Package 'MFPCA'

January 20, 2025

Type Package

Title Multivariate Functional Principal Component Analysis for Data Observed on Different Dimensional Domains

Version 1.3-10

Date 2022-09-15

Maintainer Clara Happ-Kurz <chk_R@gmx.de>

Description Calculate a multivariate functional principal component analysis for data observed on different dimensional domains. The estimation algorithm relies on univariate basis expansions for each element of the multivariate functional data (Happ & Greven, 2018) <doi:10.1080/01621459.2016.1273115>. Multivariate and univariate functional data objects are represented by S4 classes for this type of data implemented in the package 'funData'. For more details on the general concepts of both packages and a case study, see Happ-Kurz (2020) <doi:10.18637/jss.v093.i05>.

URL https://github.com/ClaraHapp/MFPCA

License GPL-2

Imports abind, foreach, irlba, Matrix(>= 1.5-0), methods, mgcv (>= 1.8-33), plyr, stats

Depends R (>= 3.2.0), funData (>= 1.3-4)

Suggests covr, fda, testthat (>= 2.0.0)

NeedsCompilation yes

SystemRequirements libfftw3 (>= 3.3.4)

RoxygenNote 7.2.1

Author Clara Happ-Kurz [aut, cre] (<https://orcid.org/0000-0003-4737-3835>)

Repository CRAN

Date/Publication 2022-09-15 08:30:02 UTC

Contents

`FCP_TPA` *The functional CP-TPA algorithm*

Description

This function implements the functional CP-TPA (FCP-TPA) algorithm, that calculates a smooth PCA for 3D tensor data (i.e. `N` observations of 2D images with dimension `S1 x S2`). The results are given in a CANDECOMP/PARAFRAC (CP) model format

$$X = \sum_{k=1}^{K} d_k \cdot u_k \circ v_k \circ w_k$$

where $\circ$ stands for the outer product, d_k is a scalar and u_k, v_k, w_k are eigenvectors for each direction of the tensor. In this representation, the outer product $v_k \circ w_k$ can be regarded as the k-th eigenimage, while $d_k \cdot u_k$ represents the vector of individual scores for this eigenimage and each observation.

Usage

```
FCP_TPA(
  X,
  K,
  penMat,
  alphaRange,
  verbose = FALSE,
  tol = 1e-04,
  maxIter = 15,
  adaptTol = TRUE
)
```

Arguments

X	The data tensor of dimensions N x S1 x S2.
K	The number of eigentensors to be calculated.
penMat	A list with entries v and w, containing a roughness penalty matrix for each direction of the image. The algorithm does not induce smoothness along observations (see Details).
alphaRange	A list of length 2 with entries v and w , containing the range of smoothness parameters to test for each direction.
verbose	Logical. If TRUE, computational details are given on the standard output during calculation of the FCP_TPA.
tol	A numeric value, giving the tolerance for relative error values in the algorithm. Defaults to 1e-4. It is automatically multiplied by 10 after maxIter steps, if adaptTol = TRUE.
maxIter	A numeric value, the maximal iteration steps. Can be doubled, if adaptTol = TRUE.
adaptTol	Logical. If TRUE, the tolerance is adapted (multiplied by 10), if the algorithm has not converged after maxIter steps and another maxIter steps are allowed with the increased tolerance, see Details. Use with caution. Defaults to TRUE.

Details

The smoothness of the eigenvectors v_k, w_k is induced by penalty matrices for both image directions, that are weighted by smoothing parameters α_{vk}, α_{wk}. The eigenvectors u_k are not smoothed, hence the algorithm does not induce smoothness along observations.

Optimal smoothing parameters are found via a nested generalized cross validation. In each iteration of the TPA (tensor power algorithm), the GCV criterion is optimized via optimize on the interval specified via alphaRange$v (or alphaRange$w, respectively).

The FCP_TPA algorithm is an iterative algorithm. Convergence is assumed if the relative difference between the actual and the previous values are all below the tolerance level tol. The tolerance level is increased automatically, if the algorithm has not converged after maxIter steps and if adaptTol = TRUE. If the algorithm did not converge after maxIter steps (or 2 * maxIter) steps, the function throws a warning.

Value

d	A vector of length K, containing the numeric weights d_k in the CP model.
U	A matrix of dimensions N x K, containing the eigenvectors u_k in the first dimension.
V	A matrix of dimensions S1 x K, containing the eigenvectors v_k in the second dimension.
W	A matrix of dimensions S2 x K, containing the eigenvectors w_k in the third dimension.

References

G. I. Allen, "Multi-way Functional Principal Components Analysis", IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing, 2013.

See Also

fcptpaBasis

Examples

```
# set.seed(1234)

N <- 100
S1 <- 75
S2 <- 75

# define "true" components
v <- sin(seq(-pi, pi, length.out = S1))
w <- exp(seq(-0.5, 1, length.out = S2))

# simulate tensor data with dimensions N x S1 x S2
X <- rnorm(N, sd = 0.5) %o% v %o% w

# create penalty matrices (penalize first differences for each dimension)
Pv <- crossprod(diff(diag(S1)))
Pw <- crossprod(diff(diag(S2)))

# estimate one eigentensor
res <- FCP_TPA(X, K = 1, penMat = list(v = Pv, w = Pw),
            alphaRange = list(v = c(1e-4, 1e4), w = c(1e-4, 1e4)),
            verbose = TRUE)

# plot the results and compare to true values
plot(res$V)
points(v/sqrt(sum(v^2)), pch = 20)
legend("topleft", legend = c("True", "Estimated"), pch = c(20, 1))

plot(res$W)
points(w/sqrt(sum(w^2)), pch = 20)
legend("topleft", legend = c("True", "Estimated"), pch = c(20, 1))
```

MFPCA *Multivariate functional principal component analysis for functions on different (dimensional) domains*

Description

This function calculates a multivariate functional principal component analysis (MFPCA) based on i.i.d. observations $x_1, \ldots, x_N$ of a multivariate functional data-generating process $X = (X^{(1)}, \ldots X^{(p)})$

with elements $X^{(j)} \in L^2(\mathcal{T}_j)$ defined on a domain $\mathcal{T}_j \subset IR^{d_j}$. In particular, the elements can be defined on different (dimensional) domains. The results contain the mean function, the estimated multivariate functional principal components $\hat{\psi}_1, \ldots, \hat{\psi}_M$ (having the same structure as x_i), the associated eigenvalues $\hat{\nu}_1 \geq \ldots \geq \hat{\nu}_M > 0$ and the individual scores $\hat{\rho}_{im} = < \widehat{x_i, \psi_m} >$. Moreover, estimated trajectories for each observation based on the truncated Karhunen-Loeve representation

$$\hat{x}_i = \sum_{m=1}^{M} \hat{\rho}_{im} \hat{\psi}_m$$

are given if desired (`fit = TRUE`). The implementation of the observations $x_i = (x_i^{(1)}, \ldots, x_i^{(p)})$, $i = 1, \ldots, N$, the mean function and multivariate functional principal components $\hat{\psi}_1, \ldots, \hat{\psi}_M$ uses the `multiFunData` class, which is defined in the package **funData**.

Usage

```
MFPCA(
  mFData,
  M,
  uniExpansions,
  weights = rep(1, length(mFData)),
  fit = FALSE,
  approx.eigen = FALSE,
  bootstrap = FALSE,
  nBootstrap = NULL,
  bootstrapAlpha = 0.05,
  bootstrapStrat = NULL,
  verbose = options()$verbose
)
```

Arguments

`mFData`	A `multiFunData` object containing the `N` observations.
`M`	The number of multivariate functional principal components to calculate.
`uniExpansions`	A list characterizing the (univariate) expansion that is calculated for each element. See Details.
`weights`	An optional vector of weights, defaults to `1` for each element. See Details.
`fit`	Logical. If `TRUE`, a truncated multivariate Karhunen-Loeve representation for the data is calculated based on the estimated scores and eigenfunctions.
`approx.eigen`	Logical. If `TRUE`, the eigenanalysis problem for the estimated covariance matrix is solved approximately using the **irlba** package, which is much faster. If the number `M` of eigenvalues to calculate is high with respect to the number of observations in `mFData` or the number of estimated univariate eigenfunctions, the approximation may be inappropriate. In this case, approx.eigen is set to `FALSE` and the function throws a warning. Defaults to `FALSE`.
`bootstrap`	Logical. If `TRUE`, pointwise bootstrap confidence bands are calculated for the multivariate functional principal components. Defaults to `FALSE`. See Details.

nBootstrap	The number of bootstrap iterations to use. Defaults to NULL, which leads to an error, if bootstrap = TRUE.
bootstrapAlpha	A vector of numerics (or a single number) giving the significance level for bootstrap intervals. Defaults to 0.05.
bootstrapStrat	A stratification variable for bootstrap. Must be a factor of length nObs(mFData) or NULL (default). If NULL, no stratification is made in the bootstrap resampling, i.e. the curves are sampled with replacement. If bootstrapStrat is not NULL, the curves are resampled with replacement within the groups defined by bootstrapStrat, hence keeping the group proportions fixed.
verbose	Logical. If TRUE, the function reports extra-information about the progress (incl. timestamps). Defaults to options()$verbose.

Details

Weighted MFPCA: If the elements vary considerably in domain, range or variation, a weight vector $w_1, \ldots, w_p$ can be supplied and the MFPCA is based on the weighted scalar product

$$<< f, g >>_w = \sum_{j=1}^{p} w_j \int_{\mathcal{T}_j} f^{(j)}(t) g^{(j)}(t) \mathrm{d}t$$

and the corresponding weighted covariance operator Γ_w.

Bootstrap: If bootstrap = TRUE, pointwise bootstrap confidence bands are generated for the multivariate eigenvalues $\hat{\nu}_1, \ldots, \hat{\nu}_M$ as well as for multivariate functional principal components $\hat{\psi}_1, \ldots, \hat{\psi}_M$. The parameter nBootstrap gives the number of bootstrap iterations. In each iteration, the observations are resampled on the level of (multivariate) functions and the whole MFPCA is recalculated. In particular, if the univariate basis depends on the data (FPCA approaches), basis functions and scores are both re-estimated. If the basis functions are fixed (e.g. splines), the scores from the original estimate are used to speed up the calculations. The confidence bands for the eigenfunctions are calculated separately for each element as pointwise percentile bootstrap confidence intervals. Analogously, the confidence bands for the eigenvalues are also percentile bootstrap confidence bands. The significance level(s) can be defined by the bootstrapAlpha parameter, which defaults to 5%. As a result, the MFPCA function returns a list CI of the same length as bootstrapAlpha, containing the lower and upper bounds of the confidence bands for the principal components as multiFunData objects of the same structure as mFData. The confidence bands for the eigenvalues are returned in a list CIvalues, containing the upper and lower bounds for each significance level.

Univariate Expansions: The multivariate functional principal component analysis relies on a univariate basis expansion for each element $X^{(j)}$. The univariate basis representation is calculated using the univDecomp function, that passes the univariate functional observations and optional parameters to the specific function. The univariate decompositions are specified via the uniExpansions argument in the MFPCA function. It is a list of the same length as the mFData object, i.e. having one entry for each element of the multivariate functional data. For each element, uniExpansion must specify at least the type of basis functions to use. Additionally, one may add further parameters. The following basis representations are supported:

- Given basis functions. Then uniExpansions[[j]] = list(type = "given", functions,scores, ortho), where functions is a funData object on the same domain as mFData, containing

the given basis functions. The parameters `scores` and `ortho` are optional. `scores` is an `N` x `K` matrix containing the scores (or coefficients) of the observed functions for the given basis functions, where `N` is the number of observed functions and `K` is the number of basis functions. Note that the scores need to be demeaned to give meaningful results. If scores are not supplied, they are calculated using the given basis functions. The parameter `ortho` specifies whether the given basis functions are orthonormal `orhto = TRUE` or not `ortho = FALSE`. If `ortho` is not supplied, the functions are treated as non-orthogonal. `scores` and `ortho` are not checked for plausibility, use them at your own risk!

- Univariate functional principal component analysis. Then `uniExpansions[[j]] = list(type = "uFPCA", nbasis, pve, npc, makePD)`, where `nbasis,pve,npc,makePD` are parameters passed to the `PACE` function for calculating the univariate functional principal component analysis.
- Basis functions expansions from the package **fda**. Then `uniExpansions[[j]] = list(type = "fda", ...)`, where `...` are passed to `funData2fd`, which heavily builds on `eval.fd`. If **fda** is not available, a warning is thrown.
- Spline basis functions (not penalized). Then `uniExpansions[[j]] = list(type = "splines1D", bs, m, k)`, where `bs,m,k` are passed to the functions `univDecomp` and `univExpansion`. For two-dimensional tensor product splines, use `type = "splines2D"`.
- Spline basis functions (with smoothness penalty). Then `uniExpansions[[j]] = list(type = "splines1Dpen", bs, m,k)`, where `bs,m,k` are passed to the functions `univDecomp` and `univExpansion`. Analogously to the unpenalized case, use `type = "splines2Dpen"` for 2D penalized tensor product splines.
- Cosine basis functions. Use `uniExpansions[[j]] = list(type = "DCT2D",qThresh, parallel)` for functions one two-dimensional domains (images) and `type = "DCT3D"` for 3D images. The calculation is based on the discrete cosine transform (DCT) implemented in the C-library `fftw3`. If this library is not available, the function will throw a warning. `qThresh` gives the quantile for hard thresholding the basis coefficients based on their absolute value. If `parallel = TRUE`, the coefficients for different images are calculated in parallel.

See `univDecomp` and `univExpansion` for details.

Value

An object of class `MFPCAfit` containing the following components:

`values`	A vector of estimated eigenvalues $\hat{\nu}_1, \ldots, \hat{\nu}_M$.
`functions`	A `multiFunData` object containing the estimated multivariate functional principal components $\hat{\psi}_1, \ldots, \hat{\psi}_M$.
`scores`	A matrix of dimension `N` x `M` containing the estimated scores $\hat{\rho}_{im}$.
`vectors`	A matrix representing the eigenvectors associated with the combined univariate score vectors. This might be helpful for calculating predictions.
`normFactors`	The normalizing factors used for calculating the multivariate eigenfunctions and scores. This might be helpful when calculation predictions.
`meanFunction`	A multivariate functional data object, corresponding to the mean function. The MFPCA is applied to the de-meaned functions in `mFData`.
`fit`	A `multiFunData` object containing estimated trajectories for each observation based on the truncated Karhunen-Loeve representation and the estimated scores and eigenfunctions.

CI — A list of the same length as bootstrapAlpha, containing the pointwise lower and upper bootstrap confidence bands for each eigenfunction and each significance level in form of multiFunData objects (only if bootstrap = TRUE).

CIvalues — A list of the same length as bootstrapAlpha, containing the lower and upper bootstrap confidence bands for each eigenvalue and each significance level (only if bootstrap = TRUE).

References

C. Happ, S. Greven (2018): Multivariate Functional Principal Component Analysis for Data Observed on Different (Dimensional) Domains. Journal of the American Statistical Association, 113(522): 649-659. DOI: doi:10.1080/01621459.2016.1273115

C. Happ-Kurz (2020): Object-Oriented Software for Functional Data. Journal of Statistical Software, 93(5): 1-38. DOI: doi:10.18637/jss.v093.i05

See Also

See Happ-Kurz (2020. doi:10.18637/jss.v093.i05) for a general introduction to the **funData** package and it's interplay with **MFPCA**. This file also includes a case study on how to use MFPCA. Useful functions: multiFunData, PACE, univDecomp, univExpansion, summary, plot, scoreplot

Examples

```
oldPar <- par(no.readonly = TRUE)

set.seed(1)

### simulate data (one-dimensional domains)
sim <-  simMultiFunData(type = "split", argvals = list(seq(0,1,0.01), seq(-0.5,0.5,0.02)),
                        M = 5, eFunType = "Poly", eValType = "linear", N = 100)

# MFPCA based on univariate FPCA
uFPCA <- MFPCA(sim$simData, M = 5, uniExpansions = list(list(type = "uFPCA"),
                                                        list(type = "uFPCA")))
summary(uFPCA)
plot(uFPCA) # plot the eigenfunctions as perturbations of the mean
scoreplot(uFPCA) # plot the scores

# MFPCA based on univariate spline expansions
splines <- MFPCA(sim$simData, M = 5, uniExpansions = list(list(type = "splines1D", k = 10),
                                                          list(type = "splines1D", k = 10)),
                 fit = TRUE) # calculate reconstruction, too
summary(splines)
plot(splines) # plot the eigenfunctions as perturbations of the mean
scoreplot(splines) # plot the scores

### Compare estimates to true eigenfunctions
# flip to make results more clear
uFPCA$functions <- flipFuns(sim$trueFuns, uFPCA$functions)
splines$functions <- flipFuns(sim$trueFuns, splines$functions)
```

```
par(mfrow = c(1,2))
plot(sim$trueFuns[[1]], main = "Eigenfunctions\n1st Element", lwd = 2)
plot(uFPCA$functions[[1]], lty = 2, add = TRUE)
plot(splines$functions[[1]], lty = 3, add = TRUE)

plot(sim$trueFuns[[2]], main = "Eigenfunctions\n2nd Element", lwd = 2)
plot(uFPCA$functions[[2]], lty = 2, add = TRUE)
plot(splines$functions[[2]], lty = 3, add = TRUE)
legend("bottomleft", c("True", "uFPCA", "splines"), lty = 1:3, lwd = c(2,1,1))

# Test reconstruction for the first 10 observations
plot(sim$simData[[1]], obs = 1:10, main = "Reconstruction\n1st Element", lwd = 2)
plot(splines$fit[[1]], obs = 1:10, lty = 2, col = 1, add = TRUE)

plot(sim$simData[[2]], obs = 1:10, main = "Reconstruction\n2nd Element", lwd = 2)
plot(splines$fit[[2]], obs = 1:10, lty = 2, col = 1, add = TRUE)
legend("bottomleft", c("True", "Reconstruction"), lty = c(1,2), lwd = c(2,1))

# MFPCA with Bootstrap-CI for the first 2 eigenfunctions
### ATTENTION: Takes long

splinesBoot <- MFPCA(sim$simData, M = 2, uniExpansions = list(list(type = "splines1D", k = 10),
                                                              list(type = "splines1D", k = 10)),
            bootstrap = TRUE, nBootstrap = 100, bootstrapAlpha = c(0.05, 0.1), verbose = TRUE)
summary(splinesBoot)

plot(splinesBoot$functions[[1]], ylim = c(-2,1.5))
plot(splinesBoot$CI$alpha_0.05$lower[[1]], lty = 2, add = TRUE)
plot(splinesBoot$CI$alpha_0.05$upper[[1]], lty = 2, add = TRUE)
plot(splinesBoot$CI$alpha_0.1$lower[[1]], lty = 3, add = TRUE)
plot(splinesBoot$CI$alpha_0.1$upper[[1]], lty = 3, add = TRUE)
abline(h = 0, col = "gray")

plot(splinesBoot$functions[[2]], ylim = c(-1,2.5))
plot(splinesBoot$CI$alpha_0.05$lower[[2]], lty = 2, add = TRUE)
plot(splinesBoot$CI$alpha_0.05$upper[[2]], lty = 2, add = TRUE)
plot(splinesBoot$CI$alpha_0.1$lower[[2]], lty = 3, add = TRUE)
plot(splinesBoot$CI$alpha_0.1$upper[[2]], lty = 3, add = TRUE)
abline(h = 0, col = "gray")
legend("topleft", c("Estimate", "95% CI", "90% CI"), lty = 1:3, lwd = c(2,1,1))

# Plot 95% confidence bands for eigenvalues
plot(1:2, splinesBoot$values, pch = 20, ylim = c(0, 1.5),
     main = "Estimated eigenvalues with 95% CI",
     xlab = "Eigenvalue no.", ylab = "")
arrows(1:2, splinesBoot$CIvalues$alpha_0.05$lower,
       1:2, splinesBoot$CIvalues$alpha_0.05$upper,
       length = 0.05, angle = 90, code = 3)
points(1:2, sim$trueVals[1:2], pch = 20, col = 4)
legend("topright", c("Estimate", "True value"), pch = 20, col = c(1,4))


### simulate data (two- and one-dimensional domains)
```

```
### ATTENTION: Takes long

set.seed(2)
sim <-  simMultiFunData(type = "weighted",
           argvals = list(list(seq(0,1,0.01), seq(-1,1,0.02)), list(seq(-0.5,0.5,0.01))),
                M = list(c(4,5), 20), eFunType = list(c("Fourier", "Fourier"), "Poly"),
                eValType = "exponential", N = 150)

# MFPCA based on univariate spline expansions (for images) and univariate FPCA (for functions)
pca <- MFPCA(sim$simData, M = 10,
             uniExpansions = list(list(type = "splines2D", k = c(10,12)),
                                  list(type = "uFPCA")))
summary(pca)
plot(pca) # plot the eigenfunctions as perturbations of the mean
scoreplot(pca) # plot the scores

### Compare to true eigenfunctions
# flip to make results more clear
pca$functions <- flipFuns(sim$trueFuns[1:10], pca$functions)

par(mfrow = c(5,2), mar = rep(2,4))
for(m in 2:6) # for m = 1, image.plot (used in plot(funData)) produces an error...
{
  plot(sim$trueFuns[[1]], main = paste("True, m = ", m), obs = m)
  plot(pca$functions[[1]], main = paste("Estimate, m = ", m), obs = m)
}

par(mfrow = c(1,1))
plot(sim$trueFuns[[2]], main = "Eigenfunctions (2nd element)", lwd = 2, obs=  1:5)
plot(pca$functions[[2]], lty = 2, add = TRUE, obs=  1:5)
legend("bottomleft", c("True", "MFPCA"), lty = 1:2, lwd = c(2,1))

par(oldPar)
```

`multivExpansion` *Calculate multivariate basis expansion*

Description

Calculate multivariate basis expansion

Usage

```
multivExpansion(multiFuns, scores)
```

Arguments

`multiFuns`	A multivariate functional data object, containing the multivariate basis functions
`scores`	A matrix containing the scores for each observation in each row. The number of columns must match the number of basis functions.

Value

A `multiFunData` object containing the expanded functions for each observation.

`PACE` *Univariate functional principal component analysis by smoothed covariance*

Description

This function calculates a univariate functional principal components analysis by smoothed covariance based on code from `fpca.sc` in package **refund**.

Usage

```
PACE(
  funDataObject,
  predData = NULL,
  nbasis = 10,
  pve = 0.99,
  npc = NULL,
  makePD = FALSE,
  cov.weight.type = "none"
)
```

Arguments

`funDataObject`	An object of class `funData` or `irregFunData` containing the functional data observed, for which the functional principal component analysis is calculated. If the data is sampled irregularly (i.e. of class `irregFunData`), `funDataObject` is transformed to a `funData` object first.
`predData`	An object of class `funData`, for which estimated trajectories based on a truncated Karhunen-Loeve representation should be estimated. Defaults to `NULL`, which implies prediction for the given data.
`nbasis`	An integer, representing the number of B-spline basis functions used for estimation of the mean function and bivariate smoothing of the covariance surface. Defaults to `10` (cf. `fpca.sc` in **refund**).
`pve`	A numeric value between 0 and 1, the proportion of variance explained: used to choose the number of principal components. Defaults to `0.99` (cf. `fpca.sc` in **refund**).
`npc`	An integer, giving a prespecified value for the number of principal components. Defaults to `NULL`. If given, this overrides `pve` (cf. `fpca.sc` in **refund**).
`makePD`	Logical: should positive definiteness be enforced for the covariance surface estimate? Defaults to `FALSE` (cf. `fpca.sc` in **refund**).

`cov.weight.type`
: The type of weighting used for the smooth covariance estimate. Defaults to `"none"`, i.e. no weighting. Alternatively, `"counts"` (corresponds to `fpca.sc` in **refund**) weights the pointwise estimates of the covariance function by the number of observation points.

Value

`mu`
: A `funData` object with one observation, corresponding to the mean function.

`values`
: A vector containing the estimated eigenvalues.

`functions`
: A `funData` object containing the estimated functional principal components.

`scores`
: An matrix of estimated scores for the observations in `funDataObject`. Each row corresponds to the scores of one observation.

`fit`
: A `funData` object containing the estimated trajectories based on the truncated Karhunen-Loeve representation and the estimated scores and functional principal components for `predData` (if this is not `NULL`) or `funDataObject` (if `predData` is `NULL`).

`npc`
: The number of functional principal components: either the supplied `npc`, or the minimum number of basis functions needed to explain proportion `pve` of the variance in the observed curves (cf. `fpca.sc` in **refund**).

`sigma2`
: The estimated measurement error variance (cf. `fpca.sc` in **refund**).

`estVar`
: The estimated smooth variance function of the data.

Warning

This function works only for univariate functional data observed on one-dimensional domains.

See Also

`funData`, `fpcaBasis`, `univDecomp`

Examples

```
oldPar <- par(no.readonly = TRUE)

# simulate data
sim <- simFunData(argvals = seq(-1,1,0.01), M = 5, eFunType = "Poly",
                  eValType = "exponential", N = 100)

# calculate univariate FPCA
pca <- PACE(sim$simData, npc = 5)

# Plot the results
par(mfrow = c(1,2))
plot(sim$trueFuns, lwd = 2, main = "Eigenfunctions")
# flip estimated functions for correct signs
plot(flipFuns(sim$trueFuns,pca$functions), lty = 2, add = TRUE)
legend("bottomright", c("True", "Estimate"), lwd = c(2,1), lty = c(1,2))
```

```
plot(sim$simData, lwd = 2, main = "Some Observations", obs = 1:7)
plot(pca$fit, lty = 2, obs = 1:7, add = TRUE) # estimates are almost equal to true values
legend("bottomright", c("True", "Estimate"), lwd = c(2,1), lty = c(1,2))

par(oldPar)
```

`plot.MFPCAfit` *Plot MFPCA results*

Description

Plots the eigenfunctions as perturbations of the mean (i.e. the mean function plus/minus a constant factor times each eigenfunction separately). If all elements have a one-dimensional domain, the plots can be combined, otherwise the effects of adding and subtracting are shown in two separate rows for each eigenfunction.

Usage

```
## S3 method for class 'MFPCAfit'
plot(
  x,
  plotPCs = seq_len(nObs(x$functions)),
  stretchFactor = NULL,
  combined = FALSE,
  ...
)
```

Arguments

`x`	An object of class `MFPCAfit`, typically returned by the MFPCA function.
`plotPCs`	The principal components to be plotted. Defaults to all components in the `MFPCAfit` object.
`stretchFactor`	The factor by which the principal components are multiplied before adding / subtracting them from the mean function. If `NULL` (the default), the median absolute value of the scores of each eigenfunction is used.
`combined`	Logical: Should the plots be combined? (Works only if all dimensions are one-dimensional). Defaults to `FALSE`.
`...`	Further graphical parameters passed to the plot.funData functions for functional data.

Value

A plot of the principal components as perturbations of the mean.

See Also

`MFPCA`, `plot.funData`

Examples

```
# Simulate multivariate functional data on one-dimensonal domains
# and calculate MFPCA (cf. MFPCA help)
set.seed(1)
# simulate data (one-dimensional domains)
sim <-  simMultiFunData(type = "split", argvals = list(seq(0,1,0.01), seq(-0.5,0.5,0.02)),
                        M = 5, eFunType = "Poly", eValType = "linear", N = 100)
# MFPCA based on univariate FPCA
PCA <- MFPCA(sim$simData, M = 5, uniExpansions = list(list(type = "uFPCA"),
                                                      list(type = "uFPCA")))

# Plot the results
plot(PCA, combined = TRUE) # combine addition and subtraction in one plot
```

`predict.MFPCAfit` *Function prediction based on MFPCA results*

Description

Predict functions based on a truncated multivariate Karhunen-Loeve representation:

$$\hat{x} = \hat{m}u + \sum_{m=1}^{M} \rho_m \hat{\psi}_m$$

with estimated mean function $\hat{\mu}$ and principal components ψ_m. The scores ρ_m can be either estimated (reconstruction of observed functions) or user-defined (construction of new functions).

Usage

```
## S3 method for class 'MFPCAfit'
predict(object, scores = object$scores, ...)
```

Arguments

`object`	An object of class `MFPCAfit`, typically resulting from a MFPCA function call.
`scores`	A matrix containing the score values. The number of columns in `scores` must equal the number of principal components in `object`. Each row represents one curve. Defaults to the estimated scores in `object`, which yields reconstructions of the original data used for the MFPCA calculation.
`...`	Arguments passed to or from other methods.

Value

A `multiFunData` object containing the predicted functions.

See Also

MFPCA

Examples

```
#' # Simulate multivariate functional data on one-dimensonal domains
# and calculate MFPCA (cf. MFPCA help)
set.seed(1)
# simulate data (one-dimensional domains)
sim <-  simMultiFunData(type = "split", argvals = list(seq(0,1,0.01), seq(-0.5,0.5,0.02)),
                        M = 5, eFunType = "Poly", eValType = "linear", N = 100)
# MFPCA based on univariate FPCA
PCA <- MFPCA(sim$simData, M = 5, uniExpansions = list(list(type = "uFPCA"),
                                                      list(type = "uFPCA")))

# Reconstruct the original data
pred <- predict(PCA) # default reconstructs data used for the MFPCA fit

# plot the results: 1st element
plot(sim$simData[[1]]) # original data
plot(pred[[1]], add = TRUE, lty = 2) # reconstruction

# plot the results: 2nd element
plot(sim$simData[[2]]) # original data
plot(pred[[2]], add = TRUE, lty = 2) # reconstruction
```

`print.MFPCAfit` *Print the results of a Multivariate Functional Principal Component Analysis*

Description

A `print` function for class `MFPCAfit`.

Usage

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

Arguments

`x`	An object of class `MFPCAfit`, usually returned by a call to MFPCA.
`...`	Arguments passed to or from other methods.

Value

No return value, called for side effects

print.summary.MFPCAfit

Print summary of a Multivariate Functional Principal Component Analysis

Description

A `print` method for class `MFPCAfit.summary`

Usage

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

Arguments

`x`	An object of class `MFPCAfit.summary`, usually returned by a call to `MFPCAfit.summary`.
`...`	Arguments passed to or from other methods.

Value

No return value, called for side effects

scoreplot

Scoreplot Generic

Description

Redirects to `plot.default`

Usage

```
scoreplot(PCAobject, ...)
```

Arguments

`PCAobject`	A principal component object
`...`	Arguments passed from or to other methods

Value

A bivariate plot of scores.

`scoreplot.MFPCAfit` *Plot the Scores of a Multivariate Functional Principal Component Analysis*

Description

This function plots two scores of a multivariate functional principal component analysis for each observation.

Usage

```
## S3 method for class 'MFPCAfit'
scoreplot(PCAobject, choices = 1:2, scale = FALSE, ...)
```

Arguments

`PCAobject`	An object of class `MFPCAfit`, typically returned by the MFPCA function.
`choices`	The indices of the scores that should by displayed. Defaults to `1:2`, i.e. the scores corresponding to the two leading modes of variability in the data.
`scale`	Logical. Should the scores be scaled by the estimated eigenvalues to emphasize the proportions of total variance explained by the components. Defaults to `FALSE`.
`...`	Further parameters passed to the `plot.default` function.

Value

A bivariate plot of scores.

See Also

MFPCA

Examples

```
# and calculate MFPCA (cf. MFPCA help)
set.seed(1)
# simulate data (one-dimensional domains)
sim <-  simMultiFunData(type = "split", argvals = list(seq(0,1,0.01), seq(-0.5,0.5,0.02)),
                        M = 5, eFunType = "Poly", eValType = "linear", N = 100)
# MFPCA based on univariate FPCA
PCA <- MFPCA(sim$simData, M = 5, uniExpansions = list(list(type = "uFPCA"),
                                                      list(type = "uFPCA")))

# Plot the first two scores
scoreplot(PCA) # no scaling (default)
scoreplot(PCA, scale = TRUE) # scale the scores by the first two eigenvalues
```

`screeplot.MFPCAfit` *Screeplot for Multivariate Functional Principal Component Analysis*

Description

This function plots the proportion of variance explained by the leading eigenvalues in an MFPCA against the number of the principal component.

Usage

```
## S3 method for class 'MFPCAfit'
screeplot(
  x,
  npcs = min(10, length(x$values)),
  type = "lines",
  ylim = NULL,
  main = deparse(substitute(x)),
  ...
)
```

Arguments

`x`	An object of class MFPCAfit, typically returned by a call to `MFPCA`.
`npcs`	The number of eigenvalued to be plotted. Defaults to all eigenvalues if their number is less or equal to 10, otherwise show only the leading first 10 eigenvalues.
`type`	The type of screeplot to be plotted. Can be either `"lines"` or `"barplot"`. Defaults to `"lines"`.
`ylim`	The limits for the y axis. Can be passed either as a vector of length 2 or as `NULL` (default). In the second case, `ylim` is set to `(0,max(pve))`, with `pve` the proportion of variance explained by the principal components to be plotted.
`main`	The title of the plot. Defaults to the variable name of `x`.
`...`	Other graphic parameters passed to `plot.default` (for `type = "lines"`) or `barplot` (for `type = "barplot"`).

Value

A screeplot, showing the decrease of the principal component score.

See Also

`MFPCA`, `screeplot`

Examples

```
# Simulate multivariate functional data on one-dimensonal domains
# and calculate MFPCA (cf. MFPCA help)
set.seed(1)
# simulate data (one-dimensional domains)
sim <-  simMultiFunData(type = "split", argvals = list(seq(0,1,0.01), seq(-0.5,0.5,0.02)),
                        M = 5, eFunType = "Poly", eValType = "linear", N = 100)
# MFPCA based on univariate FPCA
PCA <- MFPCA(sim$simData, M = 5, uniExpansions = list(list(type = "uFPCA"),
                                                      list(type = "uFPCA")))

# screeplot
screeplot(PCA) # default options
screeplot(PCA, npcs = 3, type = "barplot", main= "Screeplot")
```

`summary.MFPCAfit` *Summarize a Multivariate Functional Principal Component Analysis*

Description

A `summary` method for class `MFPCAfit`

Usage

```
## S3 method for class 'MFPCAfit'
summary(object, ...)
```

Arguments

`object`	An object of class `MFPCAfit`, usually returned by a call to MFPCA.
`...`	Arguments passed to or from other methods.

Value

An object of class `summary.MFPCAfit`

`ttv` *Tensor times vector calculation*

Description

Functionality adapted from the MATLAB tensor toolbox (https://www.tensortoolbox.org/).

Usage

```
ttv(A, v, dim)
```

Arguments

A	An array.
v	A list of the same length as dim.
dim	A vector specifying the dimensions for the multiplication.

Details

Let A be a tensor with dimensions $d_1 \times d_2 \times \ldots \times d_p$ and let v be a vector of length d_i. Then the tensor-vector-product along the i-th dimension is defined as

$$B_{j_1 \ldots j_{i-1} j_{i+1} \ldots j_d} = \sum_{i=1}^{d_i} A_{j_1 \ldots j_{i-1} i j_{i+1} \ldots j_d} \cdot v_i.$$

It can hence be seen as a generalization of the matrix-vector product.

The tensor-vector-product along several dimensions between a tensor A and multiple vectors v_1, ..., v_k ($k \leq p$) is defined as a series of consecutive tensor-vector-product along the different dimensions. For consistency, the multiplications are calculated from the dimension of the highest order to the lowest.

Value

An array, the result of the multiplication.

References

B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping, ACM Transactions on Mathematical Software 32(4):635-653, December 2006.

See Also

UMPCA

Examples

```
# create a three-mode tensor
a1 <- seq(0,1, length.out = 10)
a2 <- seq(-1,1, length.out = 20)
a3 <- seq(-pi, pi, length.out = 15)
A <-a1 %o% a2 %o% a3
dim(A)

# multiply along different dimensions
dim(ttv(A = A, v = list(rnorm(10)), dim = 1))
dim(ttv(A = A, v = list(rnorm(20)), dim = 2))
dim(ttv(A = A, v = list(rnorm(15)), dim = 3))

# multiply along more than one dimension
length(ttv(A = A, v = list(rnorm(10), rnorm(15)), dim = c(1,3)))
```

UMPCA *UMPCA: Uncorrelated Multilinear Principle Component Analysis*

Description

This function implements the uncorrelated multilinear principal component analysis for tensors of dimension 2, 3 or 4. The code is basically the same as in the MATLAB toolbox UMPCA by Haiping Lu (Link: https://www.mathworks.com/matlabcentral/fileexchange/35432-uncorrelated-multilinear-princip see also references).

Usage

```
UMPCA(TX, numP)
```

Arguments

TX	The input training data in tensorial representation, the last mode is the sample mode. For Nth-order tensor data, TX is of (N+1)th-order with the (N+1)-mode to be the sample mode. E.g., 30x20x10x100 for 100 samples of size 30x20x10.
numP	The dimension of the projected vector, denoted as P in the paper. It is the number of elementary multilinear projections (EMPs) in tensor-to-vector projection.

Value

Us	The multilinear projection, consisting of numP (P in the paper) elementary multilinear projections (EMPs), each EMP is consisted of N vectors, one in each mode.
TXmean	The mean of the input training samples TX.
odrIdx	The ordering index of projected features in decreasing variance.

Warning

As this algorithm aims more at uncorrelated features than at an optimal reconstruction of the data, hence it might give poor results when used for the univariate decomposition of images in MFPCA.

References

Haiping Lu, K.N. Plataniotis, and A.N. Venetsanopoulos, "Uncorrelated Multilinear Principal Component Analysis for Unsupervised Multilinear Subspace Learning", IEEE Transactions on Neural Networks, Vol. 20, No. 11, Page: 1820-1836, Nov. 2009.

Examples

```
set.seed(12345)

 # define "true" components
 a <- sin(seq(-pi, pi, length.out = 100))
 b <- exp(seq(-0.5, 1, length.out = 150))

 # simulate tensor data
 X <- a %o% b %o% rnorm(80, sd = 0.5)

 # estimate one component
 UMPCAres <- UMPCA(X, numP = 1)

 # plot the results and compare to true values
 plot(UMPCAres$Us[[1]][,1])
 points(a/sqrt(sum(a^2)), pch = 20) # eigenvectors are defined only up to a sign change!
 legend("topright", legend = c("True", "Estimated"), pch = c(20, 1))

 plot(UMPCAres$Us[[2]][,1])
 points(b/sqrt(sum(b^2)), pch = 20)
 legend("topleft", legend = c("True", "Estimated"), pch = c(20, 1))
```

`univDecomp` *Univariate basis decomposition*

Description

This function calculates a univariate basis decomposition for a (univariate) functional data object.

Usage

```
univDecomp(type, funDataObject, ...)
```

Arguments

`type`	A character string, specifying the basis for which the decomposition is to be calculated.
`funDataObject`	A `funData` object, representing the (univariate) functional data samples.
`...`	Further parameters, passed to the function for the particular basis to use.

Details

Functional data $X_i(t)$ can often be approximated by a linear combination of basis functions $b_k(t)$

$$X_i(t) = \sum_{k=1}^{K} \theta_{ik} b_k(t), i = 1, \ldots, N.$$

The basis functions may be prespecified (such as spline basis functions or Fourier bases) or can be estimated from the data (e.g. by functional principal component analysis) and are the same for all observations $X_1(t), \ldots, X_n(t)$. The coefficients (or scores) θ_{ik} reflect the weight of each basis function $b_k(t)$ for the observed function $X_i(t)$ and can be used to characterize the individual observations.

Value

`scores`	A matrix of scores (coefficients) for each observation based on the prespecified basis functions.
`B`	A matrix containing the scalar products of the basis functions. Can be `NULL` if the basis functions are orthonormal.
`ortho`	Logical. If `TRUE`, the basis functions are all orthonormal.
`functions`	A functional data object, representing the basis functions. Can be `NULL` if the basis functions are not estimated from the data, but have a predefined form. See Details.

Warning

The options `type = "DCT2D"` and `type = "DCT3D"` have not been tested with ATLAS/MKL/OpenBLAS.

See Also

`MFPCA`, `univExpansion`, `fpcaBasis`, `splineBasis1D`, `splineBasis1Dpen`, `splineBasis2D`, `splineBasis2Dpen`, `umpcaBasis`, `fcptpaBasis`, `fdaBasis`, `dctBasis2D`, `dctBasis3D`

Examples

```
# generate some data
dat <- simFunData(argvals = seq(0,1,0.01), M = 5,
                  eFunType = "Poly", eValType = "linear", N = 100)$simData

# decompose the data in univariate functional principal components...
decFPCA <- univDecomp(type = "uFPCA", funDataObject = dat, npc = 5)
str(decFPCA)

# or in splines (penalized)
decSplines <- univDecomp(type = "splines1Dpen", funDataObject = dat) # use mgcv's default params
str(decSplines)
```

`univExpansion` *Calculate a univariate basis expansion*

Description

This function calculates a univariate basis expansion based on given scores (coefficients) and basis functions.

Usage

```
univExpansion(
  type,
  scores,
  argvals = ifelse(!is.null(functions), functions@argvals, NULL),
  functions,
  params = NULL
)
```

Arguments

type	A character string, specifying the basis for which the decomposition is to be calculated.
scores	A matrix of scores (coefficients) for each observation based on the given basis functions.
argvals	A list, representing the domain of the basis functions. If functions is not NULL, the usual default is functions@argvals. See funData and the underlying expansion functions for details.
functions	A functional data object, representing the basis functions. Can be NULL if the basis functions are not estimated from observed data, but have a predefined form. See Details.
params	A list containing the parameters for the particular basis to use.

Details

This function calculates functional data $X_i(t), i = 1 \dots N$ that is represented as a linear combination of basis functions $b_k(t)$

$$X_i(t) = \sum_{k=1}^{K} \theta_{ik} b_k(t), i = 1, \dots, N.$$

The basis functions may be prespecified (such as spline basis functions or Fourier bases) or can be estimated from observed data (e.g. by functional principal component analysis). If type = "default" (i.e. a linear combination of arbitrary basis functions is to be calculated), both scores and basis functions must be supplied.

Value

An object of class funData with N observations on argvals, corresponding to the linear combination of the basis functions.

Warning

The options type = "spline2Dpen", type = "DCT2D" and type = "DCT3D" have not been tested with ATLAS/MKL/OpenBLAS.

See Also

MFPCA, splineFunction1D, splineFunction2D, splineFunction2Dpen, dctFunction2D, dctFunction3D, expandBasisFunction

Examples

```
oldPar <- par(no.readonly = TRUE)
par(mfrow = c(1,1))

set.seed(1234)

### Spline basis ###
# simulate coefficients (scores) for N = 10 observations and K = 8 basis functions
N <- 10
K <- 8
scores <- t(replicate(n = N, rnorm(K, sd = (K:1)/K)))
dim(scores)

# expand spline basis on [0,1]
funs <- univExpansion(type = "splines1D", scores = scores, argvals = list(seq(0,1,0.01)),
                      functions = NULL, # spline functions are known, need not be given
                      params = list(bs = "ps", m = 2, k = K)) # params for mgcv

plot(funs, main = "Spline reconstruction")

### PCA basis ###
# simulate coefficients (scores) for N = 10 observations and K = 8 basis functions
N <- 10
K <- 8

scores <- t(replicate(n = N, rnorm(K, sd = (K:1)/K)))
dim(scores)

# Fourier basis functions as eigenfunctions
eFuns <- eFun(argvals = seq(0,1,0.01), M = K, type = "Fourier")

# expand eigenfunction basis
funs <-  univExpansion(type = "uFPCA", scores = scores,
                       argvals = NULL, # use argvals of eFuns (default)
                       functions = eFuns)

plot(funs, main = "PCA reconstruction")

par(oldPar)
```

Index