Package ‘fdars’

March 5, 2026

Title Functional Data Analysis in 'Rust'

Version 0.3.3

Description Functional data analysis tools with a high-performance 'Rust' backend. Provides methods for functional data manipulation, depth computation, distance metrics, regression, and statistical testing. Supports both 1D functional data (curves) and 2D functional data (surfaces). Methods are described in Ramsay and Silverman (2005, ISBN:978-0-387-40080-8) ``Functional Data Analysis'' and Ferraty and Vieu (2006, ISBN:978-0-387-30369-7) ``Nonparametric Functional Data Analysis''.

License MIT + file LICENSE

URL `https://sipemu.github.io/fdars-r/`, `https://github.com/sipemu/fdars-r`

BugReports `https://github.com/sipemu/fdars-r/issues`

Encoding UTF-8

RoxygenNote 7.3.3

SystemRequirements Cargo (Rust's package manager), rustc

Imports methods, ggplot2

Suggests testthat (>= 3.0.0), fda.usc, fda, knitr, rmarkdown, dplyr, tidyr, ggforce, gridExtra, patchwork

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation yes

Author Simon Müller [aut, cre]

Maintainer Simon Müller <`simon.mueller@muon-stat.com`>

Repository CRAN

Date/Publication 2026-03-05 19:40:08 UTC

Contents

addError *Add Measurement Error to Functional Data*

Description

Adds independent Gaussian noise to functional data observations.

Usage

```
addError(fdataobj, sd = 0.1, type = c("pointwise", "curve"), seed = NULL)
```

Arguments

`fdataobj`	An object of class `fdata`.
`sd`	Standard deviation of the Gaussian noise.
`type`	Type of noise: **pointwise** (Default) Independent noise at each evaluation point. Each f_i(t_j) gets independent noise. **curve** Common noise level per curve. Each curve gets a single random value added to all its points.
`seed`	Optional integer random seed for reproducibility.

Value

An `fdata` object with added noise.

See Also

`simFunData`, `sparsify`

Examples

```
t <- seq(0, 1, length.out = 100)
fd_clean <- simFunData(n = 20, argvals = t, M = 5, seed = 42)
fd_noisy <- addError(fd_clean, sd = 0.1)

oldpar <- par(mfrow = c(1, 2))
plot(fd_clean, main = "Clean Data")
plot(fd_noisy, main = "With Noise (sd = 0.1)")
par(oldpar)

# Higher noise level
fd_very_noisy <- addError(fd_clean, sd = 0.5)
plot(fd_very_noisy, main = "High Noise (sd = 0.5)")
```

`AKer.cos` *Asymmetric Cosine Kernel*

Description

Asymmetric Cosine Kernel

Usage

```
AKer.cos(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 for u < 0).

Examples

```
u <- seq(-0.5, 1.5, length.out = 100)
plot(u, AKer.cos(u), type = "l", main = "Asymmetric Cosine Kernel")
```

`AKer.epa` *Asymmetric Epanechnikov Kernel*

Description

Asymmetric Epanechnikov Kernel

Usage

```
AKer.epa(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [0, 1]).

Examples

```
u <- seq(-0.5, 1.5, length.out = 100)
plot(u, AKer.epa(u), type = "l", main = "Asymmetric Epanechnikov Kernel")
```

AKer.norm *Asymmetric Normal Kernel*

Description

Asymmetric Normal Kernel

Usage

```
AKer.norm(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 for u < 0).

Examples

```
u <- seq(-0.5, 3, length.out = 100)
plot(u, AKer.norm(u), type = "l", main = "Asymmetric Normal Kernel")
```

AKer.quar *Asymmetric Quartic Kernel*

Description

Asymmetric Quartic Kernel

Usage

```
AKer.quar(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [0, 1]).

Examples

```
u <- seq(-0.5, 1.5, length.out = 100)
plot(u, AKer.quar(u), type = "l", main = "Asymmetric Quartic Kernel")
```

AKer.tri *Asymmetric Triweight Kernel*

Description

Asymmetric Triweight Kernel

Usage

```
AKer.tri(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [0, 1]).

Examples

```
u <- seq(-0.5, 1.5, length.out = 100)
plot(u, AKer.tri(u), type = "l", main = "Asymmetric Triweight Kernel")
```

AKer.unif *Asymmetric Uniform Kernel*

Description

Asymmetric Uniform Kernel

Usage

```
AKer.unif(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [0, 1]).

Examples

```
u <- seq(-0.5, 1.5, length.out = 100)
plot(u, AKer.unif(u), type = "l", main = "Asymmetric Uniform Kernel")
```

`analyze.peak.timing` *Analyze Peak Timing Variability*

Description

For short series (e.g., 3-5 years of yearly data), this function detects one peak per cycle and analyzes how peak timing varies between cycles. Uses Fourier basis smoothing for peak detection.

Usage

```
analyze.peak.timing(fdataobj, period, smooth_nbasis = NULL)
```

Arguments

`fdataobj`	An fdata object.
`period`	Known period (e.g., 365 for daily data with yearly seasonality).
`smooth_nbasis`	Number of Fourier basis functions for smoothing. If NULL, uses GCV for automatic selection (range 5-25). Default: NULL.

Details

The variability score is computed as std_timing / 0.1, capped at 1. A score > 0.5 suggests peaks are shifting substantially between cycles. The timing_trend indicates if peaks are systematically moving earlier or later over time.

Fourier basis smoothing is ideal for seasonal signals because it naturally captures periodic patterns.

Value

A list with components:

peak_times Vector of peak times

peak_values Vector of peak values

normalized_timing Position within cycle (0-1 scale)

mean_timing Mean normalized timing

std_timing Standard deviation of normalized timing

range_timing Range of normalized timing (max - min)

variability_score Variability score (0 = stable, 1 = highly variable)

timing_trend Linear trend in timing (positive = peaks getting later)

cycle_indices Cycle indices (1-indexed)

Examples

```
# 5 years of yearly data where peak shifts
t <- seq(0, 5, length.out = 365 * 5)
periods <- c(1, 1, 1, 1, 1)  # 5 complete years
# Peaks shift: March (0.2), April (0.3), May (0.4), April (0.3), March (0.2)
peak_phases <- c(0.2, 0.3, 0.4, 0.3, 0.2)
X <- sin(2 * pi * t + rep(peak_phases, each = 365))
fd <- fdata(matrix(X, nrow = 1), argvals = t)

result <- analyze.peak.timing(fd, period = 1)
print(result$variability_score)  # Shows timing variability
```

---

as.fdata.irregFdata *Convert Irregular Functional Data to Regular Grid*

Description

Creates a regular `fdata` object from an `irregFdata` object by interpolating or placing `NA` at unobserved points.

Usage

```
## S3 method for class 'irregFdata'
as.fdata(x, argvals = NULL, method = c("na", "linear", "nearest"), ...)

as.fdata(x, ...)

## S3 method for class 'fdata'
as.fdata(x, ...)
```

Arguments

`x`	An object of class `irregFdata`.
`argvals`	Target regular grid. If `NULL`, uses the union of all observation points.
`method`	Interpolation method: **na** (Default) Only fill exact matches; other points are `NA` **linear** Linear interpolation between observed points **nearest** Nearest neighbor interpolation
`...`	Additional arguments (ignored).

Value

An `fdata` object with `NA` for unobserved points (unless interpolated).

See Also

`sparsify`, `irregFdata`

Examples

```
# Create sparse data
t <- seq(0, 1, length.out = 100)
fd <- simFunData(n = 10, argvals = t, M = 5, seed = 42)
ifd <- sparsify(fd, minObs = 10, maxObs = 30, seed = 123)

# Convert back to regular grid with NA
fd_na <- as.fdata(ifd)

# Convert with linear interpolation
fd_interp <- as.fdata(ifd, method = "linear")
```

---

autoperiod *Autoperiod: Hybrid FFT + ACF Period Detection*

Description

Implements the Autoperiod algorithm (Vlachos et al. 2005) which combines FFT-based candidate detection with ACF validation and gradient ascent refinement for robust period estimation.

Usage

```
autoperiod(
  fdataobj,
  n_candidates = 5,
  gradient_steps = 10,
  detrend_method = c("none", "linear", "auto")
)
```

Arguments

`fdataobj`	An fdata object.
`n_candidates`	Maximum number of FFT peaks to consider as candidates. Default: 5. More candidates increases robustness but also computation time.
`gradient_steps`	Number of gradient ascent steps for period refinement. Default: 10. More steps improves precision.
`detrend_method`	Detrending method to apply before period estimation: **"none"** No detrending (default) **"linear"** Remove linear trend **"auto"** Automatic AIC-based selection of detrending method

Details

The Autoperiod algorithm works in three stages:

1. **Candidate Detection**: Finds peaks in the FFT periodogram
2. **ACF Validation**: Validates each candidate using the autocorrelation function. Checks that the ACF shows a peak at the candidate period, and applies harmonic analysis to distinguish fundamental periods from harmonics.
3. **Gradient Refinement**: Refines each candidate using gradient ascent on the ACF to find the exact period that maximizes the ACF peak.

The final period is chosen based on the product of normalized FFT power and ACF validation score.

Value

A list of class "autoperiod_result" with components:

period Best detected period

confidence Combined confidence (normalized FFT power * ACF validation)

fft_power FFT power at the detected period

acf_validation ACF validation score (0-1)

n_candidates Number of candidates evaluated

candidates Data frame of all candidate periods with their scores

References

Vlachos, M., Yu, P., & Castelli, V. (2005). On periodicity detection and structural periodic similarity. In Proceedings of the 2005 SIAM International Conference on Data Mining.

See Also

`sazed` for an ensemble method, `estimate.period` for simpler single-method estimation

Examples

```
# Generate seasonal data with period = 2
t <- seq(0, 20, length.out = 400)
X <- matrix(sin(2 * pi * t / 2) + 0.1 * rnorm(400), nrow = 1)
fd <- fdata(X, argvals = t)

# Detect period using Autoperiod
result <- autoperiod(fd)
print(result)

# View all candidates
print(result$candidates)
```

autoplot.fdata *Create a ggplot for fdata objects*

Description

For 1D functional data, plots curves as lines with optional coloring by external variables. For 2D functional data, plots surfaces as heatmaps with contour lines.

Usage

```
## S3 method for class 'fdata'
autoplot(
  object,
  color = NULL,
  alpha = NULL,
  show.mean = FALSE,
  show.ci = FALSE,
  ci.level = 0.9,
  palette = NULL,
  ...
)
```

Arguments

`object`	An object of class 'fdata'.
`color`	Optional vector for coloring curves. Can be: • Numeric vector: curves colored by continuous scale (viridis) • Factor/character: curves colored by discrete groups Must have length equal to number of curves.
`alpha`	Transparency of individual curve lines. Default is 0.7 for basic plots, but automatically reduced to 0.3 when `show.mean = TRUE` or `show.ci = TRUE` to reduce visual clutter and allow mean curves to stand out. Can be explicitly set to override the default.
`show.mean`	Logical. If TRUE and color is categorical, overlay group mean curves with thicker lines (default FALSE).
`show.ci`	Logical. If TRUE and color is categorical, show pointwise confidence interval ribbons per group (default FALSE).
`ci.level`	Confidence level for CI ribbons (default 0.90 for 90 percent).
`palette`	Optional named vector of colors for categorical coloring, e.g., c("A" = "blue", "B" = "red").
`...`	Additional arguments (currently ignored).

Details

Use `autoplot()` to get the ggplot object without displaying it. Use `plot()` to display the plot (returns invisibly).

Value

A ggplot object.

Examples

```
library(ggplot2)
# Get ggplot object without displaying
fd <- fdata(matrix(rnorm(200), 20, 10))
p <- autoplot(fd)

# Customize the plot
p + theme_minimal()

# Color by numeric variable
y <- rnorm(20)
autoplot(fd, color = y)

# Color by category with mean and CI
groups <- factor(rep(c("A", "B"), each = 10))
autoplot(fd, color = groups, show.mean = TRUE, show.ci = TRUE)
```

`autoplot.irregFdata` *Autoplot method for irregFdata objects*

Description

Autoplot method for irregFdata objects

Usage

```
## S3 method for class 'irregFdata'
autoplot(object, ..., alpha = 0.7)
```

Arguments

`object`	An irregFdata object.
`...`	Additional arguments (ignored).
`alpha`	Transparency for lines.

Value

A ggplot object.

basis.aic *AIC for Basis Representation*

Description

Computes the Akaike Information Criterion for a basis representation. Lower AIC indicates better model (balancing fit and complexity).

Usage

```
basis.aic(
  fdataobj,
  nbasis,
  type = c("bspline", "fourier"),
  lambda = 0,
  pooled = TRUE
)
```

Arguments

`fdataobj`	An fdata object.
`nbasis`	Number of basis functions.
`type`	Basis type: "bspline" (default) or "fourier".
`lambda`	Smoothing/penalty parameter (default 0).
`pooled`	Logical. If TRUE (default), compute a single AIC across all curves. If FALSE, compute AIC for each curve and return the mean.

Details

AIC is computed as:

$$AIC = n \log(RSS/n) + 2 \cdot edf$$

When `pooled = TRUE`, the criterion uses total observations and total effective degrees of freedom (n_curves * edf). When `pooled = FALSE`, the criterion is computed for each curve separately and the mean is returned.

Value

The AIC value (scalar).

basis.bic *BIC for Basis Representation*

Description

Computes the Bayesian Information Criterion for a basis representation. BIC penalizes complexity more strongly than AIC for larger samples.

Usage

```
basis.bic(
  fdataobj,
  nbasis,
  type = c("bspline", "fourier"),
  lambda = 0,
  pooled = TRUE
)
```

Arguments

`fdataobj`	An fdata object.
`nbasis`	Number of basis functions.
`type`	Basis type: "bspline" (default) or "fourier".
`lambda`	Smoothing/penalty parameter (default 0).
`pooled`	Logical. If TRUE (default), compute a single BIC across all curves. If FALSE, compute BIC for each curve and return the mean.

Details

BIC is computed as:

$$BIC = n\log(RSS/n) + \log(n) \cdot edf$$

When `pooled = TRUE`, the criterion uses total observations and total effective degrees of freedom (n_curves * edf). When `pooled = FALSE`, the criterion is computed for each curve separately and the mean is returned.

Value

The BIC value (scalar).

`basis.gcv` *GCV Score for Basis Representation*

Description

Computes the Generalized Cross-Validation score for a basis representation. Lower GCV indicates better fit with appropriate complexity.

Usage

```
basis.gcv(
  fdataobj,
  nbasis,
  type = c("bspline", "fourier"),
  lambda = 0,
  pooled = TRUE
)
```

Arguments

`fdataobj`	An fdata object.
`nbasis`	Number of basis functions.
`type`	Basis type: "bspline" (default) or "fourier".
`lambda`	Smoothing/penalty parameter (default 0, no penalty).
`pooled`	Logical. If TRUE (default), compute a single GCV across all curves. If FALSE, compute GCV for each curve and return the mean.

Details

GCV is computed as:

$$GCV = \frac{RSS/n}{(1 - edf/n)^2}$$

where RSS is the residual sum of squares and edf is the effective degrees of freedom (trace of the hat matrix).

When `pooled = TRUE`, the criterion is computed globally across all curves. When `pooled = FALSE`, the criterion is computed for each curve separately and the mean is returned. Use `pooled = FALSE` when curves have heterogeneous noise levels.

Value

The GCV score (scalar).

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(sin(2 * pi * t) + rnorm(50, sd = 0.1), nrow = 1)
fd <- fdata(X, argvals = t)

# Compare GCV for different nbasis
gcv_5 <- basis.gcv(fd, nbasis = 5)
gcv_10 <- basis.gcv(fd, nbasis = 10)
gcv_20 <- basis.gcv(fd, nbasis = 20)
```

---

basis2fdata *Basis Representation Functions for Functional Data*

Description

Functions for representing functional data using basis expansions, including B-splines, Fourier bases, and P-splines with penalization. Reconstruct Functional Data from Basis Coefficients

Usage

```
basis2fdata(
  coefs,
  argvals,
  nbasis = NULL,
  type = c("bspline", "fourier"),
  rangeval = NULL
)
```

Arguments

`coefs`	Coefficient matrix `[n x nbasis]` where n is the number of curves and nbasis is the number of basis functions. Can also be a vector for a single curve.
`argvals`	Numeric vector of evaluation points for reconstruction.
`nbasis`	Number of basis functions. If NULL, inferred from ncol(coefs).
`type`	Basis type: "bspline" (default) or "fourier".
`rangeval`	Range of argvals. Default: `range(argvals)`.

Details

Given basis coefficients, reconstruct the functional data by evaluating the basis expansion at specified argument values.

The reconstruction computes `X(t) = sum(coef_k * B_k(t))` where `B_k` are the basis functions evaluated at `argvals`.

Value

An fdata object with reconstructed curves.

Examples

```
# Create some functional data
t <- seq(0, 1, length.out = 100)
X <- matrix(sin(2 * pi * t), nrow = 1)
fd <- fdata(X, argvals = t)

# Project to basis and reconstruct
coefs <- fdata2basis(fd, nbasis = 15, type = "fourier")
fd_recon <- basis2fdata(coefs, argvals = t, type = "fourier")
```

---

basis2fdata_2d *Reconstruct 2D Functional Data from Tensor Product Basis Coefficients*

Description

Reconstructs 2D surfaces from tensor product basis coefficients.

Usage

```
basis2fdata_2d(
  coefs,
  argvals,
  nbasis.s,
  nbasis.t,
  type = c("bspline", "fourier")
)
```

Arguments

coefs	Coefficient matrix [n x (nbasis.s * nbasis.t)].
argvals	List with two numeric vectors for s and t coordinates.
nbasis.s	Number of basis functions in s direction.
nbasis.t	Number of basis functions in t direction.
type	Basis type: "bspline" (default) or "fourier".

Value

A 2D fdata object.

`boxplot.fdata` *Functional Boxplot*

Description

Creates a functional boxplot for visualizing the distribution of functional data. The boxplot shows the median curve, central 50 percent envelope, fence (equivalent to whiskers), and outliers.

Usage

```
## S3 method for class 'fdata'
boxplot(
  x,
  prob = 0.5,
  factor = 1.5,
  depth.func = depth.MBD,
  show.outliers = TRUE,
  col.median = "black",
  col.envelope = "magenta",
  col.fence = "pink",
  col.outliers = "red",
  ...
)
```

Arguments

`x`	An object of class 'fdata'.
`prob`	Proportion of curves for the central region (default 0.5 for 50 percent).
`factor`	Factor for fence calculation (default 1.5, as in standard boxplots).
`depth.func`	Depth function to use. Default is depth.MBD.
`show.outliers`	Logical. If TRUE (default), show outlier curves.
`col.median`	Color for median curve (default "black").
`col.envelope`	Color for central envelope (default "magenta").
`col.fence`	Color for fence region (default "pink").
`col.outliers`	Color for outlier curves (default "red").
`...`	Additional arguments passed to depth function.

Details

The functional boxplot (Sun & Genton, 2011) generalizes the standard boxplot to functional data using depth ordering:

- **Median**: The curve with maximum depth
- **Central region**: Envelope of curves with top 50 percent depth
- **Fence**: 1.5 times the envelope width beyond the central region
- **Outliers**: Curves that exceed the fence at any point

Value

A list of class 'fbplot' with components:

median Index of the median curve

central Indices of curves in the central region

outliers Indices of outlier curves

depth Depth values for all curves

plot The ggplot object

References

Sun, Y. and Genton, M.G. (2011). Functional boxplots. *Journal of Computational and Graphical Statistics*, 20(2), 316-334.

See Also

depth.MBD for the default depth function, outliers.boxplot for outlier detection using functional boxplots

Examples

```
# Create functional data with outliers
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:28) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.2)
X[29, ] <- sin(2*pi*t) + 2  # Magnitude outlier
X[30, ] <- cos(2*pi*t)       # Shape outlier
fd <- fdata(X, argvals = t)

# Create functional boxplot
fbp <- boxplot(fd)
```

---

cfd.autoperiod *CFDAutoperiod: Clustered Filtered Detrended Autoperiod*

Description

Implements the CFDAutoperiod algorithm (Puech et al. 2020) which applies first-order differencing for detrending, then uses density-based clustering of spectral peaks and ACF validation on the original signal.

Usage

```
cfd.autoperiod(fdataobj, cluster_tolerance = 0.1, min_cluster_size = 1)
```

Arguments

`fdataobj`	An fdata object.
`cluster_tolerance`	Relative tolerance for clustering nearby period candidates. Default: 0.1 (10% relative difference). Candidates within this tolerance are grouped into clusters.
`min_cluster_size`	Minimum number of candidates required to form a valid cluster. Default: 1.

Details

The CFDAutoperiod algorithm works in five stages:

1. **Differencing**: First-order differencing removes polynomial trends
2. **FFT**: Computes periodogram on the detrended signal
3. **Peak Detection**: Finds all peaks above the noise floor
4. **Clustering**: Groups nearby period candidates into clusters
5. **ACF Validation**: Validates cluster centers using the ACF of the original (non-differenced) signal

This method is particularly effective for:

- Signals with strong polynomial trends
- Multiple concurrent periodicities
- Signals where spectral leakage creates nearby spurious peaks

Value

A list of class "cfd_autoperiod_result" with components:

period Primary detected period (best validated cluster center)

confidence Combined confidence (ACF validation * spectral power)

acf_validation ACF validation score for the primary period

n_periods Number of validated period clusters

periods All detected period cluster centers

confidences Confidence scores for each detected period

References

Puech, T., Boussard, M., D'Amato, A., & Millerand, G. (2020). A fully automated periodicity detection in time series. In Advanced Analytics and Learning on Temporal Data (pp. 43-54). Springer.

See Also

`autoperiod` for the original Autoperiod algorithm, `sazed` for an ensemble method

Examples

```
# Generate data with trend
t <- seq(0, 20, length.out = 400)
X <- matrix(0.2 * t + sin(2 * pi * t / 2), nrow = 1)
fd <- fdata(X, argvals = t)

# CFDAutoperiod handles trends via differencing
result <- cfd.autoperiod(fd)
print(result)

# Multiple periods detected
X2 <- matrix(sin(2 * pi * t / 2) + 0.5 * sin(2 * pi * t / 5), nrow = 1)
fd2 <- fdata(X2, argvals = t)
result2 <- cfd.autoperiod(fd2)
print(result2$periods)  # All detected periods
```

`classify.seasonality` *Classify Seasonality Type*

Description

Classifies the type of seasonality in functional data. Particularly useful for short series (3-5 years) to identify stable vs variable timing patterns.

Usage

```
classify.seasonality(
  fdataobj,
  period,
  strength_threshold = NULL,
  timing_threshold = NULL
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Known seasonal period.
`strength_threshold`	Threshold for seasonal/non-seasonal (default: 0.3).
`timing_threshold`	Max std of normalized timing for "stable" (default: 0.05).

Details

Classification types:

- StableSeasonal: Regular peaks with consistent timing

- VariableTiming: Regular peaks but timing shifts between cycles
- IntermittentSeasonal: Some cycles seasonal, some not
- NonSeasonal: No clear seasonality

Value

A list with components:

is_seasonal Logical: is the series seasonal overall?

has_stable_timing Logical: is peak timing stable across cycles?

timing_variability Timing variability score (0-1)

seasonal_strength Overall seasonal strength

cycle_strengths Per-cycle seasonal strength

weak_seasons Indices of weak/missing seasons (0-indexed)

classification One of: "StableSeasonal", "VariableTiming", "IntermittentSeasonal", "NonSeasonal"

peak_timing Peak timing analysis (if peaks detected)

Examples

```
# Pure seasonal signal
t <- seq(0, 10, length.out = 500)
X <- matrix(sin(2 * pi * t / 2), nrow = 1)
fd <- fdata(X, argvals = t)

result <- classify.seasonality(fd, period = 2)
print(result$classification)  # "StableSeasonal"
```

`cluster.fcm` *Fuzzy C-Means Clustering for Functional Data*

Description

Performs fuzzy c-means clustering on functional data, where each curve has a membership degree to each cluster rather than a hard assignment.

Usage

```
cluster.fcm(fdataobj, ncl, m = 2, max.iter = 100, tol = 1e-06, seed = NULL)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`ncl`	Number of clusters.
`m`	Fuzziness parameter (default 2). Must be > 1. Higher values give softer cluster boundaries.
`max.iter`	Maximum number of iterations (default 100).
`tol`	Convergence tolerance (default 1e-6).
`seed`	Optional random seed for reproducibility.

Details

Fuzzy c-means minimizes the objective function:

$$J = \sum_{i=1}^{n} \sum_{c=1}^{k} u_{ic}^{m} ||X_i - v_c||^2$$

where u_ic is the membership of curve i in cluster c, v_c is the cluster center, and m is the fuzziness parameter.

The membership degrees are updated as:

$$u_{ic} = 1/\sum_{j=1}^{k}(d_{ic}/d_{ij})^{2/(m-1)}$$

When m approaches 1, FCM becomes equivalent to hard k-means. As m increases, the clusters become softer (more overlap). m = 2 is the most common choice.

Value

A list of class 'fuzzycmeans.fd' with components:

membership Matrix of membership degrees (n x ncl). Each row sums to 1.

cluster Hard cluster assignments (argmax of membership).

centers An fdata object containing the cluster centers.

objective Final value of the objective function.

fdataobj The input functional data object.

See Also

`cluster.kmeans` for hard clustering

Examples

```
# Create functional data with THREE groups - one genuinely overlapping
set.seed(42)
t <- seq(0, 1, length.out = 50)
n <- 45
X <- matrix(0, n, 50)

# Group 1: Sine waves centered at 0
for (i in 1:15) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.2)
# Group 2: Sine waves centered at 1.5 (clearly separated from group 1)
for (i in 16:30) X[i, ] <- sin(2*pi*t) + 1.5 + rnorm(50, sd = 0.2)
# Group 3: Between groups 1 and 2 (true overlap - ambiguous membership)
for (i in 31:45) X[i, ] <- sin(2*pi*t) + 0.75 + rnorm(50, sd = 0.3)

fd <- fdata(X, argvals = t)

# Fuzzy clustering reveals the overlap
```

```
fcm <- cluster.fcm(fd, ncl = 3, seed = 123)

# Curves in group 3 (31-45) have split membership - this is the key benefit!
cat("Membership for curves 31-35 (overlap region):\n")
print(round(fcm$membership[31:35, ], 2))

# Compare to hard clustering which forces a decision
km <- cluster.kmeans(fd, ncl = 3, seed = 123)
cat("\nHard vs Fuzzy assignment for curve 35:\n")
cat("K-means cluster:", km$cluster[35], "\n")
cat("FCM memberships:", round(fcm$membership[35, ], 2), "\n")
```

`cluster.init` *K-Means++ Center Initialization*

Description

Initialize cluster centers using the k-means++ algorithm, which selects centers with probability proportional to squared distance from existing centers.

Usage

```
cluster.init(fdataobj, ncl, metric = "L2", seed = NULL)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`ncl`	Number of clusters.
`metric`	Metric to use. One of "L2", "L1", or "Linf".
`seed`	Optional random seed.

Value

An fdata object containing the initial cluster centers.

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(rnorm(30 * 50), 30, 50)
fd <- fdata(X, argvals = t)
init_centers <- cluster.init(fd, ncl = 3)
```

cluster.kmeans *Clustering Functions for Functional Data*

Description

Functions for clustering functional data, including k-means and related algorithms. Functional K-Means Clustering

Usage

```
cluster.kmeans(
  fdataobj,
  ncl,
  metric = "L2",
  max.iter = 100,
  nstart = 10,
  seed = NULL,
  draw = FALSE,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`ncl`	Number of clusters.
`metric`	Either a string ("L2", "L1", "Linf") for fast Rust-based distance computation, or a metric/semimetric function (e.g., `metric.lp`, `metric.hausdorff`, `semimetric.pca`). Using a function provides flexibility but may be slower for semimetrics computed in R.
`max.iter`	Maximum number of iterations (default 100).
`nstart`	Number of random starts (default 10). The best result (lowest within-cluster sum of squares) is returned.
`seed`	Optional random seed for reproducibility.
`draw`	Logical. If TRUE, plot the clustered curves (not yet implemented).
`...`	Additional arguments passed to the metric function.

Details

Performs k-means clustering on functional data using the specified metric. Uses k-means++ initialization for better initial centers.

When `metric` is a string ("L2", "L1", "Linf"), the entire k-means algorithm runs in Rust with parallel processing, providing 50-200x speedup.

When `metric` is a function, distances are computed using that function. Functions like `metric.lp`, `metric.hausdorff`, and `metric.DTW` have Rust backends and remain fast. Semimetric functions (`semimetric.*`) are computed in R and will be slower for large datasets.

Value

A list of class 'cluster.kmeans' with components:

cluster Integer vector of cluster assignments (1 to ncl).

centers An fdata object containing the cluster centers.

withinss Within-cluster sum of squares for each cluster.

tot.withinss Total within-cluster sum of squares.

size Number of observations in each cluster.

fdataobj The input functional data object.

Examples

```
# Create functional data with two groups
t <- seq(0, 1, length.out = 50)
n <- 30
X <- matrix(0, n, 50)
true_cluster <- rep(1:2, each = 15)
for (i in 1:n) {
  if (true_cluster[i] == 1) {
    X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
  } else {
    X[i, ] <- cos(2*pi*t) + rnorm(50, sd = 0.1)
  }
}
fd <- fdata(X, argvals = t)

# Cluster with string metric (fast Rust path)
result <- cluster.kmeans(fd, ncl = 2, metric = "L2")
table(result$cluster, true_cluster)

# Cluster with metric function (also fast - Rust backend)
result2 <- cluster.kmeans(fd, ncl = 2, metric = metric.lp)

# Cluster with semimetric (flexible but slower)
result3 <- cluster.kmeans(fd, ncl = 2, metric = semimetric.pca, ncomp = 3)
```

`cluster.optim` *Optimal Number of Clusters for Functional K-Means*

Description

Determines the optimal number of clusters for functional k-means clustering using various criteria: elbow method, silhouette score, or Calinski-Harabasz index.

Usage

```
cluster.optim(
  fdataobj,
  ncl.range = 2:10,
  criterion = c("silhouette", "CH", "elbow"),
  metric = "L2",
  max.iter = 100,
  nstart = 10,
  seed = NULL,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`ncl.range`	Range of number of clusters to evaluate. Default is 2:10.
`criterion`	Criterion to use for selecting optimal k: **"silhouette"** Mean silhouette coefficient (default). Higher is better. **"CH"** Calinski-Harabasz index. Higher is better. **"elbow"** Within-cluster sum of squares. Look for elbow in plot.
`metric`	Either a string ("L2", "L1", "Linf") or a metric function.
`max.iter`	Maximum iterations for k-means (default 100).
`nstart`	Number of random starts (default 10).
`seed`	Random seed for reproducibility.
`...`	Additional arguments passed to cluster.kmeans.

Details

Silhouette score: Measures how similar each curve is to its own cluster compared to other clusters. Values range from -1 to 1, with higher being better. Optimal k maximizes the mean silhouette.

Calinski-Harabasz index: Ratio of between-cluster to within-cluster dispersion. Higher values indicate better defined clusters. Optimal k maximizes CH.

Elbow method: Plots total within-cluster sum of squares vs k. The optimal k is at the "elbow" where adding more clusters doesn't significantly reduce WSS. This is subjective and best assessed visually using `plot()`.

Value

A list of class 'cluster.optim' with components:

optimal.k Optimal number of clusters based on criterion

criterion Name of criterion used

scores Vector of criterion values for each k

ncl.range Range of k values tested

models List of cluster.kmeans objects for each k

best.model The cluster.kmeans object for optimal k

Examples

```
# Create functional data with 3 groups
set.seed(42)
t <- seq(0, 1, length.out = 50)
n <- 60
X <- matrix(0, n, 50)
true_k <- rep(1:3, each = 20)
for (i in 1:n) {
  if (true_k[i] == 1) {
    X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
  } else if (true_k[i] == 2) {
    X[i, ] <- cos(2*pi*t) + rnorm(50, sd = 0.1)
  } else {
    X[i, ] <- sin(4*pi*t) + rnorm(50, sd = 0.1)
  }
}
fd <- fdata(X, argvals = t)

# Find optimal k using silhouette
opt <- cluster.optim(fd, ncl.range = 2:6, criterion = "silhouette")
print(opt)
plot(opt)
```

`cov` *Functional Covariance Function*

Description

Computes the covariance function (surface) for functional data. For 1D: `Cov(s, t) = E[(X(s) - mu(s))(X(t) - mu(t))]` For 2D: Covariance across the flattened domain.

Usage

```
cov(fdataobj, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`...`	Additional arguments (currently ignored).

Value

A list with components:

cov The covariance matrix (m x m for 1D, (m1*m2) x (m1*m2) for 2D)

argvals The evaluation points (same as input)

mean The mean function

Examples

```
# 1D functional data
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.2)
fd <- fdata(X, argvals = t)
cov_result <- cov(fd)
image(cov_result$cov, main = "Covariance Surface")
```

`CV.S` *Cross-Validation for Smoother Selection*

Description

Compute leave-one-out cross-validation criterion for a smoother.

Usage

```
CV.S(S.type, tt, h, y, Ker = "norm", w = NULL)
```

Arguments

`S.type`	Function to compute smoother matrix (e.g., S.NW, S.LLR).
`tt`	Evaluation points.
`h`	Bandwidth parameter.
`y`	Response vector to smooth.
`Ker`	Kernel type.
`w`	Optional weights.

Value

The cross-validation score (mean squared prediction error).

Examples

```
tt <- seq(0, 1, length.out = 50)
y <- sin(2 * pi * tt) + rnorm(50, sd = 0.1)
cv_score <- CV.S(S.NW, tt, h = 0.1, y = y)
```

`decompose` *Seasonal-Trend Decomposition*

Description

Decomposes functional data into trend, seasonal, and remainder components. Similar to STL (Seasonal-Trend decomposition using LOESS).

Usage

```
decompose(
  fdataobj,
  period = NULL,
  method = c("additive", "multiplicative"),
  trend_method = c("loess", "spline"),
  bandwidth = 0.3,
  n_harmonics = 3
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Seasonal period. If NULL, estimated automatically using FFT.
`method`	Decomposition method: **"additive"** data = trend + seasonal + remainder (default) **"multiplicative"** data = trend * seasonal * remainder
`trend_method`	Method for trend extraction: "loess" or "spline". Default: "loess".
`bandwidth`	Bandwidth for trend extraction (fraction of range). Default: 0.3.
`n_harmonics`	Number of Fourier harmonics for seasonal component. Default: 3.

Details

For additive decomposition: data = trend + seasonal + remainder. The trend is extracted using LOESS or spline smoothing, then the seasonal component is estimated by fitting Fourier harmonics to the detrended data.

For multiplicative decomposition: data = trend * seasonal * remainder. This is achieved by log-transforming the data, applying additive decomposition, and back-transforming. Use this when the seasonal amplitude grows with the trend level.

Value

A list with components:

trend fdata object with trend component

seasonal fdata object with seasonal component

remainder fdata object with remainder/residual

period Period used for decomposition

method Decomposition method ("additive" or "multiplicative")

See Also

`detrend` for simple trend removal, `seasonal.strength` for measuring seasonality

Examples

```
# Additive seasonal pattern
t <- seq(0, 20, length.out = 400)
X <- matrix(2 + 0.3 * t + sin(2 * pi * t / 2.5), nrow = 1)
fd <- fdata(X, argvals = t)

result <- decompose(fd, period = 2.5, method = "additive")
# plot(result$trend)      # Linear trend
# plot(result$seasonal)   # Sinusoidal seasonal
# plot(result$remainder)  # Residual noise

# Multiplicative pattern (amplitude grows with level)
X_mult <- matrix((2 + 0.3 * t) * (1 + 0.3 * sin(2 * pi * t / 2.5)), nrow = 1)
fd_mult <- fdata(X_mult, argvals = t)

result_mult <- decompose(fd_mult, period = 2.5, method = "multiplicative")
```

`depth` *Depth Functions for Functional Data*

Description

Functions for computing various depth measures for functional data. Compute Functional Data Depth

Usage

```
depth(
  fdataobj,
  fdataori = NULL,
  method = c("FM", "mode", "RP", "RT", "BD", "MBD", "MEI", "FSD", "KFSD", "RPD"),
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata' to compute depth for.
`fdataori`	An object of class 'fdata' as reference sample. If NULL, uses fdataobj as reference.

`method`	Depth method to use. One of "FM" (Fraiman-Muniz), "mode" (modal), "RP" (random projection), "RT" (random Tukey), "BD" (band depth), "MBD" (modified band depth), "MEI" (modified epigraph index), "FSD" (functional spatial depth), "KFSD" (kernel functional spatial depth), or "RPD" (random projection with derivatives). Default is "FM".
`...`	Additional arguments passed to the specific depth function.

Details

Unified interface for computing various depth measures for functional data.

Available methods:

FM Fraiman-Muniz depth - integrates univariate depths over domain

mode Modal depth - based on kernel density estimation

RP Random projection depth - projects to random directions

RT Random Tukey depth - halfspace depth via random projections

BD Band depth - proportion of bands containing the curve (1D only)

MBD Modified band depth - allows partial containment (1D only)

MEI Modified epigraph index - proportion of time below other curves (1D only)

FSD Functional spatial depth - based on spatial signs

KFSD Kernel functional spatial depth - smoothed FSD

RPD Random projection with derivatives - includes curve derivatives

Value

A numeric vector of depth values, one per curve in fdataobj.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))

# Different depth methods
depth(fd, method = "FM")
depth(fd, method = "mode")
depth(fd, method = "RP")
```

`depth.BD` *Band Depth*

Description

Wrapper for `depth(method = "BD")`. Useful as a function argument.

Usage

```
depth.BD(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

depth

`depth.FM` *Fraiman-Muniz Depth*

Description

Wrapper for `depth(method = "FM")`. Useful as a function argument.

Usage

```
depth.FM(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

depth

`depth.FSD` *Functional Spatial Depth*

Description

Wrapper for `depth(method = "FSD")`. Useful as a function argument.

Usage

```
depth.FSD(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

`depth`

`depth.KFSD` *Kernel Functional Spatial Depth*

Description

Wrapper for `depth(method = "KFSD")`. Useful as a function argument.

Usage

```
depth.KFSD(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

depth

depth.MBD *Modified Band Depth*

Description

Wrapper for `depth(method = "MBD")`. Useful as a function argument.

Usage

```
depth.MBD(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

depth

depth.MEI *Modified Epigraph Index*

Description

Wrapper for `depth(method = "MEI")`. Useful as a function argument.

Usage

```
depth.MEI(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

depth

---

depth.mode *Modal Depth*

Description

Wrapper for `depth(method = "mode")`. Useful as a function argument.

Usage

```
depth.mode(fdataobj, fdataori = NULL, ...)
```

Arguments

fdataobj	An object of class 'fdata'.
fdataori	Reference sample (default: fdataobj itself).
...	Additional arguments.

Value

Numeric vector of depth values.

See Also

depth

---

depth.RP *Random Projection Depth*

Description

Wrapper for `depth(method = "RP")`. Useful as a function argument.

Usage

```
depth.RP(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

`depth`

`depth.RPD` *Random Projection Depth with Derivatives*

Description

Wrapper for `depth(method = "RPD")`. Useful as a function argument.

Usage

```
depth.RPD(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

`depth`

`depth.RT` *Random Tukey Depth*

Description

Wrapper for `depth(method = "RT")`. Useful as a function argument.

Usage

```
depth.RT(fdataobj, fdataori = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataori`	Reference sample (default: fdataobj itself).
`...`	Additional arguments.

Value

Numeric vector of depth values.

See Also

`depth`

`deriv` *Compute functional derivative*

Description

Compute the numerical derivative of functional data. Uses finite differences for fast computation via Rust.

Usage

```
deriv(
  fdataobj,
  nderiv = 1,
  method = "diff",
  class.out = "fdata",
  nbasis = NULL,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`nderiv`	Derivative order (1, 2, ...). Default is 1. For 2D data, only first-order derivatives are currently supported.
`method`	Method for computing derivatives. Currently only "diff" (finite differences) is supported.
`class.out`	Output class, either "fdata" or "fd". Default is "fdata".
`nbasis`	Not used (for compatibility with fda.usc).
`...`	Additional arguments (ignored).

Details

For 1D functional data (curves), computes the nth derivative. For 2D functional data (surfaces), computes partial derivatives:

- `ds`: partial derivative with respect to s (first argument)
- `dt`: partial derivative with respect to t (second argument)
- `dsdt`: mixed partial derivative

Value

For 1D data: an 'fdata' object containing the derivative values. For 2D data: a list with components `ds`, `dt`, and `dsdt`, each an 'fdata' object containing the respective partial derivative.

Examples

```
# Create smooth curves
t <- seq(0, 2*pi, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) X[i, ] <- sin(t + i/5)
fd <- fdata(X, argvals = t)

# First derivative (should be approximately cos)
fd_deriv <- deriv(fd, nderiv = 1)
```

`detect.peaks` *Detect Peaks in Functional Data*

Description

Detects local maxima (peaks) in functional data using derivative zero-crossings. Returns peak times, values, and prominence measures.

Usage

```
detect.peaks(
  fdataobj,
  min_distance = NULL,
  min_prominence = NULL,
  smooth_first = FALSE,
  smooth_nbasis = NULL,
  detrend_method = c("none", "linear", "auto")
)
```

Arguments

`fdataobj`	An fdata object.
`min_distance`	Minimum time between peaks. Default: NULL (no constraint).
`min_prominence`	Minimum prominence for a peak (0-1 scale). Peaks with lower prominence are filtered out. Default: NULL (no filter).
`smooth_first`	Logical. If TRUE, apply Fourier basis smoothing before peak detection. Recommended for noisy data. Default: FALSE.
`smooth_nbasis`	Number of Fourier basis functions for smoothing. If NULL and smooth_first=TRUE, uses GCV to automatically select optimal nbasis (range 5-25). Default: NULL (auto).
`detrend_method`	Detrending method to apply before peak detection: **"none"** No detrending (default) **"linear"** Remove linear trend **"auto"** Automatic AIC-based selection of detrending method

Details

Peak prominence measures how much a peak stands out from its surroundings. It is computed as the height difference between the peak and the highest of the two minimum values on either side, normalized by the data range.

Fourier basis smoothing is ideal for seasonal signals because it naturally captures periodic patterns without introducing boundary artifacts.

For data with trends, use detrend_method to remove the trend before detecting peaks. This prevents the trend from affecting peak prominence calculations.

Value

A list with components:

peaks List of data frames, one per curve, with columns: time, value, prominence

inter_peak_distances List of numeric vectors with distances between consecutive peaks

mean_period Mean inter-peak distance across all curves (estimates period)

Examples

```
# Generate data with clear peaks
t <- seq(0, 10, length.out = 200)
X <- matrix(sin(2 * pi * t / 2), nrow = 1)
fd <- fdata(X, argvals = t)

# Detect peaks
peaks <- detect.peaks(fd, min_distance = 1.5)
print(peaks$mean_period)  # Should be close to 2

# With automatic Fourier smoothing (GCV selects nbasis)
peaks_smooth <- detect.peaks(fd, min_distance = 1.5, smooth_first = TRUE)

# With detrending for trending data
X_trend <- matrix(2 + 0.5 * t + sin(2 * pi * t / 2), nrow = 1)
fd_trend <- fdata(X_trend, argvals = t)
peaks_det <- detect.peaks(fd_trend, detrend_method = "linear")
```

`detect.period` *Seasonal Analysis Functions for Functional Data*

Description

Functions for analyzing seasonal patterns in functional data including period estimation, peak detection, seasonal strength measurement, and detection of seasonality changes. Detect Period with Multiple Methods

Usage

```
detect.period(
  fdataobj,
  method = c("sazed", "autoperiod", "cfd", "fft", "acf"),
  ...
)
```

Arguments

`fdataobj`	An fdata object.
`method`	Detection method to use: **"sazed"** SAZED ensemble (default) - parameter-free, most robust **"autoperiod"** Autoperiod - hybrid FFT + ACF with gradient ascent **"cfd"** CFDAutoperiod - differencing + clustering + ACF validation **"fft"** Simple FFT periodogram peak **"acf"** Simple ACF peak detection
`...`	Additional arguments passed to the underlying method: **tolerance** For SAZED: relative tolerance for voting (default 0.1)

n_candidates For Autoperiod: max FFT peaks to consider (default 5)
gradient_steps For Autoperiod: refinement steps (default 10)
cluster_tolerance For CFD: clustering tolerance (default 0.1)
min_cluster_size For CFD: minimum cluster size (default 1)
max_lag For ACF: maximum lag to search
detrend_method For FFT/ACF: "none", "linear", or "auto"

Details

Unified interface for period detection that dispatches to specialized algorithms based on the chosen method. Provides a single entry point for all period detection functionality.

Method selection guidance:

- **sazed**: Best general-purpose choice. Combines 5 methods and uses voting - robust across signal types with no tuning needed.
- **autoperiod**: Good when you need candidate details and gradient-refined estimates. Slightly more precise than SAZED.
- **cfd**: Best for signals with strong polynomial trends. Also detects multiple concurrent periods.
- **fft**: Fastest, but sensitive to noise and trends.
- **acf**: Simple but effective for clean periodic signals.

Value

A period detection result. The exact class depends on the method:

sazed_result For SAZED method

autoperiod_result For Autoperiod method

cfd_autoperiod_result For CFD method

period_estimate For FFT and ACF methods

See Also

sazed, autoperiod, cfd.autoperiod, estimate.period, detect.periods

Examples

```
# Generate seasonal data
t <- seq(0, 20, length.out = 400)
X <- matrix(sin(2 * pi * t / 2) + 0.1 * rnorm(400), nrow = 1)
fd <- fdata(X, argvals = t)

# Default (SAZED) - most robust
result <- detect.period(fd)
print(result$period)

# Autoperiod with custom settings
result <- detect.period(fd, method = "autoperiod", n_candidates = 10)
```

```
# CFDAutoperiod for trended data
X_trend <- matrix(0.3 * t + sin(2 * pi * t / 2), nrow = 1)
fd_trend <- fdata(X_trend, argvals = t)
result <- detect.period(fd_trend, method = "cfd")

# Simple FFT for speed
result <- detect.period(fd, method = "fft")
```

detect.periods *Detect Multiple Concurrent Periods*

Description

Detects multiple periodicities in functional data using iterative residual subtraction. At each iteration, the dominant period is detected using FFT, its sinusoidal component is subtracted, and the process repeats on the residual until stopping criteria are met.

Usage

```
detect.periods(
  fdataobj,
  max_periods = 3,
  min_confidence = 0.5,
  min_strength = 0.2,
  detrend_method = c("auto", "none", "linear")
)
```

Arguments

`fdataobj`	An fdata object.
`max_periods`	Maximum number of periods to detect. Default: 3.
`min_confidence`	Minimum FFT confidence to continue detection. Default: 0.5.
`min_strength`	Minimum seasonal strength to continue detection. Default: 0.2.
`detrend_method`	Detrending method to apply before period detection: **"auto"** Automatic AIC-based selection of detrending method (default) **"none"** No detrending **"linear"** Remove linear trend

Details

The function uses two stopping criteria:

- FFT confidence: How prominent the dominant frequency is
- Seasonal strength: How much variance is explained by the periodicity

Both must exceed their thresholds for detection to continue. Higher thresholds result in fewer (but more reliable) detected periods.

Periods are detected in order of amplitude (FFT power), not period length. A weak yearly cycle will be detected after a strong weekly cycle.

Trends can interfere with period detection. The default "auto" detrending automatically selects an appropriate method to remove trends.

Value

A list with components:

periods Numeric vector of detected periods

confidence FFT confidence for each period

strength Seasonal strength for each period

amplitude Amplitude of the sinusoidal component

phase Phase of the sinusoidal component (radians)

n_periods Number of periods detected

See Also

`estimate.period` for single period estimation, `detrend` for standalone detrending

Examples

```
# Signal with two periods: 2 and 7
t <- seq(0, 20, length.out = 400)
X <- sin(2 * pi * t / 2) + 0.6 * sin(2 * pi * t / 7)
fd <- fdata(matrix(X, nrow = 1), argvals = t)

# Detect multiple periods
result <- detect.periods(fd, max_periods = 3)
print(result$periods)  # Should find approximately 2 and 7
print(result$n_periods)  # Should be 2
```

`detect.seasonality.changes`
Detect Changes in Seasonality

Description

Detects points in time where seasonality starts (onset) or stops (cessation) by monitoring time-varying seasonal strength.

Usage

```
detect.seasonality.changes(
  fdataobj,
  period,
  threshold = 0.3,
  window_size = NULL,
  min_duration = NULL
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Known or estimated period.
`threshold`	Seasonal strength threshold for classification (0-1). Above threshold = seasonal, below = non-seasonal. Default: 0.3.
`window_size`	Width of sliding window for strength estimation. Default: 2 * period.
`min_duration`	Minimum duration to confirm a change. Prevents detection of spurious short-term fluctuations. Default: period.

Value

A list with components:

change_points Data frame with columns: time, type ("onset" or "cessation"), strength_before, strength_after

strength_curve Time-varying seasonal strength used for detection

Examples

```
# Signal that starts non-seasonal, becomes seasonal, then stops
t <- seq(0, 30, length.out = 600)
X <- ifelse(t < 10, rnorm(sum(t < 10), sd = 0.3),
            ifelse(t < 20, sin(2 * pi * t[t >= 10 & t < 20] / 2),
                   rnorm(sum(t >= 20), sd = 0.3)))
X <- matrix(X, nrow = 1)
fd <- fdata(X, argvals = t)

# Detect changes
changes <- detect.seasonality.changes(fd, period = 2)
print(changes$change_points)  # Should show onset ~10, cessation ~20
```

detect.seasonality.changes.auto

Detect Seasonality Changes with Automatic Threshold

Description

Detects points where seasonality starts or stops, using automatic threshold selection instead of a fixed value.

Usage

```
detect.seasonality.changes.auto(
  fdataobj,
  period,
  threshold_method = "otsu",
  threshold_value = NULL,
  window_size = NULL,
  min_duration = NULL
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Known seasonal period.
`threshold_method`	Method for threshold selection: **"fixed"** Use threshold_value as fixed threshold **"percentile"** Use threshold_value as percentile of strength distribution **"otsu"** Use Otsu's method for bimodal separation (default)
`threshold_value`	Value for "fixed" or "percentile" methods.
`window_size`	Width of sliding window for strength estimation. Default: 2 * period.
`min_duration`	Minimum duration to confirm a change. Default: period.

Details

Otsu's method automatically finds the optimal threshold for separating seasonal from non-seasonal regions based on the strength distribution. This is particularly useful when you don't know the appropriate threshold for your data.

Value

A list with components:

change_points Data frame with time, type, strength_before, strength_after

strength_curve Time-varying seasonal strength used for detection

computed_threshold The threshold that was computed/used

Examples

```
# Signal that transitions from seasonal to non-seasonal
t <- seq(0, 20, length.out = 400)
X <- ifelse(t < 10, sin(2 * pi * t / 2), rnorm(sum(t >= 10), sd = 0.3))
X <- matrix(X, nrow = 1)
fd <- fdata(X, argvals = t)

# Detect changes with Otsu threshold
changes <- detect.seasonality.changes.auto(fd, period = 2)
print(changes$computed_threshold)
```

`detect_amplitude_modulation`
Detect Amplitude Modulation in Seasonal Time Series

Description

Detects whether the amplitude of a seasonal pattern changes over time (amplitude modulation). Uses either Hilbert transform or wavelet analysis to extract the time-varying amplitude envelope.

Usage

```
detect_amplitude_modulation(
  fdataobj,
  period,
  method = c("hilbert", "wavelet"),
  modulation_threshold = 0.15,
  seasonality_threshold = 0.3
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Seasonal period in argvals units.
`method`	Method for amplitude extraction: "hilbert" (default) or "wavelet".
`modulation_threshold`	Coefficient of variation threshold for detecting modulation. Default: 0.15 (i.e., CV > 15% indicates modulation).
`seasonality_threshold`	Strength threshold for seasonality detection. Default: 0.3.

Details

The function first checks if seasonality is present using the spectral method (which is robust to amplitude modulation). If seasonality is detected, it extracts the amplitude envelope and analyzes its variation.

Hilbert method: Uses Hilbert transform to compute the analytic signal and extract instantaneous amplitude. Fast but assumes narrowband signals.

Wavelet method: Uses Morlet wavelet transform at the target period. More robust to noise and non-stationarity.

Modulation types:

- `stable`: Constant amplitude (modulation_score < threshold)
- `emerging`: Amplitude increases over time (amplitude_trend > 0.3)
- `fading`: Amplitude decreases over time (amplitude_trend < -0.3)
- `oscillating`: Amplitude varies but no clear trend
- `non_seasonal`: No seasonality detected

Value

A list with components:

is_seasonal Logical, whether seasonality is detected

seasonal_strength Overall seasonal strength (spectral method)

has_modulation Logical, whether amplitude modulation is detected

modulation_type Character: "stable", "emerging", "fading", "oscillating", or "non_seasonal"

modulation_score Coefficient of variation of amplitude (0 = stable)

amplitude_trend Trend in amplitude (-1 to 1): negative = fading, positive = emerging

amplitude_curve Time-varying amplitude (fdata object)

time_points Time points for the amplitude curve

Examples

```
# Generate data with emerging seasonality
t <- seq(0, 1, length.out = 200)
amplitude <- 0.2 + 0.8 * t  # Amplitude grows over time
X <- matrix(amplitude * sin(2 * pi * t / 0.2), nrow = 1)
fd <- fdata(X, argvals = t)

result <- detect_amplitude_modulation(fd, period = 0.2)
print(result$modulation_type)  # "emerging"
print(result$amplitude_trend)  # Positive value

# Compare Hilbert vs Wavelet methods
result_hilbert <- detect_amplitude_modulation(fd, period = 0.2, method = "hilbert")
result_wavelet <- detect_amplitude_modulation(fd, period = 0.2, method = "wavelet")
```

detrend *Remove Trend from Functional Data*

Description

Removes trend from functional data using various methods. This is useful for preprocessing data before seasonal analysis when the data has a significant trend component.

Usage

```
detrend(
  fdataobj,
  method = c("linear", "polynomial", "diff1", "diff2", "loess", "auto"),
  degree = 2,
  bandwidth = 0.3,
  return_trend = FALSE
)
```

Arguments

`fdataobj`	An fdata object.
`method`	Detrending method: **"linear"** Least squares linear fit (default) **"polynomial"** Polynomial regression of specified degree **"diff1"** First-order differencing **"diff2"** Second-order differencing **"loess"** Local polynomial regression (LOESS) **"auto"** Automatic selection via AIC
`degree`	Polynomial degree for "polynomial" method. Default: 2.
`bandwidth`	Bandwidth as fraction of data range for "loess" method. Default: 0.3.
`return_trend`	Logical. If TRUE, return both trend and detrended data. Default: FALSE.

Details

For series with polynomial trends, "linear" or "polynomial" methods are appropriate. For more complex trends, "loess" provides flexibility. The "auto" method compares linear, polynomial (degree 2 and 3), and LOESS, selecting the method with lowest AIC.

Differencing methods ("diff1", "diff2") reduce the series length by 1 or 2 points respectively. The resulting fdata has correspondingly shorter argvals.

Value

If return_trend = FALSE, an fdata object with detrended data. If return_trend = TRUE, a list with components:

detrended fdata object with detrended data

trend fdata object with estimated trend

method Method used for detrending

rss Residual sum of squares per curve

See Also

decompose for full seasonal decomposition

Examples

```
# Generate data with linear trend and seasonal component
t <- seq(0, 10, length.out = 200)
X <- matrix(2 + 0.5 * t + sin(2 * pi * t / 2), nrow = 1)
fd <- fdata(X, argvals = t)

# Detrend with linear method
fd_detrended <- detrend(fd, method = "linear")

# Now estimate period on detrended data
period <- estimate.period(fd_detrended)
print(period$period)  # Should be close to 2

# Get both trend and detrended data
result <- detrend(fd, method = "linear", return_trend = TRUE)
# plot(result$trend)  # Shows the linear trend
```

---

df_to_fdata2d *Convert DataFrame to 2D functional data*

Description

Converts a data frame in long format to a 2D fdata object (surfaces). The expected format is: one identifier column, one column for the s-dimension index, and multiple columns for the t-dimension values.

Usage

```
df_to_fdata2d(
  df,
  id_col = 1,
  s_col = 2,
  t_cols = NULL,
  names = NULL,
  metadata = NULL
)
```

Arguments

`df`	A data frame with the structure described below.
`id_col`	Name or index of the identifier column (default: 1).
`s_col`	Name or index of the s-dimension column (default: 2).
`t_cols`	Names or indices of the t-dimension value columns. If NULL (default), uses all columns after `s_col`.
`names`	Optional list with 'main', 'xlab', 'ylab', 'zlab' for labels.
`metadata`	Optional data.frame with additional covariates (one row per surface). If metadata has an "id" column or non-default row names, they must match the surface identifiers from `id_col`.

Details

The expected data frame structure is:

- Column 1 (id_col): Surface identifier (e.g., "surface_1", "surface_2")
- Column 2 (s_col): Index for the s-dimension (row index of surface)
- Columns 3+ (t_cols): Values for each t-dimension point (columns of surface)

Each unique identifier represents one surface. For each surface, there should be m1 rows (one per s-value), and m2 t-columns, resulting in an m1 x m2 surface.

Value

An object of class 'fdata' with 2D functional data.

Examples

```
# Create example data frame
df <- data.frame(
  id = rep(c("surf1", "surf2"), each = 5),
  s = rep(1:5, 2),
  t1 = rnorm(10),
  t2 = rnorm(10),
  t3 = rnorm(10)
)
fd <- df_to_fdata2d(df)
print(fd)

# With metadata
meta <- data.frame(group = c("A", "B"), value = c(1.5, 2.3))
fd <- df_to_fdata2d(df, metadata = meta)
```

eFun	*Generate Eigenfunction Basis*

Description

Evaluates orthonormal eigenfunction bases at specified argument values. These eigenfunctions can be used for Karhunen-Loeve simulation.

Usage

```
eFun(argvals, M, type = c("Fourier", "Poly", "PolyHigh", "Wiener"))
```

Arguments

`argvals`	Numeric vector of evaluation points in [0, 1].
`M`	Number of eigenfunctions to generate.
`type`	Character. Type of eigenfunction system: **Fourier** Fourier basis: 1, sqrt(2)*sin(2*pi*k*t), sqrt(2)*cos(2*pi*k*t) **Poly** Orthonormal Legendre polynomials of degrees 0, 1, ..., M-1 **PolyHigh** Orthonormal Legendre polynomials starting at degree 2 **Wiener** Wiener process eigenfunctions: sqrt(2)*sin((k-0.5)*pi*t)

Details

The eigenfunctions are orthonormal with respect to the L2 inner product: `integral(phi_j * phi_k) = 1` if `j == k`, `0` otherwise.

Fourier Suitable for periodic data. First function is constant.

Poly Orthonormalized Legendre polynomials. Good for smooth data.

PolyHigh Legendre polynomials starting at degree 2, useful when linear and constant components are handled separately.

Wiener Eigenfunctions of the Brownian motion covariance. Eigenvalues decay as 1/((k-0.5)*pi)^2.

Value

A matrix of dimension `length(argvals)` x `M` containing the eigenfunction values. Each column is an eigenfunction, normalized to have unit L2 norm on [0, 1].

See Also

`eVal`, `simFunData`

Examples

```
t <- seq(0, 1, length.out = 100)

# Generate Fourier basis
phi_fourier <- eFun(t, M = 5, type = "Fourier")
matplot(t, phi_fourier, type = "l", lty = 1,
        main = "Fourier Eigenfunctions", ylab = expression(phi(t)))

# Generate Wiener eigenfunctions
phi_wiener <- eFun(t, M = 5, type = "Wiener")
matplot(t, phi_wiener, type = "l", lty = 1,
        main = "Wiener Eigenfunctions", ylab = expression(phi(t)))

# Check orthonormality (should be identity matrix)
dt <- diff(t)[1]
gram <- t(phi_fourier) %*% phi_fourier * dt
round(gram, 2)
```

---

estimate.period *Estimate Seasonal Period using FFT*

Description

Estimates the dominant period in functional data using Fast Fourier Transform and periodogram analysis.

Usage

```
estimate.period(
  fdataobj,
  method = c("fft", "acf"),
  max_lag = NULL,
  detrend_method = c("none", "linear", "auto")
)
```

Arguments

fdataobj	An fdata object.
method	Method for period estimation: "fft" (Fast Fourier Transform, default) or "acf" (autocorrelation function).
max_lag	Maximum lag for ACF method. Default: half the series length.
detrend_method	Detrending method to apply before period estimation: **"none"** No detrending (default) **"linear"** Remove linear trend **"auto"** Automatic AIC-based selection of detrending method

Details

The function computes the periodogram of the mean curve and finds the frequency with maximum power. The confidence measure indicates how pronounced the dominant frequency is relative to the background.

For data with trends, the detrend_method parameter can significantly improve period estimation accuracy. Strong trends can mask the true seasonal period.

Value

A list with components:

period Estimated period

frequency Dominant frequency (1/period)

power Power at the dominant frequency

confidence Confidence measure (ratio of peak power to mean power)

Examples

```
# Generate seasonal data with period = 2
t <- seq(0, 10, length.out = 200)
X <- matrix(sin(2 * pi * t / 2) + rnorm(200, sd = 0.1), nrow = 1)
fd <- fdata(X, argvals = t)

# Estimate period
result <- estimate.period(fd, method = "fft")
print(result$period)  # Should be close to 2

# With trend - detrending improves estimation
X_trend <- matrix(2 + 0.5 * t + sin(2 * pi * t / 2), nrow = 1)
fd_trend <- fdata(X_trend, argvals = t)
result <- estimate.period(fd_trend, detrend_method = "linear")
```

---

eVal *Generate Eigenvalue Sequence*

Description

Generates eigenvalue sequences with different decay patterns. These control the variance contribution of each mode in Karhunen-Loeve simulation.

Usage

```
eVal(M, type = c("linear", "exponential", "wiener"))
```

Arguments

`M`	Number of eigenvalues to generate.
`type`	Character. Type of eigenvalue decay: **linear** lambda_k = 1/k for k = 1, ..., M **exponential** lambda_k = exp(-k) for k = 1, ..., M **wiener** lambda_k = 1/((k - 0.5)*pi)^2, the Wiener process eigenvalues

Details

The eigenvalues control how much variance each eigenfunction contributes to the simulated curves:

linear Slow decay, higher modes contribute more variation. Produces rougher curves.

exponential Fast decay, higher modes contribute very little. Produces smoother curves.

wiener Specific decay matching Brownian motion. Use with Wiener eigenfunctions for true Brownian motion simulation.

Value

A numeric vector of length M containing the eigenvalues in decreasing order.

See Also

`eFun`, `simFunData`

Examples

```
# Compare decay patterns
lambda_lin <- eVal(20, "linear")
lambda_exp <- eVal(20, "exponential")
lambda_wie <- eVal(20, "wiener")

plot(1:20, lambda_lin, type = "b", log = "y", ylim = c(1e-10, 1),
     main = "Eigenvalue Decay Patterns", xlab = "k", ylab = expression(lambda[k]))
lines(1:20, lambda_exp, col = "red", type = "b")
lines(1:20, lambda_wie, col = "blue", type = "b")
legend("topright", c("Linear", "Exponential", "Wiener"),
       col = c("black", "red", "blue"), lty = 1, pch = 1)
```

`fdata` *Create a functional data object*

Description

Creates an fdata object for 1D functional data (curves) or 2D functional data (surfaces). For 2D data, the internal storage uses a flattened matrix format `[n, m1*m2]` where each row represents a surface stored in row-major order.

Usage

```
fdata(
  mdata,
  argvals = NULL,
  rangeval = NULL,
  names = NULL,
  fdata2d = FALSE,
  id = NULL,
  metadata = NULL
)
```

Arguments

`mdata`	Input data. Can be: • For 1D: A matrix `[n, m]` where n is number of curves, m is number of evaluation points • For 2D: A 3D array `[n, m1, m2]` where n is number of surfaces, m1 x m2 is the grid size. Automatically detected and converted to flattened storage. • For 2D: A matrix `[n, m1*m2]` (already flattened) with argvals specifying grid dimensions • For 2D: A single surface matrix `[m1, m2]` with argvals specifying grid dimensions
`argvals`	Evaluation points. For 1D: a numeric vector. For 2D: a list with two numeric vectors specifying the s and t coordinates.
`rangeval`	Range of the argument values. For 1D: a numeric vector of length 2. For 2D: a list with two numeric vectors of length 2.
`names`	List with components 'main', 'xlab', 'ylab' for plot titles. For 2D, also 'zlab' for the surface value label.
`fdata2d`	Logical. If TRUE, create 2D functional data (surface). Automatically set to TRUE if mdata is a 3D array.
`id`	Optional character vector of identifiers for each observation. If NULL, uses row names of mdata or generates "obs_1", "obs_2", etc.
`metadata`	Optional data.frame with additional covariates (one row per observation). If metadata has an "id" column or non-default row names, they must match the `id` parameter.

Details

For 2D functional data, surfaces are stored internally as a flattened matrix where each row is a surface in row-major order. To extract a single surface as a matrix, use subsetting with `drop = TRUE` or reshape manually:

```
# Extract surface i as matrix
surface_i <- fd[i, drop = TRUE]
# Or manually:
surface_i <- matrix(fd$data[i, ], nrow = fd$dims[1], ncol = fd$dims[2])
```

Value

An object of class 'fdata' containing:

data The data matrix. For 2D: flattened `[n, m1*m2]` format

argvals Evaluation points

rangeval Range of arguments

names Plot labels

fdata2d Logical indicating if 2D

dims For 2D only: `c(m1, m2)` grid dimensions

id Character vector of observation identifiers

metadata Data frame of additional covariates (or NULL)

Examples

```
# Create 1D functional data (curves)
x <- matrix(rnorm(100), nrow = 10, ncol = 10)
fd <- fdata(x, argvals = seq(0, 1, length.out = 10))

# Create with identifiers and metadata
meta <- data.frame(group = rep(c("A", "B"), 5), endpoint = rnorm(10))
fd <- fdata(x, id = paste0("patient_", 1:10), metadata = meta)

# Access metadata
fd$id
fd$metadata$group

# Create 2D functional data from 3D array [n, m1, m2]
surfaces <- array(rnorm(500), dim = c(5, 10, 10))
fd2d <- fdata(surfaces)

# Access individual surface as matrix
surface_1 <- fd2d[1, drop = TRUE]
```

`fdata.bootstrap` *Bootstrap Functional Data*

Description

Generate bootstrap samples from functional data. Supports naive bootstrap (resampling curves with replacement) and smooth bootstrap (adding noise based on estimated covariance structure).

Usage

```
fdata.bootstrap(
  fdataobj,
  n.boot = 200,
  method = c("naive", "smooth"),
  variance = NULL,
  seed = NULL
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`n.boot`	Number of bootstrap replications (default 200).
`method`	Bootstrap method: "naive" for resampling with replacement, "smooth" for adding Gaussian noise (default "naive").
`variance`	For method="smooth", the variance of the added noise. If NULL, estimated from the data.
`seed`	Optional seed for reproducibility.

Value

A list of class 'fdata.bootstrap' with components:

boot.samples List of n.boot fdata objects, each a bootstrap sample

original The original fdata object

method The bootstrap method used

n.boot Number of bootstrap replications

Examples

```
# Create functional data
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)

# Naive bootstrap
boot_naive <- fdata.bootstrap(fd, n.boot = 100, method = "naive")

# Smooth bootstrap
boot_smooth <- fdata.bootstrap(fd, n.boot = 100, method = "smooth")
```

fdata.bootstrap.ci — *Bootstrap Confidence Intervals for Functional Statistics*

Description

Compute bootstrap confidence intervals for functional statistics such as the mean function, depth values, or regression coefficients.

Usage

```
fdata.bootstrap.ci(
  fdataobj,
  statistic,
  n.boot = 200,
  alpha = 0.05,
  method = c("percentile", "basic", "normal"),
  seed = NULL
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`statistic`	A function that computes the statistic of interest. Must take an fdata object and return a numeric vector.
`n.boot`	Number of bootstrap replications (default 200).
`alpha`	Significance level for confidence intervals (default 0.05 for 95 percent CI).
`method`	CI method: "percentile" for simple percentile method, "basic" for basic bootstrap, "normal" for normal approximation (default "percentile").
`seed`	Optional seed for reproducibility.

Value

A list of class 'fdata.bootstrap.ci' with components:

estimate The statistic computed on the original data

ci.lower Lower confidence bound

ci.upper Upper confidence bound

boot.stats Matrix of bootstrap statistics (n.boot x length(statistic))

alpha The significance level used

method The CI method used

Examples

```
# Create functional data
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)

# Bootstrap CI for the mean function (returns numeric vector)
ci_mean <- fdata.bootstrap.ci(fd,
  statistic = function(x) as.numeric(mean(x)$data),
  n.boot = 100)

# Bootstrap CI for depth values
ci_depth <- fdata.bootstrap.ci(fd,
  statistic = function(x) depth.FM(x),
  n.boot = 100)
```

`fdata.cen` *Center functional data*

Description

Subtract the mean function from each curve.

Usage

```
fdata.cen(fdataobj)
```

Arguments

`fdataobj`	An object of class 'fdata'.

Value

A centered 'fdata' object.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
fd_centered <- fdata.cen(fd)
```

`fdata2basis` *Convert Functional Data to Basis Coefficients*

Description

Project functional data onto a basis system and return coefficients. Supports B-spline and Fourier basis. Works with both regular `fdata` and irregular `irregFdata` objects.

Usage

```
fdata2basis(x, nbasis = 10, type = c("bspline", "fourier"), ...)

## S3 method for class 'fdata'
fdata2basis(x, nbasis = 10, type = c("bspline", "fourier"), ...)

## S3 method for class 'irregFdata'
fdata2basis(x, nbasis = 10, type = c("bspline", "fourier"), ...)
```

Arguments

`x`	An object of class 'fdata' or 'irregFdata'.
`nbasis`	Number of basis functions (default 10).
`type`	Type of basis: "bspline" (default) or "fourier".
`...`	Additional arguments (currently unused).

Details

For regular `fdata` objects, all curves are projected onto the same basis evaluated at the common grid points.

For irregular `irregFdata` objects, each curve is individually fitted to the basis using least squares at its own observation points. This is the preferred approach for sparse/irregularly sampled data as it avoids interpolation artifacts.

Value

A matrix of coefficients (n x nbasis).

Examples

```
# Regular fdata
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)
coefs <- fdata2basis(fd, nbasis = 10, type = "bspline")

# Irregular fdata (sparsified)
```

```
ifd <- sparsify(fd, minObs = 10, maxObs = 20, seed = 42)
coefs_irreg <- fdata2basis(ifd, nbasis = 10, type = "bspline")
```

`fdata2basis_2d` *Convert 2D Functional Data to Tensor Product Basis Coefficients*

Description

Projects 2D functional data (surfaces) onto a tensor product basis, which is the Kronecker product of two 1D bases.

Usage

```
fdata2basis_2d(
  fdataobj,
  nbasis.s = 10,
  nbasis.t = 10,
  type = c("bspline", "fourier")
)
```

Arguments

`fdataobj`	A 2D fdata object (surfaces).
`nbasis.s`	Number of basis functions in the s (first) direction.
`nbasis.t`	Number of basis functions in the t (second) direction.
`type`	Basis type: "bspline" (default) or "fourier".

Details

The tensor product basis is defined as:

$$B_{2d}(s,t) = B_s(s) \otimes B_t(t)$$

Value

A matrix of coefficients `[n x (nbasis.s * nbasis.t)]`.

Examples

```
# Create 2D surface data
s <- seq(0, 1, length.out = 20)
t <- seq(0, 1, length.out = 20)
surface <- outer(sin(2*pi*s), cos(2*pi*t))
fd2d <- fdata(array(surface, dim = c(1, 20, 20)))

# Project to tensor product basis
coefs <- fdata2basis_2d(fd2d, nbasis.s = 7, nbasis.t = 7, type = "fourier")
```

fdata2basis_cv *Cross-Validation for Basis Function Number Selection*

Description

Selects the optimal number of basis functions using k-fold cross-validation or generalized cross-validation.

Usage

```
fdata2basis_cv(
  fdataobj,
  nbasis.range = 4:20,
  type = c("bspline", "fourier"),
  criterion = c("GCV", "CV", "AIC", "BIC"),
  kfold = 10,
  lambda = 0
)
```

Arguments

`fdataobj`	An fdata object.
`nbasis.range`	Vector of nbasis values to evaluate (default: 4:20).
`type`	Basis type: "bspline" (default) or "fourier".
`criterion`	Selection criterion: "GCV" (default), "CV", "AIC", or "BIC".
`kfold`	Number of folds for k-fold CV (default 10). Ignored if criterion is "GCV", "AIC", or "BIC".
`lambda`	Smoothing parameter (default 0).

Value

A list with:

optimal.nbasis Optimal number of basis functions

scores Score for each nbasis value

nbasis.range The tested nbasis values

criterion The criterion used

fitted fdata object fitted with optimal nbasis

Examples

```
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(4 * pi * t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)

# Find optimal nbasis
cv_result <- fdata2basis_cv(fd, nbasis.range = 5:15, type = "fourier")
print(cv_result$optimal.nbasis)
```

---

fdata2fd *Convert Functional Data to fd class*

Description

Converts an fdata object to an fd object from the fda package. Requires the fda package to be installed.

Usage

```
fdata2fd(fdataobj, nbasis = 10, type = c("bspline", "fourier"))
```

Arguments

`fdataobj`	An object of class 'fdata'.
`nbasis`	Number of basis functions (default 10).
`type`	Type of basis: "bspline" (default) or "fourier".

Value

An object of class 'fd' from the fda package.

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)
fd_obj <- fdata2fd(fd, nbasis = 10)
```

fdata2pc *Convert Functional Data to Principal Component Scores*

Description

Performs functional PCA and returns principal component scores for functional data. Uses SVD on centered data.

Usage

```
fdata2pc(fdataobj, ncomp = 2, lambda = 0, norm = TRUE)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`ncomp`	Number of principal components to extract (default 2).
`lambda`	Regularization parameter (default 0, not currently used).
`norm`	Logical. If TRUE (default), normalize the scores.

Value

A list with components:

d Singular values (proportional to sqrt of eigenvalues)

rotation fdata object containing PC loadings

x Matrix of PC scores (n x ncomp)

mean Mean function (numeric vector)

fdataobj.cen Centered fdata object

call The function call

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)
pc <- fdata2pc(fd, ncomp = 3)
```

`fdata2pls` *Convert Functional Data to PLS Scores*

Description

Performs Partial Least Squares regression and returns component scores for functional data using the NIPALS algorithm.

Usage

```
fdata2pls(fdataobj, y, ncomp = 2, lambda = 0, norm = TRUE)
```

Arguments

fdataobj	An object of class 'fdata'.
y	Response vector (numeric).
ncomp	Number of PLS components to extract (default 2).
lambda	Regularization parameter (default 0, not currently used).
norm	Logical. If TRUE (default), normalize the scores.

Value

A list with components:

weights Matrix of PLS weights (m x ncomp)

scores Matrix of PLS scores (n x ncomp)

loadings Matrix of PLS loadings (m x ncomp)

call The function call

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
y <- rowMeans(X) + rnorm(20, sd = 0.1)
fd <- fdata(X, argvals = t)
pls <- fdata2pls(fd, y, ncomp = 3)
```

`fequiv.test` *Functional Equivalence Test (TOST)*

Description

Tests whether two functional means are equivalent within an equivalence margin `delta`, based on the supremum norm. Uses the approach of Dette & Kokot (2021) with a simultaneous confidence band.

Usage

```
fequiv.test(
  fdataobj1,
  fdataobj2 = NULL,
  delta,
  mu0 = NULL,
  n.boot = 1000,
  alpha = 0.05,
  method = c("multiplier", "percentile"),
  seed = NULL
)
```

Arguments

`fdataobj1`	An object of class `fdata` (first sample).
`fdataobj2`	An object of class `fdata` (second sample), or NULL for a one-sample test.
`delta`	Equivalence margin (positive scalar). The test checks whether the sup-norm of the mean difference is less than `delta`.
`mu0`	Hypothesized mean function for one-sample test. A numeric vector, a single-row `fdata`, or NULL (defaults to zero).
`n.boot`	Number of bootstrap replicates (default 1000).
`alpha`	Significance level (default 0.05). The SCB has coverage 1-2*alpha.
`method`	Bootstrap method: `"multiplier"` (Gaussian multiplier, default) or `"percentile"` (resampling).
`seed`	Optional random seed for reproducibility.

Value

An object of class `fequiv.test` with components:

statistic Observed sup-norm of the mean difference

delta Equivalence margin used

critical.value Bootstrap critical value for the SCB

scb.lower Lower bound of simultaneous confidence band

scb.upper Upper bound of simultaneous confidence band

diff.mean Estimated mean difference curve (numeric vector)

reject Logical; TRUE if equivalence is declared

p.value Bootstrap p-value

alpha Significance level used

method Bootstrap method used

argvals Argument values (time grid)

n1 Sample size of first group

n2 Sample size of second group (NA for one-sample)

boot.stats Vector of bootstrap sup-norm statistics

data.name Description of input data

Hypotheses

H0 sup_t |mu1(t) - mu2(t)| >= delta (NOT equivalent)

H1 sup_t |mu1(t) - mu2(t)| < delta (equivalent)

Equivalence is declared when the entire (1-2*alpha) simultaneous confidence band for the mean difference lies within `[-delta, delta]`.

References

Dette, H. and Kokot, K. (2021). Detecting relevant differences in the covariance operators of functional time series. *Biometrika*, 108(4):895–913.

Examples

```
set.seed(42)
t_grid <- seq(0, 1, length.out = 50)
X1 <- fdata(matrix(rnorm(30 * 50), 30, 50), argvals = t_grid)
X2 <- fdata(matrix(rnorm(25 * 50, mean = 0.1), 25, 50), argvals = t_grid)

result <- fequiv.test(X1, X2, delta = 1, n.boot = 500)
print(result)
plot(result)
```

`flm.test` *Statistical Tests for Functional Data*

Description

Functions for hypothesis testing with functional data. Test for Functional Linear Model

Usage

```
flm.test(fdataobj, y, B = 500, ...)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`B`	Number of bootstrap samples for p-value computation.
`...`	Additional arguments.

Details

Tests the goodness-of-fit for a functional linear model using the projected Cramer-von Mises statistic.

Value

A list of class 'htest' with components:

statistic The test statistic

p.value Bootstrap p-value

method Name of the test

Examples

```
fd <- fdata(matrix(rnorm(200), 20, 10))
y <- rnorm(20)

test_result <- flm.test(fd, y, B = 100)
test_result$p.value
```

`fmean.test.fdata` *Test for Equality of Functional Means*

Description

Tests whether the mean function equals a specified value.

Usage

```
fmean.test.fdata(fdataobj, mu0 = NULL, B = 500, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`mu0`	Hypothesized mean function (vector). If NULL, tests against zero.
`B`	Number of bootstrap samples for p-value computation.
`...`	Additional arguments.

Value

A list of class 'htest' with components:

statistic The test statistic

p.value Bootstrap p-value

method Name of the test

Examples

```
fd <- fdata(matrix(rnorm(200), 20, 10))

test_result <- fmean.test.fdata(fd, B = 100)
test_result$p.value
```

`fregre.basis` *Functional Basis Regression*

Description

Fits a functional linear model using basis expansion (ridge regression). Uses the anofox-regression Rust backend for efficient L2-regularized regression.

Usage

```
fregre.basis(fdataobj, y, basis.x = NULL, basis.b = NULL, lambda = 0, ...)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`basis.x`	Basis for the functional covariate (currently ignored).
`basis.b`	Basis for the coefficient function (currently ignored).
`lambda`	Smoothing/regularization parameter (L2 penalty).
`...`	Additional arguments.

Value

A fitted regression object of class 'fregre.fd' with components:

`coefficients`	Beta coefficient function values
`intercept`	Intercept term
`fitted.values`	Fitted values
`residuals`	Residuals
`lambda`	Regularization parameter used
`r.squared`	R-squared (coefficient of determination)
`mean.X`	Mean of functional covariate (for prediction)
`mean.y`	Mean of response (for prediction)
`sr2`	Residual variance
`fdataobj`	Original functional data
`y`	Response vector
`call`	The function call

`fregre.basis.cv` *Cross-Validation for Functional Basis Regression*

Description

Performs k-fold cross-validation to select the optimal regularization parameter (lambda) for functional basis regression.

Usage

```
fregre.basis.cv(fdataobj, y, kfold = 10, lambda.range = NULL, seed = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`kfold`	Number of folds for cross-validation (default 10).
`lambda.range`	Range of lambda values to try. Default is 10^seq(-4, 4, length.out = 20).
`seed`	Random seed for fold assignment.
`...`	Additional arguments passed to fregre.basis.

Value

A list with components:

optimal.lambda Optimal regularization parameter
cv.errors Mean squared prediction error for each lambda
cv.se Standard error of cv.errors
model Fitted model with optimal lambda

`fregre.np` *Nonparametric Functional Regression*

Description

Fits a functional regression model using kernel smoothing (Nadaraya-Watson). Supports fixed bandwidth (h), or k-nearest neighbors with global (kNN.gCV) or local (kNN.lCV) cross-validation.

Usage

```
fregre.np(
  fdataobj,
  y,
  h = NULL,
  knn = NULL,
  type.S = c("S.NW", "kNN.gCV", "kNN.lCV"),
  Ker = "norm",
  metric = metric.lp,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`h`	Bandwidth parameter. If NULL and knn is NULL, computed automatically.
`knn`	Number of nearest neighbors to consider for bandwidth selection. Only used when `type.S` is "kNN.gCV" or "kNN.lCV".
`type.S`	Type of smoother: "S.NW" for Nadaraya-Watson with fixed h (default), "kNN.gCV" for k-NN with global CV (single k for all observations), "kNN.lCV" for k-NN with local CV (different k per observation).
`Ker`	Kernel type for smoothing. Default is "norm" (Gaussian).
`metric`	Distance metric function. Default is metric.lp.
`...`	Additional arguments passed to metric function.

Details

Three smoothing approaches are available:

Fixed bandwidth (type.S = "S.NW"): Uses a single bandwidth h for all predictions. If h is not provided, it is set to the median of non-zero pairwise distances.

k-NN Global CV (type.S = "kNN.gCV"): Selects a single optimal k for all observations using leave-one-out cross-validation. The bandwidth at each point is set to include k neighbors.

k-NN Local CV (type.S = "kNN.lCV"): Selects an optimal k_i for each observation i, allowing adaptive smoothing. Useful when the data has varying density across the functional space.

Value

A fitted regression object of class 'fregre.np' with components:

fitted.values Fitted values

residuals Residuals

h.opt Optimal/used bandwidth (for type.S = "S.NW")

knn Number of neighbors used (for kNN methods)

k.opt Optimal k value(s) - scalar for global, vector for local

type.S Type of smoother used

Ker Kernel type used

fdataobj Original functional data

y Response vector

mdist Distance matrix

sr2 Residual variance

metric Metric function used

call The function call

Examples

```
# Create functional data
t <- seq(0, 1, length.out = 50)
n <- 50
X <- matrix(0, n, 50)
for (i in 1:n) X[i, ] <- sin(2*pi*t) * i/n + rnorm(50, sd = 0.1)
y <- rowMeans(X) + rnorm(n, sd = 0.1)
fd <- fdata(X, argvals = t)

# Fixed bandwidth
fit1 <- fregre.np(fd, y, h = 0.5)

# k-NN with global CV
fit2 <- fregre.np(fd, y, type.S = "kNN.gCV", knn = 20)

# k-NN with local CV
fit3 <- fregre.np(fd, y, type.S = "kNN.lCV", knn = 20)
```

`fregre.np.cv` *Cross-Validation for Nonparametric Functional Regression*

Description

Performs k-fold cross-validation to select the optimal bandwidth parameter (h) for nonparametric functional regression.

Usage

```
fregre.np.cv(
  fdataobj,
  y,
  kfold = 10,
  h.range = NULL,
  metric = metric.lp,
  seed = NULL,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`kfold`	Number of folds for cross-validation (default 10).
`h.range`	Range of bandwidth values to try. If NULL, automatically determined from the distance matrix.
`metric`	Distance metric function. Default is metric.lp.
`seed`	Random seed for fold assignment.
`...`	Additional arguments passed to the metric function.

Value

A list with components:

optimal.h Optimal bandwidth parameter

cv.errors Mean squared prediction error for each h

cv.se Standard error of cv.errors

model Fitted model with optimal h

`fregre.np.multi` *Nonparametric Regression with Multiple Functional Predictors*

Description

Fits a nonparametric regression model with multiple functional predictors using a weighted combination of distance matrices.

Usage

```
fregre.np.multi(
  fdataobj.list,
  y,
  weights = NULL,
  h = NULL,
  knn = NULL,
  type.S = c("S.NW", "kNN.gCV", "kNN.lCV"),
  Ker = "norm",
  metric = metric.lp,
  cv.grid = NULL,
  cv.folds = 5,
  ...
)
```

Arguments

`fdataobj.list`	A list of fdata objects (functional predictors). All must have the same number of observations.
`y`	Response vector (scalar).
`weights`	Weights for combining distances. Can be: • NULL: Equal weights (1/p for each predictor) • Numeric vector: Fixed weights (will be normalized to sum to 1) • "cv": Cross-validate to find optimal weights
`h`	Bandwidth for Nadaraya-Watson kernel (optional).
`knn`	Maximum k for k-NN methods.
`type.S`	Smoother type: "S.NW", "kNN.gCV", or "kNN.lCV".
`Ker`	Kernel type (default "norm" for Gaussian).
`metric`	Distance metric function (default metric.lp).
`cv.grid`	Grid of weight values for CV (only used if weights = "cv"). Default is seq(0, 1, by = 0.1) for 2 predictors.
`cv.folds`	Number of folds for weight CV (default 5).
`...`	Additional arguments passed to metric function.

Value

An object of class 'fregre.np.multi' containing:

fdataobj.list List of functional predictors

y Response vector

weights Weights used (or optimized)

weights.cv CV results if weights = "cv"

fitted.values Fitted values

residuals Residuals

D.list List of distance matrices

D.combined Combined distance matrix

Examples

```
# Create two functional predictors
set.seed(42)
n <- 50
m <- 30
t_grid <- seq(0, 1, length.out = m)

X1 <- matrix(0, n, m)
X2 <- matrix(0, n, m)
for (i in 1:n) {
  X1[i, ] <- sin(2 * pi * t_grid) * i/n + rnorm(m, sd = 0.1)
  X2[i, ] <- cos(2 * pi * t_grid) * i/n + rnorm(m, sd = 0.1)
}
y <- rowMeans(X1) + 0.5 * rowMeans(X2) + rnorm(n, sd = 0.1)

fd1 <- fdata(X1, argvals = t_grid)
fd2 <- fdata(X2, argvals = t_grid)

# Fit with equal weights
fit1 <- fregre.np.multi(list(fd1, fd2), y)

# Fit with cross-validated weights
fit2 <- fregre.np.multi(list(fd1, fd2), y, weights = "cv")
```

`fregre.pc` *Functional Regression*

Description

Functions for functional regression models. Functional Principal Component Regression

Usage

```
fregre.pc(fdataobj, y, ncomp = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`ncomp`	Number of principal components to use.
`...`	Additional arguments.

Details

Fits a functional linear model using principal component regression.

Value

A fitted regression object of class 'fregre.fd' with components:

`coefficients`	Beta coefficient function values
`intercept`	Intercept term
`fitted.values`	Fitted values
`residuals`	Residuals
`ncomp`	Number of components used
`mean.X`	Mean of functional covariate (for prediction)
`mean.y`	Mean of response (for prediction)
`rotation`	PC loadings (for prediction)
`l`	Indices of selected components
`lm`	Underlying linear model
`sr2`	Residual variance
`fdataobj`	Original functional data
`y`	Response vector
`call`	The function call

`fregre.pc.cv` *Cross-Validation for Functional PC Regression*

Description

Performs k-fold cross-validation to select the optimal number of principal components for functional PC regression.

Usage

```
fregre.pc.cv(fdataobj, y, kfold = 10, ncomp.range = NULL, seed = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata' (functional covariate).
`y`	Response vector.
`kfold`	Number of folds for cross-validation (default 10).
`ncomp.range`	Range of number of components to try. Default is 1 to min(n-1, ncol(data)).
`seed`	Random seed for fold assignment.
`...`	Additional arguments passed to fregre.pc.

Value

A list with components:

optimal.ncomp Optimal number of components

cv.errors Mean squared prediction error for each ncomp

cv.se Standard error of cv.errors

model Fitted model with optimal ncomp

Examples

```
# Create functional data with a linear relationship
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 100, 50)
for (i in 1:100) X[i, ] <- sin(2*pi*t) * i/100 + rnorm(50, sd = 0.1)
beta_true <- cos(2*pi*t)
y <- X %*% beta_true + rnorm(100, sd = 0.5)
fd <- fdata(X, argvals = t)

# Cross-validate to find optimal number of PCs
cv_result <- fregre.pc.cv(fd, y, ncomp.range = 1:10)
```

`GCV.S` *Generalized Cross-Validation for Smoother Selection*

Description

Compute GCV criterion: RSS / (1 - tr(S)/n)^2

Usage

```
GCV.S(S.type, tt, h, y, Ker = "norm", w = NULL)
```

Arguments

`S.type`	Function to compute smoother matrix.
`tt`	Evaluation points.
`h`	Bandwidth parameter.
`y`	Response vector.
`Ker`	Kernel type.
`w`	Optional weights.

Value

The GCV score.

Examples

```
tt <- seq(0, 1, length.out = 50)
y <- sin(2 * pi * tt) + rnorm(50, sd = 0.1)
gcv_score <- GCV.S(S.NW, tt, h = 0.1, y = y)
```

---

gmed *Geometric Median of Functional Data*

Description

Computes the geometric median (L1 median) of functional data using Weiszfeld's iterative algorithm. The geometric median minimizes the sum of L2 distances to all curves/surfaces, making it robust to outliers.

Usage

```
gmed(fdataobj, max.iter = 100, tol = 1e-06)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`max.iter`	Maximum number of iterations (default 100).
`tol`	Convergence tolerance (default 1e-6).

Details

The geometric median y minimizes:

$$\sum_{i=1}^{n} ||X_i - y||_{L2}$$

Unlike the mean (L2 center), the geometric median is robust to outliers because extreme values have bounded influence (influence function is bounded).

The Weiszfeld algorithm is an iteratively reweighted least squares method that converges to the geometric median.

Value

An fdata object containing the geometric median function (1D or 2D).

See Also

`mean.fdata` for the (non-robust) mean function, `median` for depth-based median

Examples

```
# Create functional data with an outlier
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 20, 50)
for (i in 1:19) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
X[20, ] <- sin(2*pi*t) + 5  # Large outlier
fd <- fdata(X, argvals = t)

# Compare mean vs geometric median
mean_curve <- mean(fd)
gmed_curve <- gmed(fd)

# The geometric median is less affected by the outlier
```

---

group.distance *Compute Distance/Similarity Between Groups of Functional Data*

Description

Computes various distance and similarity measures between pre-defined groups of functional curves.

Usage

```
group.distance(
  fdataobj,
  groups,
  method = c("centroid", "hausdorff", "depth", "all"),
  metric = "lp",
  p = 2,
  depth.method = "FM",
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`groups`	A factor or character vector specifying group membership for each curve. Must have length equal to the number of curves.
`method`	Distance/similarity method: • "centroid": L2 distance between group mean curves • "hausdorff": Hausdorff-style distance between groups • "depth": Depth-based overlap (similarity, not distance) • "all": Compute all methods
`metric`	Distance metric for centroid method (default "lp").
`p`	Power for Lp metric (default 2 for L2).
`depth.method`	Depth method for depth-based overlap (default "FM").
`...`	Additional arguments passed to metric functions.

Value

An object of class 'group.distance' containing:

centroid Centroid distance matrix (if method includes centroid)

hausdorff Hausdorff distance matrix (if method includes hausdorff)

depth Depth-based similarity matrix (if method includes depth)

groups Unique group labels

group.sizes Number of curves per group

method Methods used

Examples

```
# Create grouped functional data
set.seed(42)
n <- 30
m <- 50
t_grid <- seq(0, 1, length.out = m)
X <- matrix(0, n, m)
for (i in 1:15) X[i, ] <- sin(2 * pi * t_grid) + rnorm(m, sd = 0.1)
for (i in 16:30) X[i, ] <- cos(2 * pi * t_grid) + rnorm(m, sd = 0.1)
fd <- fdata(X, argvals = t_grid)
groups <- factor(rep(c("A", "B"), each = 15))

# Compute all distance measures
gd <- group.distance(fd, groups, method = "all")
print(gd)
```

`group.test` *Permutation Test for Group Differences*

Description

Tests whether groups of functional data are significantly different using permutation testing.

Usage

```
group.test(
  fdataobj,
  groups,
  n.perm = 1000,
  statistic = c("centroid", "ratio"),
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`groups`	A factor or character vector specifying group membership.
`n.perm`	Number of permutations (default 1000).
`statistic`	Test statistic: "centroid" (distance between group means) or "ratio" (between/within group variance ratio).
`...`	Additional arguments passed to distance functions.

Details

Null Hypothesis (H0): All groups come from the same distribution. That is, the group labels are exchangeable and there is no systematic difference between the functional curves in different groups.

Alternative Hypothesis (H1): At least one group differs from the others in terms of location (mean function) or dispersion.

The test works by:

1. Computing a test statistic on the observed data
2. Repeatedly permuting the group labels and recomputing the statistic
3. Calculating the p-value as the proportion of permuted statistics >= observed

Two test statistics are available:

- `"centroid"`: Sum of pairwise L2 distances between group mean functions. Sensitive to differences in group locations (means).
- `"ratio"`: Ratio of between-group to within-group variance, similar to an F-statistic. Sensitive to both location and dispersion.

A small p-value (e.g., < 0.05) indicates evidence against H0, suggesting that the groups are significantly different.

Value

An object of class 'group.test' containing:

statistic Observed test statistic

p.value Permutation p-value

perm.dist Permutation distribution of test statistic

n.perm Number of permutations used

Examples

```
set.seed(42)
n <- 30
m <- 50
t_grid <- seq(0, 1, length.out = m)
X <- matrix(0, n, m)
```

```
for (i in 1:15) X[i, ] <- sin(2 * pi * t_grid) + rnorm(m, sd = 0.1)
for (i in 16:30) X[i, ] <- cos(2 * pi * t_grid) + rnorm(m, sd = 0.1)
fd <- fdata(X, argvals = t_grid)
groups <- factor(rep(c("A", "B"), each = 15))

# Test for significant difference
gt <- group.test(fd, groups, n.perm = 500)
print(gt)
```

h.default *Default Bandwidth*

Description

Compute a default bandwidth as the 15th percentile of pairwise distances.

Usage

```
h.default(fdataobj, ...)
```

Arguments

`fdataobj`	An object of class 'fdata', or a numeric vector of evaluation points.
`...`	Additional arguments (ignored).

Value

A scalar bandwidth value.

Examples

```
tt <- seq(0, 1, length.out = 50)
h <- h.default(tt)
```

IKer.cos *Integrated Cosine Kernel*

Description

Integral of Ker.cos from -1 to u.

Usage

```
IKer.cos(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Cumulative integral values.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, IKer.cos(u), type = "l", main = "Integrated Cosine Kernel")
```

`IKer.epa` *Integrated Epanechnikov Kernel*

Description

Integral of Ker.epa from -1 to u.

Usage

```
IKer.epa(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Cumulative integral values.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, IKer.epa(u), type = "l", main = "Integrated Epanechnikov Kernel")
```

`IKer.norm` *Integrated Normal Kernel*

Description

Integrated Normal Kernel

Usage

```
IKer.norm(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Cumulative integral of normal kernel from -Inf to u.

Examples

```
u <- seq(-3, 3, length.out = 100)
plot(u, IKer.norm(u), type = "l", main = "Integrated Normal Kernel")
```

`IKer.quar` *Integrated Quartic Kernel*

Description

Integral of Ker.quar from -1 to u.

Usage

```
IKer.quar(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Cumulative integral values.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, IKer.quar(u), type = "l", main = "Integrated Quartic Kernel")
```

`IKer.tri` *Integrated Triweight Kernel*

Description

Integral of Ker.tri from -1 to u.

Usage

```
IKer.tri(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Cumulative integral values.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, IKer.tri(u), type = "l", main = "Integrated Triweight Kernel")
```

`IKer.unif` *Integrated Uniform Kernel*

Description

Integral of Ker.unif from -1 to u.

Usage

```
IKer.unif(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Cumulative integral values.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, IKer.unif(u), type = "l", main = "Integrated Uniform Kernel")
```

`inprod.fdata` *Inner Product of Functional Data*

Description

Compute the inner product of two functional data objects. <f, g> = integral(f(t) * g(t) dt)

Usage

```
inprod.fdata(fdata1, fdata2 = NULL)
```

Arguments

`fdata1`	First functional data object.
`fdata2`	Second functional data object. If NULL, computes inner products of fdata1 with itself.

Value

A matrix of inner products. If fdata1 has n1 curves and fdata2 has n2 curves, returns an n1 x n2 matrix.

Examples

```
t <- seq(0, 1, length.out = 100)
X1 <- matrix(sin(2*pi*t), nrow = 1)
X2 <- matrix(cos(2*pi*t), nrow = 1)
fd1 <- fdata(X1, argvals = t)
fd2 <- fdata(X2, argvals = t)
# Inner product of sin and cos over [0,1] should be 0
inprod.fdata(fd1, fd2)
```

`instantaneous.period` *Estimate Instantaneous Period*

Description

For signals with time-varying frequency (drifting period), estimates the instantaneous period at each time point using the Hilbert transform.

Usage

```
instantaneous.period(fdataobj)
```

Arguments

`fdataobj`	An fdata object.

Details

The Hilbert transform is used to compute the analytic signal, from which the instantaneous phase is extracted. The derivative of the phase gives the instantaneous frequency, and 1/frequency gives the period.

This is particularly useful for signals where the period is not constant, such as circadian rhythms with drift or frequency-modulated signals.

Value

A list with fdata objects:

period Instantaneous period at each time point

frequency Instantaneous frequency at each time point

amplitude Instantaneous amplitude (envelope) at each time point

Examples

```
# Chirp signal with increasing frequency
t <- seq(0, 10, length.out = 500)
freq <- 0.5 + 0.1 * t  # Frequency increases from 0.5 to 1.5
X <- matrix(sin(2 * pi * cumsum(freq) * diff(c(0, t))), nrow = 1)
fd <- fdata(X, argvals = t)

# Estimate instantaneous period
inst <- instantaneous.period(fd)
# plot(inst$period)  # Shows decreasing period over time
```

`int.simpson.irregFdata` *Utility Functions for Functional Data Analysis*

Description

Various utility functions including integration, inner products, random process generation, and prediction metrics. Simpson's Rule Integration

Usage

```
## S3 method for class 'irregFdata'
int.simpson(x, ...)

int.simpson(x, ...)

## S3 method for class 'fdata'
int.simpson(x, ...)
```

Arguments

`x`	A functional data object (`fdata` or `irregFdata`).
`...`	Additional arguments passed to methods.

Details

Integrate functional data over its domain using Simpson's rule (composite trapezoidal rule for non-uniform grids). Works with both regular `fdata` and irregular `irregFdata` objects.

Value

A numeric vector of integrals, one per curve.

Examples

```
t <- seq(0, 1, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) X[i, ] <- sin(2*pi*t)
fd <- fdata(X, argvals = t)
integrals <- int.simpson(fd)  # Should be approximately 0

# Also works with irregular data
ifd <- sparsify(fd, minObs = 20, maxObs = 50, seed = 123)
integrals_irreg <- int.simpson(ifd)
```

`irregFdata` *Create an Irregular Functional Data Object*

Description

Creates an `irregFdata` object for functional data where each observation is sampled at potentially different points. This is common in longitudinal studies, sparse sampling, and sensor data with missing values.

Usage

```
irregFdata(
  argvals,
  X,
  rangeval = NULL,
  names = NULL,
  id = NULL,
  metadata = NULL
)
```

Arguments

`argvals`	A list of numeric vectors, where `argvals[[i]]` contains the observation times for the i-th curve.
`X`	A list of numeric vectors, where `X[[i]]` contains the observed values for the i-th curve. Must have the same lengths as corresponding `argvals[[i]]`.
`rangeval`	Optional numeric vector of length 2 specifying the domain range. If `NULL`, computed from the union of all observation points.
`names`	List with components `main`, `xlab`, `ylab` for plot titles.
`id`	Optional character vector of identifiers for each observation.
`metadata`	Optional data.frame with additional covariates (one row per observation).

Value

An object of class `irregFdata` containing:

argvals List of observation time vectors

X List of value vectors

n Number of observations

rangeval Domain range

names Plot labels

id Observation identifiers

metadata Additional covariates

See Also

`sparsify`, `as.fdata.irregFdata`, `fdata`

Examples

```
# Create irregular functional data directly
argvals <- list(
  c(0.0, 0.3, 0.7, 1.0),
  c(0.0, 0.2, 0.5, 0.8, 1.0),
  c(0.1, 0.4, 0.9)
)
X <- list(
  c(0.1, 0.5, 0.3, 0.2),
  c(0.0, 0.4, 0.6, 0.4, 0.1),
  c(0.3, 0.7, 0.2)
)
ifd <- irregFdata(argvals, X)
print(ifd)
plot(ifd)
```

`is.irregular` *Check if an Object is Irregular Functional Data*

Description

Check if an Object is Irregular Functional Data

Usage

```
is.irregular(x)
```

Arguments

`x`	Any R object.

Value

`TRUE` if `x` is of class `irregFdata`, `FALSE` otherwise.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
is.irregular(fd)  # FALSE

ifd <- sparsify(fd, minObs = 3, maxObs = 7)
is.irregular(ifd)  # TRUE
```

`Ker.cos` *Cosine Kernel*

Description

Cosine Kernel

Usage

```
Ker.cos(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [-1, 1]).

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Ker.cos(u), type = "l", main = "Cosine Kernel")
```

Ker.epa *Epanechnikov Kernel*

Description

Epanechnikov Kernel

Usage

```
Ker.epa(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [-1, 1]).

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Ker.epa(u), type = "l", main = "Epanechnikov Kernel")
```

Ker.norm *Kernel Functions*

Description

Symmetric, asymmetric, and integrated kernel functions for nonparametric smoothing and density estimation. Normal (Gaussian) Kernel

Usage

```
Ker.norm(u)
```

Arguments

u	Numeric vector of evaluation points.

Value

Kernel values at u.

Examples

```
u <- seq(-3, 3, length.out = 100)
plot(u, Ker.norm(u), type = "l", main = "Normal Kernel")
```

`Ker.quar` *Quartic (Biweight) Kernel*

Description

Quartic (Biweight) Kernel

Usage

```
Ker.quar(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [-1, 1]).

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Ker.quar(u), type = "l", main = "Quartic Kernel")
```

`Ker.tri` *Triweight Kernel*

Description

Triweight Kernel

Usage

```
Ker.tri(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [-1, 1]).

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Ker.tri(u), type = "l", main = "Triweight Kernel")
```

`Ker.unif` *Uniform (Rectangular) Kernel*

Description

Uniform (Rectangular) Kernel

Usage

```
Ker.unif(u)
```

Arguments

`u`	Numeric vector of evaluation points.

Value

Kernel values at u (0 outside [-1, 1]).

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Ker.unif(u), type = "l", main = "Uniform Kernel")
```

`Kernel` *Unified Symmetric Kernel Interface*

Description

Evaluates a symmetric kernel function by name.

Usage

```
Kernel(u, type.Ker = "norm")
```

Arguments

`u`	Numeric vector of evaluation points.
`type.Ker`	Kernel type: "norm", "epa", "tri", "quar", "cos", or "unif".

Value

Kernel values at u.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Kernel(u, "epa"), type = "l")
lines(u, Kernel(u, "norm"), col = "red")
```

`kernel.add` *Add Covariance Functions*

Description

Combines two covariance functions by addition.

Usage

```
kernel.add(kernel1, kernel2)
```

Arguments

`kernel1`	First covariance function.
`kernel2`	Second covariance function.

Value

A combined covariance function.

Examples

```
# Combine Gaussian with white noise
k_signal <- kernel.gaussian(variance = 1, length_scale = 0.2)
k_noise <- kernel.whitenoise(variance = 0.1)
k_total <- kernel.add(k_signal, k_noise)

t <- seq(0, 1, length.out = 50)
fd <- make.gaussian.process(n = 5, t = t, cov = k_total)
```

`Kernel.asymmetric` *Unified Asymmetric Kernel Interface*

Description

Evaluates an asymmetric kernel function by name.

Usage

```
Kernel.asymmetric(u, type.Ker = "norm")
```

Arguments

`u`	Numeric vector of evaluation points.
`type.Ker`	Kernel type: "norm", "epa", "tri", "quar", "cos", or "unif".

Value

Kernel values at u.

Examples

```
u <- seq(-0.5, 1.5, length.out = 100)
plot(u, Kernel.asymmetric(u, "epa"), type = "l")
```

`kernel.brownian` *Brownian Motion Covariance Function*

Description

Computes the Brownian motion (Wiener process) covariance function:

$$k(s,t) = \sigma^2 \min(s,t)$$

Usage

```
kernel.brownian(variance = 1)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).

Details

The Brownian motion covariance produces sample paths that start at 0 and have independent increments. The covariance between two points equals the variance times the minimum of their positions.

This covariance is only defined for 1D domains starting at 0.

Value

A covariance function object of class 'kernel_brownian'.

See Also

`kernel.gaussian`, `make.gaussian.process`

Examples

```
# Generate Brownian motion paths
cov_func <- kernel.brownian(variance = 1)
t <- seq(0, 1, length.out = 100)
fd <- make.gaussian.process(n = 10, t = t, cov = cov_func)
plot(fd)
```

`kernel.exponential` *Exponential Covariance Function*

Description

Computes the exponential covariance function:

$$k(s,t) = \sigma^2 \exp\left(-\frac{|s-t|}{\ell}\right)$$

Usage

```
kernel.exponential(variance = 1, length_scale = 1)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).
`length_scale`	Length scale parameter ℓ (default 1).

Details

This is equivalent to the Matern covariance with $\nu = 0.5$. Sample paths are continuous but not differentiable (rough).

The exponential covariance function produces sample paths that are continuous but nowhere differentiable, resulting in rough-looking curves. It is a special case of the Matern family with $\nu = 0.5$.

Value

A covariance function object of class 'kernel_exponential'.

See Also

`kernel.gaussian`, `kernel.matern`, `make.gaussian.process`

Examples

```
# Create an exponential covariance function
cov_func <- kernel.exponential(variance = 1, length_scale = 0.2)

# Evaluate covariance matrix
t <- seq(0, 1, length.out = 50)
K <- cov_func(t)

# Generate rough GP samples
fd <- make.gaussian.process(n = 10, t = t, cov = cov_func)
plot(fd)
```

`kernel.gaussian` *Gaussian (Squared Exponential) Covariance Function*

Description

Computes the Gaussian (RBF/squared exponential) covariance function:

$$k(s,t) = \sigma^2 \exp\left(-\frac{(s-t)^2}{2\ell^2}\right)$$

Usage

```
kernel.gaussian(variance = 1, length_scale = 1)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).
`length_scale`	Length scale parameter ℓ (default 1).

Details

This kernel produces infinitely differentiable (very smooth) sample paths.

The Gaussian covariance function, also known as the squared exponential or radial basis function (RBF) kernel, is one of the most commonly used covariance functions. It produces very smooth sample paths because it is infinitely differentiable.

The length scale parameter controls how quickly the correlation decays with distance. Larger values produce smoother, more slowly varying functions.

Value

A covariance function object of class 'kernel_gaussian'.

See Also

`kernel.exponential`, `kernel.matern`, `make.gaussian.process`

Examples

```
# Create a Gaussian covariance function
cov_func <- kernel.gaussian(variance = 1, length_scale = 0.2)

# Evaluate covariance matrix on a grid
t <- seq(0, 1, length.out = 50)
K <- cov_func(t)
image(K, main = "Gaussian Covariance Matrix")

# Generate Gaussian process samples
fd <- make.gaussian.process(n = 10, t = t, cov = cov_func)
plot(fd)
```

`Kernel.integrate` *Unified Integrated Kernel Interface*

Description

Evaluates an integrated kernel function by name.

Usage

```
Kernel.integrate(u, Ker = "norm", a = -1)
```

Arguments

`u`	Numeric vector of evaluation points.
`Ker`	Kernel type: "norm", "epa", "tri", "quar", "cos", or "unif".
`a`	Lower integration bound (default -1 for symmetric kernels). Not currently used (always integrates from -1 or -Inf).

Value

Cumulative integral values.

Examples

```
u <- seq(-1.5, 1.5, length.out = 100)
plot(u, Kernel.integrate(u, "epa"), type = "l")
```

`kernel.linear` *Linear Covariance Function*

Description

Computes the linear covariance function:

$$k(s,t) = \sigma^2(s-c)(t-c)$$

Usage

```
kernel.linear(variance = 1, offset = 0)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).
`offset`	Offset parameter c (default 0).

Details

The linear covariance function produces sample paths that are linear functions. It is useful when the underlying process is expected to have a linear trend.

Value

A covariance function object of class 'kernel_linear'.

See Also

`kernel.polynomial`, `make.gaussian.process`

Examples

```
# Generate linear function samples
cov_func <- kernel.linear(variance = 1)
t <- seq(0, 1, length.out = 50)
fd <- make.gaussian.process(n = 10, t = t, cov = cov_func)
plot(fd)
```

`kernel.matern` *Matern Covariance Function*

Description

Computes the Matern covariance function with smoothness parameter ν:

$$k(s,t) = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\frac{|s-t|}{\ell}\right)^{\nu} K_{\nu}\left(\sqrt{2\nu}\frac{|s-t|}{\ell}\right)$$

Usage

```
kernel.matern(variance = 1, length_scale = 1, nu = 1.5)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).
`length_scale`	Length scale parameter ℓ (default 1).
`nu`	Smoothness parameter ν (default 1.5). Common values: • `nu = 0.5`: Exponential (continuous, not differentiable) • `nu = 1.5`: Once differentiable • `nu = 2.5`: Twice differentiable • `nu = Inf`: Gaussian/squared exponential (infinitely differentiable)

Details

where K_ν is the modified Bessel function of the second kind.

The Matern family of covariance functions provides flexible control over the smoothness of sample paths through the ν parameter. As ν increases, sample paths become smoother. The Matern family includes the exponential ($\nu = 0.5$) and approaches the Gaussian kernel as $\nu \to \infty$.

For computational efficiency, special cases $\nu \in \{0.5, 1.5, 2.5, \infty\}$ use simplified closed-form expressions.

Value

A covariance function object of class 'kernel_matern'.

See Also

`kernel.gaussian`, `kernel.exponential`, `make.gaussian.process`

Examples

```
# Create Matern covariance functions with different smoothness
cov_rough <- kernel.matern(nu = 0.5)    # Equivalent to exponential
cov_smooth <- kernel.matern(nu = 2.5)   # Twice differentiable

t <- seq(0, 1, length.out = 50)

# Compare sample paths
fd_rough <- make.gaussian.process(n = 5, t = t, cov = cov_rough, seed = 42)
fd_smooth <- make.gaussian.process(n = 5, t = t, cov = cov_smooth, seed = 42)
```

kernel.mult *Multiply Covariance Functions*

Description

Combines two covariance functions by multiplication.

Usage

```
kernel.mult(kernel1, kernel2)
```

Arguments

`kernel1`	First covariance function.
`kernel2`	Second covariance function.

Value

A combined covariance function.

Examples

```
# Multiply periodic with Gaussian for locally periodic behavior
k_periodic <- kernel.periodic(period = 0.3)
k_gaussian <- kernel.gaussian(length_scale = 0.5)
k_prod <- kernel.mult(k_periodic, k_gaussian)
```

`kernel.periodic` *Periodic Covariance Function*

Description

Computes the periodic covariance function:

$$k(s,t) = \sigma^2 \exp\left(-\frac{2\sin^2(\pi|s-t|/p)}{\ell^2}\right)$$

Usage

```
kernel.periodic(variance = 1, length_scale = 1, period = 1)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).
`length_scale`	Length scale parameter ℓ (default 1).
`period`	Period parameter p (default 1).

Details

The periodic covariance function produces sample paths that are periodic with the specified period. It is useful for modeling seasonal or cyclical patterns in functional data.

Value

A covariance function object of class 'kernel_periodic'.

See Also

`kernel.gaussian`, `make.gaussian.process`

Examples

```
# Generate periodic function samples
cov_func <- kernel.periodic(period = 0.5, length_scale = 0.5)
t <- seq(0, 2, length.out = 100)
fd <- make.gaussian.process(n = 5, t = t, cov = cov_func)
plot(fd)
```

kernel.polynomial *Polynomial Covariance Function*

Description

Computes the polynomial covariance function:

$$k(s,t) = \sigma^2(s \cdot t + c)^p$$

Usage

```
kernel.polynomial(variance = 1, offset = 0, degree = 2)
```

Arguments

`variance`	Variance parameter σ^2 (default 1).
`offset`	Offset parameter c (default 0).
`degree`	Polynomial degree p (default 2).

Details

The polynomial covariance function produces sample paths that are polynomial functions of degree at most `degree`. Setting `degree = 1` and `offset = 0` gives the linear kernel.

Value

A covariance function object of class 'kernel_polynomial'.

See Also

`kernel.linear`, `make.gaussian.process`

Examples

```
# Generate quadratic function samples
cov_func <- kernel.polynomial(degree = 2, offset = 1)
t <- seq(0, 1, length.out = 50)
fd <- make.gaussian.process(n = 10, t = t, cov = cov_func)
plot(fd)
```

`kernel.whitenoise` *White Noise Covariance Function*

Description

Computes the white noise covariance function:

$$k(s,t) = \sigma^2 \mathbf{1}_{s=t}$$

Usage

```
kernel.whitenoise(variance = 1)
```

Arguments

`variance`	Variance (noise level) parameter σ^2 (default 1).

Details

where $\mathbf{1}_{s=t}$ is 1 if $s = t$ and 0 otherwise.

The white noise covariance function represents independent noise at each point. It can be added to other covariance functions to model observation noise.

Value

A covariance function object of class 'kernel_whitenoise'.

See Also

`kernel.gaussian`, `make.gaussian.process`

Examples

```
# White noise covariance produces independent samples at each point
cov_func <- kernel.whitenoise(variance = 0.1)
t <- seq(0, 1, length.out = 50)
K <- cov_func(t)
# K is diagonal
```

kernels *Covariance Kernel Functions for Gaussian Processes*

Description

Parametric covariance functions (kernels) used to define the covariance structure of Gaussian processes. These can be used with `make_gaussian_process` to generate synthetic functional data.

Value

No return value. This page documents the family of kernel functions.

localavg.fdata *Local Averages Feature Extraction*

Description

Extracts features from functional data by computing local averages over specified intervals. This is a simple but effective dimension reduction technique for functional data.

Usage

```
localavg.fdata(fdataobj, n.intervals = 10, intervals = NULL)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`n.intervals`	Number of equal-width intervals (default 10).
`intervals`	Optional matrix of custom intervals (2 columns: start, end). If provided, `n.intervals` is ignored.

Details

Local averages provide a simple way to convert functional data to multivariate data while preserving local structure. Each curve is summarized by its average value over each interval.

This can be useful as a preprocessing step for classification or clustering methods that require fixed-dimensional input.

Value

A matrix with n rows (curves) and one column per interval, containing the local average for each curve in each interval.

Examples

```
# Create functional data
t <- seq(0, 1, length.out = 100)
X <- matrix(0, 20, 100)
for (i in 1:20) X[i, ] <- sin(2*pi*t) + rnorm(100, sd = 0.1)
fd <- fdata(X, argvals = t)

# Extract 5 local average features
features <- localavg.fdata(fd, n.intervals = 5)
dim(features)  # 20 x 5

# Use custom intervals
intervals <- cbind(c(0, 0.25, 0.5), c(0.25, 0.5, 1))
features2 <- localavg.fdata(fd, intervals = intervals)
```

`lomb.scargle` *Lomb-Scargle Periodogram*

Description

Computes the Lomb-Scargle periodogram for period detection in unevenly-sampled data. The Lomb-Scargle method is designed for irregularly spaced observations and reduces to the standard periodogram for evenly-spaced data.

Usage

```
lomb.scargle(fdataobj, oversampling = 4, nyquist_factor = 1)
```

Arguments

`fdataobj`	An fdata object. Can have regular or irregular sampling.
`oversampling`	Oversampling factor for the frequency grid. Higher values give finer frequency resolution. Default: 4.
`nyquist_factor`	Maximum frequency as a multiple of the pseudo-Nyquist frequency. Default: 1.

Details

The Lomb-Scargle periodogram is particularly useful when:

- Data has gaps or missing observations
- Sampling is not uniform (e.g., astronomical observations)
- Working with irregular functional data

The algorithm follows Scargle (1982) with significance estimation from Horne & Baliunas (1986). For each test frequency, it computes:

$$P(\omega) = \frac{1}{2\sigma^2}\left[\frac{(\sum_j (y_j - \bar{y})\cos\omega(t_j - \tau))^2}{\sum_j \cos^2\omega(t_j - \tau)} + \frac{(\sum_j (y_j - \bar{y})\sin\omega(t_j - \tau))^2}{\sum_j \sin^2\omega(t_j - \tau)}\right]$$

where τ is a phase shift chosen to make the sine and cosine terms orthogonal.

Value

A list of class "lomb_scargle_result" with components:

frequencies Vector of evaluated frequencies

periods Corresponding periods (1/frequency)

power Normalized Lomb-Scargle power at each frequency

peak_period Period with highest power

peak_frequency Frequency with highest power

peak_power Maximum power value

false_alarm_probability False alarm probability at peak

significance Significance level (1 - FAP)

References

Scargle, J.D. (1982). Studies in astronomical time series analysis. II. Statistical aspects of spectral analysis of unevenly spaced data. The Astrophysical Journal, 263, 835-853.

Horne, J.H., & Baliunas, S.L. (1986). A prescription for period analysis of unevenly sampled time series. The Astrophysical Journal, 302, 757-763.

See Also

`estimate.period`, `sazed`

Examples

```
# Regular sampling
t <- seq(0, 10, length.out = 200)
X <- matrix(sin(2 * pi * t / 2), nrow = 1)
fd <- fdata(X, argvals = t)
result <- lomb.scargle(fd)
print(result)

# Irregular sampling (simulated)
set.seed(42)
t_irreg <- sort(runif(100, 0, 10))
X_irreg <- matrix(sin(2 * pi * t_irreg / 2), nrow = 1)
fd_irreg <- fdata(X_irreg, argvals = t_irreg)
result_irreg <- lomb.scargle(fd_irreg)
```

magnitudeshape — *Magnitude-Shape Outlier Detection for Functional Data*

Description

Performs Magnitude-Shape (MS) outlier detection for functional data. Each curve is represented as a point in 2D space where the x-axis represents magnitude outlyingness and the y-axis represents shape outlyingness.

Usage

```
magnitudeshape(
  fdataobj,
  depth.func = depth.MBD,
  cutoff.quantile = 0.993,
  col.normal = "black",
  col.outliers = "red",
  label = "index",
  label_all = FALSE,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`depth.func`	Depth function to use for computing outlyingness. Default is depth.MBD.
`cutoff.quantile`	Quantile for outlier cutoff (default 0.993).
`col.normal`	Color for normal curves (default "black").
`col.outliers`	Color for outlier curves (default "red").
`label`	What to use for labeling outlier points. Options: • `"index"`: Use numeric indices (default) • `"id"`: Use observation IDs from the fdata object • A column name from the fdata metadata (e.g., `"patient_id"`) • `NULL`: No labels
`label_all`	Logical. If TRUE, label all points, not just outliers. Default FALSE.
`...`	Additional arguments passed to depth function.

Details

The MS plot (Dai & Genton, 2019) decomposes functional outlyingness into:

- **Magnitude Outlyingness (MO)**: Based on pointwise median of directional outlyingness - captures shift outliers

- **Shape Outlyingness (VO)**: Based on variability of directional outlyingness - captures shape outliers

Outliers are detected using the chi-squared distribution with cutoff at the specified quantile.

Value

A list of class 'magnitudeshape' with components:

MO Magnitude outlyingness values

VO Shape (variability) outlyingness values

outliers Indices of detected outliers

cutoff Chi-squared cutoff value used

plot The ggplot object

References

Dai, W. and Genton, M.G. (2019). Directional outlyingness for multivariate functional data. *Computational Statistics & Data Analysis*, 131, 50-65.

Examples

```
# Create functional data with outliers
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:28) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.2)
X[29, ] <- sin(2*pi*t) + 2  # Magnitude outlier
X[30, ] <- sin(4*pi*t)       # Shape outlier
fd <- fdata(X, argvals = t)

# Create MS plot
ms <- magnitudeshape(fd)

# With IDs and metadata
fd <- fdata(X, argvals = t, id = paste0("curve_", 1:30))
ms <- magnitudeshape(fd, label = "id")
```

`make.gaussian.process` *Generate Gaussian Process Samples*

Description

Generates functional data samples from a Gaussian process with the specified mean and covariance functions.

Usage

```
make.gaussian.process(n, t, cov = kernel.gaussian(), mean = 0, seed = NULL)
```

Arguments

`n`	Number of samples to generate.
`t`	Evaluation points (vector for 1D, list of two vectors for 2D).
`cov`	Covariance function (from `kernel.gaussian`, `kernel.matern`, etc.).
`mean`	Mean function. Can be a scalar (default 0), a vector of length equal to the number of evaluation points, or a function.
`seed`	Optional random seed for reproducibility.

Details

This function generates samples from a Gaussian process with the specified covariance structure. The samples are generated by computing the Cholesky decomposition of the covariance matrix and multiplying by standard normal random variables.

For 2D functional data, pass `t` as a list of two vectors representing the grid in each dimension.

Value

An fdata object containing the generated samples.

See Also

`kernel.gaussian`, `kernel.matern`, `kernel.exponential`

Examples

```
# Generate smooth GP samples with Gaussian covariance
t <- seq(0, 1, length.out = 100)
fd <- make.gaussian.process(n = 20, t = t,
                            cov = kernel.gaussian(length_scale = 0.2),
                            seed = 42)
plot(fd)

# Generate rough GP samples with exponential covariance
fd_rough <- make.gaussian.process(n = 20, t = t,
                                  cov = kernel.exponential(length_scale = 0.1),
                                  seed = 42)
plot(fd_rough)

# Generate 2D GP samples (surfaces)
s <- seq(0, 1, length.out = 20)
t2 <- seq(0, 1, length.out = 20)
fd2d <- make.gaussian.process(n = 5, t = list(s, t2),
                              cov = kernel.gaussian(length_scale = 0.3),
                              seed = 42)
plot(fd2d)

# Generate GP with non-zero mean
mean_func <- function(t) sin(2 * pi * t)
fd_mean <- make.gaussian.process(n = 10, t = t,
                                 cov = kernel.gaussian(variance = 0.1),
                                 mean = mean_func, seed = 42)
plot(fd_mean)
```

`matrix.profile` *Matrix Profile for Motif Discovery and Period Detection*

Description

Computes the Matrix Profile using the STOMP (Scalable Time series Ordered-search Matrix Profile) algorithm. The Matrix Profile stores the z-normalized Euclidean distance between each subsequence and its nearest neighbor, enabling efficient motif discovery and period detection for non-sinusoidal patterns.

Usage

```
matrix.profile(fdataobj, subsequence_length = NULL, exclusion_zone = 0.5)
```

Arguments

`fdataobj`	An fdata object.
`subsequence_length`	Length of subsequences to compare. If NULL, automatically determined as approximately 1/4 of series length.
`exclusion_zone`	Fraction of subsequence length to exclude around each position to prevent trivial self-matches. Default: 0.5.

Details

The Matrix Profile algorithm (Yeh et al., 2016; Zhu et al., 2016) is particularly suited for:

- Non-sinusoidal repeating patterns (unlike FFT/spectral methods)
- Motif discovery (finding repeated subsequences)
- Anomaly detection (subsequences with no good match)

The STOMP variant uses O(n^2) time complexity but is highly cache-efficient. For very long series (>10000 points), consider downsampling.

Period detection is based on "arc analysis": counting how often the nearest neighbor is a fixed distance away. Peaks in arc counts indicate periodicity.

Value

A list of class "matrix_profile_result" with components:

profile Numeric vector of minimum z-normalized distances at each position

profile_index Integer vector of nearest neighbor indices (1-based)

subsequence_length Subsequence length used

detected_periods Candidate periods detected from arc analysis (top 5)

arc_counts Arc counts at each index distance (for diagnostics)

primary_period Most prominent detected period

confidence Confidence score (0-1) based on arc prominence

References

Yeh, C. C. M., et al. (2016). Matrix profile I: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. ICDM 2016.

Zhu, Y., et al. (2016). Matrix profile II: Exploiting a novel algorithm and GPUs to break the one hundred million barrier for time series motifs and joins. ICDM 2016.

See Also

`estimate.period`, `sazed`, `lomb.scargle`

Examples

```
# Periodic sawtooth wave (non-sinusoidal)
t <- seq(0, 10, length.out = 200)
period <- 2
X <- matrix((t %% period) / period, nrow = 1)
fd <- fdata(X, argvals = t)

# Compute Matrix Profile
result <- matrix.profile(fd, subsequence_length = 30)
print(result)
plot(result)

# Sine wave for comparison
X_sine <- matrix(sin(2 * pi * t / period), nrow = 1)
fd_sine <- fdata(X_sine, argvals = t)
result_sine <- matrix.profile(fd_sine, subsequence_length = 30)
```

`mean.fdata` *Compute functional mean*

Description

Computes the pointwise mean function across all observations. This is an S3 method for the generic `mean` function.

Usage

```
## S3 method for class 'fdata'
mean(x, ...)
```

Arguments

x	An object of class 'fdata'.
...	Additional arguments (currently ignored).

Value

For 1D fdata: a numeric vector containing the mean function values. For 2D fdata: an fdata object containing the mean surface.

Examples

```
# 1D functional data
fd <- fdata(matrix(rnorm(100), 10, 10))
fm <- mean(fd)

# 2D functional data
X <- array(rnorm(500), dim = c(5, 10, 10))
fd2d <- fdata(X, argvals = list(1:10, 1:10), fdata2d = TRUE)
fm2d <- mean(fd2d)
```

---

mean.irregFdata *Estimate Mean Function for Irregular Data*

Description

Estimates the mean function from irregularly sampled functional data.

Usage

```
## S3 method for class 'irregFdata'
mean(
  x,
  argvals = NULL,
  method = c("basis", "kernel"),
  nbasis = 15,
  type = c("bspline", "fourier"),
  bandwidth = NULL,
  kernel = c("epanechnikov", "gaussian"),
  ...
)
```

Arguments

x	An object of class `irregFdata`.
argvals	Target grid for mean estimation. If `NULL`, uses a regular grid of 100 points.
method	Estimation method: `"basis"` (default, recommended) fits basis functions to each curve then averages; `"kernel"` uses Nadaraya-Watson kernel smoothing.

`nbasis`	Number of basis functions for `method = "basis"` (default 15).
`type`	Basis type for `method = "basis"`: `"bspline"` (default) or `"fourier"`.
`bandwidth`	Kernel bandwidth for `method = "kernel"`. If `NULL`, uses range/10.
`kernel`	Kernel type for `method = "kernel"`: `"epanechnikov"` (default) or `"gaussian"`.
`...`	Additional arguments (ignored).

Details

The `"basis"` method (default) works by:

1. Fitting basis functions to each curve via least squares
2. Reconstructing each curve on the target grid
3. Averaging the reconstructed curves

This approach preserves the functional structure and typically gives more accurate estimates than kernel smoothing.

The `"kernel"` method uses Nadaraya-Watson estimation, pooling all observations across curves. This is faster but may be less accurate for structured functional data.

Value

An `fdata` object containing the estimated mean function.

Examples

```
t <- seq(0, 1, length.out = 100)
fd <- simFunData(n = 50, argvals = t, M = 5, seed = 42)
ifd <- sparsify(fd, minObs = 10, maxObs = 30, seed = 123)

# Recommended: basis method
mean_fd <- mean(ifd)
plot(mean_fd, main = "Estimated Mean Function")

# Alternative: kernel method
mean_kernel <- mean(ifd, method = "kernel", bandwidth = 0.1)
```

`median` *Compute Functional Median*

Description

Returns the curve with maximum depth using the specified depth method.

Usage

```
median(
  fdataobj,
  method = c("FM", "mode", "RP", "RT", "BD", "MBD", "FSD", "KFSD", "RPD"),
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`method`	Depth method to use. One of "FM", "mode", "RP", "RT", "BD", "MBD", "FSD", "KFSD", or "RPD". Default is "FM".
`...`	Additional arguments passed to the depth function.

Value

The curve (as fdata object) with maximum depth.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
med <- median(fd)
med_mode <- median(fd, method = "mode")
```

`metric` *Distance Metrics for Functional Data*

Description

Functions for computing various distance metrics between functional data. Compute Distance Metric for Functional Data

Usage

```
metric(fdataobj, fdataref = NULL, method = "lp", ...)
```

Arguments

`fdataobj` An object of class 'fdata'.

`fdataref` An object of class 'fdata'. If NULL, computes self-distances.

`method` Distance method to use. One of:

- "lp" - Lp metric (default)
- "hausdorff" - Hausdorff distance
- "dtw" - Dynamic Time Warping
- "pca" - Semi-metric based on PCA scores

- "deriv" - Semi-metric based on derivatives
- "basis" - Semi-metric based on basis coefficients
- "fourier" - Semi-metric based on FFT coefficients
- "hshift" - Semi-metric with horizontal shift
- "kl" - Symmetric Kullback-Leibler divergence

... Additional arguments passed to the specific distance function.

Details

Unified interface for computing various distance metrics between functional data objects. This function dispatches to the appropriate specialized distance function based on the method parameter.

This function provides a convenient unified interface for all distance computations in fdars. The additional arguments in ... are passed to the underlying distance function:

- lp: lp, w
- hausdorff: (none)
- dtw: p, w
- pca: ncomp
- deriv: nderiv, lp
- basis: nbasis, basis, nderiv
- fourier: nfreq
- hshift: max_shift
- kl: eps, normalize

Value

A distance matrix.

Examples

```
fd <- fdata(matrix(rnorm(200), 20, 10))

# Using different distance methods
D_lp <- metric(fd, method = "lp")
D_hausdorff <- metric(fd, method = "hausdorff")
D_pca <- metric(fd, method = "pca", ncomp = 3)

# Cross-distances
fd2 <- fdata(matrix(rnorm(100), 10, 10))
D_cross <- metric(fd, fd2, method = "lp")
```

metric.DTW *Dynamic Time Warping for Functional Data*

Description

Computes the Dynamic Time Warping distance between functional data. DTW allows for non-linear alignment of curves.

Usage

```
metric.DTW(fdataobj, fdataref = NULL, p = 2, w = NULL, ...)
```

Arguments

fdataobj	An object of class 'fdata'.
fdataref	An object of class 'fdata'. If NULL, computes self-distances.
p	The p in Lp distance (default 2 for L2/Euclidean).
w	Sakoe-Chiba window constraint. Default is min(ncol(fdataobj), ncol(fdataref)). Use -1 for no window constraint.
...	Additional arguments (ignored).

Value

A distance matrix.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
D <- metric.DTW(fd)
```

metric.hausdorff *Hausdorff Metric for Functional Data*

Description

Computes the Hausdorff distance between functional data objects. The Hausdorff distance treats each curve as a set of points (t, f(t)) in 2D space and computes the maximum of the minimum distances.

Usage

```
metric.hausdorff(fdataobj, fdataref = NULL, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataref`	An object of class 'fdata'. If NULL, uses fdataobj.
`...`	Additional arguments (ignored).

Value

A distance matrix.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
D <- metric.hausdorff(fd)
```

`metric.kl` *Kullback-Leibler Divergence Metric for Functional Data*

Description

Computes the symmetric Kullback-Leibler divergence between functional data treated as probability distributions. Curves are first normalized to be valid probability density functions.

Usage

```
metric.kl(fdataobj, fdataref = NULL, eps = 1e-10, normalize = TRUE, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataref`	An object of class 'fdata'. If NULL, computes self-distances.
`eps`	Small value for numerical stability (default 1e-10).
`normalize`	Logical. If TRUE (default), curves are shifted to be non-negative and normalized to integrate to 1.
`...`	Additional arguments (ignored).

Details

The symmetric KL divergence is computed as:

$$D_{KL}(f, g) = \frac{1}{2}[KL(f||g) + KL(g||f)]$$

where

$$KL(f||g) = \int f(t) \log \frac{f(t)}{g(t)} dt$$

When `normalize = TRUE`, curves are first shifted to be non-negative (by subtracting the minimum and adding eps), then normalized to integrate to 1. This makes them valid probability density functions.

The symmetric KL divergence is always non-negative and equals zero only when the two distributions are identical. However, it does not satisfy the triangle inequality.

Value

A distance matrix based on symmetric KL divergence.

Examples

```
# Create curves that look like probability densities
t <- seq(0, 1, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) {
  # Shifted Gaussian-like curves
  X[i, ] <- exp(-(t - 0.3 - i/50)^2 / 0.02) + rnorm(100, sd = 0.01)
}
fd <- fdata(X, argvals = t)

# Compute KL divergence
D <- metric.kl(fd)
```

`metric.lp.irregFdata` *Lp Metric for Functional Data*

Description

Computes the Lp distance between functional data objects using numerical integration (Simpson's rule). Works with both regular `fdata` and irregular `irregFdata` objects.

Usage

```
## S3 method for class 'irregFdata'
metric.lp(x, p = 2, ...)

metric.lp(x, ...)

## S3 method for class 'fdata'
metric.lp(x, y = NULL, p = 2, w = 1, ...)
```

Arguments

`x`	A functional data object (`fdata` or `irregFdata`).
`p`	The order of the Lp metric (default 2 for L2 distance).
`...`	Additional arguments passed to methods.

`y`	An object of class 'fdata'. If NULL, computes self-distances for x (more efficient symmetric computation). Only supported for `fdata`.
`w`	Optional weight vector of length equal to number of evaluation points. Default is uniform weighting. Only supported for `fdata`.

Value

A distance matrix.

Examples

```
# Regular fdata
fd <- fdata(matrix(rnorm(100), 10, 10))
D <- metric.lp(fd)  # Self-distances

# Irregular fdata
ifd <- sparsify(fd, minObs = 3, maxObs = 7, seed = 42)
D_irreg <- metric.lp(ifd)
```

`norm` *Compute Lp Norm of Functional Data*

Description

Generic function to compute Lp norms for functional data objects. Works with both regular `fdata` and irregular `irregFdata` objects.

Usage

```
norm(x, p = 2, ...)

## S3 method for class 'fdata'
norm(x, p = 2, ...)

## S3 method for class 'irregFdata'
norm(x, p = 2, ...)
```

Arguments

`x`	A functional data object (`fdata` or `irregFdata`).
`p`	The order of the norm (default 2 for L2 norm).
`...`	Additional arguments passed to methods.

Value

A numeric vector of norms, one per curve.

Examples

```
# Regular fdata
fd <- fdata(matrix(rnorm(100), 10, 10))
norms <- norm(fd)

# Irregular fdata
ifd <- sparsify(fd, minObs = 3, maxObs = 7, seed = 42)
norms_irreg <- norm(ifd)
```

`normalize` *Normalize functional data*

Description

Scales each curve to have Lp norm equal to 1.

Usage

```
normalize(fdataobj, p = 2)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`p`	The order of the norm (default 2 for L2 norm).

Value

A normalized 'fdata' object where each curve has unit norm.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10), argvals = seq(0, 1, length.out = 10))
fd_norm <- normalize(fd)
norm(fd_norm)  # All values should be 1
```

`Ops.fdata` *Arithmetic Operations for Functional Data*

Description

Perform elementwise arithmetic on `fdata` objects. Supports addition, subtraction, multiplication, division, and exponentiation between two `fdata` objects or between an `fdata` object and a numeric scalar.

Usage

```
## S3 method for class 'fdata'
Ops(e1, e2)
```

Arguments

`e1, e2`	Objects of class `fdata` or numeric scalars. At least one must be of class `fdata`.

Value

An `fdata` object with the result.

Examples

```
fd1 <- fdata(matrix(1:20, 4, 5))
fd2 <- fdata(matrix(21:40, 4, 5))
fd1 + fd2
fd1 * 2
3 - fd1
```

---

optim.np *Optimize Bandwidth Using Cross-Validation*

Description

Find the optimal bandwidth by minimizing CV or GCV criterion.

Usage

```
optim.np(
  fdataobj,
  S.type,
  h.range = NULL,
  criterion = "GCV",
  Ker = "norm",
  ...
)
```

Arguments

`fdataobj`	An fdata object.
`S.type`	Smoother function (S.NW, S.LLR, etc.).
`h.range`	Range of bandwidths to search (default: data-driven).
`criterion`	"CV" or "GCV".
`Ker`	Kernel type.
`...`	Additional arguments passed to optimizer.

Value

A list with optimal bandwidth and CV/GCV score.

Examples

```
tt <- seq(0, 1, length.out = 50)
y <- sin(2 * pi * tt) + rnorm(50, sd = 0.1)
fd <- fdata(matrix(y, nrow = 1), argvals = tt)
result <- optim.np(fd, S.NW)
```

`outliergram` *Outliergram for Functional Data*

Description

Creates an outliergram plot that displays MEI (Modified Epigraph Index) versus MBD (Modified Band Depth) for outlier detection. Points below the parabolic boundary are identified as outliers, and each outlier is classified by type.

Usage

```
outliergram(fdataobj, factor = 1.5, mei_threshold = 0.25, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`factor`	Factor to adjust the outlier detection threshold. Higher values make detection less sensitive. Default is 1.5.
`mei_threshold`	Deprecated and ignored. Kept for backwards compatibility.
`...`	Additional arguments (currently ignored).

Details

The outliergram plots MEI on the x-axis versus MBD on the y-axis. For a sample of size n, the theoretical relationship is bounded by the finite-sample parabola (Arribas-Gil & Romo, 2014, Proposition 1):

$$MBD \leq a_0 + a_1 \cdot MEI + a_2 \cdot MEI^2$$

where $a_0 = -2/(n(n-1))$, $a_1 = 2(n+1)/(n-1)$, $a_2 = -2(n+1)/(n-1)$.

Shape outliers are detected using a boxplot fence on the vertical distances below the parabola: a curve is flagged when its distance exceeds $Q_3 + \text{factor} \times IQR$.

Value

An object of class 'outliergram' with components:

fdataobj The input functional data

mei MEI values for each curve

mbd MBD values for each curve

outliers Indices of detected outliers

outlier_type Character vector of outlier types ("shape") for each detected outlier

n_outliers Number of outliers detected

factor The factor used for threshold adjustment

parabola Coefficients of the parabolic boundary (a0, a1, a2)

threshold The boxplot-fence threshold for distance below the parabola

dist_to_parabola Vertical distance below the parabola for each curve (positive values indicate the point is below the parabola)

References

Arribas-Gil, A. and Romo, J. (2014). Shape outlier detection and visualization for functional data: the outliergram. *Biostatistics*, 15(4), 603-619.

See Also

depth for depth computation, magnitudeshape for an alternative outlier visualization.

Examples

```
# Create functional data with different outlier types
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 32, 50)
for (i in 1:29) X[i, ] <- sin(2 * pi * t) + rnorm(50, sd = 0.2)
X[30, ] <- sin(2 * pi * t) + 2       # magnitude outlier (high)
X[31, ] <- sin(2 * pi * t) - 2       # magnitude outlier (low)
X[32, ] <- sin(4 * pi * t)           # shape outlier
fd <- fdata(X, argvals = t)

# Create outliergram
og <- outliergram(fd)
print(og)
plot(og, color_by_type = TRUE)
```

`outliers.boxplot` *Outlier Detection using Functional Boxplot*

Description

Detects outliers based on the functional boxplot method. Curves that exceed the fence (1.5 times the central envelope width) at any point are flagged as outliers.

Usage

```
outliers.boxplot(
  fdataobj,
  prob = 0.5,
  factor = 1.5,
  depth.func = depth.MBD,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`prob`	Proportion of curves for the central region (default 0.5).
`factor`	Factor for fence calculation (default 1.5).
`depth.func`	Depth function to use. Default is depth.MBD.
`...`	Additional arguments passed to depth function.

Value

A list of class 'outliers.fdata' with components:

outliers Indices of detected outliers

depths Depth values for all curves

cutoff Not used (for compatibility)

fdataobj Original fdata object

See Also

`boxplot.fdata` for functional boxplot visualization

Examples

```
# Create functional data with outliers
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:28) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.2)
```

```
X[29, ] <- sin(2*pi*t) + 2  # Magnitude outlier
X[30, ] <- cos(2*pi*t)       # Shape outlier
fd <- fdata(X, argvals = t)

# Detect outliers
out <- outliers.boxplot(fd)
print(out)
```

`outliers.depth.pond` *Outlier Detection for Functional Data*

Description

Functions for detecting outliers in functional data using depth measures. Outlier Detection using Weighted Depth

Usage

```
outliers.depth.pond(
  fdataobj,
  nb = 200,
  dfunc = depth.mode,
  threshold_method = c("quantile", "mad", "iqr"),
  quan = 0.05,
  k = NULL,
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`nb`	Number of bootstrap samples. Default is 200.
`dfunc`	Depth function to use. Default is depth.mode.
`threshold_method`	Method for computing the outlier threshold. Options: **"quantile"** Use quantile of weighted depths (default). Curves with depth below this quantile are flagged as outliers. **"mad"** Use median - k * MAD of weighted depths. More robust to existing outliers in the data. **"iqr"** Use Q1 - k * IQR, similar to boxplot whiskers.
`quan`	Quantile for outlier cutoff when `threshold_method = "quantile"`. Default is 0.05, meaning curves with depth in the bottom 5% are flagged (95th percentile threshold). Lower values detect fewer outliers.
`k`	Multiplier for MAD or IQR methods. Default is 2.5 for MAD and 1.5 for IQR. Higher values detect fewer outliers.
`...`	Additional arguments passed to depth function.

Details

Detects outliers based on depth with bootstrap resampling. The threshold for outlier detection can be computed using different methods.

The function first computes depth values for all curves, then uses bootstrap resampling to obtain weighted depths that are more robust to sampling variability.

Threshold Methods:

- **quantile**: Flags curves with depth below the specified quantile. With `quan = 0.1`, approximately 10% of curves would be flagged under the null hypothesis of no outliers. Suitable when you expect a specific proportion of outliers.
- **mad**: Uses `median(depths) - k * MAD(depths)` as threshold. More robust because MAD is not influenced by extreme values. With k = 2.5, this corresponds roughly to a 1-2% false positive rate under normality.
- **iqr**: Uses `Q1 - k * IQR` as threshold, similar to boxplot outlier detection. With k = 1.5, corresponds to the standard boxplot fence.

Value

A list of class 'outliers.fdata' with components:

outliers Indices of detected outliers

depths Depth values for all curves

weighted_depths Bootstrap-weighted depth values

cutoff Depth cutoff used

threshold_method Method used for threshold computation

fdataobj Original fdata object

Examples

```
# Create data with outliers
set.seed(42)
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:28) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.2)
X[29, ] <- sin(2*pi*t) + 3  # outlier
X[30, ] <- -sin(2*pi*t)     # outlier
fd <- fdata(X, argvals = t)


# Default: quantile method with 95th percentile (bottom 5%)
out1 <- outliers.depth.pond(fd, nb = 50)

# More permissive: bottom 10%
out1b <- outliers.depth.pond(fd, nb = 50, quan = 0.1)

# MAD method (more robust)
out2 <- outliers.depth.pond(fd, nb = 50, threshold_method = "mad", k = 2.5)
```

```
# IQR method (boxplot-like)
out3 <- outliers.depth.pond(fd, nb = 50, threshold_method = "iqr", k = 1.5)
```

outliers.depth.trim *Outlier Detection using Trimmed Depth*

Description

Detects outliers based on depth trimming.

Usage

```
outliers.depth.trim(fdataobj, trim = 0.1, dfunc = depth.mode, ...)
```

Arguments

fdataobj	An object of class 'fdata'.
trim	Proportion of curves to consider as potential outliers. Default is 0.1 (curves with depth in bottom 10%).
dfunc	Depth function to use. Default is depth.mode.
...	Additional arguments passed to depth function.

Value

A list of class 'outliers.fdata' with components:

outliers Indices of detected outliers

depths Depth values for all curves

cutoff Depth cutoff used

Examples

```
fd <- fdata(matrix(rnorm(200), 20, 10))
out <- outliers.depth.trim(fd, trim = 0.1)
```

`outliers.lrt` *LRT-based Outlier Detection for Functional Data*

Description

Detects outliers using the Likelihood Ratio Test approach based on Febrero-Bande et al. Uses bootstrap to estimate a threshold and iteratively removes curves exceeding this threshold. Implemented in Rust for high performance with parallelized bootstrap.

Usage

```
outliers.lrt(
  fdataobj,
  nb = 200,
  smo = 0.05,
  trim = 0.1,
  seed = NULL,
  percentile = 0.99
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`nb`	Number of bootstrap replications for threshold estimation (default 200).
`smo`	Smoothing parameter for bootstrap noise (default 0.05).
`trim`	Proportion of curves to trim for robust estimation (default 0.1).
`seed`	Random seed for reproducibility.
`percentile`	Percentile of bootstrap distribution to use as threshold (default 0.99, meaning 99th percentile). Lower values make detection more sensitive (detect more outliers).

Value

A list of class 'outliers.fdata' with components:

outliers Indices of detected outliers

distances Normalized distances for all curves

threshold Bootstrap threshold used

percentile Percentile used for threshold

fdataobj Original fdata object

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:30) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
# Add an outlier
X[1, ] <- X[1, ] + 3
fd <- fdata(X, argvals = t)
out <- outliers.lrt(fd, nb = 100)

# More sensitive detection
out_sensitive <- outliers.lrt(fd, nb = 100, percentile = 0.95)
```

`outliers.thres.lrt` *LRT Outlier Detection Threshold*

Description

Computes the bootstrap threshold for LRT-based outlier detection. This is a highly parallelized Rust implementation providing significant speedup over pure R implementations.

Usage

```
outliers.thres.lrt(
  fdataobj,
  nb = 200,
  smo = 0.05,
  trim = 0.1,
  seed = NULL,
  percentile = 0.99
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`nb`	Number of bootstrap replications (default 200).
`smo`	Smoothing parameter for bootstrap noise (default 0.05).
`trim`	Proportion of curves to trim for robust estimation (default 0.1).
`seed`	Random seed for reproducibility.
`percentile`	Percentile of bootstrap distribution to use as threshold (default 0.99, meaning 99th percentile). Lower values make detection more sensitive (detect more outliers).

Value

The threshold value at the specified percentile.

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:30) X[i, ] <- sin(2*pi*t) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)
thresh <- outliers.thres.lrt(fd, nb = 100)

# More sensitive detection (95th percentile)
thresh_sensitive <- outliers.thres.lrt(fd, nb = 100, percentile = 0.95)
```

`plot.amplitude_modulation` *Plot method for amplitude_modulation objects*

Description

Plot method for amplitude_modulation objects

Usage

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

Arguments

`x`	An amplitude_modulation object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`plot.basis.auto` *Plot method for basis.auto objects*

Description

Plot method for basis.auto objects

Usage

```
## S3 method for class 'basis.auto'
plot(
  x,
  which = c("all", "fourier", "pspline"),
  show.original = TRUE,
  max.curves = 20,
  ...
)
```

Arguments

`x`	A basis.auto object.
`which`	Which curves to plot: "all" (default), "fourier", or "pspline".
`show.original`	Logical. If TRUE (default), overlay original data.
`max.curves`	Maximum number of curves to plot (default 20).
`...`	Additional arguments passed to ggplot.

Value

A `ggplot` object (invisibly).

`plot.basis.cv` *Plot method for basis.cv objects*

Description

Plot method for basis.cv objects

Usage

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

Arguments

`x`	A basis.cv object.
`...`	Additional arguments (ignored).

Value

A `ggplot` object (invisibly).

plot.cluster.fcm *Plot Method for cluster.fcm Objects*

Description

Plot Method for cluster.fcm Objects

Usage

```
## S3 method for class 'cluster.fcm'
plot(x, type = c("curves", "membership"), ...)
```

Arguments

`x`	An object of class 'cluster.fcm'.
`type`	Type of plot: "curves" (default) or "membership".
`...`	Additional arguments (currently ignored).

Value

A ggplot object.

plot.cluster.kmeans *Plot Method for cluster.kmeans Objects*

Description

Plot Method for cluster.kmeans Objects

Usage

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

Arguments

`x`	An object of class 'cluster.kmeans'.
`...`	Additional arguments (currently ignored).

Value

A ggplot object.

`plot.cluster.optim` *Plot Method for cluster.optim Objects*

Description

Plot Method for cluster.optim Objects

Usage

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

Arguments

`x`	An object of class 'cluster.optim'.
`...`	Additional arguments (currently ignored).

Value

A ggplot object.

`plot.fdata` *Plot method for fdata objects*

Description

Displays a plot of functional data. For 1D functional data, plots curves as lines with optional coloring. For 2D functional data, plots surfaces as heatmaps with contour lines.

Usage

```
## S3 method for class 'fdata'
plot(
  x,
  color = NULL,
  alpha = NULL,
  show.mean = FALSE,
  show.ci = FALSE,
  ci.level = 0.9,
  palette = NULL,
  ...
)
```

Arguments

`x`	An object of class 'fdata'.
`color`	Optional vector for coloring curves. Can be: • Numeric vector: curves colored by continuous scale (viridis) • Factor/character: curves colored by discrete groups Must have length equal to number of curves.
`alpha`	Transparency of individual curve lines. Default is 0.7 for basic plots, but automatically reduced to 0.3 when `show.mean = TRUE` or `show.ci = TRUE` to reduce visual clutter and allow mean curves to stand out. Can be explicitly set to override the default.
`show.mean`	Logical. If TRUE and color is categorical, overlay group mean curves with thicker lines (default FALSE).
`show.ci`	Logical. If TRUE and color is categorical, show pointwise confidence interval ribbons per group (default FALSE).
`ci.level`	Confidence level for CI ribbons (default 0.90 for 90 percent).
`palette`	Optional named vector of colors for categorical coloring, e.g., c("A" = "blue", "B" = "red").
`...`	Additional arguments (currently ignored).

Details

This function displays the plot immediately. To get the ggplot object without displaying (e.g., for customization), use `autoplot.fdata`.

Value

The ggplot object (invisibly).

Examples

```
library(ggplot2)
# Display plot immediately
fd <- fdata(matrix(rnorm(200), 20, 10))
plot(fd)

# To get ggplot object without displaying, use autoplot:
p <- autoplot(fd)
```

plot.fdata2pc *Plot FPCA Results*

Description

Visualize functional principal component analysis results with multiple plot types: component perturbation plots, variance explained (scree plot), or score plots.

Usage

```
## S3 method for class 'fdata2pc'
plot(
  x,
  type = c("components", "variance", "scores"),
  ncomp = 3,
  multiple = 2,
  show_both_directions = TRUE,
  ...
)
```

Arguments

`x`	An object of class 'fdata2pc' from `fdata2pc`.
`type`	Type of plot: "components" (default) shows mean +/- scaled PC loadings, "variance" shows a scree plot of variance explained, "scores" shows PC1 vs PC2 scatter plot of observations.
`ncomp`	Number of components to display (default 3 or fewer if not available).
`multiple`	Factor for scaling PC perturbations. Default is 2 (shows +/- 2*sqrt(eigenvalue)*PC).
`show_both_directions`	Logical. If TRUE (default), show both positive and negative perturbations (mean + PC and mean - PC). If FALSE, only show positive perturbation. All curves are solid lines differentiated by color.
`...`	Additional arguments passed to plotting functions.

Details

The "components" plot shows the mean function (black) with perturbations in the direction of each principal component. The perturbation is computed as: mean +/- multiple * sqrt(variance_explained) * PC_loading. All lines are solid and differentiated by color only.

The "variance" plot shows a scree plot with the proportion of variance explained by each component as a bar chart.

The "scores" plot shows a scatter plot of observations in PC space, typically PC1 vs PC2.

Value

A ggplot object (invisibly).

See Also

fdata2pc for computing FPCA.

Examples

```
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:30) X[i, ] <- sin(2*pi*t + runif(1, 0, pi)) + rnorm(50, sd = 0.1)
fd <- fdata(X, argvals = t)
pc <- fdata2pc(fd, ncomp = 3)

# Plot PC components (mean +/- perturbations)
plot(pc, type = "components")

# Scree plot
plot(pc, type = "variance")

# Score plot
plot(pc, type = "scores")
```

`plot.fequiv.test` *Plot method for fequiv.test*

Description

Creates a ggplot showing the mean difference curve, simultaneous confidence band, and equivalence margins.

Usage

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

Arguments

x	An `fequiv.test` object.
...	Additional arguments (ignored).

Value

A ggplot object (invisibly).

`plot.group.distance` *Plot method for group.distance*

Description

Plot method for group.distance

Usage

```
## S3 method for class 'group.distance'
plot(x, type = c("heatmap", "dendrogram"), which = NULL, ...)
```

Arguments

`x`	An object of class 'group.distance'.
`type`	Plot type: "heatmap" or "dendrogram".
`which`	Which distance matrix to plot. If NULL (default), uses the first available matrix from the group.distance object.
`...`	Additional arguments.

Value

A ggplot object (for heatmap) or NULL (for dendrogram, uses base graphics).

`plot.irregFdata` *Plot method for irregFdata objects*

Description

Plot method for irregFdata objects

Usage

```
## S3 method for class 'irregFdata'
plot(
  x,
  ...,
  col = NULL,
  lty = 1,
  lwd = 1,
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  add = FALSE,
  alpha = 0.7
)
```

Arguments

`x`	An irregFdata object.
`...`	Additional arguments passed to `plot`.
`col`	Colors for curves.
`lty`	Line type.
`lwd`	Line width.
`main`	Plot title.
`xlab`	X-axis label.
`ylab`	Y-axis label.
`add`	Logical. If TRUE, add to existing plot.
`alpha`	Transparency for many curves.

Value

Invisibly returns the input object `x`.

---

plot.lomb_scargle_result — *Plot method for lomb_scargle_result objects*

Description

Plot method for lomb_scargle_result objects

Usage

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

Arguments

`x`	A lomb_scargle_result object.
`...`	Additional arguments passed to `plot`.

Value

Invisibly returns `NULL`.

`plot.magnitudeshape` *Plot Method for magnitudeshape Objects*

Description

Plot Method for magnitudeshape Objects

Usage

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

Arguments

`x`	An object of class 'magnitudeshape'.
`...`	Additional arguments (ignored).

Value

The ggplot object (invisibly).

`plot.matrix_profile_result` *Plot method for matrix_profile_result objects*

Description

Plot method for matrix_profile_result objects

Usage

```
## S3 method for class 'matrix_profile_result'
plot(x, type = c("profile", "arcs", "both"), ...)
```

Arguments

`x`	A matrix_profile_result object.
`type`	Plot type: "profile", "arcs", or "both".
`...`	Additional arguments passed to `plot`.

Value

Invisibly returns `NULL`.

`plot.outliergram` *Plot Method for Outliergram Objects*

Description

Creates a scatter plot of MEI vs MBD with the parabolic boundary and identified outliers highlighted.

Usage

```
## S3 method for class 'outliergram'
plot(
  x,
  col_normal = "gray60",
  col_outlier = "red",
  color_by_type = FALSE,
  show_parabola = TRUE,
  show_threshold = TRUE,
  label = "index",
  label_all = FALSE,
  ...
)
```

Arguments

`x`	An object of class 'outliergram'.
`col_normal`	Color for normal observations. Default is "gray60".
`col_outlier`	Color for outliers (used when `color_by_type = FALSE`). Default is "red".
`color_by_type`	Logical. If TRUE, color outliers by their type. Default is FALSE.
`show_parabola`	Logical. If TRUE, draw the theoretical parabola. Default TRUE.
`show_threshold`	Logical. If TRUE, draw the adjusted threshold parabola. Default TRUE.
`label`	What to use for labeling outlier points. Options: • `"index"`: Use numeric indices (default) • `"id"`: Use observation IDs from the fdata object • A column name from the fdata metadata (e.g., `"patient_id"`)
`label_all`	Logical. If TRUE, label all points, not just outliers. Default FALSE.
`...`	Additional arguments passed to plotting functions.

Value

A `ggplot` object (invisibly).

plot.outliers.fdata *Plot method for outliers.fdata objects*

Description

Plot method for outliers.fdata objects

Usage

```
## S3 method for class 'outliers.fdata'
plot(x, col.outliers = "red", ...)
```

Arguments

`x`	An object of class 'outliers.fdata'.
`col.outliers`	Color for outlier curves (default "red").
`...`	Additional arguments (currently ignored).

Value

A ggplot object.

plot.pspline *Plot method for pspline objects*

Description

Plot method for pspline objects

Usage

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

Arguments

`x`	A pspline object.
`...`	Additional arguments passed to plot.fdata.

Value

A `ggplot` object (invisibly).

`plot.pspline.2d` *Plot method for pspline.2d objects*

Description

Plot method for pspline.2d objects

Usage

```
## S3 method for class 'pspline.2d'
plot(x, ...)
```

Arguments

`x`	A pspline.2d object.
`...`	Additional arguments passed to plot.fdata.

Value

A `ggplot` object (invisibly).

`plot.register.fd` *Plot Method for register.fd Objects*

Description

Plot Method for register.fd Objects

Usage

```
## S3 method for class 'register.fd'
plot(x, type = c("registered", "original", "both"), ...)
```

Arguments

`x`	An object of class 'register.fd'.
`type`	Type of plot: "registered" (default), "original", or "both".
`...`	Additional arguments (currently ignored).

Value

A ggplot object.

`plot.ssa_result` *Plot method for ssa_result objects*

Description

Plot method for ssa_result objects

Usage

```
## S3 method for class 'ssa_result'
plot(x, type = c("decomposition", "spectrum"), curves = 1, ...)
```

Arguments

`x`	An ssa_result object.
`type`	Plot type: "decomposition" or "spectrum".
`curves`	Indices of curves to plot.
`...`	Additional arguments (ignored).

Value

Invisibly returns `NULL`.

`plot.stl_result` *Plot method for stl_result objects*

Description

Plot method for stl_result objects

Usage

```
## S3 method for class 'stl_result'
plot(x, curves = 1, ...)
```

Arguments

`x`	An stl_result object.
`curves`	Indices of curves to plot.
`...`	Additional arguments (ignored).

Value

Invisibly returns `NULL`.

pred.MAE *Mean Absolute Error*

Description

Compute the Mean Absolute Error between predicted and actual values.

Usage

```
pred.MAE(y_true, y_pred)
```

Arguments

`y_true`	Actual values.
`y_pred`	Predicted values.

Value

The mean absolute error.

Examples

```
y_true <- c(1, 2, 3, 4, 5)
y_pred <- c(1.1, 2.2, 2.9, 4.1, 4.8)
pred.MAE(y_true, y_pred)
```

pred.MSE *Mean Squared Error*

Description

Compute the Mean Squared Error between predicted and actual values.

Usage

```
pred.MSE(y_true, y_pred)
```

Arguments

`y_true`	Actual values.
`y_pred`	Predicted values.

Value

The mean squared error.

Examples

```
y_true <- c(1, 2, 3, 4, 5)
y_pred <- c(1.1, 2.2, 2.9, 4.1, 4.8)
pred.MSE(y_true, y_pred)
```

`pred.R2` *R-Squared (Coefficient of Determination)*

Description

Compute the R-squared value between predicted and actual values.

Usage

```
pred.R2(y_true, y_pred)
```

Arguments

`y_true`	Actual values.
`y_pred`	Predicted values.

Value

The R-squared value.

Examples

```
y_true <- c(1, 2, 3, 4, 5)
y_pred <- c(1.1, 2.2, 2.9, 4.1, 4.8)
pred.R2(y_true, y_pred)
```

`pred.RMSE` *Root Mean Squared Error*

Description

Compute the Root Mean Squared Error between predicted and actual values.

Usage

```
pred.RMSE(y_true, y_pred)
```

Arguments

`y_true`	Actual values.
`y_pred`	Predicted values.

Value

The root mean squared error.

Examples

```
y_true <- c(1, 2, 3, 4, 5)
y_pred <- c(1.1, 2.2, 2.9, 4.1, 4.8)
pred.RMSE(y_true, y_pred)
```

---

predict.fregre.fd *Predict Method for Functional Regression (fregre.fd)*

Description

Predictions from a fitted functional regression model (fregre.pc or fregre.basis).

Usage

```
## S3 method for class 'fregre.fd'
predict(object, newdata = NULL, ...)
```

Arguments

`object`	A fitted model object of class 'fregre.fd'.
`newdata`	An fdata object containing new functional data for prediction. If NULL, returns fitted values from training data.
`...`	Additional arguments (ignored).

Value

A numeric vector of predicted values.

Examples

```
# Create functional data
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:30) X[i, ] <- sin(2*pi*t) * i/30 + rnorm(50, sd = 0.1)
y <- rowMeans(X) + rnorm(30, sd = 0.1)
fd <- fdata(X, argvals = t)

# Fit model
fit <- fregre.pc(fd, y, ncomp = 3)

# Predict on new data
X_new <- matrix(sin(2*pi*t) * 0.5, nrow = 1)
fd_new <- fdata(X_new, argvals = t)
predict(fit, fd_new)
```

predict.fregre.np *Predict Method for Nonparametric Functional Regression (fregre.np)*

Description

Predictions from a fitted nonparametric functional regression model.

Usage

```
## S3 method for class 'fregre.np'
predict(object, newdata = NULL, ...)
```

Arguments

`object`	A fitted model object of class 'fregre.np'.
`newdata`	An fdata object containing new functional data for prediction. If NULL, returns fitted values from training data.
`...`	Additional arguments (ignored).

Value

A numeric vector of predicted values.

Examples

```
# Create functional data
t <- seq(0, 1, length.out = 50)
X <- matrix(0, 30, 50)
for (i in 1:30) X[i, ] <- sin(2*pi*t) * i/30 + rnorm(50, sd = 0.1)
y <- rowMeans(X) + rnorm(30, sd = 0.1)
fd <- fdata(X, argvals = t)

# Fit model
fit <- fregre.np(fd, y)

# Predict on new data
X_new <- matrix(sin(2*pi*t) * 0.5, nrow = 1)
fd_new <- fdata(X_new, argvals = t)
predict(fit, fd_new)
```

predict.fregre.np.multi

Predict method for fregre.np.multi

Description

Predict method for fregre.np.multi

Usage

```
## S3 method for class 'fregre.np.multi'
predict(object, newdata.list = NULL, ...)
```

Arguments

`object`	Fitted fregre.np.multi object.
`newdata.list`	List of new fdata objects (same length as original).
`...`	Additional arguments.

Value

Predicted values.

print.amplitude_modulation

Print method for amplitude_modulation objects

Description

Print method for amplitude_modulation objects

Usage

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

Arguments

`x`	An amplitude_modulation object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.autoperiod_result` *Print method for autoperiod_result objects*

Description

Print method for autoperiod_result objects

Usage

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

Arguments

`x`	An autoperiod_result object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.basis.auto` *Print method for basis.auto objects*

Description

Print method for basis.auto objects

Usage

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

Arguments

`x`	A basis.auto object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.basis.cv *Print method for basis.cv objects*

Description

Print method for basis.cv objects

Usage

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

Arguments

x	A basis.cv object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.cfd_autoperiod_result *Print method for cfd_autoperiod_result objects*

Description

Print method for cfd_autoperiod_result objects

Usage

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

Arguments

x	A cfd_autoperiod_result object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

`print.cluster.fcm` *Print Method for cluster.fcm Objects*

Description

Print Method for cluster.fcm Objects

Usage

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

Arguments

`x`	An object of class ’cluster.fcm’.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.cluster.kmeans` *Print Method for cluster.kmeans Objects*

Description

Print Method for cluster.kmeans Objects

Usage

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

Arguments

`x`	An object of class ’cluster.kmeans’.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.cluster.optim` *Print Method for cluster.optim Objects*

Description

Print Method for cluster.optim Objects

Usage

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

Arguments

`x`	An object of class 'cluster.optim'.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.decomposition` *Print method for decomposition objects*

Description

Print method for decomposition objects

Usage

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

Arguments

`x`	A decomposition object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.fbplot *Print Method for fbplot Objects*

Description

Print Method for fbplot Objects

Usage

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

Arguments

x	An object of class 'fbplot'.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.fdata *Print method for fdata objects*

Description

Print method for fdata objects

Usage

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

Arguments

x	An object of class 'fdata'.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.fdata.bootstrap.ci *Print method for bootstrap CI*

Description

Print method for bootstrap CI

Usage

```
## S3 method for class 'fdata.bootstrap.ci'
print(x, ...)
```

Arguments

x	A fdata.bootstrap.ci object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.fdata2pc *Print Method for FPCA Results*

Description

Print Method for FPCA Results

Usage

```
## S3 method for class 'fdata2pc'
print(x, ...)
```

Arguments

x	An object of class 'fdata2pc'.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

`print.fequiv.test` *Print method for fequiv.test*

Description

Print method for fequiv.test

Usage

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

Arguments

`x`	An `fequiv.test` object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.fregre.fd` *Print method for fregre objects*

Description

Print method for fregre objects

Usage

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

Arguments

`x`	A fregre.fd object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.fregre.np *Print method for fregre.np objects*

Description

Print method for fregre.np objects

Usage

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

Arguments

x	A fregre.np object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.fregre.np.multi *Print method for fregre.np.multi*

Description

Print method for fregre.np.multi

Usage

```
## S3 method for class 'fregre.np.multi'
print(x, ...)
```

Arguments

x	A fregre.np.multi object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

`print.group.distance` *Print method for group.distance*

Description

Print method for group.distance

Usage

```
## S3 method for class 'group.distance'
print(x, digits = 3, ...)
```

Arguments

`x`	A group.distance object.
`digits`	Number of digits for printing (default 3).
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.group.test` *Print method for group.test*

Description

Print method for group.test

Usage

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

Arguments

`x`	A group.test object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.irregFdata` *Print method for irregFdata objects*

Description

Print method for irregFdata objects

Usage

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

Arguments

`x`	An irregFdata object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.kernel` *Print Method for Covariance Functions*

Description

Print Method for Covariance Functions

Usage

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

Arguments

`x`	A covariance function object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.lomb_scargle_result` *Print method for lomb_scargle_result objects*

Description

Print method for lomb_scargle_result objects

Usage

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

Arguments

`x`	A lomb_scargle_result object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.magnitudeshape` *Print Method for magnitudeshape Objects*

Description

Print Method for magnitudeshape Objects

Usage

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

Arguments

`x`	An object of class 'magnitudeshape'.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.matrix_profile_result

Print method for matrix_profile_result objects

Description

Print method for matrix_profile_result objects

Usage

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

Arguments

x	A matrix_profile_result object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.multiFunData

Print method for multiFunData objects

Description

Print method for multiFunData objects

Usage

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

Arguments

x	A multiFunData object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.multiple_periods *Print method for multiple_periods objects*

Description

Print method for multiple_periods objects

Usage

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

Arguments

`x`	A multiple_periods object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.outliergram *Print Method for Outliergram Objects*

Description

Print Method for Outliergram Objects

Usage

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

Arguments

`x`	An object of class 'outliergram'.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.outliers.fdata` *Print method for outliers.fdata objects*

Description

Print method for outliers.fdata objects

Usage

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

Arguments

`x`	An outliers.fdata object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.peak_detection` *Print method for peak_detection objects*

Description

Print method for peak_detection objects

Usage

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

Arguments

`x`	A peak_detection object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.peak_timing` *Print method for peak_timing objects*

Description

Print method for peak_timing objects

Usage

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

Arguments

`x`	A peak_timing object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.period_estimate` *Print method for period_estimate objects*

Description

Print method for period_estimate objects

Usage

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

Arguments

`x`	A period_estimate object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.pspline` *Print method for pspline objects*

Description

Print method for pspline objects

Usage

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

Arguments

`x`	A pspline object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.pspline.2d` *Print method for pspline.2d objects*

Description

Print method for pspline.2d objects

Usage

```
## S3 method for class 'pspline.2d'
print(x, ...)
```

Arguments

`x`	A pspline.2d object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.register.fd` *Print Method for register.fd Objects*

Description

Print Method for register.fd Objects

Usage

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

Arguments

`x`	An object of class 'register.fd'.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.sazed_result` *Print method for sazed_result objects*

Description

Print method for sazed_result objects

Usage

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

Arguments

`x`	A sazed_result object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.seasonality_changes

Print method for seasonality_changes objects

Description

Print method for seasonality_changes objects

Usage

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

Arguments

x	A seasonality_changes object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

print.seasonality_changes_auto

Print method for seasonality_changes_auto objects

Description

Print method for seasonality_changes_auto objects

Usage

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

Arguments

x	A seasonality_changes_auto object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

`print.seasonality_classification` *Print method for seasonality_classification objects*

Description

Print method for seasonality_classification objects

Usage

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

Arguments

`x`	A seasonality_classification object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

`print.ssa_result` *Print method for ssa_result objects*

Description

Print method for ssa_result objects

Usage

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

Arguments

`x`	An ssa_result object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object `x`.

print.stl_result *Print method for stl_result objects*

Description

Print method for stl_result objects

Usage

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

Arguments

x	An stl_result object.
...	Additional arguments (ignored).

Value

Invisibly returns the input object x.

pspline *P-spline Smoothing for Functional Data*

Description

Fits penalized B-splines (P-splines) to functional data with automatic or manual selection of the smoothing parameter.

Usage

```
pspline(
  fdataobj,
  nbasis = 20,
  lambda = 1,
  order = 2,
  lambda.select = FALSE,
  criterion = c("GCV", "AIC", "BIC"),
  lambda.range = 10^seq(-4, 4, length.out = 50)
)
```

Arguments

`fdataobj`	An fdata object.
`nbasis`	Number of B-spline basis functions (default 20).
`lambda`	Smoothing parameter. Higher values give smoother curves. If NULL and lambda.select = TRUE, selected automatically.
`order`	Order of the difference penalty (default 2, for second derivative penalty).
`lambda.select`	Logical. If TRUE, select lambda automatically using the specified criterion.
`criterion`	Criterion for lambda selection: "GCV" (default), "AIC", or "BIC".
`lambda.range`	Range of lambda values to search (log10 scale). Default: `10^seq(-4, 4, length.out = 50)`.

Details

P-splines minimize:

$$||y - Bc||^2 + \lambda c' D' Dc$$

where B is the B-spline basis matrix, c are coefficients, and D is the difference matrix of the specified order.

Value

A list of class "pspline" with:

fdata Smoothed fdata object

coefs Coefficient matrix

lambda Used or selected lambda value

edf Effective degrees of freedom

gcv/aic/bic Criterion values

nbasis Number of basis functions used

References

Eilers, P.H.C. and Marx, B.D. (1996). Flexible smoothing with B-splines and penalties. *Statistical Science*, 11(2), 89-121.

Examples

```
# Create noisy data
t <- seq(0, 1, length.out = 100)
true_signal <- sin(2 * pi * t)
noisy <- true_signal + rnorm(100, sd = 0.3)
fd <- fdata(matrix(noisy, nrow = 1), argvals = t)

# Smooth with P-splines
result <- pspline(fd, nbasis = 20, lambda = 10)
plot(fd)
lines(t, result$fdata$data[1, ], col = "red", lwd = 2)
```

```
# Automatic lambda selection
result_auto <- pspline(fd, nbasis = 20, lambda.select = TRUE)
```

---

pspline.2d *P-spline Smoothing for 2D Functional Data*

Description

Fits 2D P-splines with anisotropic penalties in both directions.

Usage

```
pspline.2d(
  fdataobj,
  nbasis.s = 10,
  nbasis.t = 10,
  lambda.s = 1,
  lambda.t = 1,
  order = 2,
  lambda.select = FALSE,
  criterion = c("GCV", "AIC", "BIC")
)
```

Arguments

`fdataobj`	A 2D fdata object.
`nbasis.s`	Number of B-spline basis functions in s direction.
`nbasis.t`	Number of B-spline basis functions in t direction.
`lambda.s`	Smoothing parameter in s direction.
`lambda.t`	Smoothing parameter in t direction.
`order`	Order of the difference penalty (default 2).
`lambda.select`	Logical. If TRUE, select lambdas automatically.
`criterion`	Criterion for selection: "GCV", "AIC", or "BIC".

Details

The 2D penalty uses Kronecker product structure:

$$P = \lambda_s(I_t \otimes P_s) + \lambda_t(P_t \otimes I_s)$$

Value

A list of class "pspline.2d" similar to `pspline()`.

r.bridge *Generate Brownian Bridge*

Description

Simulate sample paths from a Brownian bridge, which is a Brownian motion conditioned to return to 0 at time 1.

Usage

```
r.bridge(n, t, sigma = 1, seed = NULL)
```

Arguments

n	Number of sample paths.
t	Evaluation points (should include 0 and 1 for standard bridge).
sigma	Volatility (default 1).
seed	Optional random seed.

Value

An fdata object containing the simulated paths.

Examples

```
t <- seq(0, 1, length.out = 100)
bb_data <- r.bridge(n = 20, t = t)
plot(bb_data)
```

r.brownian *Generate Brownian Motion*

Description

Simulate sample paths from standard Brownian motion (Wiener process).

Usage

```
r.brownian(n, t, sigma = 1, x0 = 0, seed = NULL)
```

Arguments

n	Number of sample paths.
t	Evaluation points.
sigma	Volatility (standard deviation per unit time, default 1).
x0	Initial value (default 0).
seed	Optional random seed.

Value

An fdata object containing the simulated paths.

Examples

```
t <- seq(0, 1, length.out = 100)
bm_data <- r.brownian(n = 20, t = t)
plot(bm_data)
```

`r.ou` *Generate Ornstein-Uhlenbeck Process*

Description

Simulate sample paths from an Ornstein-Uhlenbeck process using the Euler-Maruyama discretization scheme.

Usage

```
r.ou(n, t, mu = 0, theta = 1, sigma = 1, x0 = 0, seed = NULL)
```

Arguments

`n`	Number of sample paths to generate.
`t`	Evaluation points (numeric vector).
`mu`	Long-term mean (default 0).
`theta`	Mean reversion rate (default 1).
`sigma`	Volatility (default 1).
`x0`	Initial value (default 0).
`seed`	Optional random seed.

Details

The OU process satisfies the SDE: dX(t) = -theta * X(t) dt + sigma * dW(t)

Value

An fdata object containing the simulated paths.

Examples

```
t <- seq(0, 1, length.out = 100)
ou_data <- r.ou(n = 20, t = t, theta = 2, sigma = 1)
plot(ou_data)
```

`register.fd` *Curve Registration (Alignment)*

Description

Aligns functional data by horizontal shifting to a target curve. This reduces phase variation in the sample.

Usage

```
register.fd(fdataobj, target = NULL, max.shift = 0.2)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`target`	Target curve to align to. If NULL (default), uses the mean.
`max.shift`	Maximum allowed shift as proportion of domain (default 0.2).

Details

Shift registration finds the horizontal translation that maximizes the cross-correlation between each curve and the target. This is appropriate when curves have similar shapes but differ mainly in timing.

For more complex warping, consider DTW-based methods.

Value

A list of class 'register.fd' with components:

registered An fdata object with registered (aligned) curves.

shifts Numeric vector of shift amounts for each curve.

target The target curve used for alignment.

fdataobj Original (unregistered) functional data.

See Also

`metric.DTW` for dynamic time warping distance

Examples

```
# Create phase-shifted curves
set.seed(42)
t <- seq(0, 1, length.out = 100)
X <- matrix(0, 20, 100)
for (i in 1:20) {
  phase <- runif(1, -0.1, 0.1)
  X[i, ] <- sin(2*pi*(t + phase)) + rnorm(100, sd = 0.1)
}
fd <- fdata(X, argvals = t)

# Register curves
reg <- register.fd(fd)
print(reg)

# Compare original vs registered
oldpar <- par(mfrow = c(1, 2))
plot(fd)
plot(reg$registered)
par(oldpar)
```

S.KNN *K-Nearest Neighbors Smoother Matrix*

Description

Compute a smoother matrix using adaptive bandwidth based on the k nearest neighbors. The bandwidth at each point is the distance to the k-th nearest neighbor.

Usage

```
S.KNN(tt, knn, Ker = "norm", w = NULL, cv = FALSE)
```

Arguments

`tt`	Evaluation points (numeric vector).
`knn`	Number of nearest neighbors.
`Ker`	Kernel function or name.
`w`	Optional weights vector.
`cv`	Logical. If TRUE, compute leave-one-out cross-validation matrix.

Value

An n x n smoother matrix S.

Examples

```
tt <- seq(0, 1, length.out = 50)
S <- S.KNN(tt, knn = 10)
```

S.LCR *Local Cubic Regression Smoother Matrix*

Description

Convenience function for Local Polynomial Regression with degree 3.

Usage

```
S.LCR(tt, h, Ker = "norm", w = NULL, cv = FALSE)
```

Arguments

`tt`	Evaluation points (numeric vector).
`h`	Bandwidth parameter.
`Ker`	Kernel function or name.
`w`	Optional weights vector.
`cv`	Logical. If TRUE, compute leave-one-out cross-validation matrix.

Value

An n x n smoother matrix S.

Examples

```
tt <- seq(0, 1, length.out = 50)
S <- S.LCR(tt, h = 0.15)
```

S.LLR *Local Linear Regression Smoother Matrix*

Description

Compute the Local Linear Regression (LLR) smoother matrix. LLR has better boundary bias properties than Nadaraya-Watson.

Usage

```
S.LLR(tt, h, Ker = "norm", w = NULL, cv = FALSE)
```

Arguments

`tt`	Evaluation points (numeric vector).
`h`	Bandwidth parameter.
`Ker`	Kernel function or name. One of "norm", "epa", "tri", "quar", "cos", "unif".
`w`	Optional weights vector of length n.
`cv`	Logical. If TRUE, compute leave-one-out cross-validation matrix.

Value

An n x n smoother matrix S.

Examples

```
tt <- seq(0, 1, length.out = 50)
S <- S.LLR(tt, h = 0.1)
```

`S.LPR` *Local Polynomial Regression Smoother Matrix*

Description

Compute the Local Polynomial Regression smoother matrix of degree p. Special cases: p=0 is Nadaraya-Watson, p=1 is Local Linear Regression.

Usage

```
S.LPR(tt, h, p = 1, Ker = "norm", w = NULL, cv = FALSE)
```

Arguments

`tt`	Evaluation points (numeric vector).
`h`	Bandwidth parameter.
`p`	Polynomial degree (default 1 for local linear).
`Ker`	Kernel function or name.
`w`	Optional weights vector.
`cv`	Logical. If TRUE, compute leave-one-out cross-validation matrix.

Value

An n x n smoother matrix S.

Examples

```
tt <- seq(0, 1, length.out = 50)
S <- S.LPR(tt, h = 0.1, p = 2)  # Local quadratic regression
```

S.NW *Smoothing Functions for Functional Data*

Description

Functions for computing smoothing matrices and applying kernel smoothing to functional data. Nadaraya-Watson Kernel Smoother Matrix

Usage

```
S.NW(tt, h, Ker = "norm", w = NULL, cv = FALSE)
```

Arguments

`tt`	Evaluation points (numeric vector).
`h`	Bandwidth parameter.
`Ker`	Kernel function or name. One of "norm", "epa", "tri", "quar", "cos", "unif", or a custom function.
`w`	Optional weights vector of length n.
`cv`	Logical. If TRUE, compute leave-one-out cross-validation matrix (diagonal is zero).

Details

Compute the Nadaraya-Watson kernel smoother matrix.

Value

An n x n smoother matrix S such that smooth(y) = S %*% y.

Examples

```
tt <- seq(0, 1, length.out = 50)
S <- S.NW(tt, h = 0.1)
dim(S)  # 50 x 50
```

sazed — *SAZED: Spectral-ACF Zero-crossing Ensemble Detection*

Description

A parameter-free ensemble method for robust period detection that combines five different detection approaches and uses majority voting to determine the consensus period.

Usage

```
sazed(fdataobj, tolerance = 0.1, detrend_method = c("none", "linear", "auto"))
```

Arguments

`fdataobj`	An fdata object.
`tolerance`	Relative tolerance for considering periods equal when voting. Default: 0.1 (10% relative difference). Use smaller values for stricter matching, larger values for more lenient matching.
`detrend_method`	Detrending method to apply before period estimation: **"none"** No detrending (default) **"linear"** Remove linear trend **"auto"** Automatic AIC-based selection of detrending method

Details

SAZED combines five detection methods:

1. **Spectral**: Finds peaks in the FFT periodogram above the noise floor
2. **ACF Peak**: Identifies the first significant peak in the autocorrelation function
3. **ACF Average**: Computes a weighted mean of ACF peak locations
4. **Zero-crossing**: Estimates period from ACF zero-crossing intervals
5. **Spectral Diff**: Applies FFT to first-differenced signal (trend removal)

The final period is chosen by majority voting: periods within the tolerance are grouped together, and the group with the most members determines the consensus. The returned period is the average of the agreeing estimates.

This method is particularly robust because:

- It requires no tuning parameters (tolerance has sensible defaults)
- Multiple methods must agree for high confidence
- Differencing component handles trends automatically
- Works well across different signal types

Value

A list of class "sazed_result" with components:

period Consensus period (average of agreeing components)

confidence Confidence score (0-1, proportion of agreeing components)

agreeing_components Number of components that agreed on the period

components List of individual component estimates:

- **spectral** Period from FFT periodogram peak
- **acf_peak** Period from first ACF peak
- **acf_average** Weighted average of ACF peaks
- **zero_crossing** Period from ACF zero crossings
- **spectral_diff** Period from FFT on differenced signal

See Also

`estimate.period` for single-method estimation, `detect.periods` for detecting multiple concurrent periods

Examples

```
# Generate seasonal data with period = 2
t <- seq(0, 20, length.out = 400)
X <- matrix(sin(2 * pi * t / 2) + 0.1 * rnorm(400), nrow = 1)
fd <- fdata(X, argvals = t)

# Detect period using SAZED
result <- sazed(fd)
print(result)  # Shows consensus period and component details

# With trend - SAZED's spectral_diff component handles this
X_trend <- matrix(0.3 * t + sin(2 * pi * t / 2), nrow = 1)
fd_trend <- fdata(X_trend, argvals = t)
result <- sazed(fd_trend)
```

`scale_minmax` *Min-Max scaling for functional data*

Description

Scales each curve to the range $[0, 1]$ (or custom range). This preserves the shape while normalizing the range.

Usage

```
scale_minmax(fdataobj, min = 0, max = 1)

## S3 method for class 'fdata'
scale_minmax(fdataobj, min = 0, max = 1)

## S3 method for class 'irregFdata'
scale_minmax(fdataobj, min = 0, max = 1)
```

Arguments

`fdataobj`	An object of class ’fdata’.
`min`	Target minimum value (default 0).
`max`	Target maximum value (default 1).

Value

A scaled ’fdata’ object where each curve is in the specified range.

Examples

```
fd <- fdata(matrix(rnorm(100) * 10 + 50, 10, 10), argvals = seq(0, 1, length.out = 10))
fd_scaled <- scale_minmax(fd)
# Check: each curve now in [0, 1]
apply(fd_scaled$data, 1, range)
```

`sd` *Functional Standard Deviation*

Description

Computes the pointwise standard deviation function of functional data.

Usage

```
sd(fdataobj, ...)
```

Arguments

`fdataobj`	An object of class ’fdata’.
`...`	Additional arguments (currently ignored).

Value

An fdata object containing the standard deviation function (1D or 2D).

Examples

```
# 1D functional data
fd <- fdata(matrix(rnorm(100), 10, 10))
s <- sd(fd)

# 2D functional data
X <- array(rnorm(500), dim = c(5, 10, 10))
fd2d <- fdata(X, argvals = list(1:10, 1:10), fdata2d = TRUE)
s2d <- sd(fd2d)
```

`seasonal.strength` *Measure Seasonal Strength*

Description

Computes the strength of seasonality in functional data. Values range from 0 (no seasonality) to 1 (pure seasonal signal).

Usage

```
seasonal.strength(
  fdataobj,
  period = NULL,
  method = c("variance", "spectral", "wavelet"),
  n_harmonics = 3,
  detrend_method = c("none", "linear", "auto")
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Known or estimated period. If NULL, period is estimated automatically using FFT.
`method`	Method for computing strength: **"variance"** Variance decomposition: Var(seasonal) / Var(total) **"spectral"** Spectral: power at seasonal frequencies / total power **"wavelet"** Wavelet: Morlet wavelet power at seasonal period / total variance
`n_harmonics`	Number of Fourier harmonics to use (for variance method). Default: 3.
`detrend_method`	Detrending method to apply before computing strength: **"none"** No detrending (default) **"linear"** Remove linear trend **"auto"** Automatic AIC-based selection of detrending method

Details

The variance method decomposes the signal into a seasonal component (using Fourier basis with the specified period) and computes the proportion of variance explained by the seasonal component.

The spectral method computes the proportion of total spectral power that falls at the seasonal frequency and its harmonics.

The wavelet method uses a Morlet wavelet to measure power at the target period. It provides time-localized frequency information and is robust to non-stationary signals.

Trends can artificially lower the seasonal strength measure by contributing non-seasonal variance. Use detrend_method to remove trends before computing strength.

Value

A numeric value between 0 and 1 representing seasonal strength.

Examples

```
# Pure seasonal signal
t <- seq(0, 10, length.out = 200)
X <- matrix(sin(2 * pi * t / 2), nrow = 1)
fd_seasonal <- fdata(X, argvals = t)
seasonal.strength(fd_seasonal, period = 2)  # Should be close to 1

# Pure noise
X_noise <- matrix(rnorm(200), nrow = 1)
fd_noise <- fdata(X_noise, argvals = t)
seasonal.strength(fd_noise, period = 2)  # Should be close to 0

# Trending data - detrending improves strength estimate
X_trend <- matrix(2 + 0.5 * t + sin(2 * pi * t / 2), nrow = 1)
fd_trend <- fdata(X_trend, argvals = t)
seasonal.strength(fd_trend, period = 2, detrend_method = "linear")
```

`seasonal.strength.curve` *Time-Varying Seasonal Strength*

Description

Computes seasonal strength at each time point using a sliding window, allowing detection of how seasonality changes over time.

Usage

```
seasonal.strength.curve(
  fdataobj,
  period,
  window_size = NULL,
  method = c("variance", "spectral")
)
```

Arguments

`fdataobj`	An fdata object.
`period`	Known or estimated period.
`window_size`	Width of the sliding window. Recommended: 2 * period.
`method`	Method for computing strength: "variance" or "spectral".

Value

An fdata object containing the time-varying seasonal strength curve.

Examples

```
# Signal that transitions from seasonal to non-seasonal
t <- seq(0, 20, length.out = 400)
X <- ifelse(t < 10, sin(2 * pi * t / 2), rnorm(length(t[t >= 10]), sd = 0.5))
X <- matrix(X, nrow = 1)
fd <- fdata(X, argvals = t)

# Compute time-varying strength
ss <- seasonal.strength.curve(fd, period = 2, window_size = 4)
# plot(ss)  # Shows strength declining around t = 10
```

`select.basis.auto` *Automatic Per-Curve Basis Type and Number Selection*

Description

Selects the optimal basis type (Fourier or P-spline) and number of basis functions for each curve individually using model selection criteria. This is useful when working with mixed datasets containing both seasonal and non-seasonal curves.

Usage

```
select.basis.auto(
  fdataobj,
  criterion = c("GCV", "AIC", "BIC"),
  nbasis.range = NULL,
  lambda.pspline = NULL,
  use.seasonal.hint = TRUE
)
```

Arguments

`fdataobj`	An fdata object.
`criterion`	Model selection criterion: "GCV" (default), "AIC", or "BIC".

`nbasis.range`	Optional numeric vector of length 2 specifying `c(min, max)` for nbasis search range. If NULL, automatic ranges are used: Fourier 3-25, P-spline 6-40 (or limited by data length).
`lambda.pspline`	Smoothing parameter for P-splines. If NULL (default), lambda is automatically selected from a grid for each curve.
`use.seasonal.hint`	Logical. If TRUE (default), uses FFT-based seasonality detection to inform basis preference. Seasonal curves start Fourier search from 5 basis functions.

Details

For each curve, the function searches over:

- Fourier basis: odd nbasis values from 3 (or 5 if seasonal) to min(m/3, 25)
- P-spline basis: nbasis from 6 to min(m/2, 40), with lambda from grid {0.001, 0.01, 0.1, 1, 10, 100} if lambda.pspline is NULL

The function uses parallel processing (via Rust/rayon) for efficiency when processing multiple curves.

Value

A list of class "basis.auto" with:

basis.type Character vector ("pspline" or "fourier") for each curve

nbasis Integer vector of selected nbasis per curve

score Numeric vector of best criterion scores

coefficients List of coefficient vectors for each curve

fitted fdata object with fitted values

edf Numeric vector of effective degrees of freedom

seasonal.detected Logical vector indicating detected seasonality

lambda Numeric vector of lambda values (NA for Fourier curves)

criterion Character string of criterion used

original Original fdata object

See Also

`fdata2basis_cv` for global basis selection, `pspline` for P-spline fitting

Examples

```
# Generate mixed data: some seasonal, some polynomial
set.seed(42)
t <- seq(0, 10, length.out = 100)

# 3 seasonal curves
X_seasonal <- matrix(0, 3, 100)
```

```
for (i in 1:3) {
  X_seasonal[i, ] <- sin(2 * pi * t / 2.5) + rnorm(100, sd = 0.2)
}

# 3 polynomial curves
X_poly <- matrix(0, 3, 100)
for (i in 1:3) {
  X_poly[i, ] <- 0.1 * t^2 - t + rnorm(100, sd = 0.5)
}

fd <- fdata(rbind(X_seasonal, X_poly), argvals = t)

# Auto-select optimal basis for each curve
result <- select.basis.auto(fd)
print(result)

# Should detect: first 3 as Fourier, last 3 as P-spline
table(result$basis.type)
```

---

semimetric.basis *Semi-metric based on Basis Expansion*

Description

Computes a semi-metric based on the L2 distance of basis expansion coefficients. Supports B-spline and Fourier basis.

Usage

```
semimetric.basis(
  fdataobj,
  fdataref = NULL,
  nbasis = 5,
  basis = "bspline",
  nderiv = 0,
  ...
)
```

Arguments

fdataobj	An object of class 'fdata'.
fdataref	An object of class 'fdata'. If NULL, uses fdataobj.
nbasis	Number of basis functions. Default is 5.
basis	Type of basis: "bspline" (default) or "fourier".
nderiv	Derivative order to compute distance on (default 0).
...	Additional arguments (ignored).

Value

A distance matrix based on basis coefficients.

Examples

```
# Create curves
t <- seq(0, 1, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) X[i, ] <- sin(2*pi*t + i/5) + rnorm(100, sd = 0.1)
fd <- fdata(X, argvals = t)

# Compute distance based on B-spline coefficients
D <- semimetric.basis(fd, nbasis = 7)
```

---

semimetric.deriv *Semi-metric based on Derivatives*

Description

Computes a semi-metric based on the Lp distance of the nderiv-th derivative of functional data.

Usage

```
semimetric.deriv(fdataobj, fdataref = NULL, nderiv = 1, lp = 2, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataref`	An object of class 'fdata'. If NULL, uses fdataobj.
`nderiv`	Derivative order (1, 2, ...). Default is 1.
`lp`	The p in Lp metric. Default is 2 (L2 distance).
`...`	Additional arguments passed to deriv.

Value

A distance matrix based on derivative distances.

Examples

```
# Create smooth curves
t <- seq(0, 2*pi, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) X[i, ] <- sin(t + i/5)
fd <- fdata(X, argvals = t)

# Compute distance based on first derivative
D <- semimetric.deriv(fd, nderiv = 1)
```

semimetric.fourier *Semi-metric based on Fourier Coefficients (FFT)*

Description

Computes a semi-metric based on the L2 distance of Fourier coefficients computed via Fast Fourier Transform (FFT). This is more efficient than the Fourier basis option in semimetric.basis for large nfreq.

Usage

```
semimetric.fourier(fdataobj, fdataref = NULL, nfreq = 5, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataref`	An object of class 'fdata'. If NULL, uses fdataobj.
`nfreq`	Number of Fourier frequencies to use (excluding DC). Default is 5.
`...`	Additional arguments (ignored).

Details

The Fourier coefficients are computed using FFT and normalized by the number of points. The distance is the L2 distance between the magnitude of the first nfreq+1 coefficients (DC + nfreq frequencies).

This function uses Rust's rustfft library for efficient FFT computation, making it faster than R's base fft for large datasets.

Value

A distance matrix based on Fourier coefficients.

Examples

```
# Create curves with different frequency content
t <- seq(0, 1, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) X[i, ] <- sin(2*pi*i*t) + rnorm(100, sd = 0.1)
fd <- fdata(X, argvals = t)

# Compute distance based on Fourier coefficients
D <- semimetric.fourier(fd, nfreq = 10)
```

`semimetric.hshift` *Semi-metric based on Horizontal Shift (Time Warping)*

Description

Computes a semi-metric based on the minimum L2 distance after optimal horizontal shifting of curves. This is useful for comparing curves that may have phase differences.

Usage

```
semimetric.hshift(fdataobj, fdataref = NULL, max_shift = -1, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataref`	An object of class 'fdata'. If NULL, uses fdataobj.
`max_shift`	Maximum shift in number of grid points. Default is m/4 where m is the number of evaluation points. Use -1 for automatic.
`...`	Additional arguments (ignored).

Details

For each pair of curves, this function computes:

$$d(f,g) = \min_{|h| \le h_{max}} ||f(t) - g(t+h)||$$

where h is the horizontal shift in discrete units.

This semi-metric is useful when comparing curves with phase shifts, such as ECG signals with different heart rates or periodic signals with different phases.

Value

A distance matrix based on minimum L2 distance after shift.

Examples

```
# Create curves with phase shifts
t <- seq(0, 2*pi, length.out = 100)
X <- matrix(0, 10, 100)
for (i in 1:10) X[i, ] <- sin(t + i*0.2) + rnorm(100, sd = 0.1)
fd <- fdata(X, argvals = t)

# Compute distance accounting for phase shifts
D <- semimetric.hshift(fd, max_shift = 10)
```

semimetric.pca *Semi-metric based on Principal Components*

Description

Computes a semi-metric based on the first ncomp principal component scores.

Usage

```
semimetric.pca(fdataobj, fdataref = NULL, ncomp = 2, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`fdataref`	An object of class 'fdata'. If NULL, uses fdataobj.
`ncomp`	Number of principal components to use.
`...`	Additional arguments (ignored).

Value

A distance matrix based on PC scores.

Examples

```
fd <- fdata(matrix(rnorm(200), 20, 10))
D <- semimetric.pca(fd, ncomp = 3)
```

simFunData *Simulate Functional Data via Karhunen-Loeve Expansion*

Description

Generates functional data samples using a truncated Karhunen-Loeve representation: `f_i(t) = mean(t) + sum_{k=1}^M xi_{ik} * phi_k(t)` where `xi_{ik} ~ N(0, lambda_k)`.

Usage

```
simFunData(
  n,
  argvals,
  M,
  eFun.type = c("Fourier", "Poly", "PolyHigh", "Wiener"),
  eVal.type = c("linear", "exponential", "wiener"),
  mean = NULL,
  seed = NULL
)
```

Arguments

`n`	Number of curves to generate.
`argvals`	Numeric vector of evaluation points.
`M`	Number of basis functions (eigenfunctions) to use.
`eFun.type`	Type of eigenfunction basis: `"Fourier"`, `"Poly"`, `"PolyHigh"`, or `"Wiener"`.
`eVal.type`	Type of eigenvalue decay: `"linear"`, `"exponential"`, or `"wiener"`.
`mean`	Mean function. Can be: • `NULL` for zero mean • A numeric vector of length equal to `argvals` • A function that takes `argvals` as input
`seed`	Optional integer random seed for reproducibility.

Details

The Karhunen-Loeve expansion provides a natural way to simulate Gaussian functional data with a specified covariance structure. The eigenvalues control the variance contribution of each mode, while the eigenfunctions determine the shape of variation.

The theoretical covariance function is: `Cov(X(s), X(t)) = sum_{k=1}^M lambda_k * phi_k(s) * phi_k(t)`

Value

An `fdata` object containing the simulated functional data.

See Also

`eFun`, `eVal`, `addError`, `make.gaussian.process`

Examples

```
t <- seq(0, 1, length.out = 100)

# Basic simulation with Fourier basis
fd <- simFunData(n = 20, argvals = t, M = 5,
                 eFun.type = "Fourier", eVal.type = "linear")
plot(fd, main = "Simulated Functional Data (Fourier, Linear)")

# Smoother curves with exponential decay
fd_smooth <- simFunData(n = 20, argvals = t, M = 10,
                        eFun.type = "Fourier", eVal.type = "exponential")
plot(fd_smooth, main = "Smooth Simulated Data (Exponential Decay)")

# Wiener process simulation
fd_wiener <- simFunData(n = 20, argvals = t, M = 10,
                        eFun.type = "Wiener", eVal.type = "wiener", seed = 42)
plot(fd_wiener, main = "Wiener Process Simulation")
```

```
# With mean function
mean_fn <- function(t) sin(2 * pi * t)
fd_mean <- simFunData(n = 20, argvals = t, M = 5, mean = mean_fn, seed = 42)
plot(fd_mean, main = "Simulated Data with Sinusoidal Mean")
```

`simMultiFunData` *Simulate Multivariate Functional Data*

Description

Generates multivariate (vector-valued) functional data where each component is simulated via Karhunen-Loeve expansion with potentially different eigenfunctions, eigenvalues, and domains.

Usage

```
simMultiFunData(
  n,
  argvals,
  M,
  eFun.type = "Fourier",
  eVal.type = "linear",
  mean = NULL,
  seed = NULL
)
```

Arguments

`n`	Number of multivariate curves to generate.
`argvals`	List of numeric vectors, one per component.
`M`	Integer or integer vector. Number of basis functions per component. If a single integer, used for all components.
`eFun.type`	Character or character vector specifying eigenfunction type for each component. See `eFun` for options.
`eVal.type`	Character or character vector specifying eigenvalue decay for each component. See `eVal` for options.
`mean`	List of mean functions (one per component), or `NULL`.
`seed`	Optional integer random seed.

Value

A list of class `multiFunData` containing:

components List of `fdata` objects, one per component

n Number of observations

p Number of components

See Also

simFunData, eFun, eVal

Examples

```
# Two-component multivariate functional data
t1 <- seq(0, 1, length.out = 100)
t2 <- seq(0, 0.5, length.out = 50)

mfd <- simMultiFunData(
  n = 20,
  argvals = list(t1, t2),
  M = c(5, 3),
  eFun.type = c("Fourier", "Wiener"),
  eVal.type = c("exponential", "linear")
)

# Plot first component
plot(mfd$components[[1]], main = "Component 1")
```

---

sparsify — *Convert Regular Functional Data to Irregular by Subsampling*

Description

Creates an `irregFdata` object from regular `fdata` by randomly selecting a subset of observation points for each curve.

Usage

```
sparsify(fdataobj, minObs = 5, maxObs = NULL, prob = NULL, seed = NULL)
```

Arguments

`fdataobj`	An object of class `fdata`.
`minObs`	Minimum number of observations to keep per curve.
`maxObs`	Maximum number of observations to keep per curve. If `NULL`, uses the total number of points.
`prob`	Sampling probability function. If `NULL`, uniform sampling is used. Otherwise, a function that takes `argvals` and returns sampling weights (not necessarily normalized).
`seed`	Optional integer random seed for reproducibility.

Details

For each curve, the function:

1. Draws a random number of points to keep between `minObs` and `maxObs`
2. Samples that many points (without replacement) from the grid
3. If `prob` is provided, sampling is weighted accordingly

Common probability functions:

- Uniform: `NULL` (default)
- More points in middle: `function(t) dnorm(t, mean = 0.5, sd = 0.2)`
- More points at ends: `function(t) 1 - dnorm(t, mean = 0.5, sd = 0.2)`

Value

An object of class `irregFdata`.

See Also

`irregFdata`, `as.fdata.irregFdata`, `addError`

Examples

```
# Create regular functional data
t <- seq(0, 1, length.out = 100)
fd <- simFunData(n = 20, argvals = t, M = 5, seed = 42)

# Uniform sparsification
ifd <- sparsify(fd, minObs = 10, maxObs = 30, seed = 123)
print(ifd)
plot(ifd)

# Non-uniform: more observations in the middle
prob_middle <- function(t) dnorm(t, mean = 0.5, sd = 0.2)
ifd_middle <- sparsify(fd, minObs = 15, maxObs = 25, prob = prob_middle, seed = 123)
plot(ifd_middle, main = "More Observations in Middle")
```

`ssa.fd` *Singular Spectrum Analysis (SSA) for Time Series Decomposition*

Description

Performs Singular Spectrum Analysis on functional data to decompose each curve into trend, seasonal (oscillatory), and noise components. SSA is a model-free, non-parametric technique based on singular value decomposition of the trajectory matrix.

Usage

```
ssa.fd(fdataobj, window.length = NULL, n.components = 10)
```

Arguments

`fdataobj`	An fdata object.
`window.length`	Embedding window length (L). If NULL, automatically determined as min(n/2, 50). Larger values capture longer-term patterns.
`n.components`	Number of SVD components to extract. Default: 10. More components allow finer decomposition but increase noise.

Details

The SSA algorithm consists of four stages:

1. Embedding: The time series is converted into a trajectory matrix by arranging lagged versions of the series as columns.

2. SVD Decomposition: Singular value decomposition of the trajectory matrix produces orthogonal components.

3. Grouping: Components are grouped into trend (slowly varying), seasonal (oscillatory), and noise. Auto-detection uses sign change analysis and autocorrelation.

4. Reconstruction: Diagonal averaging (Hankelization) converts grouped trajectory matrices back to time series.

SSA is particularly suited for:

- Short time series where spectral methods fail
- Noisy data with weak periodic signals
- Non-stationary data with changing trend
- Separating multiple periodicities

Value

A list of class "ssa_result" with components:

trend fdata object containing reconstructed trend component

seasonal fdata object containing reconstructed seasonal component

noise fdata object containing noise/residual component

singular.values Singular values from SVD (sorted descending)

contributions Proportion of variance explained by each component

window.length Window length used

n.components Number of components extracted

detected.period Auto-detected period (if any)

confidence Confidence score for detected period

call The function call

References

Golyandina, N., & Zhigljavsky, A. (2013). Singular Spectrum Analysis for Time Series. Springer.

Elsner, J. B., & Tsonis, A. A. (1996). Singular Spectrum Analysis: A New Tool in Time Series Analysis. Plenum Press.

See Also

stl.fd, decompose, estimate.period

Examples

```
# Signal with trend + seasonal + noise
t <- seq(0, 10, length.out = 200)
X <- matrix(0.05 * t + sin(2 * pi * t / 1.5) + rnorm(length(t), sd = 0.3), nrow = 1)
fd <- fdata(X, argvals = t)

# Perform SSA
result <- ssa.fd(fd)
print(result)

# Plot components
plot(result)

# Examine singular value spectrum (scree plot)
plot(result, type = "spectrum")
```

standardize *Standardize functional data (z-score normalization)*

Description

Transforms each curve to have mean 0 and standard deviation 1. This is useful for comparing curve shapes regardless of their level or scale.

Usage

```
standardize(fdataobj)

## S3 method for class 'fdata'
standardize(fdataobj)

## S3 method for class 'irregFdata'
standardize(fdataobj)
```

Arguments

fdataobj	An object of class ’fdata’.

Value

A standardized ’fdata’ object where each curve has mean 0 and sd 1.

Examples

```
fd <- fdata(matrix(rnorm(100) * 10 + 50, 10, 10), argvals = seq(0, 1, length.out = 10))
fd_std <- standardize(fd)
# Check: each curve now has mean ~0 and sd ~1
rowMeans(fd_std$data)
apply(fd_std$data, 1, sd)
```

`stl.fd` *STL Decomposition: Seasonal and Trend decomposition using LOESS*

Description

Performs STL (Seasonal and Trend decomposition using LOESS) on functional data following Cleveland et al. (1990). This is a robust iterative procedure that separates a time series into trend, seasonal, and remainder components.

Usage

```
stl.fd(fdataobj, period, s.window = NULL, t.window = NULL, robust = TRUE)
```

Arguments

`fdataobj`	An fdata object.
`period`	Integer. The seasonal period (number of observations per cycle).
`s.window`	Seasonal smoothing window. Must be odd. If NULL, defaults to 7. Larger values produce smoother seasonal components.
`t.window`	Trend smoothing window. Must be odd. If NULL, automatically calculated based on period and s.window.
`robust`	Logical. If TRUE, performs robustness iterations to downweight outliers using bisquare weighting. Default: TRUE.

Details

The STL algorithm proceeds as follows:

Inner Loop (repeated n.inner times):

1. Detrending: Subtract current trend estimate
2. Cycle-subseries smoothing: Smooth values at each seasonal position across cycles
3. Low-pass filtering: Remove high-frequency noise
4. Detrending the smoothed cycle-subseries
5. Deseasonalizing: Subtract seasonal from original data
6. Trend smoothing: Apply LOESS to deseasonalized data

Outer Loop (for robustness):

1. Compute residuals from current decomposition
2. Calculate robustness weights using bisquare function
3. Re-run inner loop with weighted smoothing

STL is particularly effective for:

- Long time series with many cycles
- Data with outliers (when robust=TRUE)
- Slowly changing seasonal patterns

Value

A list of class "stl_result" with components:

trend fdata object containing trend components

seasonal fdata object containing seasonal components

remainder fdata object containing remainder (residual) components

weights Matrix of robustness weights (1 = full weight, 0 = outlier)

period The period used

s.window Seasonal smoothing window used

t.window Trend smoothing window used

inner.iterations Number of inner loop iterations

outer.iterations Number of outer (robustness) iterations

call The function call

References

Cleveland, R. B., Cleveland, W. S., McRae, J. E., & Terpenning, I. (1990). STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Journal of Official Statistics, 6(1), 3-73.

See Also

decompose, detrend, seasonal.strength

Examples

```
# Create seasonal data with trend
t <- seq(0, 20, length.out = 400)
period <- 2  # corresponds to 40 observations
period_obs <- 40
X <- matrix(0.05 * t + sin(2 * pi * t / period) + rnorm(length(t), sd = 0.2), nrow = 1)
fd <- fdata(X, argvals = t)

# Perform STL decomposition
result <- stl.fd(fd, period = period_obs)
print(result)
```

```
# Plot the decomposition
plot(result)

# Non-robust version (faster but sensitive to outliers)
result_fast <- stl.fd(fd, period = period_obs, robust = FALSE)
```

`summary.basis.auto` *Summary method for basis.auto objects*

Description

Summary method for basis.auto objects

Usage

```
## S3 method for class 'basis.auto'
summary(object, ...)
```

Arguments

`object`	A basis.auto object.
`...`	Additional arguments (ignored).

Value

Invisibly returns a data frame with per-curve basis selection details.

`summary.fdata` *Summary method for fdata objects*

Description

Summary method for fdata objects

Usage

```
## S3 method for class 'fdata'
summary(object, ...)
```

Arguments

`object`	An object of class ’fdata’.
`...`	Additional arguments (ignored).

Value

Invisibly returns a summary list with descriptive statistics.

`summary.irregFdata` *Summary method for irregFdata objects*

Description

Summary method for irregFdata objects

Usage

```
## S3 method for class 'irregFdata'
summary(object, ...)
```

Arguments

`object`	An irregFdata object.
`...`	Additional arguments (ignored).

Value

Invisibly returns the input object.

`trimmed` *Compute Functional Trimmed Mean*

Description

Computes the trimmed mean by excluding curves with lowest depth.

Usage

```
trimmed(
  fdataobj,
  trim = 0.1,
  method = c("FM", "mode", "RP", "RT", "BD", "MBD", "FSD", "KFSD", "RPD"),
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`trim`	Proportion of curves to trim (default 0.1).
`method`	Depth method to use. One of "FM", "mode", "RP", "RT", "BD", "MBD", "FSD", "KFSD", or "RPD". Default is "FM".
`...`	Additional arguments passed to the depth function.

Value

An fdata object containing the trimmed mean function.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
tm <- trimmed(fd, trim = 0.2)
tm_mode <- trimmed(fd, trim = 0.2, method = "mode")
```

`trimvar` *Compute Functional Trimmed Variance*

Description

Computes the trimmed variance by excluding curves with lowest depth.

Usage

```
trimvar(
  fdataobj,
  trim = 0.1,
  method = c("FM", "mode", "RP", "RT", "BD", "MBD", "FSD", "KFSD", "RPD"),
  ...
)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`trim`	Proportion of curves to trim (default 0.1).
`method`	Depth method to use. One of "FM", "mode", "RP", "RT", "BD", "MBD", "FSD", "KFSD", or "RPD". Default is "FM".
`...`	Additional arguments passed to the depth function.

Value

An fdata object containing the trimmed variance function.

Examples

```
fd <- fdata(matrix(rnorm(100), 10, 10))
tv <- trimvar(fd, trim = 0.2)
tv_mode <- trimvar(fd, trim = 0.2, method = "mode")
```

`var` *Functional Variance*

Description

Computes the pointwise variance function of functional data.

Usage

```
var(fdataobj, ...)
```

Arguments

`fdataobj`	An object of class 'fdata'.
`...`	Additional arguments (currently ignored).

Value

An fdata object containing the variance function (1D or 2D).

Examples

```
# 1D functional data
fd <- fdata(matrix(rnorm(100), 10, 10))
v <- var(fd)

# 2D functional data
X <- array(rnorm(500), dim = c(5, 10, 10))
fd2d <- fdata(X, argvals = list(1:10, 1:10), fdata2d = TRUE)
v2d <- var(fd2d)
```

`[.fdata` *Subset method for fdata objects*

Description

Subset method for fdata objects

Usage

```
## S3 method for class 'fdata'
x[i, j, drop = FALSE]
```

Arguments

`x`	An object of class 'fdata'.
`i`	Row indices (which curves to keep).
`j`	Column indices (which time points to keep).
`drop`	Logical. If TRUE and only one curve selected, return vector.

Value

An `fdata` object containing the selected subset.

`[.irregFdata` *Subset method for irregFdata objects*

Description

Subset method for irregFdata objects

Usage

```
## S3 method for class 'irregFdata'
x[i, ...]
```

Arguments

`x`	An irregFdata object.
`i`	Indices of observations to select.
`...`	Additional arguments (ignored).

Value

An `irregFdata` object containing the selected subset.

Index