dim`	The dimension of R^dim in which data is constructed.
`type`	The method used for the construction.
`Ks`	A vector of positive integer values specifying the range of k values.
`n.lmax`	The number of gaussians to use.
`y.min`	The mimimum value of the range of y values.
`y.max`	The maximum value of the range of y values.
`out.dir`	The output directory where the generated data is stored.
`verbose`	Set it to TRUE, to print progress messages.

Value

A list containing the length of the longest stretch of isomorphic MS graphs for each generated dataset

---

`thresholded.sigmoid` *Sigmoidal Function with Lower and Upper Thresholds*

Description

Creates a sigmoidal function that smoothly transitions from approximately 0 to approximately 1 between two specified threshold values. This is useful for creating soft thresholds in data processing or for probability-based filtering.

Usage

```
thresholded.sigmoid(x, lower_threshold, upper_threshold, steepness = 10)
```

Arguments

`x`	Numeric vector or scalar. The input values to transform.
`lower_threshold`	Numeric scalar. The lower threshold value where the function begins its transition from 0 to 1. For values well below this threshold, the function returns values close to 0.

`upper_threshold`	Numeric scalar. The upper threshold value where the function completes its transition from 0 to 1. For values well above this threshold, the function returns values close to 1.
`steepness`	Numeric scalar. Controls the steepness of the transition between thresholds. Higher values create a sharper transition. Default is 10.

Value

A numeric vector of the same length as `x` with values ranging from approximately 0 to approximately 1.

Examples

```
# Define thresholds
q90 <- 0.1  # Lower threshold (e.g., 90th percentile)
q95 <- 0.15 # Upper threshold (e.g., 95th percentile)

# Apply to a sequence of values
x_seq <- seq(0, 0.3, length.out = 100)
y_values <- thresholded.sigmoid(x_seq, q90, q95)

# Plot the result
plot(x_seq, y_values, type = "l",
     xlab = "Input Value", ylab = "Transformed Value",
     main = "Thresholded Sigmoid Function")
abline(v = c(q90, q95), lty = 2, col = c("blue", "red"))
```

`total.associations` *Calculate Total Associations from Geodesic Analysis*

Description

Calculates total associations (TAs) and total absolute associations (TAAs) between the distance along a geodesic and some function over the geodesic.

Usage

```
total.associations(obj, ED = NULL, verbose = FALSE)
```

Arguments

`obj`	A list result from E.geodesic.X() containing at least: **X** A matrix of data **data.dir** Directory containing saved results
`ED`	Numeric vector of mean disease/outcome values over the geodesic. If NULL (default), calculates associations with distance along geodesic.
`verbose`	Logical indicating whether to print progress messages. Default is FALSE.

Details

This function processes results from geodesic analysis, calculating associations between variables along the geodesic path. When ED is NULL, it calculates associations with the distance along the geodesic. When ED is provided, it calculates associations with the specified outcome variable.

Value

A matrix with columns:

TA Total Association

TAA Total Absolute Association

pval p-value for the association test

Row names correspond to column names of the input matrix X.

Examples

```
## Not run:
# Assuming obj is a result from E.geodesic.X()
# Calculate associations with distance
ta_results <- total.associations(obj)

# Calculate associations with an outcome
ED <- rnorm(400)  # Example outcome vector
ta_results_ED <- total.associations(obj, ED = ED)

## End(Not run)
```

`tr.exponential.kernel` *Truncated exponential kernel*

Description

The function equals to (1-t)*exp(-t) for t within (-1, 1) and 0 otherwise; t = abs(x/bw)

Usage

```
tr.exponential.kernel(x, bw = 1)
```

Arguments

`x`	A numeric vector.
`bw`	A bandwidth numeric parameter.

`triangle.plot` *triangle.plot*

Description

Custom plot function to draw vertices as triangles in an igraph plot.

Usage

```
triangle.plot(coords, v = NULL, params)
```

Arguments

`coords`	A matrix of vertex coordinates. Each row should contain the x and y coordinates of a vertex.
`v`	An optional vector of vertex indices to be plotted. If NULL, all vertices will be plotted.
`params`	A list of plotting parameters, typically obtained from the igraph `params` function.

Details

This function is used to plot vertices as equilateral triangles in an igraph graph. The triangles are sized to have approximately the same area as circles of the same vertex size. It is intended to be used with the igraph `add_shape` function to add the "triangle" shape to graph vertices.

The function adjusts the size of the triangles to match the area of circles with the same vertex size. This ensures visual consistency in the plot. The function also handles vertex colors, frame colors, and frame widths.

Examples

```
## Not run:
  library(igraph)

  # Define the triangle plot function
  triangle.plot <- function(coords, v = NULL, params) {
      vertex.color <- params("vertex", "color")
      if (length(vertex.color) != 1 && !is.null(v)) {
          vertex.color <- vertex.color\[v\]
      }
      vertex.frame.color <- params("vertex", "frame.color")
      if (length(vertex.frame.color) != 1 && !is.null(v)) {
          vertex.frame.color <- vertex.frame.color\[v\]
      }
      vertex.frame.width <- params("vertex", "frame.width")
      if (length(vertex.frame.width) != 1 && !is.null(v)) {
          vertex.frame.width <- vertex.frame.width\[v\]
      }
      vertex.size <- 1/200 * params("vertex", "size")
      if (length(vertex.size) != 1 && !is.null(v)) {
          vertex.size <- vertex.size\[v\]
      }
```

```
        vertex.size <- rep(vertex.size, length.out = nrow(coords))

        # Adjust the size for the triangle
        side.length <- sqrt(4 * pi / sqrt(3)) * vertex.size / 2

        vertex.frame.color\[vertex.frame.width <= 0\] <- NA
        vertex.frame.width\[vertex.frame.width <= 0\] <- 1

        for (i in 1:nrow(coords)) {
            x <- coords\[i, 1\]
            y <- coords\[i, 2\]
            size <- side.length\[i\]
            polygon(x + size * c(cos(pi/2), cos(7*pi/6), cos(11*pi/6)),
                    y + size * c(sin(pi/2), sin(7*pi/6), sin(11*pi/6)),
                    col = vertex.color\[i\], border = vertex.frame.color\[i\],
                    lwd = vertex.frame.width\[i\])
        }
    }

    # Add the triangle shape to igraph
    add_shape("triangle", clip = shapes(shape = "circle")$clip, plot = triangle.plot)

    # Example graph
    g <- make_ring(10)
    V(g)$shape <- rep(c("circle", "triangle"), length.out = vcount(g))
    plot(g, vertex.size = 15, vertex.color = "skyblue")

## End(Not run)
```

`triangular.kernel` *Triangular kernel*

Description

The function equals to 1 - abs(x/bw) within (-bw, bw) and 0 otherwise

Usage

```
triangular.kernel(x, bw = 1)
```

Arguments

`x`	A numeric vector.
`bw`	A bandwidth numeric parameter.

`two.factor.analysis` *Two-Factor Analysis for Contingency Tables with Proportion Tests*

Description

Performs comprehensive analysis of contingency tables for two categorical variables, including proportion tests and optional LaTeX output. Supports both two-sample proportion comparisons between groups and one-sample tests against expected proportions.

Usage

```
two.factor.analysis(
  y,
  x,
  out.dir,
  latex.file = NULL,
  label = NA,
  do.prop.test = TRUE,
  expected.prop = NULL
)
```

Arguments

`y`	factor, the grouping variable (e.g., treatment groups or community state types)
`x`	factor, the binary outcome variable (e.g., success/failure, case/control)
`out.dir`	character, path to the output directory where results will be saved
`latex.file`	character or NULL, optional path to LaTeX output file. If NULL, defaults to `file.path(out.dir, paste0(label, ".tex"))`
`label`	character or NA, label for output files and LaTeX tables. If NA, defaults to "fp" for file naming
`do.prop.test`	logical, whether to perform proportion tests (default: TRUE)
`expected.prop`	numeric or NULL, if provided (must be between 0 and 1), performs one-sample proportion test against this expected value. If NULL, performs two-sample tests between groups

Details

The function performs the following steps:

1. Validates inputs and creates output directory if needed
2. Constructs contingency table and calculates row proportions
3. Performs proportion tests:
   - If `expected.prop` is provided: one-sample tests comparing each group's proportion to the expected value
   - If `expected.prop` is NULL: two-sample tests comparing each group against all other groups combined
4. Formats results combining proportions, counts, and p-values
5. Saves outputs as RDA, CSV, and optionally LaTeX files

Value

A list with the following components:

f contingency table of frequencies

p matrix of row percentages

pval numeric vector of p-values from proportion tests

all.f formatted results matrix combining percentages, counts, and p-values

Examples

```
## Not run:
# Create example data
set.seed(123)
treatment <- factor(rep(c("A", "B", "C"), each = 100))
outcome <- factor(rbinom(300, 1, rep(c(0.3, 0.5, 0.4), each = 100)),
                  labels = c("Failure", "Success"))

# Two-sample proportion tests
result <- two.factor.analysis(
  y = treatment,
  x = outcome,
  out.dir = tempdir(),
  label = "treatment_analysis"
)

# One-sample tests against expected proportion of 0.4
result2 <- two.factor.analysis(
  y = treatment,
  x = outcome,
  out.dir = tempdir(),
  label = "treatment_vs_expected",
  expected.prop = 0.4
)

## End(Not run)
```

`ugg.get.path.data` *Create a Refined Graph with Uniformly Spaced Grid Vertices and creates local paths throught the specified ref vertex*

Description

Creates a refined version of an input graph by adding grid vertices (points) along its edges. The grid vertices are placed to maintain approximately uniform spacing throughout the graph structure. This function is particularly useful for tasks that require a denser sampling of points along the graph edges, such as graph-based interpolation or spatial analysis.

Analyzes paths in a graph centered around a reference vertex, computing distances, kernel weights, and associated values along these paths. The function identifies both single paths and composite paths that meet the minimum size requirement, with the reference vertex serving as a central point in the path structure.

Usage

```
ugg.get.path.data(
  adj.list,
  weight.list,
  grid.size,
  y,
  ref.vertex,
  bandwidth,
  dist.normalization.factor = 1.01,
  min.path.size = 5L,
  diff.threshold = 5L,
  kernel.type = 7L,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list where each element `i` is an integer vector containing the indices of vertices adjacent to vertex `i`. Vertex indices must be 1-based. The graph structure must be undirected, meaning if vertex `j` appears in `adj.list[[i]]`, then vertex `i` must appear in `adj.list[[j]]`.
`weight.list`	A list matching the structure of adj.list, where each element contains the corresponding edge weights (typically distances or lengths). `weight.list[[i]][j]` should contain the weight of the edge between vertex `i` and vertex `adj.list[[i]][j]`. Weights must be positive numbers.
`grid.size`	A positive integer specifying the desired number of grid vertices to add. The actual number of added vertices may differ slightly from this target due to the distribution of edge lengths in the graph.
`y`	A numeric vector of values associated with each vertex in the graph.
`ref.vertex`	An integer specifying the reference vertex around which paths are constructed (1-based indexing).
`bandwidth`	A positive numeric value specifying the maximum allowable path distance from the reference vertex.
`dist.normalization.factor`	A numeric value between 0 and 1 for normalizing distances in kernel calculations (default: 1.01).
`min.path.size`	An integer specifying the minimum number of vertices required in a valid path (default: 5).
`diff.threshold`	An integer specifying the number of vertices after the ref vertex that two paths have to have different (set intersection is empty) to produce a composite path from these two paths
`kernel.type`	An integer specifying the kernel function type: - 1: Epanechnikov - 2: Triangular - 4: Laplace - 5: Normal - 6: Biweight - 7: Tricube (default)
`verbose`	Logical indicating whether to print progress information. Default is FALSE.

Value

A list where each element represents a path and contains:

`vertices`	Integer vector of path vertices (1-based indices)

`ref_vertex`	Integer indicating the reference vertex (1-based index)
`rel_center_offset`	Numeric value indicating relative position of reference vertex (0 = center, 0.5 = endpoint)
`total_weight`	Numeric value representing total path length
`x_path`	Numeric vector of cumulative distances along path from start
`w_path`	Numeric vector of kernel weights for each vertex
`y_path`	Numeric vector of y-values for path vertices

`uggmalo` *Uniform Grid Graph Model-Averaged Local Linear Regression (UGGMALO)*

Description

Note: This is an experimental function and may produce errors under certain conditions. Use with caution and please report issues.

Implements the Uniform Grid Graph Model-Averaged Local linear regression (UGGMALO) algorithm for estimating conditional expectations of functions defined over vertices of a graph using local path linear models with model averaging.

The algorithm performs the following main steps:

1. Computes graph diameter and determines bandwidth range
2. Creates a uniform grid representation of the input graph
3. For each candidate bandwidth:
    - Processes paths through grid vertices
    - Fits local logistic models to path data
    - Computes weighted predictions and errors
4. Determines optimal bandwidth based on cross-validation errors

The algorithm uses weighted logistic regression on paths through the graph to create local models, which are then combined using weighted averaging. Model evaluation is performed using leave-one-out cross-validation with Brier score errors.

Usage

```
uggmalo(
  adj.list,
  weight.list,
  y,
  best.models.coverage.factor = 0.9,
  min.bw.factor = 0.05,
  max.bw.factor = 0.5,
  n.bws = 50L,
  grid.size = 100L,
  start.vertex = 1L,
  snap.tolerance = 0.1,
  dist.normalization.factor = 1.01,
  min.path.size = 5L,
  diff.threshold = 5L,
  kernel.type = 7L,
  fit.quadratic = FALSE,
  tolerance = 1e-08,
  n.bb = 0L,
  p = 0.95,
  n.perms = 0L,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element i contains the indices of vertices adjacent to vertex i (1-based indexing).
`weight.list`	A list of numeric vectors representing the weights of the edges. Must have the same structure as `adj.list`.
`y`	A numeric vector of observations at each vertex. Length must match the number of vertices.
`best.models.coverage.factor`	Numeric scalar between 0.5 and 1.0 controlling model coverage. Default: 0.9
`min.bw.factor`	Numeric scalar. Minimum bandwidth factor relative to graph diameter. Must be positive. Default: 0.05
`max.bw.factor`	Numeric scalar. Maximum bandwidth factor relative to graph diameter. Must be greater than `min.bw.factor`. Default: 0.5
`n.bws`	Integer. Number of bandwidths to test between `min.bw.factor` and `max.bw.factor`. Must be positive. Default: 50
`grid.size`	Integer. Size of the evaluation grid for predictions. Must be positive. Default: 100
`start.vertex`	Integer. Index of the starting vertex (1-based). Must be between 1 and the number of vertices. Default: 1
`snap.tolerance`	Numeric scalar. Tolerance for snapping distances to grid points. Must be positive. Default: 0.1
`dist.normalization.factor`	Numeric scalar. Factor for normalizing distances. Must be greater than 1. Default: 1.01
`min.path.size`	Integer. Minimum path size for distance calculations. Must be at least 5. Default: 5
`diff.threshold`	Integer. Threshold for difference in path lengths. Must be at least 5. Default: 5
`kernel.type`	Integer between 0 and 7. Type of kernel to use: • 0: Uniform kernel • 1: Triangular kernel • 2: Epanechnikov kernel • 3: Quartic kernel • 4: Triweight kernel • 5: Tricube kernel • 6: Gaussian kernel • 7: Cosine kernel

	Default: 7
`fit.quadratic`	Logical. Whether to fit quadratic terms in the local models. Default: FALSE
`tolerance`	Numeric scalar. Convergence tolerance for optimization. Must be positive. Default: 1e-8
`n.bb`	Integer. Number of bag bootstrap iterations. Must be non-negative. Default: 0
`p`	Numeric scalar between 0 and 1. Probability parameter for bootstrap. Default: 0.95
`n.perms`	Integer. Number of permutations for uncertainty estimation. Must be non-negative. Default: 0
`verbose`	Logical. Whether to print progress messages. Default: FALSE

Value

A list of class "uggmalo" containing:

candidate_bws Numeric vector of candidate bandwidths tested

bw_predictions Matrix of predictions for each bandwidth (vertices × bandwidths)

mean_errors Numeric vector of mean cross-validation errors for each bandwidth

opt_bw_idx Integer. Index of the optimal bandwidth (1-based)

predictions Numeric vector of predictions using the optimal bandwidth

graph_diameter Numeric. Diameter of the graph

Examples

```
## Not run:
# Create a simple chain graph
set.seed(123)  # For reproducibility
n.vertices <- 20
adj.list <- lapply(1:n.vertices, function(i) {
  # Simple chain graph
  if (i == 1) c(2L)
  else if (i == n.vertices) c(i - 1L)
  else c(i - 1L, i + 1L)
})

dist.list <- lapply(1:n.vertices, function(i) {
  if (i == 1) c(1.0)
  else if (i == n.vertices) c(1.0)
  else c(1.0, 1.0)
})

# Create parabolic signal with noise
x <- seq(-1, 1, length.out = n.vertices)
true.signal <- 3 * x^2 - 2 * x + 1  # Parabola
noise <- rnorm(n.vertices, mean = 0, sd = 0.3)
y <- true.signal + noise

# Run estimation with default parameters
result <- uggmalo(adj.list, dist.list, y)

# Compare true vs estimated values
plot(1:n.vertices, y, pch = 19, col = "gray50",
     xlab = "Vertex", ylab = "Value",
     main = "UGGMALO: True vs Estimated")
lines(1:n.vertices, true.signal, col = "blue", lwd = 2)
lines(1:n.vertices, result$predictions, col = "red", lwd = 2)
legend("topright", c("Observations", "True signal", "UGGMALO estimate"),
       col = c("gray50", "blue", "red"),
       pch = c(19, NA, NA), lty = c(NA, 1, 1), lwd = 2)

# Run with custom parameters for comparison
result2 <- uggmalo(adj.list, dist.list, y,
                   min.bw.factor = 0.1,
                   max.bw.factor = 0.8,
                   n.bws = 30,
                   verbose = TRUE)

## End(Not run)
```

---

uggmalo.bayesian.bootstrap.with.uncertainty

Bayesian Bootstrap with Uncertainty for UGGMALO Predictions

Description

Note: This is an experimental function and may produce errors under certain conditions. Use with caution and please report issues.

Combines Bayesian bootstrap with permutation-based uncertainty estimation for UGGMALO predictions on a graph. This function accounts for both the uncertainty in p-value estimation from permutation tests and the spatial structure of the data.

The function performs these steps:

1. Generates Dirichlet weights for Bayesian bootstrap
2. Resamples from permutation distributions to account for p-value uncertainty
3. Applies spatial smoothing using UGGMALO to account for graph structure
4. Computes credible intervals from the bootstrap distribution

Usage

```
uggmalo.bayesian.bootstrap.with.uncertainty(
  perm.results,
  true.predictions,
  graph,
  n.bootstrap = 1000L,
  n.cores = 14L
)
```

Arguments

`perm.results`	Numeric matrix where each row corresponds to a vertex and each column contains predictions from a permutation run. Dimensions should be (number of vertices $\times$ number of permutations).

`true.predictions`	Numeric vector of original UGGMALO predictions for each vertex. Length must match the number of rows in `perm.results`.
`graph`	List containing graph structure with required components: **pruned_adj_list** Adjacency list representation of the graph (list of integer vectors) **pruned_dist_list** List of numeric vectors containing edge weights/distances
`n.bootstrap`	Integer. Number of bootstrap iterations. Must be positive. Default: 1000
`n.cores`	Integer. Number of CPU cores to use for parallel processing. Default: 14

Value

A list of class "uggmalo_bootstrap" containing:

`ci.lower`	Numeric vector of lower bounds of 95% credible intervals
`ci.upper`	Numeric vector of upper bounds of 95% credible intervals
`bootstrap.distribution`	Matrix of bootstrap estimates where each column represents one bootstrap iteration (vertices × iterations)

Known Issues

This function currently fails with the error "Reference vertex not found in path vertices" (line 1944 in centered_paths.cpp). This is a known bug that will be addressed in future releases.

See Also

`uggmalo`

Examples

```
## Not run:
# WARNING: This is an experimental function with known issues
# The following example demonstrates the intended usage but currently fails
# with error: "Reference vertex not found in path vertices"

if (requireNamespace("foreach", quietly = TRUE) &&
    requireNamespace("doParallel", quietly = TRUE)) {

  # Create example data
  n.vertices <- 20
  n.perms <- 100

  # Simulated permutation results
  perm.results <- matrix(rnorm(n.vertices * n.perms),
                         nrow = n.vertices, ncol = n.perms)

  # True predictions
  true.predictions <- rnorm(n.vertices)

  # Simple graph structure
  graph <- list(
    pruned_adj_list = lapply(1:n.vertices, function(i) {
      # Simple chain graph
```

```
        if (i == 1) c(2L)
        else if (i == n.vertices) c(i - 1L)
        else c(i - 1L, i + 1L)
      }),
      pruned_dist_list = lapply(1:n.vertices, function(i) {
        if (i == 1) c(1.0)
        else if (i == n.vertices) c(1.0)
        else c(1.0, 1.0)
      })
    )

    # Run bootstrap (with fewer iterations for example)
    # NOTE: This currently fails with a known bug
    results <- uggmalo.bayesian.bootstrap.with.uncertainty(
      perm.results = perm.results,
      true.predictions = true.predictions,
      graph = graph,
      n.bootstrap = 100,
      n.cores = 2
    )

    # Plot credible intervals (if successful)
    plot(1:n.vertices, true.predictions, pch = 19,
         ylim = range(c(results$ci.lower, results$ci.upper)),
         xlab = "Vertex", ylab = "Prediction",
         main = "UGGMALO Predictions with 95% Credible Intervals")
    arrows(1:n.vertices, results$ci.lower,
           1:n.vertices, results$ci.upper,
           length = 0.05, angle = 90, code = 3)
}

## End(Not run)
```

`uggmalog` *Uniform Grid Graph Model-Averaged LOGistic regression (UGGMA-LOG)*

Description

Implements a sophisticated algorithm for analyzing weighted graphs using local path logistic models with model averaging. The algorithm performs the following main steps:

1. Computes graph diameter and determines bandwidth range
2. Creates a uniform grid representation of the input graph
3. For each candidate bandwidth:
   - Processes paths through grid vertices
   - Fits local logistic models to path data
   - Computes weighted predictions and errors
4. Determines optimal bandwidth based on cross-validation errors

The algorithm uses weighted logistic regression on paths through the graph to create local models, which are then combined using weighted averaging. Model evaluation is performed using leave-one-out cross-validation with Brier score errors.

Usage

```
uggmalog(
  adj.list,
  weight.list,
  y,
  best.models.coverage.factor = 0.9,
  min.bw.factor = 0.05,
  max.bw.factor = 0.5,
  n.bws = 50L,
  grid.size = 100L,
  start.vertex = 1L,
  snap.tolerance = 0.1,
  dist.normalization.factor = 1.01,
  min.path.size = 5L,
  diff.threshold = 5L,
  kernel.type = 7L,
  fit.quadratic = FALSE,
  max.iterations = 100L,
  ridge.lambda = 0,
  tolerance = 1e-08,
  verbose = FALSE
)
```

Arguments

`adj.list`	A list of integer vectors representing the adjacency list of the graph. Each element `i` contains the indices of vertices adjacent to vertex `i`. Uses 1-based indexing.
`weight.list`	A list of numeric vectors representing the weights of the edges. Must have the same structure as `adj.list`.
`y`	A numeric vector of observations at each vertex. Length must match the number of vertices in the graph.
`best.models.coverage.factor`	Numeric scalar between 0.5 and 1.0. Controls the proportion of best models used in model averaging. Default: 0.9
`min.bw.factor`	Numeric scalar. Minimum bandwidth factor relative to graph diameter. Must be positive. Default: 0.05
`max.bw.factor`	Numeric scalar. Maximum bandwidth factor relative to graph diameter. Must be greater than `min.bw.factor`. Default: 0.5
`n.bws`	Positive integer. Number of bandwidths to test between `min.bw.factor` and `max.bw.factor`. Default: 50
`grid.size`	Positive integer. Size of the evaluation grid for predictions. Default: 100
`start.vertex`	Positive integer. Index of the starting vertex (1-based). Must be between 1 and the number of vertices. Default: 1
`snap.tolerance`	Positive numeric scalar. Tolerance for snapping distances to grid points. Default: 0.1
`dist.normalization.factor`	Numeric scalar greater than 1. Factor for normalizing distances. Default: 1.01
`min.path.size`	Positive integer. Minimum path size for distance calculations. Default: 5

`diff.threshold`	Non-negative integer. Threshold for difference in path lengths. Default: 5
`kernel.type`	Integer between 0 and 7. Type of kernel to use: • 0: Uniform kernel • 1: Triangular kernel • 2: Epanechnikov kernel • 3: Quartic (biweight) kernel • 4: Triweight kernel • 5: Tricube kernel • 6: Gaussian kernel • 7: Cosine kernel (default)
`fit.quadratic`	Logical scalar. Whether to fit quadratic terms in the local models. Default: FALSE
`max.iterations`	Positive integer. Maximum number of iterations for optimization. Default: 100
`ridge.lambda`	Non-negative numeric scalar. Ridge regression parameter. Default: 0.0
`tolerance`	Positive numeric scalar. Convergence tolerance for optimization. Default: 1e-8
`verbose`	Logical scalar. Whether to print progress messages. Default: FALSE

Details

The algorithm is particularly useful for prediction on graphs where local structure is important. The model averaging approach helps to reduce overfitting and provides more stable predictions.

The bandwidth selection is performed automatically using cross-validation, choosing the bandwidth that minimizes the mean Brier score error.

Value

A list with class `"uggmalog"` containing:

`candidate_bws`	Numeric vector of candidate bandwidths tested
`bw_predictions`	Numeric matrix of predictions for each bandwidth (rows: vertices, columns: bandwidths)
`mean_errors`	Numeric vector of mean cross-validation errors for each bandwidth
`opt_bw_idx`	Integer scalar. Index of the optimal bandwidth (1-based)
`predictions`	Numeric vector of predictions using the optimal bandwidth
`graph_diameter`	Numeric scalar. Computed diameter of the graph

Note

This function requires compilation of C++ code. The adjacency list uses R's standard 1-based indexing, which is internally converted to 0-based indexing for the C++ implementation.

Examples

```
## Not run:
# Create a simple chain graph
n <- 10
adj <- vector("list", n)
weights <- vector("list", n)
```

```
# Build chain: 1 -- 2 -- 3 -- ... -- n
for (i in 1:n) {
  adj[[i]] <- integer(0)
  weights[[i]] <- numeric(0)

  if (i > 1) {
    adj[[i]] <- c(adj[[i]], i - 1)
    weights[[i]] <- c(weights[[i]], 1.0)
  }

  if (i < n) {
    adj[[i]] <- c(adj[[i]], i + 1)
    weights[[i]] <- c(weights[[i]], 1.0)
  }
}

# Generate some response data
set.seed(123)
y <- sin(seq(0, pi, length.out = n)) + rnorm(n, sd = 0.1)

# Run UGGMALOG
result <- uggmalog(adj, weights, y, verbose = TRUE)

# Print optimal bandwidth index
cat("Optimal bandwidth index:", result$opt_bw_idx, "\n")

# More complex example: Grid graph
# Create a 5x5 grid graph
grid_size <- 5
n <- grid_size^2
adj <- vector("list", n)
weights <- vector("list", n)

# Function to convert 2D coordinates to 1D index
coord_to_idx <- function(i, j) (i - 1) * grid_size + j

# Build grid adjacency
for (i in 1:grid_size) {
  for (j in 1:grid_size) {
    idx <- coord_to_idx(i, j)
    adj[[idx]] <- integer(0)
    weights[[idx]] <- numeric(0)

    # Add horizontal edges
    if (j > 1) {
      adj[[idx]] <- c(adj[[idx]], coord_to_idx(i, j - 1))
      weights[[idx]] <- c(weights[[idx]], 1.0)
    }
    if (j < grid_size) {
      adj[[idx]] <- c(adj[[idx]], coord_to_idx(i, j + 1))
      weights[[idx]] <- c(weights[[idx]], 1.0)
    }

    # Add vertical edges
    if (i > 1) {
      adj[[idx]] <- c(adj[[idx]], coord_to_idx(i - 1, j))
      weights[[idx]] <- c(weights[[idx]], 1.0)
    }
    if (i < grid_size) {
      adj[[idx]] <- c(adj[[idx]], coord_to_idx(i + 1, j))
      weights[[idx]] <- c(weights[[idx]], 1.0)
    }
  }
}

# Generate response based on distance from center
center <- (grid_size + 1) / 2
y <- numeric(n)
for (i in 1:grid_size) {
  for (j in 1:grid_size) {
    dist_from_center <- sqrt((i - center)^2 + (j - center)^2)
    y[coord_to_idx(i, j)] <- exp(-dist_from_center / 2) + rnorm(1, sd = 0.05)
  }
}

# Run UGGMALOG with custom parameters
result <- uggmalog(
  adj.list = adj,
  weight.list = weights,
  y = y,
  n.bws = 30,
  kernel.type = 5,  # Tricube kernel
  fit.quadratic = TRUE,
  verbose = TRUE
)

## End(Not run)
```

ulogit *Fit Univariate Logistic Regression Model*

Description

Fits a univariate logistic regression model to binary response data using Newton-Raphson optimization with optional ridge regularization. The function implements numerical safeguards to ensure stable convergence and provides leave-one-out cross-validation errors for model assessment.

Usage

```
ulogit(
  x,
  y,
  w = NULL,
  max.iterations = 100L,
  ridge.lambda = 0.002,
  max.beta = 100,
  tolerance = 1e-08,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of predictor values. Must contain only finite values (no NA, NaN, or Inf).
`y`	Binary response vector containing only 0 or 1 values. Must be the same length as `x`.
`w`	Optional numeric vector of non-negative observation weights. If `NULL` (default), all observations are given equal weight. Must be the same length as `x` if provided.
`max.iterations`	Maximum number of iterations for Newton-Raphson optimization. Must be a positive integer. Default is 100.
`ridge.lambda`	Ridge regularization parameter to improve numerical stability. Must be non-negative. Default is 0.002. Higher values provide more regularization but may increase bias.
`max.beta`	Maximum allowed absolute value for coefficient estimates. Used to prevent numerical overflow. Must be positive. Default is 100.0.
`tolerance`	Convergence tolerance for optimization. The algorithm stops when the relative change in log-likelihood is less than this value. Must be positive. Default is 1e-8.
`verbose`	Logical flag to enable detailed output during optimization. If `TRUE`, prints iteration progress. Default is `FALSE`.

Details

The function fits a logistic regression model of the form:

$$logit(p_i) = \beta_0 + \beta_1 x_i$$

where p_i is the probability of success for observation i.

The Newton-Raphson algorithm is used for maximum likelihood estimation with optional ridge regularization. The ridge penalty adds $\lambda\beta^2$ to the negative log-likelihood, which helps stabilize the estimation when the data are nearly separable.

Leave-one-out cross-validation (LOOCV) errors are computed efficiently using the hat matrix diagonal elements, providing a measure of predictive accuracy without requiring repeated model fitting.

Value

A list of class `"ulogit"` containing:

predictions Numeric vector of fitted probabilities for each observation

errors Numeric vector of leave-one-out cross-validation errors

weights Numeric vector of weights used in model fitting

converged Logical indicating whether the algorithm converged

iterations Integer giving the number of iterations used

call The matched call

References

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning (2nd ed.). Springer.

See Also

`eigen.ulogit` for an alternative implementation using Eigen, `glm` for multivariate logistic regression

Examples

```
# Basic usage with simulated data
set.seed(123)
x <- seq(0, 1, length.out = 100)
true_prob <- 1/(1 + exp(-(2*x - 1)))
y <- rbinom(100, 1, prob = true_prob)
fit <- ulogit(x, y)

# Plot results
plot(x, y, pch = 16, col = ifelse(y == 1, "blue", "red"),
     main = "Univariate Logistic Regression")
lines(x, fit$predictions, lwd = 2)
legend("topleft", c("y = 1", "y = 0", "Fitted"),
       col = c("blue", "red", "black"),
       pch = c(16, 16, NA), lty = c(NA, NA, 1))

# Example with weights
w <- runif(100, 0.5, 1.5)
fit_weighted <- ulogit(x, y, w = w)

# Example with increased regularization
fit_regularized <- ulogit(x, y, ridge.lambda = 0.1)

# Compare LOOCV errors
cat("Standard model LOOCV error:", mean(fit$errors), "\n")
cat("Weighted model LOOCV error:", mean(fit_weighted$errors), "\n")
cat("Regularized model LOOCV error:", mean(fit_regularized$errors), "\n")
```

`univariate.gkmm` *Adaptive Neighborhood Size Graph K-Means for Univariate Data*

Description

Performs nonparametric regression using graph-based k-means with adaptive neighborhood size selection. The method constructs chain graphs from sorted predictor values and uses cross-validation to select the optimal neighborhood size.

Usage

```
univariate.gkmm(
  x,
  y,
  y.true = NULL,
  use.median = FALSE,
  h.min = 2,
  h.max = min(30, length(x)),
  n.CVs = 1000,
  n.CV.folds = 10,
  p = 0.95,
  n.bb = 500,
  ikernel = 1,
  n.cores = 1,
  dist.normalization.factor = 1.01,
  epsilon = 1e-15,
  seed = NULL
)
```

Arguments

`x`	Numeric vector of predictor values.
`y`	Numeric vector of response values.
`y.true`	Optional numeric vector of true response values for performance evaluation.
`use.median`	Logical; if TRUE uses median, if FALSE uses mean for predictions. Default is FALSE.
`h.min`	Minimum neighborhood size to consider (>= 2). Default is 2.
`h.max`	Maximum neighborhood size to consider. Default is min(30, length(x)).
`n.CVs`	Number of cross-validation iterations. Default is 1000.
`n.CV.folds`	Number of CV folds (>= 2). Default is 10.
`p`	Probability level for credible intervals (0 < p < 1). Default is 0.95.
`n.bb`	Number of Bayesian bootstrap iterations (0 to skip). Default is 500.
`ikernel`	Integer specifying kernel function (1-6). Default is 1.
`n.cores`	Number of CPU cores for parallel processing. Default is 1.
`dist.normalization.factor`	Distance normalization factor (> 1). Default is 1.01.
`epsilon`	Small positive value for numerical stability. Default is 1e-15.
`seed`	Random seed for reproducibility. Default is NULL.

Details

The algorithm:

1. Sorts data by predictor values
2. Constructs chain graphs with varying neighborhood sizes
3. Uses cross-validation to select optimal neighborhood size
4. Computes predictions using kernel-weighted means
5. Optionally performs Bayesian bootstrap for uncertainty quantification

Value

An object of class "ugkmm" containing:

`h_values`	Vector of tested neighborhood sizes
`h_cv_errors`	Cross-validation errors for each h
`mean_cv_error`	Mean CV error across all h values

opt_h	Optimal neighborhood size
predictions	Fitted values
bb_predictions	Bootstrap central tendency estimates
opt_ci_lower	Lower credible interval bounds
opt_ci_upper	Upper credible interval bounds
x_sorted	Sorted predictor values
y_sorted	Response values (sorted by x)
y_true_sorted	True values (sorted by x) if provided

See Also

plot.ugkmm for plotting methods summary.ugkmm for summary statistics

Examples

```
## Not run:
# Generate example data
set.seed(123)
x <- seq(0, 10, length.out = 100)
y <- sin(x) + rnorm(100, 0, 0.1)

# Fit model
result <- univariate.gkmm(x, y, h.min = 2, h.max = 20, n.bb = 100)

# Plot results
plot(result)

# Summary
summary(result)

## End(Not run)
```

update.graph.spectral.lowess *Update Graph Spectral LOWESS Fit*

Description

Updates a graph spectral LOWESS fit with new parameters

Usage

```
## S3 method for class 'graph.spectral.lowess'
update(object, formula, ..., evaluate = TRUE)
```

Arguments

object	A 'graph.spectral.lowess' object
formula	Changes to the model (currently unused)
...	Additional arguments to override in the new call
evaluate	Should the updated call be evaluated? (default: TRUE)

Value

Updated graph.spectral.lowess object (if evaluate = TRUE) or updated call

---

`update.graph.spectral.ma.lowess`
Update Graph Spectral MA LOWESS Fit

Description

Updates a graph spectral model-averaged LOWESS fit with new parameters, particularly useful for changing the blending coefficient.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
update(object, formula, ..., evaluate = TRUE)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`formula`	Changes to the model (currently unused)
`...`	Additional arguments to override in the new call
`evaluate`	Should the updated call be evaluated? (default: TRUE)

Value

Updated graph.spectral.ma.lowess object (if evaluate = TRUE) or updated call

---

`update.mabilog` *Update Mabilog Model*

Description

Update a mabilog model with new parameters

Usage

```
## S3 method for class 'mabilog'
update(object, ...)
```

Arguments

`object`	A 'mabilog' object
`...`	Arguments to update in the model call

Value

A new mabilog object

`update.mabilo_plus` *Update Mabilo Plus Model*

Description

Update a mabilo_plus model with new parameters

Usage

```
## S3 method for class 'mabilo_plus'
update(object, ...)
```

Arguments

`object`	A 'mabilo_plus' object
`...`	Arguments to update in the model call

Value

A new mabilo_plus object

`upgmalo` *Univariate Path Graph Model Averaging Local Linear Model*

Description

Implements adaptive neighborhood selection for univariate data using path graph model averaging local linear (UPGMALO) estimation. The function automatically constructs a chain graph from sorted predictor values and applies the PGMALO methodology.

Usage

```
upgmalo(
  x,
  y,
  y.true = NULL,
  use.median = TRUE,
  h.min = 4L,
  h.max = min(30L, length(x) - 2L),
  p = 0.95,
  n.bb = 50L,
  bb.max.distance.deviation = 1L,
  n.CVs = 100L,
  n.CV.folds = 10L,
  seed = 0L,
  kernel.type = 7L,
  dist.normalization.factor = 1.01,
  epsilon = 1e-15,
  verbose = FALSE
)
```

Arguments

`x`	Numeric vector of predictor values. Will be automatically sorted if not already in ascending order.
`y`	Numeric vector of response values corresponding to x.
`y.true`	Optional numeric vector of true response values for computing prediction errors.
`use.median`	Logical; if TRUE uses median instead of mean for bootstrap interval estimation (default: TRUE).
`h.min`	Integer; minimum neighborhood size to consider. Must be even and at least 4 (default: 4).
`h.max`	Integer; maximum neighborhood size to consider. Must be even and at most n-2 (default: min(30, n-2)).
`p`	Numeric; confidence level for bootstrap intervals (default: 0.95).
`n.bb`	Integer; number of bootstrap iterations (default: 50).
`bb.max.distance.deviation`	Integer; maximum distance deviation for bootstrap samples (default: 1).
`n.CVs`	Integer; number of cross-validation iterations (default: 100).
`n.CV.folds`	Integer; number of cross-validation folds (default: 10).
`seed`	Integer; random seed for reproducibility (default: 0).
`kernel.type`	Integer between 1 and 10 specifying kernel function (default: 7 for uniform kernel).
`dist.normalization.factor`	Numeric; distance normalization factor (default: 1.01).
`epsilon`	Numeric; numerical stability parameter (default: 1e-15).
`verbose`	Logical; if TRUE prints progress messages (default: FALSE).

Details

This function is a convenience wrapper that constructs a chain graph from univariate data and applies the PGMALO methodology. The chain graph connects each point to its immediate neighbors in the sorted order of x values.

Value

An S3 object of class "upgmalo" containing all components from `pgmalo` plus:

x_sorted Sorted predictor values

y_sorted Response values corresponding to sorted x

y_true_sorted True values corresponding to sorted x (if provided)

h_min Minimum h value used

h_max Maximum h value used

max_h_index Index range for h values

See Also

`pgmalo` for general graph version, `plot.upgmalo` for visualization, `summary.upgmalo` for model summaries

Examples

```
## Not run:
# Generate nonlinear data
n <- 50
x <- seq(0, 2*pi, length.out = n)
y_true <- sin(x)
y <- y_true + rnorm(n, 0, 0.2)

# Fit model
  fit <- upgmalo(x, y, y.true = y_true, h.max = 20, n.CVs = 20)

  # Plot results
  plot(fit)

## End(Not run)
```

`v1.torus.knot` *Torus knot*

Description

Torus knot

Usage

```
v1.torus.knot(n, k, m)
```

Arguments

`n`	The desired number of points to generate.
`k`	Torus knot first parameter.
`m`	Torus knot second parameter.

`validate.maximal.packing` *Validate a Maximal Packing*

Description

Validates whether a given vertex packing is maximal and correctly satisfies the distance constraints. A packing is valid if all vertices are separated by at least the specified radius, and is maximal if no additional vertices can be added without violating this constraint.

Usage

```
validate.maximal.packing(
  adj.list,
  weight.list,
  packing.vertices,
  max.packing.radius
)
```

Arguments

`adj.list`	A list where each element is a vector of adjacent vertex indices (1-based) for the corresponding vertex.
`weight.list`	A list where each element is a vector of edge weights corresponding to the adjacencies in `adj.list`.
`packing.vertices`	An integer vector of vertex indices (1-based) that form the packing to be validated.
`max.packing.radius`	A numeric value representing the minimum distance that should separate any two vertices in the packing.

Details

This function performs two key validations:

1. **Packing Property**: Verifies that all vertices in the packing are separated by at least `max.packing.radius`.
2. **Maximality**: Verifies that no additional vertex can be added to the packing without violating the packing property.

The function uses the `igraph` package to compute shortest path distances between vertices using Dijkstra's algorithm.

Value

A list with class "packing_validation" containing validation results:

`valid`	Logical indicating whether the packing satisfies the distance constraint
`min.packing.distance`	The minimum distance found between any two packing vertices
`max.coverage.distance`	The maximum distance from any non-packing vertex to its nearest packing vertex
`violations`	A data frame containing details of any violations found, or NULL if none. Contains columns: type, vertex1, vertex2, distance
`is.maximal`	Logical indicating whether the packing is maximal (no vertices can be added)
`potential.additions`	Integer vector of vertex indices that could potentially be added to the packing if it's not maximal, or NULL if maximal

Examples

```
## Not run:
# Create a simple path graph with 6 vertices
adj.list <- list(
  c(2),          # vertex 1 connects to 2
  c(1, 3),       # vertex 2 connects to 1 and 3
  c(2, 4),       # vertex 3 connects to 2 and 4
  c(3, 5),       # vertex 4 connects to 3 and 5
  c(4, 6),       # vertex 5 connects to 4 and 6
  c(5)           # vertex 6 connects to 5
)
```

```
weight.list <- list(
  c(1), c(1, 1), c(1, 1), c(1, 1), c(1, 1), c(1)
)

# Test a packing with vertices 1 and 4
packing <- c(1, 4)
radius <- 3

# Validate the packing
result <- validate.maximal.packing(adj.list, weight.list, packing, radius)
print(result)

## End(Not run)
```

---

vars.approx.monotonically.assoc.with.geodesic

Given TA and TAA from E.geodesic.X(), this routine identifies variables monotonically associated with the distance along the geodesic

---

Description

Given TA and TAA from E.geodesic.X(), this routine identifies variables monotonically associated with the distance along the geodesic

Usage

```
vars.approx.monotonically.assoc.with.geodesic(X, eps = 0.4)
```

Arguments

X	A matrix of a data frame with TA and delta1 columns.
eps	The positive constant specifying the size of a neighborhood around the diagonal and anti-diagonal of the (TA, TAA) graph.

Details

This function analyzes the relationship between variables and geodesic distance by examining the TA (turn angle) and TAA (turn angle acceleration) values. Variables are considered monotonically associated if their (TA, TAA) values fall within an epsilon neighborhood of either the diagonal (monotonically increasing) or anti-diagonal (monotonically decreasing) in the unit square.

Value

A logical vector of length equal to the number of rows in X, where TRUE indicates that the corresponding variable is approximately monotonically associated with the geodesic distance (either increasing or decreasing).

Examples

```
# Assuming X has columns TA and TAA from E.geodesic.X output
# monotonic_vars <- vars.approx.monotonically.assoc.with.geodesic(X, eps = 0.4)
# X[monotonic_vars, ] # subset to monotonic variables
```

`vcov.mabilog` *Extract Variance-Covariance Matrix*

Description

Computes an approximate variance-covariance matrix for the predictions Note: This is a simplified approximation for binomial variance

Usage

```
## S3 method for class 'mabilog'
vcov(object, ...)
```

Arguments

`object`	A 'mabilog' object
`...`	Additional arguments (currently unused)

Value

Diagonal variance-covariance matrix

`vcov.mabilo_plus` *Extract Variance-Covariance Matrix*

Description

Computes an approximate variance-covariance matrix for the predictions

Usage

```
## S3 method for class 'mabilo_plus'
vcov(object, type = c("ma", "sm"), ...)
```

Arguments

`object`	A 'mabilo_plus' object
`type`	Character string specifying which predictions to use: "ma" (default) or "sm"
`...`	Additional arguments (currently unused)

Value

Variance-covariance matrix

vector.norm *The norm of a vector.*

Description

The norm of a vector.

Usage

```
vector.norm(x)
```

Arguments

`x`	A numeric vector.

verify.maximal.packing *Verify Maximal Packing Created by create.maximal.packing*

Description

A convenience function to verify the correctness of a maximal packing created using the `create.maximal.packing` function. This function validates both the packing property (minimum separation distance) and maximality (no vertices can be added).

Usage

```
verify.maximal.packing(packing.result, verbose = TRUE)
```

Arguments

`packing.result`	The result returned by `create.maximal.packing`, which must be an object of class "maximal_packing".
`verbose`	Logical indicating whether to print detailed validation results. Default is TRUE.

Details

This function takes the output of `create.maximal.packing` and verifies two key properties:

1. The packing vertices are all separated by at least `max_packing_radius`
2. The packing is maximal (no more vertices can be added without violating the distance constraint)

```
When \code{verbose = TRUE}, the function prints:
\itemize{
  \item Graph diameter
  \item Packing radius used
  \item Number of vertices in the packing
  \item Minimum distance between packing vertices
  \item Maximum coverage distance
  \item Validity and maximality status
  \item Any violations or potential additions (if applicable)
}
```

Value

A logical value: TRUE if the packing is both valid (satisfies the distance constraint) and maximal (no vertices can be added), FALSE otherwise.

See Also

`create.maximal.packing`, `validate.maximal.packing`

Examples

```
## Not run:
# Create a simple cycle graph
n <- 10
adj.list <- lapply(1:n, function(i) {
  c(ifelse(i == 1, n, i - 1), ifelse(i == n, 1, i + 1))
})
weight.list <- lapply(1:n, function(i) c(1, 1))

# Create maximal packing
result <- create.maximal.packing(adj.list, weight.list, grid.size = 3)

# Verify the packing with detailed output
is_valid <- verify.maximal.packing(result, verbose = TRUE)

# Verify quietly
is_valid <- verify.maximal.packing(result, verbose = FALSE)

## End(Not run)
```

`vert.error.bar` *Add Vertical Error Bar to Plot*

Description

Adds a vertical error bar to an existing 2D plot

Usage

```
vert.error.bar(x, ymin, ymax, dx = 0.025, lwd = 1, col = "red")
```

Arguments

`x`	X-coordinate for the error bar.
`ymin`	Lower limit of error bar.
`ymax`	Upper limit of error bar.
`dx`	Horizontal offset for end caps.

`lwd`	Line width.
`col`	Color of error bar.

Value

Invisibly returns NULL.

Examples

```
plot(1:10, rnorm(10), ylim = c(-3, 3))
vert.error.bar(5, -1, 1, col = "red")
```

`vertices` *Extract Vertices from a Local Extrema Object*

Description

Generic function to extract vertices from various objects.

Usage

```
vertices(object, ...)
```

Arguments

`object`	An object from which to extract vertices.
`...`	Additional arguments passed to methods.

Value

The extracted vertices (format depends on the method).

`vertices.local_extrema` *Extract Vertices of a Specific Local Extremum*

Description

Extracts the vertices belonging to a specific local extremum identified by its label. Returns all vertices in the neighborhood of the specified extremum.

Usage

```
## S3 method for class 'local_extrema'
vertices(object, label, include.center = TRUE, ...)
```

Arguments

`object`	An object of class `"local_extrema"`.
`label`	Character string specifying the label of the extremum (e.g., "M1", "m2").
`include.center`	Logical; if `TRUE` (default), include the center vertex.
`...`	Additional arguments (currently ignored).

Value

A numeric vector of vertex indices in the extremum's neighborhood.

Note

This method requires that the C++ backend provides neighborhood vertex information in the `neighborhood_vertices` component of the result.

Examples

```
# Create example data
adj.list <- list(c(2), c(1,3), c(2,4), c(3,5), c(4))
weight.list <- list(c(1), c(1,1), c(1,1), c(1,1), c(1))
y <- c(1, 3, 2, 5, 1)

# Detect maxima
maxima <- detect.local.extrema(adj.list, weight.list, y, 2, 2)

# Extract vertices for the first maximum (if it exists)
if (length(maxima$vertices) > 0) {
  v <- vertices(maxima, maxima$labels[1])
}
```

`visualize.grid.function` *Visualize a Function on a Grid Graph*

Description

Creates visualizations of a function defined on a grid graph, including 2D heatmap, 3D surface plot (if rgl is available), and contour plot.

Creates multiple visualizations of a function defined on a grid graph including heatmap with contours, 3D perspective plot, and optionally an interactive 3D plot.

Usage

```
visualize.grid.function(
  grid.size,
  z,
  centers = NULL,
  title = "Function on Grid Graph"
)
```

```
visualize.grid.function(
  grid.size,
  z,
  centers = NULL,
  title = "Function on Grid Graph"
)
```

Arguments

grid.size	Integer; The size of the square grid (grid.size x grid.size)
z	Numeric vector; Function values at each vertex of the grid graph, ordered row-wise (length = grid.size^2)
centers	Optional integer vector; Indices of special vertices to highlight (e.g., local maxima or centers)
title	Character string; Title for the plots (default: "Function on Grid Graph")

Details

The function creates a side-by-side visualization with:

- Left panel: Heatmap with contour lines and optional center points
- Right panel: 3D perspective plot of the surface

If the rgl package is available, an additional interactive 3D visualization is created in a separate window.

The grid vertices are numbered from 1 to grid.size^2, going row by row. The function values in z should be ordered to match this numbering.

Value

Invisibly returns the function values

Invisibly returns the input z values

Examples

```
## Not run:
# Create a grid graph
grid.size <- 20
grid <- create.grid.graph(grid.size, grid.size)

# Generate a mixture of Gaussians
centers <- c(1, grid.size * grid.size / 2 + grid.size / 2)
y <- generate.graph.gaussian.mixture(
  grid$adj.list,
  grid$weight.list,
  centers
)

# Visualize the function
visualize.grid.function(grid.size, y, centers)

## End(Not run)

## Not run:
```

```
# Create a simple example with a Gaussian-like function on a 20x20 grid
grid.size <- 20
n_vertices <- grid.size^2

# Generate coordinates
x <- rep(1:grid.size, grid.size) / grid.size
y <- rep(1:grid.size, each=grid.size) / grid.size

# Create a function with two peaks
z <- exp(-10*((x-0.3)^2 + (y-0.3)^2)) + 0.5*exp(-8*((x-0.7)^2 + (y-0.6)^2))

# Find local maxima (simple approach)
centers <- which(z > 0.9)

# Visualize
visualize.grid.function(grid.size, z, centers)

## End(Not run)
```

visualize.smoothing.steps

Visualize Smoothing Steps from Graph Flow Complex

Description

Creates 3D visualizations showing the step-by-step smoothing process applied to spurious extrema in a graph flow complex. Requires detailed recording to have been enabled during the computation.

Usage

```
visualize.smoothing.steps(
  gflow_result,
  plot_res,
  step_by_step = TRUE,
  animation_delay = 0.5
)
```

Arguments

`gflow_result`	A `gflow_cx` object created with `detailed.recording = TRUE`
`plot_res`	A graph plotting result containing layout information
`step_by_step`	Logical; if TRUE (default), pauses between steps for interactive viewing. If FALSE, saves snapshots to files.
`animation_delay`	Numeric; delay in seconds between steps when `step_by_step = TRUE`. Default is 0.5.

Details

This function requires the `rgl` package for 3D visualization. When `step_by_step = FALSE`, it saves PNG snapshots of each smoothing step.

The visualization shows:

- The graph structure in 3D with z-coordinates from function values
- Local minima as blue spheres
- Local maxima as red spheres
- The progression of smoothing for each spurious extremum

Value

Invisibly returns NULL. Creates 3D visualizations as side effect.

Examples

```
## Not run:
# Requires interactive graphics
if (interactive() && requireNamespace("rgl", quietly = TRUE)) {
  # Create example with spurious extrema
  adj.list <- list(c(2,3), c(1,3,4), c(1,2,4), c(2,3))
  weight.list <- list(c(1,1), c(1,1,1), c(1,1,1), c(1,1))
  y <- c(0, 1, 0.9, 0.1)  # Spurious max at vertex 3

  # Compute with detailed recording
  result <- create.gflow.cx(
    adj.list, weight.list, y,
    hop.idx.thld = 2,
    detailed.recording = TRUE,
    verbose = FALSE
  )

  # Create layout (assuming plot.graph function exists)
  # plot_res <- plot.graph(list(adj.list=adj.list, weight.list=weight.list))

  # Visualize smoothing steps
  # visualize.smoothing.steps(result, plot_res)
}

## End(Not run)
```

`wasserstein.distance` *Compute Wasserstein Distance Between Two Datasets*

Description

This function calculates the Wasserstein distance (also known as Earth Mover's Distance) between two datasets. The Wasserstein distance can be intuitively understood as the minimum "cost" of transforming one distribution into another, where cost is measured as the amount of probability mass that needs to be moved, multiplied by the distance it needs to be moved.

Imagine each distribution as a pile of earth, and the Wasserstein distance as the minimum amount of work required to transform one pile into the other. This makes it particularly useful for comparing distributions with different supports or when KL-divergence might be undefined or infinite.

The algorithm uses the transport package to compute the distance efficiently. It's worth noting that while this isn't a direct proxy for relative entropy, it provides a robust measure of dissimilarity between distributions.

Usage

```
wasserstein.distance(X, Y)
```

Arguments

`X`	A matrix or data frame representing the first dataset.
`Y`	A matrix or data frame representing the second dataset.

Value

A numeric value representing the Wasserstein distance between X and Y.

References

Villani, C. (2008). Optimal transport: old and new (Vol. 338). Springer Science & Business Media.

Examples

```
X <- matrix(rnorm(1000), ncol = 2)
Y <- matrix(rnorm(1000, mean = 1), ncol = 2)
result <- wasserstein.distance(X, Y)
print(result)
```

`wasserstein.distance.1D` *Calculate Wasserstein Distance Between 1D Samples*

Description

This function computes the Wasserstein distance (also known as Earth Mover's Distance) between two samples from one-dimensional distributions.

Usage

```
wasserstein.distance.1D(x, y)
```

Arguments

`x`	A numeric vector representing a sample from the first distribution.
`y`	A numeric vector representing a sample from the second distribution.

Details

The Wasserstein distance is calculated by sorting both input samples and computing the average absolute difference between the sorted values. This implementation uses a C function for efficient computation.

Both input vectors must have the same length, contain only numeric values, and all elements must be finite.

Value

A single numeric value representing the Wasserstein distance between the two samples.

See Also

https://en.wikipedia.org/wiki/Wasserstein_metric for more information on the Wasserstein distance.

Examples

```
x <- rnorm(1000)
y <- rnorm(1000, mean = 1)
dist <- wasserstein.distance.1D(x, y)
print(dist)
```

`wasserstein.divergence` *Compute Wasserstein Divergence Between Two Point Sets*

Description

This function calculates a modified Wasserstein divergence between two point sets X and Y. It uses k-nearest neighbors in X to define local neighborhoods, and computes the Wasserstein distance between corresponding neighborhoods in X and Y.

Usage

```
wasserstein.divergence(X, Y, k)
```

Arguments

X	A numeric matrix where each row represents a point in n-dimensional space.
Y	A numeric matrix with the same dimensions as X, representing the second point set.
k	An integer specifying the number of nearest neighbors to consider.

Details

The function first checks the validity of inputs, then uses k-nearest neighbors to define local neighborhoods in X. For each neighborhood, it computes the 2-Wasserstein distance between the corresponding points in X and Y, using Euclidean ground distance. The final divergence is the sum of these local Wasserstein distances.

Value

A numeric value representing the sum of Wasserstein distances between local neighborhoods.

Note

This function requires the 'FNN' package for k-nearest neighbor calculations and the 'transport' package for Wasserstein distance computations.

Examples

```
## Not run:
X <- matrix(rnorm(1000 * 3), nrow = 1000, ncol = 3)
Y <- matrix(rnorm(1000 * 3), nrow = 1000, ncol = 3)
div <- wasserstein.divergence(X, Y, k = 10)

## End(Not run)
```

`wasserstein.ipNNuv` *Local Wasserstein distance to the uniform and delta at 1 of the inner products of the NN unit vectors.*

Description

For each point x of a state space, S, the unit vectors starting at x and ending at x's K-NN's are defined. The routine estimates the entropy of the inner products <n_i, n_K> of all these unit vectors with the last one.

Usage

```
wasserstein.ipNNuv(
  S,
  K = 25,
  use.transport = TRUE,
  n.breaks = 50,
  ref.i = 2813,
  use.geodesic.knn = TRUE,
  nn.i = NULL,
  nn.d = NULL,
  verbose = TRUE
)
```

Arguments

`S`	A state space.
`K`	The number of nearest neighbors to use for the identification of the end points
`use.transport`	Set to TRUE, to use wasserstein1d() from transport package.
`n.breaks`	The number of bin for the estimate of distribution of cosine of the angle values.
`ref.i`	The index of the reference point whose cosine angles will be used as the reference for the Wd1 estimates.

`use.geodesic.knn`	Set to TRUE, if geodesic.knn() is to be used instead of get.knn().
`nn.i`	A matrix of indices of the nearest neighbors.
`nn.d`	A matrix of distances to the nearest neighbors.
`verbose`	Set to TRUE for messages indicating which part of the routine is currently run.

Value

A list with cosine of the angle values for each state of the state space.

`wasserstein1d.test` *Performs Permutation Test for the Wasserstein Distance Between Two 1D Samples*

Description

Performs a permutation test to assess the significance of the Wasserstein distance between two 1D samples. The test determines if the observed Wasserstein distance is significantly different from the distances obtained by permuting the samples, under the null hypothesis that the samples come from the same distribution. The function supports parallel computation for faster processing.

Usage

```
wasserstein1d.test(x, y, n.perms = 10000, n.cores = 7)
```

Arguments

`x`	A numeric vector representing the first 1D sample.
`y`	A numeric vector representing the second 1D sample.
`n.perms`	An integer specifying the number of permutations to use for the permutation test. Default value is 10,000.
`n.cores`	An integer specifying the number of cores to use for parallel computation. If `n.cores` is greater than 1, the null Wasserstein distances are computed in parallel. If `n.cores` is 1 (default), the computation is done serially.

Details

The function computes the actual Wasserstein distance between the two provided samples, then performs `n.perms` permutations of the concatenated samples and computes the Wasserstein distance for each permutation. The p-value is computed as the proportion of permuted distances that are at least as large as the actual distance.

Value

A list containing:

- `wasserstein1d`: The Wasserstein distance between `x` and `y`.
- `null.wasserstein1d`: A numeric vector of Wasserstein distances between permuted `x` and `y`, obtained from `n.perms` permutations.
- `p.value`: The p-value of the permutation test, representing the proportion of permuted Wasserstein distances that are greater than or equal to the observed distance.

See Also

`wasserstein1d` for computing the Wasserstein distance between two 1D samples.

Examples

```
## Two samples from the same distribution
x <- rnorm(100); y <- rnorm(100)
res <- wasserstein1d.test(x, y, n.perms = 1000, n.cores = 1)
str(res)

## Two samples from different distributions
x <- rnorm(100)
y <- rnorm(100, mean = 2)
## Perform the permutation test with 1000 permutations, serially
res <- wasserstein1d.test(x, y, n.perms = 1000, n.cores = 1)
str(res)
## Perform the permutation test with 1000 permutations, using 2 cores
res <- wasserstein1d.test(x, y, n.perms = 1000, n.cores = 2)
str(res)
```

`weighted.p.value` *Weighted P-value Calculation*

Description

Computes the weighted p-value for a sample from a distribution quantifying uncertainty of estimation of a value for which the p-value is to be computed, given a normal null distribution with specified mean and standard deviation. The weighted p-value takes into account the uncertainty in the value for which the classical p-value would normally be computed.

Usage

```
weighted.p.value(u, mu, sigma, alternative = c("two.sided", "less", "greater"))
```

Arguments

`u`	A numeric vector representing the sample drawn from a distribution quantifying uncertainty of estimation of a value for which the p-value is to be computed.
`mu`	The mean of the normal null distribution.
`sigma`	The standard deviation of the normal null distribution.
`alternative`	Character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less".

Details

The weighted p-value approach is useful when there is uncertainty in the test statistic itself. Instead of computing a single p-value based on a point estimate, this method:

1. Computes p-values for each value in the uncertainty distribution
2. Averages these p-values to obtain a weighted p-value

This approach is particularly valuable in:

- Bayesian contexts where posterior distributions represent uncertainty
- Measurement error scenarios
- Bootstrap or simulation-based inference

The weighted p-value can be interpreted as the expected p-value over the uncertainty distribution of the parameter.

Value

A single numeric value representing the weighted p-value.

References

For theoretical foundations of weighted p-values in the context of uncertainty quantification, see relevant literature on Bayesian p-values and posterior predictive checks.

Examples

```
# Example 1: Uncertainty represented by a normal distribution
u_sample <- rnorm(1000, mean = 1.5, sd = 0.5)

# Test against null hypothesis N(0, 1)
p_value <- weighted.p.value(u_sample, mu = 0, sigma = 1)
cat("Weighted p-value:", p_value, "\n")

# Compare with classical p-value using the mean
classical_p <- pnorm(mean(u_sample), mean = 0, sd = 1, lower.tail = FALSE)
cat("Classical p-value:", classical_p, "\n")

# Example 2: Two-sided test
u_sample2 <- rnorm(1000, mean = -0.5, sd = 0.3)
p_two_sided <- weighted.p.value(u_sample2, mu = 0, sigma = 1,
                                 alternative = "two.sided")
cat("Two-sided weighted p-value:", p_two_sided, "\n")

# Example 3: Using with bootstrap samples
# Simulate some data and bootstrap the mean
set.seed(123)
original_data <- rnorm(30, mean = 1.2, sd = 2)
boot_means <- replicate(1000, mean(sample(original_data, replace = TRUE)))

# Weighted p-value using bootstrap distribution
p_boot <- weighted.p.value(boot_means, mu = 0, sigma = 2/sqrt(30))
cat("Bootstrap-based weighted p-value:", p_boot, "\n")
```

---

weighted.p.value.summary   *Weighted P-value Summary*

Description

Provides a comprehensive summary of weighted p-value analysis including comparison with classical p-value and visualization options.

Usage

```
weighted.p.value.summary(
  u,
  mu,
  sigma,
  alternative = c("two.sided", "less", "greater"),
  plot = FALSE
)
```

Arguments

`u`	A numeric vector representing the uncertainty distribution.
`mu`	The mean of the normal null distribution.
`sigma`	The standard deviation of the normal null distribution.
`alternative`	Character string specifying the alternative hypothesis.
`plot`	Logical; if TRUE, produces a visualization of the analysis.

Value

A list containing:

`weighted.p.value`	The weighted p-value
`classical.p.value`	Classical p-value based on mean of u
`summary.stats`	Summary statistics of the uncertainty distribution
`interpretation`	Text interpretation of results

Examples

```
u_sample <- rnorm(1000, mean = 1.5, sd = 0.5)
summary_results <- weighted.p.value.summary(u_sample, mu = 0, sigma = 1)
print(summary_results)
```

`weights.graph.spectral.ma.lowess` *Extract Model Weights from Graph Spectral MA LOWESS*

Description

Extracts the weights assigned to each bandwidth model in the averaging process. This function is specific to the model-averaged version.

Usage

```
## S3 method for class 'graph.spectral.ma.lowess'
weights(object, vertex = NULL, ...)
```

Arguments

`object`	A 'graph.spectral.ma.lowess' object
`vertex`	Optional vertex index to get weights for specific vertex
`...`	Additional arguments (currently unused)

Value

Matrix of model weights (vertices x bandwidths) or vector for specific vertex

---

`wgraph.prune.long.edges` *Prune Long Edges in a Weighted Graph*

Description

This function prunes long edges in a weighted graph based on the existence of shorter alternative paths. It uses a C++ implementation for efficiency and returns the pruned graph along with information about the pruning process.

Usage

```
wgraph.prune.long.edges(
  graph,
  edge.lengths,
  alt.path.len.ratio.thld,
  use.total.length.constraint = TRUE,
  verbose = FALSE
)
```

Arguments

`graph`	A list of integer vectors representing the adjacency list of the graph. Each element of the list corresponds to a vertex, and contains the indices of its neighboring vertices. The graph should use 1-based indexing (as is standard in R).
`edge.lengths`	A list of numeric vectors representing the lengths of edges. Each element corresponds to a vertex, and contains the lengths of edges to its neighbors. The structure should match that of the `graph` parameter.
`alt.path.len.ratio.thld`	A single numeric value representing the threshold for the alternative path length ratio. Edges are pruned if an alternative path is found with length less than this ratio times the original edge length.
`use.total.length.constraint`	A logical value. If TRUE, the total length of the alternative path must be less than the original edge length. If FALSE, each edge in the alternative path must be shorter than the original edge length. Default is TRUE.
`verbose`	A logical value. If TRUE, progress information will be printed during the pruning process. Default is FALSE.

Details

The function first converts the input graph to 0-based indexing for the C++ function, then calls the C++ implementation to perform the pruning. The pruned graph is converted back to 1-based indexing before being returned.

The pruning process iterates through edges from longest to shortest. For each edge, it searches for an alternative path. If a path is found that is shorter than the threshold ratio times the original edge length, the edge is removed from the graph.

Value

A list containing four elements:

`adj_list`	A list of integer vectors representing the adjacency list of the pruned graph.
`edge_lengths_list`	A list of numeric vectors representing the edge lengths in the pruned graph.
`path_lengths`	A numeric vector of alternative path lengths found during pruning.
`edge_lengths`	A numeric vector of original edge lengths corresponding to path_lengths.

Note

- The function assumes the input graph is undirected.
- The input graph and edge lengths must be consistent (same length and structure).
- The function may modify the order of vertices in the adjacency lists.

Examples

```
## Not run:
# Create a simple weighted graph
graph <- list(c(2,3), c(1,3), c(1,2))
edge.lengths <- list(c(1,2), c(1,3), c(2,3))
threshold <- 0.9

# Prune the graph
result <- wgraph.prune.long.edges(graph, edge.lengths, threshold,
                                  use.total.length.constraint = TRUE,
                                  verbose = TRUE)
# Examine the results
print(result$adj_list)
print(result$edge_lengths_list)
print(result$path_lengths)
print(result$edge_lengths)

## End(Not run)
```

`winsorize` *Winsorize a numeric vector*

Description

Replaces extreme values in a numeric vector with less extreme values. Values below the p-th percentile are set to the p-th percentile, and values above the (1-p)-th percentile are set to the (1-p)-th percentile.

Usage

```
winsorize(x, p = 0.01)
```

Arguments

`x`	A numeric vector to be winsorized
`p`	The proportion of data to be winsorized on each tail (default is 0.01). Must be between 0 and 0.25.

Value

A numeric vector of the same length as `x` with extreme values replaced. NA values in the input are preserved in the output.

See Also

`winsorize.zscore` for robust z-score winsorization

Examples

```
# Simple example
x <- 1:10
winsorize(x, p = 0.2)

# With normally distributed data
set.seed(123)
y <- rnorm(100)
y_wins <- winsorize(y, p = 0.05)

# Compare ranges
range(y)
range(y_wins)
```

`winsorize.zscore` *Winsorized Z-score normalization*

Description

Winsorized Z-score normalization

Usage

```
winsorize.zscore(data, limits = c(0.01, 0.01))
```

Arguments

`data`	A numeric matrix or data frame where rows are samples and columns are features
`limits`	A numeric vector of length 2 specifying the lower and upper proportion of values to be winsorized (default: c(0.01, 0.01) for 1% at each tail)

Value

A matrix of winsorized and Z-scored values

Examples

```
# Example with random data
set.seed(123)
example.data <- matrix(rnorm(100, 5, 2), ncol=5)
# Add some outliers
example.data[1,1] <- 25
example.data[2,3] <- -15
normalized.data <- winsorize.zscore(example.data)
```

`wpredict.1D` *Predicts the mean values of linear models, beta, over grid points of x*

Description

It is a variation on predict.1D() with nn.w passed to it instead of nn.d and nn.kernel for calculation of nn.w inside the corresponding C routine.

Usage

```
wpredict.1D(beta, nn.i, nn.w, nn.x, nx, max.K, y.binary = FALSE)
```

Arguments

`beta`	Coefficients of the local linear model.
`nn.i`	A matrix of indices of x NN's of xgrid.
`nn.w`	A matrix of NN weights.
`nn.x`	The values of predictor variable over NN's of xgrid.
`nx`	The length of x.
`max.K`	A vector of **indices** indicating the range where weights are not 0.
`y.binary`	Set to TRUE if y is a binary variable.

Index