Package 'fda'

December 16, 2020

Version 5.1.9

Date 2020-12-16

Title Functional Data Analysis

Author J. O. Ramsay <ramsay@psych.mcgill.ca> [aut,cre],
Spencer Graves <spencer.graves@effectivedefense.org> [ctb],
Giles Hooker <gjh27@cornell.edu> [ctb]

Maintainer J. O. Ramsay <ramsay@psych.mcgill.ca>

Depends R (>= 3.5), splines, Matrix, fds

Suggests deSolve, lattice

Description These functions were developed to support functional data analysis as described in Ramsay, J. O. and Silverman, B. W. (2005) Functional Data Analysis. New York: Springer and in Ramsay, J. O., Hooker, Giles, and Graves, Spencer (2009). Functional Data Analysis with R and Matlab (Springer). The package includes data sets and script files working many examples including all but one of the 76 figures in this latter book. Matlab versions are available by ftp from <http://www.psych.mcgill.ca/misc/fda/downloads/FDAfuns/>.

License GPL (>= 2)

URL http://www.functionaldata.org

LazyData true

NeedsCompilation no

Repository CRAN

Date/Publication 2020-12-16 19:40:02 UTC

R topics documented:

`fda-package` *Functional Data Analysis in R*

Description

Functions and data sets companion to Ramsay, J. O.; Hooker, Giles; and Graves, Spencer (2010) Functional Data Analysis with R and Matlab, plus Ramsay, J. O., and Silverman, B. W. (2006) Functional Data Analysis, 2nd ed. and (2002) Applied Functional Data Analysis (Springer). This includes finite bases approximations (such as splines and Fourier series) to functions fit to data smoothing on the integral of the squared deviations from an arbitrary differential operator.

Details

Package:	fda
Type:	Package
Version:	2.2.6
Date:	2011-02-03
License:	GPL-2
LazyLoad:	yes

Author(s)

J. O. Ramsay, <ramsay@psych.mcgill.ca>, Hadley Wickham <h.wickham@gmail.com>, Spencer Graves <spencer.graves@prodsyse.com>, Giles Hooker <gjh27@cornell.edu>

Maintainer: J. O. Ramsay <ramsay@psych.mcgill.ca>

References

Ramsay, J. O.; Hooker, Giles; and Graves, Spencer (2010) *Functional Data Analysis with R and Matlab*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

Examples

```
##
## As noted in the Preface to Ramsay, Hooker and Graves (p. v),
## the fda package includes scripts to reproduce all but one of the
## figures in the book.
##
## These figures can be found and run as follows:
##
## Not run:
scriptsDir <- system.file('scripts', package='fda')
Rscripts <- dir(scriptsDir, full.names=TRUE, pattern='R$')
fdarm <- grep('fdarm', Rscripts, value=TRUE)
chapters <- length(fdarm)
# NOTE:  If R fails in any of these scripts,
# this for loop will not end normally,
# and the abnormal termination will be displayed:
for(ch in 1:chapters){
  cat('Running', fdarm[ch], '\n')
  invisible(source(fdarm[ch]))
}

## End(Not run)
```

`AmpPhaseDecomp` *Decomposition for Amplitude and Phase Variation*

Description

Registration is the process of aligning peaks, valleys and other features in a sample of curves. Once the registration has taken place, this function computes two mean squared error measures, one for amplitude variation, and the other for phase variation. It also computes a squared multiple correlation index of the amount of variation in the unregistered functions is due to phase.

Usage

```
AmpPhaseDecomp(xfd, yfd, hfd, rng=xrng)
```

Arguments

xfd	a functional data object containing the unregistered curves.
yfd	a functional data object containing the registered curves.
hfd	a functional data object containing the strictly monotone warping functions $h(t)$. This is typically returned by the functions landmarkreg and register.fd.
rng	a vector of length 2 specifying a range of values over which the decomposition is to be computed. Both values must be within the range of the functional data objects in the argument. By default the whole range of the functional data objects is used.

Details

The decomposition can yield negative values for MS.phas if the registration does not improve the alignment of the curves, or if used to compare two registration processes based on different principles, such as is the case for functions landmarkreg and register.fd.

Value

a named list with the following components:

MS.amp	the mean squared error for amplitude variation.
MS.phas	the mean squared error for phase variation.
RSQR	the squared correlation measure of the proportion of the total variation that is due to phase variation.
C	a constant required for the decomposition. Its value is one if the derivatives the warping functions are independent of the squared registered functions.

See Also

landmarkreg, register.fd, smooth.morph

Examples

```
#See the analysis for the growth data in the examples.
```

---

arithmetic.basisfd *Arithmatic on functional basis objects*

Description

Arithmatic on functional basis objects

Usage

```
## S3 method for class 'basisfd'
basis1 == basis2
```

Arguments

`basis1, basis2` functional basis object

Value

basisobj1 == basisobj2 returns a logical scalar.

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

`basisfd`, `basisfd.product` `arithmetic.fd`

`arithmetic.fd` *Arithmetic on functional data ('fd') objects*

Description

Arithmetic on functional data objects

Usage

```
## S3 method for class 'fd'
e1 + e2
## S3 method for class 'fd'
e1 - e2
## S3 method for class 'fd'
e1 * e2
plus.fd(e1, e2, basisobj=NULL)
minus.fd(e1, e2, basisobj=NULL)
times.fd(e1, e2, basisobj=NULL)
```

Arguments

`e1, e2`	object of class 'fd' or a numeric vector. Note that 'e1+e2' will dispatch to plus.fd(e1, e2) only if e1 has class 'fd'. Similarly, 'e1-e2' or 'e1*e2' will dispatch to minus.fd(e1, e2) or time.fd(e1, e2), respectively, only if e1 is of class 'fd'.
`basisobj`	reference basis; defaults to e1[['basis']] * e2[['basis']]; ignored for `plus.fd` and `minus.fd`.

Value

A function data object corresponding to the pointwise sum, difference or product of e1 and e2.

If both arguments are functional data objects, the bases are the same, and the coefficient matrices are the same dims, the indicated operation is applied to the coefficient matrices of the two objects. In other words, e1+e2 is obtained for this case by adding the coefficient matrices from e1 and e2.

If e1 or e2 is a numeric scalar, that scalar is applied to the coefficient matrix of the functional data object.

If either e1 or e2 is a numeric vector, it must be the same length as the number of replicated functional observations in the other argument.

When both arguments are functional data objects, they need not have the same bases. However, if they don't have the same number of replicates, then one of them must have a single replicate. In the second case, the singleton function is replicated to match the number of replicates of the other function. In either case, they must have the same number of functions. When both arguments are functional data objects, and the bases are not the same, the basis used for the sum is constructed to be of higher dimension than the basis for either factor according to rules described in function TIMES for two basis objects.

See Also

basisfd, basisfd.product, exponentiate.fd

Examples

```
##
## add a parabola to itself
##
bspl4 <- create.bspline.basis(nbasis=4)
parab4.5 <- fd(c(3, -1, -1, 3)/3, bspl4)

coef2 <- matrix(c(6, -2, -2, 6)/3, 4)
dimnames(coef2) <- list(NULL, 'reps 1')

all.equal(coef(parab4.5+parab4.5), coef2)


##
## Same example with interior knots at 1/3 and 1/2
##
bspl5.3 <- create.bspline.basis(breaks=c(0, 1/3, 1))
plot(bspl5.3)
x. <- seq(0, 1, .1)
para4.5.3 <- smooth.basis(x., 4*(x.-0.5)^2, fdParobj=bspl5.3)$fd
plot(para4.5.3)

bspl5.2 <- create.bspline.basis(breaks=c(0, 1/2, 1))
plot(bspl5.2)
para4.5.2 <- smooth.basis(x., 4*(x.-0.5)^2, fdParobj=bspl5.2)$fd
plot(para4.5.2)
```

```
#str(para4.5.3+para4.5.2)

coef2. <- matrix(0, 9, 1)
dimnames(coef2.) <- list(NULL, 'rep1')

all.equal(coef(para4.5.3-para4.5.2), coef2.)


##
## product
##
quart <- para4.5.3*para4.5.2







# norder(quart) = norder(para4.5.2)+norder(para4.5.3)-1 = 7

norder(quart) == (norder(para4.5.2)+norder(para4.5.3)-1)


# para4.5.2 with knot at 0.5 and para4.5.3 with knot at 1/3
# both have (2 end points + 1 interior knot) + norder-2
#     = 5 basis functions
# quart has (2 end points + 2 interior knots)+norder-2
#     = 9 basis functions
# coefficients look strange because the knots are at
# (1/3, 1/2) and not symmetrical


all.equal(as.numeric(coef(quart)),
0.1*c(90, 50, 14, -10, 6, -2, -2, 30, 90)/9)


plot(para4.5.3*para4.5.2) # quartic, not parabolic ...

##
## product with Fourier bases
##
f3 <- fd(c(0,0,1), create.fourier.basis())
f3^2 # number of basis functions = 7?

##
## fd+numeric
##
coef1 <- matrix(c(6, 2, 2, 6)/3, 4)
dimnames(coef1) <- list(NULL, 'reps 1')

all.equal(coef(parab4.5+1), coef1)
```

```
all.equal(1+parab4.5, parab4.5+1)


##
## fd-numeric
##
coefneg <- matrix(c(-3, 1, 1, -3)/3, 4)
dimnames(coefneg) <- list(NULL, 'reps 1')

all.equal(coef(-parab4.5), coefneg)


plot(parab4.5-1)

plot(1-parab4.5)
```

---

as.array3 *Reshape a vector or array to have 3 dimensions.*

Description

Coerce a vector or array to have 3 dimensions, preserving dimnames if feasible. Throw an error if length(dim(x)) > 3.

Usage

```
as.array3(x)
```

Arguments

x	A vector or array.

Details

1. dimx <- dim(x); ndim <- length(dimx)
2. if(ndim==3)return(x).
3. if(ndim>3)stop.
4. x2 <- as.matrix(x)
5. dim(x2) <- c(dim(x2), 1)
6. xnames <- dimnames(x)
7. if(is.list(xnames))dimnames(x2) <- list(xnames[[1]], xnames[[2]], NULL)

Value

A 3-dimensional array with names matching x

Author(s)

Spencer Graves

See Also

dim, dimnames checkDims3

Examples

```
##
## vector -> array
##
as.array3(c(a=1, b=2))

##
## matrix -> array
##
as.array3(matrix(1:6, 2))
as.array3(matrix(1:6, 2, dimnames=list(letters[1:2], LETTERS[3:5])))

##
## array -> array
##
as.array3(array(1:6, 1:3))

##
## 4-d array
##
## Not run:
as.array3(array(1:24, 1:4))
Error in as.array3(array(1:24, 1:4)) :
  length(dim(array(1:24, 1:4)) = 4 > 3

## End(Not run)
```

---

as.fd *Convert a spline object to class 'fd'*

Description

Translate a spline object of another class into the Functional Data (class fd) format.

Usage

```
as.fd(x, ...)
## S3 method for class 'fdSmooth'
as.fd(x, ...)
## S3 method for class 'function'
as.fd(x, ...)
```

```
## S3 method for class 'smooth.spline'
as.fd(x, ...)
```

Arguments

x	an object to be converted to class fd.
...	optional arguments passed to specific methods, currently unused.

Details

The behavior depends on the class and nature of x.

- as.fd.fdSmoothextract the fd component
- as.fd.function Create an fd object from a function of the form created by splinefun. This will translate method = 'fmn' and 'natural' but not 'periodic': 'fmn' splines are isomorphic to standard B-splines with coincident boundary knots, which is the basis produced by create.bspline.basis. 'natural' splines occupy a subspace of this space, with the restriction that the second derivative at the end points is zero (as noted in the Wikipedia spline article). 'periodic' splines do not use coincident boundary knots and are not currently supported in fda; instead, fda uses finite Fourier bases for periodic phenomena.
- as.fd.smooth.spline Create an fd object from a smooth.spline object.

Author(s)

Spencer Graves

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

spline entry in *Wikipedia* https://en.wikipedia.org/wiki/Spline_(mathematics)

See Also

fd splinefun

Examples

```
##
## as.fd.fdSmooth
##
girlGrowthSm <- with(growth,
  smooth.basisPar(argvals=age, y=hgtf, lambda=0.1))
girlGrowth.fd <- as.fd(girlGrowthSm)

##
## as.fd.function(splinefun(...), ...)
```

```
##
x2 <- 1:7
y2 <- sin((x2-0.5)*pi)
f <- splinefun(x2, y2)
fd. <- as.fd(f)
x. <- seq(1, 7, .02)
fx. <- f(x.)
fdx. <- eval.fd(x., fd.)

# range(y2, fx., fdx.) generates an error 2012.04.22

rfdx <- range(fdx.)

plot(range(x2), range(y2, fx., rfdx), type='n')
points(x2, y2)
lines(x., sin((x.-0.5)*pi), lty='dashed')
lines(x., f(x.), col='blue')
lines(x., eval.fd(x., fd.), col='red', lwd=3, lty='dashed')
# splinefun and as.fd(splineful(...)) are close
# but quite different from the actual function
# apart from the actual 7 points fitted,
# which are fitted exactly
# ... and there is no information in the data
# to support a better fit!

# Translate also a natural spline
fn <- splinefun(x2, y2, method='natural')
fn. <- as.fd(fn)
lines(x., fn(x.), lty='dotted', col='blue')
lines(x., eval.fd(x., fn.), col='green', lty='dotted', lwd=3)

## Not run:
# Will NOT translate a periodic spline
fp <- splinefun(x, y, method='periodic')
as.fd(fp)
#Error in as.fd.function(fp) :
#  x (fp)  uses periodic B-splines, and as.fd is programmed
#   to translate only B-splines with coincident boundary knots.

## End(Not run)

##
## as.fd.smooth.spline
##
cars.spl <- with(cars, smooth.spline(speed, dist))
cars.fd <- as.fd(cars.spl)

plot(dist~speed, cars)
lines(cars.spl)
sp. <- with(cars, seq(min(speed), max(speed), len=101))
d. <- eval.fd(sp., cars.fd)
lines(sp., d., lty=2, col='red', lwd=3)
```

as.POSIXct1970 as.POXIXct *for number of seconds since the start of 1970.*

Description

as.POSIXct.numeric requires origin to be specified. This assumes that it is the start of 1970.

Usage

```
as.POSIXct1970(x, tz="GMT", ...)
```

Arguments

x	a numeric vector of times in seconds since the start of 1970. (If x is not numeric, call as.POSIXct.)
tz	A timezone specification to be used for the conversion, if one is required. System-specific (see time zones), but "" is the current timezone, and "GMT" is UTC (Universal Time, Coordinated).
...	optional arguments to pass to as.POSIXct.

Details

o1970 <- strptime('1970-01-01', ' o1970. <- as.POSIXct(o1970), as.POSIXct(x, origin=o1970.)

Value

Returns a vector of class POSIXct.

Author(s)

Spencer Graves

See Also

as.POSIXct, ISOdate, strptime

Examples

```
sec <- c(0, 1, 60, 3600, 24*3600, 31*24*3600, 365*24*3600)
Sec <- as.POSIXct1970(sec)

all.equal(sec, as.numeric(Sec))
```

`axisIntervals` *Mark Intervals on a Plot Axis*

Description

Adds an axis (axisintervals) or two axes (axesIntervals) to the current plot with tick marks delimiting interval described by labels

Usage

```
axisIntervals(side=1, atTick1=fda::monthBegin.5, atTick2=fda::monthEnd.5,
      atLabels=fda::monthMid, labels=month.abb, cex.axis=0.9, ...)
axesIntervals(side=1:2, atTick1=fda::monthBegin.5, atTick2=fda::monthEnd.5,
      atLabels=fda::monthMid, labels=month.abb, cex.axis=0.9, las=1, ...)
```

Arguments

`side`	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
`atTick1`	the points at which tick-marks marking the starting points of the intervals are to be drawn. This defaults to 'monthBegin.5' to mark monthly periods for an annual cycle. These are constructed by calling axis(side, at=atTick1, labels=FALSE, ...). For more detail on this, see 'axis'.
`atTick2`	the points at which tick-marks marking the ends of the intervals are to be drawn. This defaults to 'monthEnd.5' to mark monthly periods for an annual cycle. These are constructed by calling axis(side, at=atTick2, labels=FALSE, ...). Use atTick2=NA to rely only on atTick1. For more detail on this, see 'axis'.
`atLabels`	the points at which 'labels' should be typed. These are constructed by calling axis(side, at=atLabels, tick=FALSE, ...). For more detail on this, see 'axis'.
`labels`	Labels to be typed at locations 'atLabels'. This is accomplished by calling axis(side, at=atLabels, labels=labels, tick=FALSE, ...). For more detail on this, see 'axis'.
`cex.axis`	Character expansion (magnification) used for axis annotations ('labels' in this function call) relative to the current setting of 'cex'. For more detail, see 'par'.
`las`	line axis style; see `par`.
`...`	additional arguments passed to `axis`.

Value

The value from the third (labels) call to 'axis'. This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

`axesIntervals` calls `axisIntervals(side[1],...)` then `axis(side[2],...)`.

Side Effects

An axis is added to the current plot.

Author(s)

Spencer Graves

See Also

axis, par monthBegin.5 monthEnd.5 monthMid month.abb monthLetters

Examples

```
daybasis65 <- create.fourier.basis(c(0, 365), 65)

daytempfd <- with(CanadianWeather, smooth.basis(
       day.5,  dailyAv[,,"Temperature.C"],
       daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd )

with(CanadianWeather, plotfit.fd(
      dailyAv[,,"Temperature.C"], argvals=day.5,
          daytempfd, index=1, titles=place, axes=FALSE) )
# Label the horizontal axis with the month names
axisIntervals(1)
axis(2)
# Depending on the physical size of the plot,
# axis labels may not all print.
# In that case, there are 2 options:
# (1) reduce 'cex.lab'.
# (2) Use different labels as illustrated by adding
#     such an axis to the top of this plot

with(CanadianWeather, plotfit.fd(
      dailyAv[,,"Temperature.C"], argvals=day.5,
          daytempfd, index=1, titles=place, axes=FALSE) )
# Label the horizontal axis with the month names
axesIntervals()

axisIntervals(3, labels=monthLetters, cex.lab=1.2, line=-0.5)
# 'line' argument here is passed to 'axis' via '...'
```

---

basisfd.product *Product of two basisfd objects*

Description

pointwise multiplication method for basisfd class

Usage

```
## S3 method for class 'basisfd'
basisobj1 * basisobj2
```

Arguments

basisobj1, basisobj2
: objects of class basisfd

Details

TIMES for (two basis objects sets up a basis suitable for expanding the pointwise product of two functional data objects with these respective bases. In the absence of a true product basis system in this code, the rules followed are inevitably a compromise: (1) if both bases are B-splines, the norder is the sum of the two orders - 1, and the breaks are the union of the two knot sequences, each knot multiplicity being the maximum of the multiplicities of the value in the two break sequences. That is, no knot in the product knot sequence will have a multiplicity greater than the multiplicities of this value in the two knot sequences. The rationale this rule is that order of differentiability of the product at eachy value will be controlled by whichever knot sequence has the greater multiplicity. In the case where one of the splines is order 1, or a step function, the problem is dealt with by replacing the original knot values by multiple values at that location to give a discontinuous derivative. (2) if both bases are Fourier bases, AND the periods are the the same, the product is a Fourier basis with number of basis functions the sum of the two numbers of basis fns. (3) if only one of the bases is B-spline, the product basis is B-spline with the same knot sequence and order two higher. (4) in all other cases, the product is a B-spline basis with number of basis functions equal to the sum of the two numbers of bases and equally spaced knots.

See Also

basisfd

Examples

```
f1 <- create.fourier.basis()
f1.2 <- f1*f1

all.equal(f1.2, create.fourier.basis(nbasis=5))
```

---

bifd *Create a bivariate functional data object*

Description

This function creates a bivariate functional data object, which consists of two bases for expanding a functional data object of two variables, s and t, and a set of coefficients defining this expansion. The bases are contained in "basisfd" objects.

Usage

```
bifd (coef=matrix(0,2,1), sbasisobj=create.bspline.basis(),
      tbasisobj=create.bspline.basis(), fdnames=defaultnames)
```

Arguments

coef	a two-, three-, or four-dimensional array containing coefficient values for the expansion of each set of bivariate function values=terms of a set of basis function values If 'coef' is two dimensional, this implies that there is only one variable and only one replication. In that case, the first and second dimensions correspond to the basis functions for the first and second argument, respectively. If 'coef' is three dimensional, this implies that there are multiple replicates on only one variable. In that case, the first and second dimensions correspond to the basis functions for the first and second argument, respectively, and the third dimension corresponds to replications. If 'coef' has four dimensions, the fourth dimension corresponds to variables.
sbasisobj	a functional data basis object for the first argument s of the bivariate function.
tbasisobj	a functional data basis object for the second argument t of the bivariate function.
fdnames	A list of length 4 containing dimnames for 'coefs' if it is a 4-dimensional array. If it is only 2- or 3-dimensional, the later components of fdnames are not applied to 'coefs'. In any event, the components of fdnames describe the following: (1) The row of 'coefs' corresponding to the bases in sbasisobj. Defaults to sbasisobj[["names"]] if non-null and of the proper length, or to existing dimnames(coefs)[[1]] if non-null and of the proper length, and to 's1', 's2', ..., otherwise. (2) The columns of 'coefs' corresponding to the bases in tbasisobj. Defaults to tbasisobj[["names"]] if non-null and of the proper length, or to existing dimnames(coefs)[[2]] if non-null and of the proper length, and to 't1', 't2', ..., otherwise. (3) The replicates. Defaults to dimnames(coefs)[[3]] if non-null and of the proper length, and to 'rep1', ..., otherwise. (4) Variable names. Defaults to dimnames(coefs)[[4]] if non-null and of the proper length, and to 'var1', ..., otherwise.

Value

A bivariate functional data object = a list of class 'bifd' with the following components:

coefs	the input 'coefs' possible with dimnames from dfnames if provided or from sbasisobj$names and tbasisobsj$names
sbasisobj	a functional data basis object for the first argument s of the bivariate function.
tbasisobj	a functional data basis object for the second argument t of the bivariate function.
bifdnames	a list of length 4 giving names for the dimensions of coefs, with one or two unused lists of names if length(dim(coefs)) is only two or one, respectively.

Author(s)

Spencer Graves

See Also

basisfd objAndNames

Examples

```
Bspl2 <- create.bspline.basis(nbasis=2, norder=1)
Bspl3 <- create.bspline.basis(nbasis=3, norder=2)

(bBspl2.3 <- bifd(array(1:6, dim=2:3), Bspl2, Bspl3))
str(bBspl2.3)
```

---

bifdPar *Define a Bivariate Functional Parameter Object*

Description

Functional parameter objects are used as arguments to functions that estimate functional parameters, such as smoothing functions like smooth.basis. A bivariate functional parameter object supplies the analogous information required for smoothing bivariate data using a bivariate functional data object $x(s,t)$. The arguments are the same as those for fdPar objects, except that two linear differential operator objects and two smoothing parameters must be applied, each pair corresponding to one of the arguments s and t of the bivariate functional data object.

Usage

```
bifdPar(bifdobj, Lfdobjs=int2Lfd(2), Lfdobjt=int2Lfd(2), lambdas=0, lambdat=0,
      estimate=TRUE)
```

Arguments

bifdobj	a bivariate functional data object.
Lfdobjs	either a nonnegative integer or a linear differential operator object for the first argument s. If NULL, Lfdobjs depends on bifdobj[['sbasis']][['type']]: • bspline Lfdobjs <- int2Lfd(max(0, norder-2)), where norder = norder(bifdobj[['sbasis']]). • fourier Lfdobjs = a harmonic acceleration operator: Lfdobj <-vec2Lfd(c(0,(2*pi/diff(rngs))^2,0),rngs) where rngs = bifdobj[['sbasis']][['rangeval']]. • anything elseLfdobj <- int2Lfd(0)
Lfdobjt	either a nonnegative integer or a linear differential operator object for the first argument t. If NULL, Lfdobjt depends on bifdobj[['tbasis']][['type']]: • bspline Lfdobj <- int2Lfd(max(0, norder-2)), where norder = norder(bifdobj[['tbasis']]).

- fourier Lfdobj = a harmonic acceleration operator:
  `Lfdobj <-vec2Lfd(c(0,(2*pi/diff(rngt))^2,0),rngt)`
  where rngt = bifdobj[['tbasis']][['rangeval']].
- anything elseLfdobj <- int2Lfd(0)

`lambdas` a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter $x(s,t)$ as a function of s..

`lambdat` a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter $x(s,t)$ as a function of t..

`estimate` not currently used.

Value

a bivariate functional parameter object (i.e., an object of class `bifdPar`), which is a list with the following components:

`bifd` a functional data object (i.e., with class `bifd`)

`Lfdobjs` a linear differential operator object (i.e., with class `Lfdobjs`)

`Lfdobjt` a linear differential operator object (i.e., with class `Lfdobjt`)

`lambdas` a nonnegative real number

`lambdat` a nonnegative real number

`estimate` not currently used

Source

Ramsay, James O., Hooker, Giles, and Graves, Spencer (2009) *Functional Data Analysis in R and Matlab*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York

See Also

`linmod`

Examples

```
#See the prediction of precipitation using temperature as
#the independent variable in the analysis of the daily weather
#data, and the analysis of the Swedish mortality data.
```

bsplinepen *B-Spline Penalty Matrix*

Description

Computes the matrix defining the roughness penalty for functions expressed in terms of a B-spline basis.

Usage

```
bsplinepen(basisobj, Lfdobj=2, rng=basisobj$rangeval, returnMatrix=FALSE)
```

Arguments

basisobj	a B-spline basis object.
Lfdobj	either a nonnegative integer or a linear differential operator object.
rng	a vector of length 2 defining range over which the basis penalty is to be computed.
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Details

A roughness penalty for a function $x(t)$ is defined by integrating the square of either the derivative of $x(t)$ or, more generally, the result of applying a linear differential operator L to it. The most common roughness penalty is the integral of the square of the second derivative, and this is the default. To apply this roughness penalty, the matrix of inner products of the basis functions (possibly after applying the linear differential operator to them) defining this function is necessary. This function just calls the roughness penalty evaluation function specific to the basis involved.

Value

a symmetric matrix of order equal to the number of basis functions defined by the B-spline basis object. Each element is the inner product of two B-spline basis functions after applying the derivative or linear differential operator defined by Lfdobj.

Examples

```
##
## bsplinepen with only one basis function
##
bspl1.1 <- create.bspline.basis(nbasis=1, norder=1)
pen1.1 <- bsplinepen(bspl1.1, 0)

##
## bspline pen for a cubic spline with knots at seq(0, 1, .1)
##
```

```
basisobj <- create.bspline.basis(c(0,1),13)
#  compute the 13 by 13 matrix of inner products of second derivatives
penmat <- bsplinepen(basisobj)

##
## with rng of class Date or POSIXct
##
# Date
invasion1 <- as.Date('1775-09-04')
invasion2 <- as.Date('1812-07-12')
earlyUS.Canada <- c(invasion1, invasion2)
BspInvade1 <- create.bspline.basis(earlyUS.Canada)
Binvadmat <- bsplinepen(BspInvade1)

# POSIXct
AmRev.ct <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev1.ct <- create.bspline.basis(AmRev.ct)
Brevmat <- bsplinepen(BspRev1.ct)
```

`bsplineS` *B-spline Basis Function Values*

Description

Evaluates a set of B-spline basis functions, or a derivative of these functions, at a set of arguments.

Usage

```
bsplineS(x, breaks, norder=4, nderiv=0, returnMatrix=FALSE)
```

Arguments

`x`	A vector of argument values at which the B-spline basis functions are to be evaluated.
`breaks`	A strictly increasing set of break values defining the B-spline basis. The argument values `x` should be within the interval spanned by the break values.
`norder`	The order of the B-spline basis functions. The order less one is the degree of the piece-wise polynomials that make up any B-spline function. The default is order 4, meaning piece-wise cubic.
`nderiv`	A nonnegative integer specifying the order of derivative to be evaluated. The derivative must not exceed the order. The default derivative is 0, meaning that the basis functions themselves are evaluated.
`returnMatrix`	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Value

a matrix of function values. The number of rows equals the number of arguments, and the number of columns equals the number of basis functions.

Examples

```
# Minimal example:  A B-spline of order 1 (i.e., a step function)
# with 0 interior knots:
bS <- bsplineS(seq(0, 1, .2), 0:1, 1, 0)

# check

all.equal(bS, matrix(1, 6))


#  set up break values at 0.0, 0.2,..., 0.8, 1.0.
breaks <- seq(0,1,0.2)
#  set up a set of 11 argument values
x <- seq(0,1,0.1)
#  the order willl be 4, and the number of basis functions
#  is equal to the number of interior break values (4 here)
#  plus the order, for a total here of 8.
norder <- 4
#  compute the 11 by 8 matrix of basis function values
basismat <- bsplineS(x, breaks, norder)

# use sparse Matrix representation to save memory
Basismat <- bsplineS(x, breaks, norder, returnMatrix=TRUE)

# check

class(Basismat)=='dgCMatrix'



all.equal(basismat, as.matrix(Basismat))
```

`CanadianWeather` *Canadian average annual weather cycle*

Description

Daily temperature and precipitation at 35 different locations in Canada averaged over 1960 to 1994.

Usage

```
CanadianWeather
daily
```

Format

'CanadianWeather' and 'daily' are lists containing essentially the same data. 'CanadianWeather' may be preferred for most purposes; 'daily' is included primarily for compatibility with scripts written before the other format became available and for compatibility with the Matlab 'fda' code.

- CanadianWeather A list with the following components:
  - dailyAv a three dimensional array c(365, 35, 3) summarizing data collected at 35 different weather stations in Canada on the following:
    [,,1] = [,, 'Temperature.C']: average daily temperature for each day of the year
    [,,2] = [,, 'Precipitation.mm']: average daily rainfall for each day of the year rounded to 0.1 mm.
    [,,3] = [,, 'log10precip']: base 10 logarithm of Precipitation.mm after first replacing 27 zeros by 0.05 mm (Ramsay and Silverman 2006, p. 248).
  - place Names of the 35 different weather stations in Canada whose data are summarized in 'dailyAv'. These names vary between 6 and 11 characters in length. By contrast, daily[["place"]] which are all 11 characters, with names having fewer characters being extended with trailing blanks.
  - province names of the Canadian province containing each place
  - coordinates a numeric matrix giving 'N.latitude' and 'W.longitude' for each place.
  - region Which of 4 climate zones contain each place: Atlantic, Pacific, Continental, Arctic.
  - monthlyTemp A matrix of dimensions (12, 35) giving the average temperature in degrees celcius for each month of the year.
  - monthlyPrecip A matrix of dimensions (12, 35) giving the average daily precipitation in millimeters for each month of the year.
  - geogindex Order the weather stations from East to West to North
- daily A list with the following components:
  - place Names of the 35 different weather stations in Canada whose data are summarized in 'dailyAv'. These names are all 11 characters, with shorter names being extended with trailing blanks. This is different from CanadianWeather[["place"]], where trailing blanks have been dropped.
  - tempav a matrix of dimensions (365, 35) giving the average temperature in degrees celcius for each day of the year. This is essentially the same as CanadianWeather[["dailyAv"]][,, "Temperature.C"].
  - precipav a matrix of dimensions (365, 35) giving the average temperature in degrees celcius for each day of the year. This is essentially the same as CanadianWeather[["dailyAv"]][,, "Precipitation.mm"].

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York

See Also

monthAccessories MontrealTemp

Examples

```
##
## 1.  Plot (latitude & longitude) of stations by region
##
with(CanadianWeather, plot(-coordinates[, 2], coordinates[, 1], type='n',
                           xlab="West Longitude", ylab="North Latitude",
                           axes=FALSE) )
Wlon <- pretty(CanadianWeather$coordinates[, 2])
axis(1, -Wlon, Wlon)
axis(2)

rgns <- 1:4
names(rgns) <- c('Arctic', 'Atlantic', 'Continental', 'Pacific')
Rgns <- rgns[CanadianWeather$region]
with(CanadianWeather, points(-coordinates[, 2], coordinates[, 1],
                             col=Rgns, pch=Rgns) )
legend('topright', legend=names(rgns), col=rgns, pch=rgns)

##
## 2.  Plot dailyAv[, 'Temperature.C'] for 4 stations
##
data(CanadianWeather)
# Expand the left margin to allow space for place names
op <- par(mar=c(5, 4, 4, 5)+.1)
# Plot
stations <- c("Pr. Rupert", "Montreal", "Edmonton", "Resolute")
matplot(day.5, CanadianWeather$dailyAv[, stations, "Temperature.C"],
        type="l", axes=FALSE, xlab="", ylab="Mean Temperature (deg C)")
axis(2, las=1)
# Label the horizontal axis with the month names
axis(1, monthBegin.5, labels=FALSE)
axis(1, monthEnd.5, labels=FALSE)
axis(1, monthMid, monthLetters, tick=FALSE)
# Add the monthly averages
matpoints(monthMid, CanadianWeather$monthlyTemp[, stations])
# Add the names of the weather stations
mtext(stations, side=4,
      at=CanadianWeather$dailyAv[365, stations, "Temperature.C"],
     las=1)
# clean up
par(op)
```

`cca.fd` *Functional Canonical Correlation Analysis*

Description

Carry out a functional canonical correlation analysis with regularization or roughness penalties on the estimated canonical variables.

Usage

```
cca.fd(fdobj1, fdobj2=fdobj1, ncan = 2,
       ccafdPar1=fdPar(basisobj1, 2, 1e-10),
       ccafdPar2=ccafdPar1, centerfns=TRUE)
```

Arguments

fdobj1	a functional data object.
fdobj2	a functional data object. By default this is fdobj1, in which case the first argument must be a bivariate functional data object.
ncan	the number of canonical variables and weight functions to be computed. The default is 2.
ccafdPar1	a functional parameter object defining the first set of canonical weight functions. The object may contain specifications for a roughness penalty. The default is defined using the same basis as that used for fdobj1 with a slight penalty on its second derivative.
ccafdPar2	a functional parameter object defining the second set of canonical weight functions. The object may contain specifications for a roughness penalty. The default is ccafdParobj1.
centerfns	if TRUE, the functions are centered prior to analysis. This is the default.

Value

an object of class cca.fd with the 5 slots:

ccwtfd1	a functional data object for the first canonical variate weight function
ccwtfd2	a functional data object for the second canonical variate weight function
cancorr	a vector of canonical correlations
ccavar1	a matrix of scores on the first canonical variable.
ccavar2	a matrix of scores on the second canonical variable.

See Also

plot.cca.fd, varmx.cca.fd, pca.fd

Examples

```
#  Canonical correlation analysis of knee-hip curves

gaittime  <- (1:20)/21
gaitrange <- c(0,1)
gaitbasis <- create.fourier.basis(gaitrange,21)
lambda    <- 10^(-11.5)
harmaccelLfd <- vec2Lfd(c(0, 0, (2*pi)^2, 0))

gaitfdPar <- fdPar(gaitbasis, harmaccelLfd, lambda)
gaitfd <- smooth.basis(gaittime, gait, gaitfdPar)$fd
```

```
ccafdPar <- fdPar(gaitfd, harmaccelLfd, 1e-8)
ccafd0    <- cca.fd(gaitfd[,1], gaitfd[,2], ncan=3, ccafdPar, ccafdPar)
#  display the canonical correlations
round(ccafd0$ccacorr[1:6],3)
#  compute a VARIMAX rotation of the canonical variables
ccafd <- varmx.cca.fd(ccafd0)
#  plot the canonical weight functions
plot.cca.fd(ccafd)
```

`center.fd` *Center Functional Data*

Description

Subtract the pointwise mean from each of the functions in a functional data object; that is, to center them on the mean function.

Usage

```
center.fd(fdobj)
```

Arguments

`fdobj`	a functional data object to be centered.

Value

a functional data object whose mean is zero.

See Also

`mean.fd`, `sum.fd`, `stddev.fd`, `std.fd`

Examples

```
daytime    <- (1:365)-0.5
daybasis   <- create.fourier.basis(c(0,365), 365)
harmLcoef  <- c(0,(2*pi/365)^2,0)
harmLfd    <- vec2Lfd(harmLcoef, c(0,365))
templambda <- 0.01
tempfdPar  <- fdPar(daybasis, harmLfd, templambda)

# do not run on CRAN because it takes too long.
tempfd     <- smooth.basis(daytime,
       CanadianWeather$dailyAv[,,"Temperature.C"], tempfdPar)$fd

tempctrfd  <- center.fd(tempfd)

plot(tempctrfd, xlab="Day", ylab="deg. C",
     main = "Centered temperature curves")
```

checkDims3 *Compare dimensions and dimnames of arrays*

Description

Compare selected dimensions and dimnames of arrays, coercing objects to 3-dimensional arrays and either give an error or force matching.

Usage

```
checkDim3(x, y=NULL, xdim=1, ydim=1, defaultNames='x',
         subset=c('xiny', 'yinx', 'neither'),
         xName=substring(deparse(substitute(x)), 1, 33),
         yName=substring(deparse(substitute(y)), 1, 33) )
checkDims3(x, y=NULL, xdim=2:3, ydim=2:3, defaultNames='x',
         subset=c('xiny', 'yinx', 'neither'),
         xName=substring(deparse(substitute(x)), 1, 33),
         yName=substring(deparse(substitute(y)), 1, 33) )
```

Arguments

x, y	arrays to be compared. If y is missing, x is used. Currently, both x and y can have at most 3 dimensions. If either has more, an error will be thrown. If either has fewer, it will be expanded to 3 dimensions using as.array3.
xdim, ydim	For checkDim3, these are positive integers indicating which dimension of x will be compared with which dimension of y. For checkDims3, these are positive integer vectors of the same length, passed one at a time to checkDim3. The default here is to force matching dimensions for plotfit.fd.
defaultNames	Either NULL, FALSE or a character string or vector or list. If NULL, no checking is done of dimnames. If FALSE, an error is thrown unless the corresponding dimensions of x and y match exactly. If it is a character string, vector, or list, it is used as the default names if neither x nor y have dimenames for the compared dimensions. If it is a character vector that is too short, it is extended to the required length using paste(defaultNames, 1:ni), where ni = the required length. If it is a list, it should have length (length(xdim)+1). Each component must be either a character vector or NULL. If neither x nor y have dimenames for the first compared dimensions, defaultNames[[1]] will be used instead unless it is NULL, in which case the last component of defaultNames will be used. If it is null, an error is thrown.
subset	If 'xiny', and any(dim(y)[ydim] < dim(x)[xdim]), an error is thrown. Else if any(dim(y)[ydim] > dim(x)[xdim]) the larger is reduced to match the smaller. If 'yinx', this procedure is reversed. If 'neither', any dimension mismatch generates an error.

xName, yName names of the arguments x and y, used only to in error messages.

Details

For checkDims3, confirm that xdim and ydim have the same length, and call checkDim3 for each pair.

For checkDim3, proceed as follows:

1. if((xdim>3) | (ydim>3)) throw an error.

2. ixperm <- list(1:3, c(2, 1, 3), c(3, 2, 1))[xdim]; iyperm <- list(1:3, c(2, 1, 3), c(3, 2, 1))[ydim];

3. x3 <- aperm(as.array3(x), ixperm); y3 <- aperm(as.array3(y), iyperm)

4. xNames <- dimnames(x3); yNames <- dimnames(y3)

5. Check subset. For example, for subset='xiny', use the following: `if(is.null(xNames)){ if(dim(x3)[1]>dim(y3)[1]) stop else y. <-y3[1:dim(x3)[1],,] dimnames(x) <-list(yNames[[1]],NULL,NULL) } else { if(is.null(xNames[[1]])){ if(dim(x3)[1]>dim(y3)[1]) stop else y. <-y3[1:dim(x3)[1],,] dimnames(x3)[[1]] <-yNames[[1]] } else { if(any(!is.element(xNames[[1]],yNames[[1]])))stop else y. <-y3[xNames[[1]],,] } }`

6. return(list(x=aperm(x3, ixperm), y=aperm(y., iyperm)))

Value

a list with components x and y.

Author(s)

Spencer Graves

See Also

as.array3 plotfit.fd

Examples

```
# Select the first two rows of y
stopifnot(all.equal(
checkDim3(1:2, 3:5),
list(x=array(1:2, c(2,1,1), list(c('x1','x2'), NULL, NULL)),
     y=array(3:4, c(2,1,1), list(c('x1','x2'), NULL, NULL)) )
))

# Select the first two rows of a matrix y
stopifnot(all.equal(
checkDim3(1:2, matrix(3:8, 3)),
list(x=array(1:2,         c(2,1,1), list(c('x1','x2'), NULL, NULL)),
     y=array(c(3:4, 6:7), c(2,2,1), list(c('x1','x2'), NULL, NULL)) )
))

# Select the first column of y
stopifnot(all.equal(
checkDim3(1:2, matrix(3:8, 3), 2, 2),
```

```
list(x=array(1:2,         c(2,1,1), list(NULL, 'x', NULL)),
     y=array(3:5, c(3,1,1), list(NULL, 'x', NULL)) )
))

# Select the first two rows and the first column of y
stopifnot(all.equal(
checkDims3(1:2, matrix(3:8, 3), 1:2, 1:2),
list(x=array(1:2, c(2,1,1), list(c('x1','x2'), 'x', NULL)),
     y=array(3:4, c(2,1,1), list(c('x1','x2'), 'x', NULL)) )
))

# Select the first 2 rows of y
x1 <- matrix(1:4, 2, dimnames=list(NULL, LETTERS[2:3]))
x1a <- x1. <- as.array3(x1)
dimnames(x1a)[[1]] <- c('x1', 'x2')
y1 <- matrix(11:19, 3, dimnames=list(NULL, LETTERS[1:3]))
y1a <- y1. <- as.array3(y1)
dimnames(y1a)[[1]] <- c('x1', 'x2', 'x3')

stopifnot(all.equal(
checkDim3(x1, y1),
list(x=x1a, y=y1a[1:2, , , drop=FALSE])
))

# Select columns 2 & 3 of y
stopifnot(all.equal(
checkDim3(x1, y1, 2, 2),
list(x=x1., y=y1.[, 2:3, , drop=FALSE ])
))

# Select the first 2 rows and  columns 2 & 3 of y
stopifnot(all.equal(
checkDims3(x1, y1, 1:2, 1:2),
list(x=x1a, y=y1a[1:2, 2:3, , drop=FALSE ])
))

# y = columns 2 and 3 of x
x23 <- matrix(1:6, 2, dimnames=list(letters[2:3], letters[1:3]))
x23. <- as.array3(x23)
stopifnot(all.equal(
checkDim3(x23, xdim=1, ydim=2),
list(x=x23., y=x23.[, 2:3,, drop=FALSE ])
))

# Transfer dimnames from y to x
x4a <- x4 <- matrix(1:4, 2)
y4 <- matrix(5:8, 2, dimnames=list(letters[1:2], letters[3:4]))
dimnames(x4a) <- dimnames(t(y4))
stopifnot(all.equal(
checkDims3(x4, y4, 1:2, 2:1),
list(x=as.array3(x4a), y=as.array3(y4))
))
```

```
# as used in plotfit.fd
daybasis65 <- create.fourier.basis(c(0, 365), 65)

daytempfd <- with(CanadianWeather, smooth.basis(
       day.5, dailyAv[,,"Temperature.C"],
       daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd )

defaultNms <- with(daytempfd, c(fdnames[2], fdnames[3], x='x'))
subset <- checkDims3(CanadianWeather$dailyAv[, , "Temperature.C"],
               daytempfd$coef, defaultNames=defaultNms)
# Problem:  dimnames(...)[[3]] = '1'
# Fix:
subset3 <- checkDims3(
        CanadianWeather$dailyAv[, , "Temperature.C", drop=FALSE],
               daytempfd$coef, defaultNames=defaultNms)
```

---

checkLogicalInteger *Does an argument satisfy required conditions?*

Description

Check whether an argument is a logical vector of a certain length or a numeric vector in a certain range and issue an appropriate error or warning if not:

`checkLogical` throws an error or returns FALSE with a warning unless `x` is a logical vector of exactly the required `length`.

`checkNumeric` throws an error or returns FALSE with a warning unless `x` is either NULL or a `numeric` vector of at most `length` with `x` in the desired range.

`checkLogicalInteger` returns a logical vector of exactly `length` unless `x` is neither NULL nor `logical` of the required `length` nor `numeric` with `x` in the desired range.

Usage

```
checkLogical(x, length., warnOnly=FALSE)
checkNumeric(x, lower, upper, length., integer=TRUE, unique=TRUE,
             inclusion=c(TRUE,TRUE), warnOnly=FALSE)
checkLogicalInteger(x, length., warnOnly=FALSE)
```

Arguments

`x`	an object to be checked
`length.`	The required length for `x` if `logical` and not NULL or the maximum length if `numeric`.
`lower, upper`	lower and upper limits for `x`.
`integer`	logical: If true, a `numeric` `x` must be `integer`.
`unique`	logical: TRUE if duplicates are NOT allowed in `x`.

inclusion	logical vector of length 2, similar to link[ifultools]{checkRange}: if(inclusion[1]) (lower <= x) else (lower < x) if(inclusion[2]) (x <= upper) else (x < upper)
warnOnly	logical: If TRUE, violations are reported as warnings, not as errors.

Details

1. xName <- deparse(substitute(x)) to use in any required error or warning.

2. if(is.null(x)) handle appropriately: Return FALSE for checkLogical, TRUE for checkNumeric and rep(TRUE, length.) for checkLogicalInteger.

3. Check class(x).

4. Check other conditions.

Value

checkLogical returns a logical vector of the required length., unless it issues an error message.

checkNumeric returns a numeric vector of at most length. with all elements between lower and upper, and optionally unique, unless it issues an error message.

checkLogicalInteger returns a logical vector of the required length., unless it issues an error message.

Author(s)

Spencer Graves

Examples

```
##
## checkLogical
##
checkLogical(NULL, length=3, warnOnly=TRUE)
checkLogical(c(FALSE, TRUE, TRUE), length=4, warnOnly=TRUE)
checkLogical(c(FALSE, TRUE, TRUE), length=3)

##
## checkNumeric
##
checkNumeric(NULL, lower=1, upper=3)
checkNumeric(1:3, 1, 3)
checkNumeric(1:3, 1, 3, inclusion=FALSE, warnOnly=TRUE)
checkNumeric(pi, 1, 4, integer=TRUE, warnOnly=TRUE)
checkNumeric(c(1, 1), 1, 4, warnOnly=TRUE)
checkNumeric(c(1, 1), 1, 4, unique=FALSE, warnOnly=TRUE)

##
## checkLogicalInteger
##
checkLogicalInteger(NULL, 3)
checkLogicalInteger(c(FALSE, TRUE), warnOnly=TRUE)
```

```
checkLogicalInteger(1:2, 3)
checkLogicalInteger(2, warnOnly=TRUE)
checkLogicalInteger(c(2, 4), 3, warnOnly=TRUE)

##
## checkLogicalInteger names its calling function
## rather than itself as the location of error detection
## if possible
##
tstFun <- function(x, length., warnOnly=FALSE){
   checkLogicalInteger(x, length., warnOnly)
}
tstFun(NULL, 3)
tstFun(4, 3, warnOnly=TRUE)

tstFun2 <- function(x, length., warnOnly=FALSE){
   tstFun(x, length., warnOnly)
}
tstFun2(4, 3, warnOnly=TRUE)
```

`coef.fd` *Extract functional coefficients*

Description

Obtain the coefficients component from a functional object (functional data, class `fd`, functional parameter, class `fdPar`, a functional smooth, class `fdSmooth`, or a Taylor spline representation, class `Taylor`.

Usage

```
## S3 method for class 'fd'
coef(object, ...)
## S3 method for class 'fdPar'
coef(object, ...)
## S3 method for class 'fdSmooth'
coef(object, ...)
## S3 method for class 'fd'
coefficients(object, ...)
## S3 method for class 'fdPar'
coefficients(object, ...)
## S3 method for class 'fdSmooth'
coefficients(object, ...)
```

Arguments

`object`	An object whose functional coefficients are desired
`...`	other arguments

Details

Functional representations are evaluated by multiplying a basis function matrix times a coefficient vector, matrix or 3-dimensional array. (The basis function matrix contains the basis functions as columns evaluated at the `evalarg` values as rows.)

Value

A numeric vector or array of the coefficients.

See Also

`coef` `fd` `fdPar` `smooth.basisPar` `smooth.basis`

Examples

```
##
## coef.fd
##
bspl1.1 <- create.bspline.basis(norder=1, breaks=0:1)
fd.bspl1.1 <- fd(0, basisobj=bspl1.1)
coef(fd.bspl1.1)


##
## coef.basisPar
##
rangeval <- c(-3,3)
#  set up some standard normal data
x <- rnorm(50)
#  make sure values within the range
x[x < -3] <- -2.99
x[x >  3] <-  2.99
#  set up basis for W(x)
basisobj <- create.bspline.basis(rangeval, 11)
#  set up initial value for Wfdobj
Wfd0 <- fd(matrix(0,11,1), basisobj)
WfdParobj <- fdPar(Wfd0)

coef(WfdParobj)


##
## coef.fdSmooth
##

girlGrowthSm <- with(growth, smooth.basisPar(argvals=age, y=hgtf,
                                             lambda=0.1)$fd)
coef(girlGrowthSm)
```

cor.fd *Correlation matrix from functional data object(s)*

Description

Compute a correlation matrix for one or two functional data objects.

Usage

```
cor.fd(evalarg1, fdobj1, evalarg2=evalarg1, fdobj2=fdobj1)
```

Arguments

evalarg1	a vector of argument values for fdobj1.
evalarg2	a vector of argument values for fdobj2.
fdobj1, fdobj2	functional data objects

Details

1. var1 <- var.fd(fdobj1)
2. evalVar1 <- eval.bifd(evalarg1, evalarg1, var1)
3. if(missing(fdobj2)) Convert evalVar1 to correlations
4. else:

4.1. var2 <- var.fd(fdobj2)

4.2. evalVar2 <- eval.bifd(evalarg2, evalarg2, var2)

4.3. var12 <- var.df(fdobj1, fdobj2)

4.4. evalVar12 <- eval.bifd(evalarg1, evalarg2, var12)

4.5. Convert evalVar12 to correlations

Value

A matrix or array:

With one or two functional data objects, fdobj1 and possibly fdobj2, the value is a matrix of dimensions length(evalarg1) by length(evalarg2) giving the correlations at those points of fdobj1 if missing(fdobj2) or of correlations between eval.fd(evalarg1, fdobj1) and eval.fd(evalarg2, fdobj2).

With a single multivariate data object with k variables, the value is a 4-dimensional array of dim = c(nPts, nPts, 1, choose(k+1, 2)), where nPts = length(evalarg1).

See Also

mean.fd, sd.fd, std.fd stdev.fd var.fd

Examples

```
daybasis3 <- create.fourier.basis(c(0, 365))
daybasis5 <- create.fourier.basis(c(0, 365), 5)
tempfd3 <- with(CanadianWeather, smooth.basis(
       day.5, dailyAv[,,"Temperature.C"],
       daybasis3, fdnames=list("Day", "Station", "Deg C"))$fd )
precfd5 <- with(CanadianWeather, smooth.basis(
       day.5, dailyAv[,,"log10precip"],
       daybasis5, fdnames=list("Day", "Station", "Deg C"))$fd )

# Correlation matrix for a single functional data object
(tempCor3 <- cor.fd(seq(0, 356, length=4), tempfd3))

# Cross correlation matrix between two functional data objects
# Compare with structure described above under 'value':
(tempPrecCor3.5 <- cor.fd(seq(0, 365, length=4), tempfd3,
                          seq(0, 356, length=6), precfd5))

# The following produces contour and perspective plots

daybasis65 <- create.fourier.basis(rangeval=c(0, 365), nbasis=65)
daytempfd <- with(CanadianWeather, smooth.basis(
       day.5, dailyAv[,,"Temperature.C"],
       daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd )
dayprecfd <- with(CanadianWeather, smooth.basis(
       day.5, dailyAv[,,"log10precip"],
       daybasis65, fdnames=list("Day", "Station", "log10(mm)"))$fd )

str(tempPrecCor <- cor.fd(weeks, daytempfd, weeks, dayprecfd))
# dim(tempPrecCor)= c(53, 53)

op <- par(mfrow=c(1,2), pty="s")
contour(weeks, weeks, tempPrecCor,
        xlab="Average Daily Temperature",
        ylab="Average Daily log10(precipitation)",
        main=paste("Correlation function across locations\n",
          "for Canadian Anual Temperature Cycle"),
        cex.main=0.8, axes=FALSE)
axisIntervals(1, atTick1=seq(0, 365, length=5), atTick2=NA,
            atLabels=seq(1/8, 1, 1/4)*365,
            labels=paste("Q", 1:4) )
axisIntervals(2, atTick1=seq(0, 365, length=5), atTick2=NA,
            atLabels=seq(1/8, 1, 1/4)*365,
            labels=paste("Q", 1:4) )
persp(weeks, weeks, tempPrecCor,
      xlab="Days", ylab="Days", zlab="Correlation")
mtext("Temperature-Precipitation Correlations", line=-4, outer=TRUE)
par(op)

# Correlations and cross correlations
# in a bivariate functional data object
gaittime   <- (1:20)/21
```

```
gaitbasis5 <- create.fourier.basis(c(0,1),nbasis=5)
gaitfd5    <- smooth.basis(gaittime, gait, gaitbasis5)$fd

gait.t3 <- (0:2)/2
(gaitCor3.5 <- cor.fd(gait.t3, gaitfd5))
# Check the answers with manual computations
gait3.5 <- eval.fd(gait.t3, gaitfd5)
all.equal(cor(t(gait3.5[,,1])), gaitCor3.5[,,,1])
# TRUE
all.equal(cor(t(gait3.5[,,2])), gaitCor3.5[,,,3])
# TRUE
all.equal(cor(t(gait3.5[,,2]), t(gait3.5[,,1])),
               gaitCor3.5[,,,2])
# TRUE

# NOTE:
dimnames(gaitCor3.5)[[4]]
# [1] Hip-Hip
# [2] Knee-Hip
# [3] Knee-Knee
# If [2] were "Hip-Knee", then
# gaitCor3.5[,,,2] would match
# cor(t(gait3.5[,,1]), t(gait3.5[,,2]))
# *** It does NOT.  Instead, it matches:
# cor(t(gait3.5[,,2]), t(gait3.5[,,1]))
```

---

covPACE *Estimate of the covariance surface*

Description

Function covPACE does a bivariate smoothing for estimating the covariance surface for data that has not yet been smoothed

Usage

```
covPACE(data,rng , time, meanfd, basis, lambda, Lfdobj)
```

Arguments

data	a matrix object or list – If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If y is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. – If it is a list, each element must be a matrix object, the rows correspond to argument values per individual. First column corresponds to time points and following columns to argument values per variable.

rng	a vector of length 2 defining a restricted range where the data was observed
time	Array with time points where data was taken. length(time) == dim(data)[1]
meanfd	Fd object corresponding to the mean function of the data
basis	basisfd object for smoothing the covariate function
lambda	a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter
Lfdobj	a linear differential operator object for smoothing penalty of the estimated functional parameter

Value

a list with these two named entries:

cov.estimate	an object of class "bifd" object or a list of "bifd" elements
meanfd	a functional data object giving the mean function

CRAN *Test if running as CRAN*

Description

This function allows package developers to run tests themselves that should not run on CRAN or with "R CMD check –as-cran" because of compute time constraints with CRAN tests.

Usage

```
CRAN(CRAN_pattern, n_R_CHECK4CRAN)
```

Arguments

CRAN_pattern	a regular expressions to apply to the names of Sys.getenv() to identify possible CRAN parameters. Defaults to Sys.getenv('_CRAN_pattern_') if available and '^_R_' if not.
n_R_CHECK4CRAN	Assume this is CRAN if at least n_R_CHECK4CRAN elements of Sys.getenv() have names matching x. Defaults to Sys.getenv('_n_R_CHECK4CRAN_') if available and 5 if not.

Details

The "Writing R Extensions" manual says that "R CMD check" can be customized "by setting environment variables _R_CHECK_*_:, as described in" the Tools section of the "R Internals" manual.

'R CMD check' was tested with R 3.0.1 under Fedora 18 Linux and with Rtools 3.0 from April 16, 2013 under Windows 7. With the '–as-cran' option, 7 matches were found; without it, only 3 were found. These numbers were unaffected by the presence or absence of the '–timings' parameter. On this basis, the default value of n_R_CHECK4CRAN was set at 5.

1. x. <- Sys.getenv()
2. Fix CRAN_pattern and n_R_CHECK4CRAN if missing.
3. Let i be the indices of x. whose names match all the patterns in the vector x.
4. Assume this is CRAN if length(i) >= n_R_CHECK4CRAN

Value

a logical scalar with attributes 'Sys.getenv' containing the results of `Sys.getenv()` and 'matches' contining `i` per step 3 above.

See Also

Sys.getenv

Examples

```
cran <- CRAN()
str(cran)
gete <- attr(cran, 'Sys.getenv')
(ngete <- names(gete))

iget <- grep('^_', names(gete))
gete[iget]

## Not run:
if(CRAN()){
  stop('CRAN')
} else {
  stop('NOT CRAN')
}

## End(Not run)
```

`create.basis` *Create Basis Set for Functional Data Analysis*

Description

Functional data analysis proceeds by selecting a finite basis set and fitting data to it. The current `fda` package supports fitting via least squares penalized with lambda times the integral over the (finite) support of the basis set of the squared deviations from a linear differential operator.

Details

The most commonly used basis in fda is probably B-splines. For periodic phenomena, Fourier bases are quite useful. A constant basis is provided to facilitation arithmetic with functional data objects. To restrict attention to solutions of certain differential equations, it may be useful to use a corresponding basis set such as exponential, monomial or power basis sets.

Power bases support the use of negative and fractional powers, while monomial bases are restricted only to nonnegative integer exponents.

The polygonal basis is essentially a B-spline of order 2, degree 1.

The following summarizes arguments used by some or all of the current create.basis functions:

- rangeval a vector of length 2 giving the lower and upper limits of the range of permissible values for the function argument.

  For bspline bases, this can be inferred from range(breaks). For polygonal bases, this can be inferred from range(argvals). In all other cases, this defaults to 0:1.

- nbasis an integer giving the number of basis functions.

  This is not used for two of the create.basis functions: For constant this is 1, so there is no need to specify it. For polygonal bases, it is length(argvals), and again there is no need to specify it.

  For bspline bases, if nbasis is not specified, it defaults to (length(breaks) + norder - 2) if breaks is provided. Otherwise, nbasis defaults to 20 for bspline bases.

  For exponential bases, if nbasis is not specified, it defaults to length(ratevec) if ratevec is provided. Otherwise, in fda_2.0.2, ratevec defaults to 1, which makes nbasis = 1; in fda_2.0.4, ratevec will default to 0:1, so nbasis will then default to 2.

  For monomial and power bases, if nbasis is not specified, it defaults to length(exponents) if exponents is provided. Otherwise, nbasis defaults to 2 for monomial and power bases. (Temporary exception: In fda_2.0.2, the default nbasis for power bases is 1. This will be increased to 2 in fda_2.0.4.)

  In addition to rangeval and nbasis, all but constant bases have one or two parameters unique to that basis type or shared with one other:

  - bspline Argument norder = the order of the spline, which is one more than the degree of the polynomials used. This defaults to 4, which gives cubic splines.
    Argument breaks = the locations of the break or join points; also called knots. This defaults to seq(rangeval[1], rangeval[2], nbasis-norder+2).
  - polygonal Argument argvals = the locations of the break or join points; also called knots. This defaults to seq(rangeval[1], rangeval[2], nbasis).
  - fourier Argument period defaults to diff(rangeval).
  - exponential Argument ratevec. In fda_2.0.2, this defaulted to 1. In fda_2.0.3, it will default to 0:1.
  - monomial, power Argument exponents. Default = 0:(nbasis-1). For monomial bases, exponents must be distinct nonnegative integers. For power bases, they must be distinct real numbers.

  Beginning with fda_2.1.0, the last 6 arguments for all the create.basis functions will be as follows; some but not all are available in the previous versions of fda:

  - dropind a vector of integers specifiying the basis functions to be dropped, if any.

- quadvals a matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of `quadvals` contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to 1, 4, 2, 4, ..., 2, 4, 1.
- values a list of matrices with one row for each row of `quadvals` and one column for each basis function. The elements of the list correspond to the basis functions and their derivatives evaluated at the quadrature points contained in the first column of `quadvals`.
- basisvalues A list of lists, allocated by code such as vector("list",1). This field is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each list within the vector corresponds to a specific set of argument values, and must have at least two components, which may be tagged as you wish. 'The first component in an element of the list vector contains the argument values. The second component in an element of the list vector contains a matrix of values of the basis functions evaluated at the arguments in the first component. The third and subsequent components, if present, contain matrices of values their derivatives up to a maximum derivative order. Whenever function getbasismatrix is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system. Even this time can be avoided by direct retrieval of the desired array. For example, you might set up a vector of argument values called "evalargs" along with a matrix of basis function values for these argument values called "basismat". You might want too use tags like "args" and "values", respectively for these. You would then assign them to `basisvalues` with code such as the following:
  basisobj\$basisvalues <- vector("list",1)
  basisobj\$basisvalues[[1]] <- list(args=evalargs, values=basismat)
- names either a character vector of the same length as the number of basis functions or a simple stem used to construct such a vector.
  For `bspline` bases, this defaults to paste('bspl', norder, '.', 1:nbreaks, sep='').
  For other bases, there are crudely similar defaults.
- axes an optional list used by selected `plot` functions to create custom `axes`. If this `axes` argument is not NULL, functions `plot.basisfd`, `plot.fd`, `plot.fdSmooth` `plotfit.fd`, `plotfit.fdSmooth`, and `plot.Lfd` will create axes via `do.call(x$axes[[1]],x$axes[-1])`. The primary example of this is to create `CanadianWeather` plots using `list("axesIntervals")`

Author(s)

J. O. Ramsay and Spencer Graves

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

create.bspline.basis create.constant.basis create.exponential.basis create.fourier.basis create.monomial.basis create.polygonal.basis create.power.basis

---

`create.bspline.basis` *Create a B-spline Basis*

Description

Functional data objects are constructed by specifying a set of basis functions and a set of coefficients defining a linear combination of these basis functions. The B-spline basis is used for non-periodic functions. B-spline basis functions are polynomial segments jointed end-to-end at at argument values called knots, breaks or join points. The segments have specifiable smoothness across these breaks. B-spline basis functions have the advantages of very fast computation and great flexibility. A polygonal basis generated by `create.polygonal.basis` is essentially a B-spline basis of order 2, degree 1. Monomial and polynomial bases can be obtained as linear transformations of certain B-spline bases.

Usage

```
create.bspline.basis(rangeval=NULL, nbasis=NULL, norder=4,
      breaks=NULL, dropind=NULL, quadvals=NULL, values=NULL,
      basisvalues=NULL, names="bspl")
```

Arguments

`rangeval`	a numeric vector of length 2 defining the interval over which the functional data object can be evaluated; default value is `if(is.null(breaks)) 0:1 else range(breaks)`. If `length(rangeval) == 1` and `rangeval <= 0`, this is an error. Otherwise, if `length(rangeval) == 1`, `rangeval` is replaced by `c(0,rangeval)`. If length(rangeval)>2 and neither `breaks` nor `nbasis` are provided, this extra long `rangeval` argument is assigned to `breaks`, and then `rangeval = range(breaks)`. NOTE: Nonnumerics are also accepted provided `sum(is.na(as.numeric(rangeval))) == 0`. However, as of July 2, 2012, nonnumerics may not work for `argvals` in other `fda` functions.
`nbasis`	an integer variable specifying the number of basis functions. This 'nbasis' argument is ignored if `breaks` is supplied, in which case nbasis = nbreaks + norder - 2, where nbreaks = length(breaks). If `breaks` is not supplied and `nbasis` is, then nbreaks = nbasis - norder + 2, and breaks = seq(rangevals[1], rangevals[2], nbreaks).
`norder`	an integer specifying the order of b-splines, which is one higher than their degree. The default of 4 gives cubic splines.

`breaks` a vector specifying the break points defining the b-spline. Also called knots, these are a strictly increasing sequence of junction points between piecewise polynomial segments. They must satisfy `breaks[1]` = `rangeval[1]` and `breaks[nbreaks]` = `rangeval[2]`, where `nbreaks` is the length of `breaks`. There must be at least 2 values in `breaks`.

As for rangeval, must satisfy `sum(is.na(as.numeric(breaks))) == 0`.

`dropind` a vector of integers specifying the basis functions to be dropped, if any. For example, if it is required that a function be zero at the left boundary, this is achieved by dropping the first basis function, the only one that is nonzero at that point.

`quadvals` a matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of `quadvals` contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to 1, 4, 2, 4, ..., 2, 4, 1.

`values` a list containing the basis functions and their derivatives evaluated at the quadrature points contained in the first column of `quadvals` .

`basisvalues` a vector of lists, allocated by code such as `vector("list",1)`. This argument is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each list within the vector corresponds to a specific set of argument values, and must have at least two components, which may be tagged as you wish. The first component in an element of the list vector contains the argument values. The second component in an element of the list vector contains a matrix of values of the basis functions evaluated at the arguments in the first component. The third and subsequent components, if present, contain matrices of values their derivatives up to a maximum derivative order. Whenever function `getbasismatrix()` is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system.

`names` either a character vector of the same length as the number of basis functions or a single character string to which `norder,"."` and `1:nbasis` are appended as `paste(names,norder,".",1:nbasis,sep="")`. For example, if `norder` = 4, this defaults to `'bspl4.1','bspl4.2'`, ...

Details

Spline functions are constructed by joining polynomials end-to-end at argument values called *break points* or *knots*. First, the interval is subdivided into a set of adjoining intervals separated the knots. Then a polynomial of order m (degree $m-1$) is defined for each interval. To make the resulting piecewise polynomial smooth, two adjoining polynomials are constrained to have their values and all their derivatives up to order $m-2$ match at the point where they join.

Consider as an illustration the very common case where the order is 4 for all polynomials, so that degree of each polynomials is 3. That is, the polynomials are *cubic*. Then at each break point or knot, the values of adjacent polynomials must match, and so also for their first and second derivatives. Only their third derivatives will differ at the point of junction.

The number of degrees of freedom of a cubic spline function of this nature is calculated as follows. First, for the first interval, there are four degrees of freedom. Then, for each additional interval, the polynomial over that interval now has only one degree of freedom because of the requirement for matching values and derivatives. This means that the number of degrees of freedom is the number of interior knots (that is, not counting the lower and upper limits) plus the order of the polynomials:

`nbasis = norder + length(breaks) -2`

The consistency of the values of `nbasis`, `norder` and `breaks` is checked, and an error message results if this equation is not satisfied.

B-splines are a set of special spline functions that can be used to construct any such piecewise polynomial by computing the appropriate linear combination. They derive their computational convenience from the fact that any B-spline basis function is nonzero over at most m adjacent intervals. The number of basis functions is given by the rule above for the number of degrees of freedom.

The number of intervals controls the flexibility of the spline; the more knots, the more flexible the resulting spline will be. But the position of the knots also plays a role. Where do we position the knots? There is room for judgment here, but two considerations must be kept in mind: (1) you usually want at least one argument value between two adjacent knots, and (2) there should be more knots where the curve needs to have sharp curvatures such as a sharp peak or valley or an abrupt change of level, but only a few knots are required where the curve is changing very slowly.

This function automatically includes `norder` replicates of the end points rangeval. By contrast, the analogous functions splineDesign and spline.des in the `splines` package do NOT automatically replicate the end points. To compare answers, the end knots must be replicated manually when using splineDesign or spline.des.

Value

a basis object of the type `bspline`

References

Ramsay, James O., Hooker, Giles, and Graves, Spencer (2009), *Functional data analysis with R and Matlab*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

`basisfd`, `create.constant.basis`, `create.exponential.basis`, `create.fourier.basis`, `create.monomial.basis`, `create.polygonal.basis`, `create.power.basis` `splineDesign` `spline.des`

Examples

```
##
## The simplest basis currently available with this function:
##
```

```
bspl1.1 <- create.bspline.basis(norder=1)
plot(bspl1.1)
# 1 basis function, order 1 = degree 0 = step function:

# should be the same as above:
b1.1 <- create.bspline.basis(0:1, nbasis=1, norder=1, breaks=0:1)

all.equal(bspl1.1, b1.1)


bspl2.2 <- create.bspline.basis(norder=2)
plot(bspl2.2)

bspl3.3 <- create.bspline.basis(norder=3)
plot(bspl3.3)

bspl4.4 <- create.bspline.basis()
plot(bspl4.4)

bspl1.2 <- create.bspline.basis(norder=1, breaks=c(0,.5, 1))
plot(bspl1.2)
# 2 bases, order 1 = degree 0 = step functions:
# (1) constant 1 between 0 and 0.5 and 0 otherwise
# (2) constant 1 between 0.5 and 1 and 0 otherwise.

bspl2.3 <- create.bspline.basis(norder=2, breaks=c(0,.5, 1))
plot(bspl2.3)
# 3 bases:  order 2 = degree 1 = linear
# (1) line from (0,1) down to (0.5, 0), 0 after
# (2) line from (0,0) up to (0.5, 1), then down to (1,0)
# (3) 0 to (0.5, 0) then up to (1,1).

bspl3.4 <- create.bspline.basis(norder=3, breaks=c(0,.5, 1))
plot(bspl3.4)
# 4 bases:  order 3 = degree 2 = parabolas.
# (1) (x-.5)^2 from 0 to .5, 0 after
# (2) 2*(x-1)^2 from .5 to 1, and a parabola
#     from (0,0 to (.5, .5) to match
# (3 & 4) = complements to (2 & 1).

bSpl4. <- create.bspline.basis(c(-1,1))
plot(bSpl4.)
# Same as bSpl4.23 but over (-1,1) rather than (0,1).

# set up the b-spline basis for the lip data, using 23 basis functions,
#   order 4 (cubic), and equally spaced knots.
#  There will be 23 - 4 = 19 interior knots at 0.05, ..., 0.95
lipbasis <- create.bspline.basis(c(0,1), 23)
plot(lipbasis)

bSpl.growth <- create.bspline.basis(growth$age)
# cubic spline (order 4)
```

```
bSpl.growth6 <- create.bspline.basis(growth$age,norder=6)
# quintic spline (order 6)

##
## Nonnumeric rangeval
##

# Date
July4.1776 <- as.Date('1776-07-04')
Apr30.1789 <- as.Date('1789-04-30')
AmRev <- c(July4.1776, Apr30.1789)
BspRevolution <- create.bspline.basis(AmRev)

# POSIXct
July4.1776ct <- as.POSIXct1970('1776-07-04')
Apr30.1789ct <- as.POSIXct1970('1789-04-30')
AmRev.ct <- c(July4.1776ct, Apr30.1789ct)
BspRev.ct <- create.bspline.basis(AmRev.ct)
```

`create.constant.basis` *Create a Constant Basis*

Description

Create a constant basis object, defining a single basis function whose value is everywhere 1.0.

Usage

```
create.constant.basis(rangeval=c(0, 1), names="const", axes=NULL)
```

Arguments

`rangeval`	a vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated. However, this is seldom used since the value of the basis function does not depend on the range or any argument values.
`names`	a character vector of length 1.
`axes`	an optional list used by selected `plot` functions to create custom `axes`. If this `axes` argument is not `NULL`, functions `plot.basisfd`, `plot.fd`, `plot.fdSmooth` `plotfit.fd`, `plotfit.fdSmooth`, and `plot.Lfd` will create axes via `x$axes[[1]]` and `x$axes[-1]`. The primary example of this uses `list("axesIntervals",...)`, e.g., with `Fourier` bases to create `CanadianWeather` plots

Value

a basis object with type component `const`.

See Also

basisfd, create.bspline.basis, create.exponential.basis, create.fourier.basis, create.monomial.basis, create.polygonal.basis, create.power.basis

Examples

```
basisobj <- create.constant.basis(c(-1,1))
```

---

create.exponential.basis

Create an Exponential Basis

Description

Create an exponential basis object defining a set of exponential functions with rate constants in argument ratevec.

Usage

```
create.exponential.basis(rangeval=c(0,1), nbasis=NULL, ratevec=NULL,
                         dropind=NULL, quadvals=NULL, values=NULL,
                         basisvalues=NULL, names='exp', axes=NULL)
```

Arguments

rangeval	a vector of length 2 containing the initial and final values of the interval over which the functional data object can be evaluated.
nbasis	the number of basis functions. Default = `if(is.null(ratevec)) 2 else length(ratevec)`.
ratevec	a vector of length `nbasis` of rate constants defining basis functions of the form `exp(rate*x)`. Default = 0:(nbasis-1).
dropind	a vector of integers specifiying the basis functions to be dropped, if any. For example, if it is required that a function be zero at the left boundary, this is achieved by dropping the first basis function, the only one that is nonzero at that point.
quadvals	a matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of `quadvals` contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to 1, 4, 2, 4, ..., 2, 4, 1.
values	a list of matrices with one row for each row of `quadvals` and one column for each basis function. The elements of the list correspond to the basis functions and their derivatives evaluated at the quadrature points contained in the first column of `quadvals`.

basisvalues
: A list of lists, allocated by code such as vector("list",1). This field is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each list within the vector corresponds to a specific set of argument values, and must have at least two components, which may be tagged as you wish. 'The first component in an element of the list vector contains the argument values. The second component in an element of the list vector contains a matrix of values of the basis functions evaluated at the arguments in the first component. The third and subsequent components, if present, contain matrices of values their derivatives up to a maximum derivative order. Whenever function getbasismatrix is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system. Even this time can be avoided by direct retrieval of the desired array. For example, you might set up a vector of argument values called "evalargs" along with a matrix of basis function values for these argument values called "basismat". You might want too use names like "args" and "values", respectively for these. You would then assign them to `basisvalues` with code such as the following:

  basisobj\$basisvalues <- vector("list",1)

  basisobj\$basisvalues[[1]] <- list(args=evalargs, values=basismat)

names
: either a character vector of the same length as the number of basis functions or a simple stem used to construct such a vector.

  For `exponential` bases, this defaults to paste('exp', 0:(nbasis-1), sep='').

axes
: an optional list used by selected `plot` functions to create custom `axes`. If this `axes` argument is not `NULL`, functions `plot.basisfd`, `plot.fd`, `plot.fdSmooth` `plotfit.fd`, `plotfit.fdSmooth`, and `plot.Lfd` will create axes via `x$axes[[1]]` and `x$axes[-1]`. The primary example of this uses `list("axesIntervals",...)`, e.g., with `Fourier` bases to create `CanadianWeather` plots

Details

Exponential functions are of the type $exp(bx)$ where b is the rate constant. If $b = 0$, the constant function is defined.

Value

a basis object with the type `expon`.

See Also

`basisfd`, `create.bspline.basis`, `create.constant.basis`, `create.fourier.basis`, `create.monomial.basis`, `create.polygonal.basis`, `create.power.basis`

Examples

```
#  Create an exponential basis over interval [0,5]
#  with basis functions 1, exp(-t) and exp(-5t)
basisobj <- create.exponential.basis(c(0,5),3,c(0,-1,-5))
```

```
#  plot the basis
plot(basisobj)
```

`create.fourier.basis` *Create a Fourier Basis*

Description

Create an Fourier basis object defining a set of Fourier functions with specified period.

Usage

```
create.fourier.basis(rangeval=c(0, 1), nbasis=3,
              period=diff(rangeval), dropind=NULL, quadvals=NULL,
              values=NULL, basisvalues=NULL, names=NULL,
              axes=NULL)
```

Arguments

`rangeval`	a vector of length 2 containing the initial and final values of the interval over which the functional data object can be evaluated.
`nbasis`	positive odd integer: If an even number is specified, it is rounded up to the nearest odd integer to preserve the pairing of sine and cosine functions. An even number of basis functions only makes sense when there are always only an even number of observations at equally spaced points; that case can be accommodated using dropind = nbasis-1 (because the bases are `const`, `sin`, `cos`, ...).
`period`	the width of any interval over which the Fourier functions repeat themselves or are periodic.
`dropind`	an optional vector of integers specifiying basis functions to be dropped.
`quadvals`	an optional matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of `quadvals` contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to 1, 4, 2, 4, ..., 2, 4, 1.
`values`	an optional list of matrices with one row for each row of `quadvals` and one column for each basis function. The elements of the list correspond to the basis functions and their derivatives evaluated at the quadrature points contained in the first column of `quadvals`.
`basisvalues`	an optional list of lists, allocated by code such as vector("list",1). This field is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each sublist corresponds to a specific set of argument values, and must have at least two components: a vector of argument values and a matrix of the values the basis functions evaluated at the arguments in the first component. Third and subsequent components, if present, contain matrices of values their

derivatives. Whenever function getbasismatrix is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system. Even this time can be avoided by direct retrieval of the desired array. For example, you might set up a vector of argument values called "evalargs" along with a matrix of basis function values for these argument values called "basismat". You might want too use tags like "args" and "values", respectively for these. You would then assign them to `basisvalues` with code such as the following:

basisobj\$basisvalues <- vector("list",1)

basisobj\$basisvalues[[1]] <- list(args=evalargs, values=basismat)

`names` either a character vector of the same length as the number of basis functions or a simple stem used to construct such a vector.

If `nbasis` = 3, `names` defaults to c('const', 'cos', 'sin'). If `nbasis` > 3, `names` defaults to c('const', outer(c('cos', 'sin'), 1:((nbasis-1)/2), paste, sep='')).

If names = NA, no names are used.

`axes` an optional list used by selected `plot` functions to create custom `axes`. If this `axes` argument is not `NULL`, functions `plot.basisfd`, `plot.fd`, `plot.fdSmooth` `plotfit.fd`, `plotfit.fdSmooth`, and `plot.Lfd` will create axes via `x$axes[[1]]` and `x$axes[-1]`. The primary example of this is to create `CanadianWeather` plots using `list("axesIntervals")`

Details

Functional data objects are constructed by specifying a set of basis functions and a set of coefficients defining a linear combination of these basis functions. The Fourier basis is a system that is usually used for periodic functions. It has the advantages of very fast computation and great flexibility. If the data are considered to be nonperiod, the Fourier basis is usually preferred. The first Fourier basis function is the constant function. The remainder are sine and cosine pairs with integer multiples of the base period. The number of basis functions generated is always odd.

Value

a basis object with the type `fourier`.

See Also

`basisfd`, `create.bspline.basis`, `create.constant.basis`, `create.exponential.basis`, `create.monomial.basis`, `create.polygonal.basis`, `create.power.basis`

Examples

```
# Create a minimal Fourier basis for annual data
#  using 3 basis functions
yearbasis3 <- create.fourier.basis(c(0,365),
                    axes=list("axesIntervals") )
#  plot the basis
plot(yearbasis3)
```

```
# Identify the months with letters
plot(yearbasis3, axes=list('axesIntervals', labels=monthLetters))

# The same labels as part of the basis object
yearbasis3. <- create.fourier.basis(c(0,365),
       axes=list("axesIntervals", labels=monthLetters) )
plot(yearbasis3.)

# set up the Fourier basis for the monthly temperature data,
#  using 9 basis functions with period 12 months.
monthbasis <- create.fourier.basis(c(0,12), 9, 12.0)

#  plot the basis
plot(monthbasis)

# Create a false Fourier basis using 1 basis function.
falseFourierBasis <- create.fourier.basis(nbasis=1)
#  plot the basis:  constant
plot(falseFourierBasis)
```

`create.monomial.basis` *Create a Monomial Basis*

Description

Creates a set of basis functions consisting of powers of the argument.

Usage

```
create.monomial.basis(rangeval=c(0, 1), nbasis=NULL,
         exponents=NULL, dropind=NULL, quadvals=NULL,
         values=NULL, basisvalues=NULL, names='monomial',
         axes=NULL)
```

Arguments

`rangeval`	a vector of length 2 containing the initial and final values of the interval over which the functional data object can be evaluated.
`nbasis`	the number of basis functions = `length(exponents)`. Default = if(is.null(exponents)) 2 else length(exponents).
`exponents`	the nonnegative integer powers to be used. By default, these are 0, 1, 2, ..., (nbasis-1).
`dropind`	a vector of integers specifiying the basis functions to be dropped, if any. For example, if it is required that a function be zero at the left boundary when rangeval[1] = 0, this is achieved by dropping the first basis function, the only one that is nonzero at that point.

quadvals — a matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of quadvals contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to 1, 4, 2, 4, ..., 2, 4, 1.

values — a list of matrices with one row for each row of quadvals and one column for each basis function. The elements of the list correspond to the basis functions and their derivatives evaluated at the quadrature points contained in the first column of quadvals.

basisvalues — A list of lists, allocated by code such as vector("list",1). This field is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each list within the vector corresponds to a specific set of argument values, and must have at least two components, which may be tagged as you wish. 'The first component in an element of the list vector contains the argument values. The second component in an element of the list vector contains a matrix of values of the basis functions evaluated at the arguments in the first component. The third and subsequent components, if present, contain matrices of values their derivatives up to a maximum derivative order. Whenever function getbasismatrix is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system. Even this time can be avoided by direct retrieval of the desired array. For example, you might set up a vector of argument values called "evalargs" along with a matrix of basis function values for these argument values called "basismat". You might want too use names like "args" and "values", respectively for these. You would then assign them to basisvalues with code such as the following:

basisobj\$basisvalues <- vector("list",1)

basisobj\$basisvalues[[1]] <- list(args=evalargs, values=basismat)

names — either a character vector of the same length as the number of basis functions or a simple stem used to construct such a vector.

For monomial bases, this defaults to paste('monomial', 1:nbreaks, sep='').

axes — an optional list used by selected plot functions to create custom axes. If this axes argument is not NULL, functions plot.basisfd, plot.fd, plot.fdSmooth plotfit.fd, plotfit.fdSmooth, and plot.Lfd will create axes via x$axes[[1]] and x$axes[-1]. The primary example of this uses list("axesIntervals",...), e.g., with Fourier bases to create CanadianWeather plots.

Value

a basis object with the type monom.

See Also

basisfd, link{create.basis} create.bspline.basis, create.constant.basis, create.fourier.basis, create.exponential.basis, create.polygonal.basis, create.power.basis

Examples

```
##
## simplest example: one constant 'basis function'
##
m0 <- create.monomial.basis(nbasis=1)
plot(m0)

##
## Create a monomial basis over the interval [-1,1]
##  consisting of the first three powers of t
##
basisobj <- create.monomial.basis(c(-1,1), 5)
#  plot the basis
plot(basisobj)

##
## rangeval of class Date or POSIXct
##
# Date
invasion1 <- as.Date('1775-09-04')
invasion2 <- as.Date('1812-07-12')
earlyUS.Canada <- c(invasion1, invasion2)
BspInvade1 <- create.monomial.basis(earlyUS.Canada)

# POSIXct
AmRev.ct <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev1.ct <- create.monomial.basis(AmRev.ct)
```

`create.polygonal.basis` *Create a Polygonal Basis*

Description

A basis is set up for constructing polygonal lines, consisting of straight line segments that join together.

Usage

```
create.polygonal.basis(rangeval=NULL, argvals=NULL, dropind=NULL,
        quadvals=NULL, values=NULL, basisvalues=NULL, names='polygon',
        axes=NULL)
```

Arguments

`rangeval`	a numeric vector of length 2 defining the interval over which the functional data object can be evaluated; default value is `if(is.null(argvals)) 0:1 else range(argvals)`.

If length(rangeval) == 1 and rangeval <= 0, this is an error. Otherwise, if length(rangeval) == 1, rangeval is replaced by c(0,rangeval).

If length(rangeval)>2 and argvals is not provided, this extra long rangeval argument is assigned to argvals, and then rangeval = range(argvale).

argvals — a strictly increasing vector of argument values at which line segments join to form a polygonal line.

dropind — a vector of integers specifiying the basis functions to be dropped, if any. For example, if it is required that a function be zero at the left boundary, this is achieved by dropping the first basis function, the only one that is nonzero at that point.

quadvals — a matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of quadvals contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to These are proportional to 1, 4, 2, 4, ..., 2, 4, 1.

values — a list containing the basis functions and their derivatives evaluated at the quadrature points contained in the first column of quadvals .

basisvalues — A list of lists, allocated by code such as vector("list",1). This is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each sublist corresponds to a specific set of argument values, and must have at least two components, which may be named as you wish. The first component of a sublist contains the argument values. The second component contains a matrix of values of the basis functions evaluated at the arguments in the first component. The third and subsequent components, if present, contain matrices of values their derivatives up to a maximum derivative order. Whenever function getbasismatrix is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system. Even this time can be avoided by direct retrieval of the desired array. For example, you might set up a vector of argument values called "evalargs" along with a matrix of basis function values for these argument values called "basismat". You might want too use tags like "args" and "values", respectively for these. You would then assign them to basisvalues with code such as the following:

basisobj\$basisvalues <- vector("list",1)

basisobj\$basisvalues[[1]] <- list(args=evalargs, values=basismat)

names — either a character vector of the same length as the number of basis functions or a single character string to which 1:nbasis are appended as paste(names,1:nbasis,sep=''. For example, if nbasis = 4, this defaults to c('polygon1','polygon2','polygon3','polygon4').

axes — an optional list used by selected plot functions to create custom axes. If this axes argument is not NULL, functions plot.basisfd, plot.fd, plot.fdSmooth plotfit.fd, plotfit.fdSmooth, and plot.Lfd will create axes via x\$axes[[1]] and x\$axes[-1]. The primary example of this uses list("axesIntervals",...), e.g., with Fourier bases to create CanadianWeather plots

Details

The actual basis functions consist of triangles, each with its apex over an argument value. Note that in effect the polygonal basis is identical to a B-spline basis of order 2 and a knot or break value at each argument value. The range of the polygonal basis is set to the interval defined by the smallest and largest argument values.

Value

a basis object with the type `polyg`.

See Also

`basisfd`, `create.bspline.basis`, `create.constant.basis`, `create.exponential.basis`, `create.fourier.basis`, `create.monomial.basis`, `create.power.basis`

Examples

```
#  Create a polygonal basis over the interval [0,1]
#  with break points at 0, 0.1, ..., 0.95, 1
(basisobj <- create.polygonal.basis(seq(0,1,0.1)))
#  plot the basis
plot(basisobj)
```

`create.power.basis` *Create a Power Basis Object*

Description

The basis system is a set of powers of argument x. That is, a basis function would be `x^exponent`, where `exponent` is a vector containing a set of powers or exponents. The power basis would normally only be used for positive values of x, since the power of a negative number is only defined for nonnegative integers, and the exponents here can be any real numbers.

Usage

```
create.power.basis(rangeval=c(0, 1), nbasis=NULL, exponents=NULL,
            dropind=NULL, quadvals=NULL, values=NULL,
            basisvalues=NULL, names='power', axes=NULL)
```

Arguments

`rangeval`	a vector of length 2 with the first element being the lower limit of the range of argument values, and the second the upper limit. Of course the lower limit must be less than the upper limit.
`nbasis`	the number of basis functions = `length(exponents)`. Default = if(is.null(exponents)) 2 else length(exponents).
`exponents`	a numeric vector of length `nbasis` containing the powers of `x` in the basis.

dropind — a vector of integers specifiying the basis functions to be dropped, if any. For example, if it is required that a function be zero at the left boundary, this is achieved by dropping the first basis function, the only one that is nonzero at that point.

quadvals — a matrix with two columns and a number of rows equal to the number of quadrature points for numerical evaluation of the penalty integral. The first column of quadvals contains the quadrature points, and the second column the quadrature weights. A minimum of 5 values are required for each inter-knot interval, and that is often enough. For Simpson's rule, these points are equally spaced, and the weights are proportional to 1, 4, 2, 4, ..., 2, 4, 1.

values — a list of matrices with one row for each row of quadvals and one column for each basis function. The elements of the list correspond to the basis functions and their derivatives evaluated at the quadrature points contained in the first column of quadvals.

basisvalues — A list of lists, allocated by code such as vector("list",1). This field is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each list within the vector corresponds to a specific set of argument values, and must have at least two components, which may be tagged as you wish. 'The first component in an element of the list vector contains the argument values. The second component in an element of the list vector contains a matrix of values of the basis functions evaluated at the arguments in the first component. The third and subsequent components, if present, contain matrices of values their derivatives up to a maximum derivative order. Whenever function getbasismatrix is called, it checks the first list in each row to see, first, if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match. This takes time, of course, but is much faster than re-evaluation of the basis system. Even this time can be avoided by direct retrieval of the desired array. For example, you might set up a vector of argument values called "evalargs" along with a matrix of basis function values for these argument values called "basismat". You might want too use names like "args" and "values", respectively for these. You would then assign them to basisvalues with code such as the following:

basisobj\$basisvalues <- vector("list",1)

basisobj\$basisvalues[[1]] <- list(args=evalargs, values=basismat)

names — either a character vector of the same length as the number of basis functions or a simple stem used to construct such a vector.

For power bases, this defaults to paste(power', 0:(nbasis-1), sep='').

axes — an optional list used by selected plot functions to create custom axes. If this axes argument is not NULL, functions plot.basisfd, plot.fd, plot.fdSmooth plotfit.fd, plotfit.fdSmooth, and plot.Lfd will create axes via x$axes[[1]] and x$axes[-1]. The primary example of this uses list("axesIntervals",...), e.g., with Fourier bases to create CanadianWeather plots

Details

The power basis differs from the monomial basis in two ways. First, the powers may be nonintegers. Secondly, they may be negative. Consequently, a power basis is usually used with arguments that only take positive values, although a zero value can be tolerated if none of the powers are negative.

Value

a basis object of type power.

See Also

basisfd, create.bspline.basis, create.constant.basis, create.exponential.basis, create.fourier.basis, create.monomial.basis, create.polygonal.basis,

Examples

```
#  Create a power basis over the interval [1e-7,1]
#  with powers or exponents -1, -0.5, 0, 0.5 and 1
basisobj <- create.power.basis(c(1e-7,1), 5, seq(-1,1,0.5))
#  plot the basis
plot(basisobj)
```

CSTR *Continuously Stirred Tank Reactor*

Description

Functions for solving the Continuously Stirred Tank Reactor (CSTR) Ordinary Differential Equations (ODEs). A solution for observations where metrology error is assumed to be negligible can be obtained via lsoda(y, Time, CSTR2, parms); CSTR2 calls CSTR2in. When metrology error can not be ignored, use CSTRfn (which calls CSTRfitLS). To estimate parameters in the CSTR differential equation system (kref, EoverR, a, and / or b), pass CSTRres to nls. If nls fails to converge, first use optim or nlminb with CSTRsse, then pass the estimates to nls.

Usage

```
CSTR2in(Time, condition =
   c('all.cool.step', 'all.hot.step', 'all.hot.ramp', 'all.cool.ramp',
     'Tc.hot.exponential', 'Tc.cool.exponential', 'Tc.hot.ramp',
     'Tc.cool.ramp', 'Tc.hot.step', 'Tc.cool.step'),
   tau=1)
CSTR2(Time, y, parms)

CSTRfitLS(coef, datstruct, fitstruct, lambda, gradwrd=FALSE)
CSTRfn(parvec, datstruct, fitstruct, CSTRbasis, lambda, gradwrd=TRUE)
CSTRres(kref=NULL, EoverR=NULL, a=NULL, b=NULL,
        datstruct, fitstruct, CSTRbasis, lambda, gradwrd=FALSE)
CSTRsse(par, datstruct, fitstruct, CSTRbasis, lambda)
```

Arguments

Time	The time(s) for which computation(s) are desired
condition	a character string with the name of one of ten preprogrammed input scenarios.
tau	time for exponential decay of exp(-1) under condition = 'Tc.hot.exponential' or 'Tc.cool.exponential'; ignored for other values of 'condition'.
y	Either a vector of length 2 or a matrix with 2 columns giving the observation(s) on Concentration and Temperature for which computation(s) are desired
parms	a list of CSTR model parameters passed via the lsoda 'parms' argument. This list consists of the following 3 components: • fitstruct a list with 12 components describing the structure for fitting. This is the same as the 'fitstruct' argument of 'CSTRfitLS' and 'CSTRfn' without the 'fit' component; see below. • condition a character string identifying the inputs to the simulation. Currently, any of the following are accepted: 'all.cool.step', 'all.hot.step', 'all.hot.ramp', 'all.cool.ramp', 'Tc.hot.exponential', 'Tc.cool.exponential', 'Tc.hot.ramp', 'Tc.cool.ramp', 'Tc.hot.step', or 'Tc.cool.step'. • Tlim end time for the computations.
coef	a matrix with one row for each basis function in fitstruct and columns c("Conc", "Temp") or a vector form of such a matrix.
datstruct	a list describing the structure of the data. CSTRfitLS uses the following components: • basismat, Dbasismat basis coefficent matrices with one row for each observation and one column for each basis vector. These are typically produced by code something like the following: basismat <- eval.basis(Time, CSTRbasis) Dbasismat <- eval.basis(Time, CSTRbasis, 1) • Cwt, Twt scalar variances of 'fd' functional data objects for Concentration and Temperature used to place the two series on comparable scales. • y a matrix with 2 columns for the observed 'Conc' and 'Temp'. • quadbasismat, Dquadbasismat basis coefficient matrices with one row for each quadrature point and one column for each basis vector. These are typically produced by code something like the following: quadbasismat <- eval.basis(quadpts, CSTRbasis) Dquadbasismat <- eval.basis(quadpts, CSTRbasis, 1) • Fc, F., CA0, T0, Tc input series for CSTRfitLS and CSTRfn as the output list produced by CSTR2in. • quadpts Quadrature points created by 'quadset' and stored in CSTRbasis[["quadvals"]][, "quadpts"]. • quadwts Quadrature weights created by 'quadset' and stored in CSTRbasis[["quadvals"]][, "quadpts"].
fitstruct	a list with 14 components: • V volume in cubic meters • Cp concentration in cal/(g.K) for computing betaTC and betaTT; see details below.

- rho density in grams per cubic meter
- delH cal/kmol
- Cpc concentration in cal/(g.K) used for computing alpha; see details below.
- Tref reference temperature.
- kref reference value
- EoverR E/R in units of K/1e4
- a scale factor for Fco in alpha; see details below.
- b power of Fco in alpha; see details below.
- Tcin Tc input temperature vector.
- fit logical vector of length 2 indicating whether Contentration or Temperature or both are considered to be observed and used for parameter estimation.
- coef0 data.frame(Conc = Cfdsmth[["coef"]], Temp = Tfdsmth[["coef"]]), where Cfdsmth and Tfdsmth are the objects returned by smooth.basis when applied to the observations on Conc and Temp, respectively.
- estimate logical vector of length 4 indicating which of kref, EoverR, a and b are taken from 'parvec'; all others are taken from 'fitstruct'.

`lambda` a 2-vector of rate parameters 'lambdaC' and 'lambdaT'.

`gradwrd` a logical scalar TRUE if the gradient is to be returned as well as the residuals matrix.

`parvec, par` initial values for the parameters specified by fitstruct[["estimate"]] to be estimated.

`CSTRbasis` Quadrature basis returned by 'quadset'.

`kref, EoverR, a, b`
the kref, EoverR, a, and b coefficients of the CSTR model as individual arguments of CSTRres to support using 'nls' with the CSTR model. Those actually provided by name will be estimated; the others will be taken from '.fitstruct'; see details.

Details

Ramsay et al. (2007) considers the following differential equation system for a continuously stirred tank reactor (CSTR):

dC/dt = (-betaCC(T, F.in)*C + F.in*C.in)

dT/dt = (-betaTT(Fcvec, F.in)*T + betaTC(T, F.in)*C + alpha(Fcvec)*T.co)

where

betaCC(T, F.in) = kref*exp(-1e4*EoverR*(1/T - 1/Tref)) + F.in

betaTT(Fcvec, F.in) = alpha(Fcvec) + F.in

betaTC(T, F.in) = (-delH/(rho*Cp))*betaCC(T, F.in)

$$alpha(Fcvec) = (a * Fcvec^(b + 1)/(K1 * (Fcvec + K2 * Fcvec^b)))$$

K1 = V*rho*Cp

K2 = 1/(2*rhoc*Cpc)

The four functions CSTR2in, CSTR2, CSTRfitLS, and CSTRfn compute coefficients of basis vectors for two different solutions to this set of differential equations. Functions CSTR2in and CSTR2 work with 'lsoda' to provide a solution to this system of equations. Functions CSTSRitLS and CSTRfn are used to estimate parameters to fit this differential equation system to noisy data. These solutions are conditioned on specified values for kref, EoverR, a, and b. The other function, CSTRres, support estimation of these parameters using 'nls'.

CSTR2in translates a character string 'condition' into a data.frame containing system inputs for which the reaction of the system is desired. CSTR2 calls CSTR2in and then computes the corresponding predicted first derivatives of CSTR system outputs according to the right hand side of the system equations. CSTR2 can be called by 'lsoda' in the 'deSolve' package to actually solve the system of equations. To solve the CSTR equations for another set of inputs, the easiest modification might be to change CSTR2in to return the desired inputs. Another alternative would be to add an argument 'input.data.frame' that would be used in place of CSTR2in when present.

CSTRfitLS computes standardized residuals for systems outputs Conc, Temp or both as specified by fitstruct[["fit"]], a logical vector of length 2. The standardization is sqrt(datstruct[["Cwt"]]) and / or sqrt(datstruct[["Twt"]]) for Conc and Temp, respectively. CSTRfitLS also returns standardized deviations from the predicted first derivatives for Conc and Temp.

CSTRfn uses a Gauss-Newton optimization to estimates the coefficients of CSTRbasis to minimize the weighted sum of squares of residuals returned by CSTRfitLS.

CSTRres provides an interface between 'nls' and 'CSTRfn'. It gets the parameters to be estimated via the official function arguments, kref, EoverR, a, and / or b. The subset of these parameters to estimate must be specified both directly in the function call to 'nls' and indirectly via fitstruct[["estimate"]]. CSTRres gets the other CSTRfn arguments (datstruct, fitstruct, CSTRbasis, and lambda) via the 'data' argument of 'nls'.

CSTRsse computes sum of squares of residuals for use with optim or nlminb.

Value

`CSTR2in` a matrix with number of rows = length(Time) and columns for F., CA0, T0, Tcin, and Fc. This gives the inputs to the CSTR simulation for the chosen 'condition'.

`CSTR2` a list with one component being a matrix with number of rows = length(tobs) and 2 columns giving the first derivatives of Conc and Temp according to the right hand side of the differential equation. CSTR2 calls CSTR2in to get its inputs.

`CSTRfitLS` a list with one or two components as follows:

- res a list with two components
  Sres = a matrix giving the residuals between observed and predicted datstruct[["y"]] divided by sqrt(datstruct[[c("Cwt", "Twt")]]) so the result is dimensionless. dim(Sres) = dim(datstruct[["y"]]). Thus, if datstruct[["y"]] has only one column, 'Sres' has only one column.
  Lres = a matrix with two columns giving the difference between left and right hand sides of the CSTR differential equation at all the quadrature points. dim(Lres) = c(nquad, 2).
- Dres If gradwrd=TRUE, a list with two components:

DSres = a matrix with one row for each element of res[["Sres"]] and two columns for each basis function.
DLres = a matrix with two rows for each quadrature point and two columns for each basis function.
If gradwrd=FALSE, this component is not present.

`CSTRfn` a list with five components:

- res the 'res' component of the final 'CSTRfitLS' object reformatted with its component Sres first followed by Lres, using with(CSTRfitLS(...)[["res"]], c(Sres, Lres)).
- Dres one of two very different gradient matrices depending on the value of 'gradwrd'.
  If gradwrd = TRUE, Dres is a matrix with one row for each observation value to match and one column for each parameter taken from 'parvec' per fitstruct[["estimate"]]. Also, if fitstruct[["fit"]] = c(1,1), CSTRfn tries to match both Concentration and Temperature, and rows corresponding to Concentration come first following by rows corresponding to Temperature.
  If gradwrd = FALSE, this is the 'Dres' component of the final 'CSTRfitLS' object reformatted as follows:
  Dres <- with(CSTRfitLS(...)[["Dres"]], rbind(DSres, DLres))
- fitstruct a list components matching the 'fitstruct' input, with coefficients estimated replaced by their initial values from parvec and with coef0 replace by its final estimate.
- df estimated degrees of freedom as the trace of the appropriate matrix.
- gcv the Generalized cross validation estimate of the mean square error, as discussed in Ramsay and Silverman (2006, sec. 5.4).

`CSTRres` the 'res' component of CSTRfd(...) as a column vector. This allows us to use 'nls' with the CSTR model. This can be especially useful as 'nls' has several helper functions to facilitate evaluating goodness of fit and and uncertainty in parameter estimates.

`CSTRsse` sum(res*res) from CSTRfd(...). This allows us to use 'optim' or 'nlminb' with the CSTR model. This can also be used to obtain starting values for 'nls' in cases where 'nls' fails to converge from the initial provided starting values. Apart from 'par', the other arguments 'datstruct', 'fitstruct', 'CSTRbasis', and 'lambda', must be passed via '...' in 'optim' or 'nlminb'.

References

Ramsay, J. O., Hooker, G., Cao, J. and Campbell, D. (2007) Parameter estimation for differential equations: A generalized smoothing approach (with discussion). *Journal of the Royal Statistical Society*, Series B, 69, 741-796.

Ramsay, J. O., and Silverman, B. W. (2006) *Functional Data Analysis*, 2nd ed., Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

lsoda nls

Examples

```
###
###
### 1.  lsoda(y, times, func=CSTR2, parms=...)
###
###
#  The system of two nonlinear equations has five forcing or
#  input functions.
#  These equations are taken from
#  Marlin, T. E. (2000) Process Control, 2nd Edition, McGraw Hill,
#  pages 899-902.
##
##  Set up the problem
##
fitstruct <- list(V    = 1.0,#  volume in cubic meters
                  Cp   = 1.0,#  concentration in cal/(g.K)
                  rho  = 1.0,#  density in grams per cubic meter
                  delH = -130.0,# cal/kmol
                  Cpc  = 1.0,#  concentration in cal/(g.K)
                  rhoc = 1.0,#  cal/kmol
                  Tref = 350)#  reference temperature
#  store true values of known parameters
EoverRtru = 0.83301#   E/R in units K/1e4
kreftru   = 0.4610 #   reference value
atru      = 1.678#     a in units (cal/min)/K/1e6
btru      = 0.5#       dimensionless exponent

#% enter these parameter values into fitstruct

fitstruct[["kref"]]   = kreftru#
fitstruct[["EoverR"]] = EoverRtru#  kref = 0.4610
fitstruct[["a"]]      = atru#       a in units (cal/min)/K/1e6
fitstruct[["b"]]      = btru#       dimensionless exponent

Tlim  = 64#    reaction observed over interval [0, Tlim]
delta = 1/12#  observe every five seconds
tspan = seq(0, Tlim, delta)#

coolStepInput <- CSTR2in(tspan, 'all.cool.step')

#  set constants for ODE solver

#  cool condition solution
#  initial conditions

Cinit.cool = 1.5965#  initial concentration in kmol per cubic meter
Tinit.cool = 341.3754# initial temperature in deg K
yinit = c(Conc = Cinit.cool, Temp=Tinit.cool)
```

```
#  load cool input into fitstruct

fitstruct[["Tcin"]] = coolStepInput[, "Tcin"];

#  solve  differential equation with true parameter values

if (require(deSolve)) {
coolStepSoln <- lsoda(y=yinit, times=tspan, func=CSTR2,
  parms=list(fitstruct=fitstruct, condition='all.cool.step', Tlim=Tlim) )
}
###
###
### 2.  CSTRfn
###
###

# See the script in '~R\library\fda\scripts\CSTR\CSTR_demo.R'
#  for more examples.
```

`cycleplot.fd` *Plot Cycles for a Periodic Bivariate Functional Data Object*

Description

A plotting function for data such as the knee-hip angles in the gait data or temperature-precipitation curves for the weather data.

Usage

```
cycleplot.fd(fdobj, matplt=TRUE, nx=201, ...)
```

Arguments

`fdobj`	a bivariate functional data object to be plotted.
`matplt`	if TRUE, all cycles are plotted simultaneously; otherwise each cycle in turn is plotted.
`nx`	the number of argument values in a fine mesh to be plotted. Increase the default number of 201 if the curves have a lot of detail in them.
`...`	additional plotting parameters such as axis labels and etc. that are used in all plot functions.

Value

None

Side Effects

A plot of the cycles

See Also

`plot.fd`, `plotfit.fd`,

`Data2fd` *Create a functional data object from data*

Description

This function converts an array `y` of function values plus an array `argvals` of argument values into a functional data object. This function tries to do as much for the user as possible in setting up a call to function `smooth.basis`. Be warned that the result may not be a satisfactory smooth of the data, and consequently that it may be necessary to use function `smooth.basis` instead, the help file for which provides a great deal more information than is provided here. Also, function `Data2fd` can swap the first two arguments, `argvals` and `y` if it appears that they have been included in reverse order. A warning message is returned if this swap takes place. Any such automatic decision, though, has the possibility of being wrong, and the results should be carefully checked. Preferably, the order of the arguments should be respected: `argvals` comes first and `y` comes second.

Usage

```
Data2fd(argvals=NULL, y=NULL, basisobj=NULL, nderiv=NULL,
        lambda=3e-8/diff(as.numeric(range(argvals))),
        fdnames=NULL, covariates=NULL, method="chol",
        dfscale=1)
```

Arguments

`argvals`	a set of argument values. If this is a vector, the same set of argument values is used for all columns of `y`. If `argvals` is a matrix, the columns correspond to the columns of `y`, and contain the argument values for that replicate or case. Dimensions for `argvals` must match the first dimensions of `y`, though `y` can have more dimensions. For example, if dim(y) = c(9, 5, 2), `argvals` can be a vector of length 9 or a matrix of dimensions c(9, 5) or an array of dimensions c(9, 5, 2).
`y`	an array containing sampled values of curves. If `y` is a vector, only one replicate and variable are assumed. If `y` is a matrix, rows must correspond to argument values and columns to replications or cases, and it will be assumed that there is only one variable per observation. If `y` is a three-dimensional array, the first dimension (rows) corresponds to argument values, the second (columns) to replications, and the third (layers) to variables within replications. Missing values are permitted, and the number of values may vary from one replication to another. If this is the case, the number of rows must

equal the maximum number of argument values, and columns of y having fewer values must be padded out with NA's.

basisobj — One of the following:

- basisfd a functional basis object (class basisfd).
- fd a functional data object (class fd), from which its basis component is extracted.
- fdPar a functional parameter object (class fdPar), from which its basis component is extracted.
- integer an integer giving the order of a B-spline basis, create.bspline.basis(argvals,norder=ba
- numeric vector specifying the knots for a B-spline basis, create.bspline.basis(basisobj)
- NULL Defaults to create.bspline.basis(argvals).

nderiv — Smoothing typically specified as an integer order for the derivative whose square is integrated and weighted by lambda to smooth. By default, if basisobj[['type']] == 'bspline', the smoothing operator is int2Lfd(max(0,norder-2)).

A general linear differential operator can also be supplied.

lambda — weight on the smoothing operator specified by nderiv.

fdnames — Either a character vector of length 3 or a named list of length 3. In either case, the three elements correspond to the following:

- argname name of the argument, e.g. "time" or "age".
- repname a description of the cases, e.g. "reps" or "weather stations"
- value the name of the observed function value, e.g. "temperature"

If fdnames is a list, the components provide labels for the levels of the corresponding dimension of y.

covariates — the observed values in y are assumed to be primarily determined by the height of the curve being estimated. However, from time to time certain values can also be influenced by other known variables. For example, multi-year sets of climate variables may be also determined by the presence of absence of an El Nino event, or a volcanic eruption. One or more of these covariates can be supplied as an n by p matrix, where p is the number of such covariates. When such covariates are available, the smoothing is called "semi-parametric." Matrices or arrays of regression coefficients are then estimated that define the impacts of each of these covariates for each curve and each variable.

method — by default the function uses the usual textbook equations for computing the coefficients of the basis function expansions. But, as in regression analysis, a price is paid in terms of rounding error for such computations since they involved cross-products of basis function values. Optionally, if method is set equal to the string "qr", the computation uses an algorithm based on the qr-decomposition which is more accurate, but will require substantially more computing time when n is large, meaning more than 500 or so. The default is "chol", referring the Choleski decomposition of a symmetric positive definite matrix.

dfscale — the generalized cross-validation or "gcv" criterion that is often used to determine the size of the smoothing parameter involves the subtraction of an measure of degrees of freedom from n. Chong Gu has argued that multiplying this degrees of freedom measure by a constant slightly greater than 1, such as 1.2, can produce better decisions about the level of smoothing to be used. The default value is, however, 1.0.

Details

This function tends to be used in rather simple applications where there is no need to control the roughness of the resulting curve with any great finesse. The roughness is essentially controlled by how many basis functions are used. In more sophisticated applications, it would be better to use the function `smooth.basisPar`.

Value

an object of the `fd` class containing:

- coefs the coefficient array
- basis a basis object
- fdnames a list containing names for the arguments, function values and variables

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

`smooth.basisPar`, `smooth.basis`, `project.basis`, `smooth.fd`, `smooth.monotone`, `smooth.pos`, `day.5`

Examples

```
##
## Simplest possible example:  constant function
##
# 1 basis, order 1 = degree 0 = constant function
b1.1 <- create.bspline.basis(nbasis=1, norder=1)
# data values: 1 and 2, with a mean of 1.5
y12 <- 1:2
# smooth data, giving a constant function with value 1.5
fd1.1 <- Data2fd(y12, basisobj=b1.1)
plot(fd1.1)
# now repeat the analysis with some smoothing, which moves the
# toward 0.
fd1.1.5 <- Data2fd(y12, basisobj=b1.1, lambda=0.5)
#  values of the smooth:
# fd1.1.5 = sum(y12)/(n+lambda*integral(over arg=0 to 1 of 1))
#         = 3 / (2+0.5) = 1.2
eval.fd(seq(0, 1, .2), fd1.1.5)
##
## step function smoothing
##
# 2 step basis functions: order 1 = degree 0 = step functions
b1.2 <- create.bspline.basis(nbasis=2, norder=1)
```

```
#  fit the data without smoothing
fd1.2 <- Data2fd(1:2, basisobj=b1.2)
# plot the result:  A step function:  1 to 0.5, then 2
op <- par(mfrow=c(2,1))
plot(b1.2, main='bases')
plot(fd1.2, main='fit')
par(op)
##
## Simple oversmoothing
##
# 3 step basis functions: order 1 = degree 0 = step functions
b1.3 <- create.bspline.basis(nbasis=3, norder=1)
#  smooth the data with smoothing
fd1.3.5 <- Data2fd(y12, basisobj=b1.3, lambda=0.5)
#  plot the fit along with the points
plot(0:1, c(0, 2), type='n')
points(0:1, y12)
lines(fd1.3.5)
# Fit = penalized least squares with penalty =
#          = lambda * integral(0:1 of basis^2),
#            which shrinks the points towards 0.
# X1.3 = matrix(c(1,0, 0,0, 0,1), 2)
# XtX = crossprod(X1.3) = diag(c(1, 0, 1))
# penmat = diag(3)/3
#        = 3x3 matrix of integral(over arg=0:1 of basis[i]*basis[j])
# Xt.y = crossprod(X1.3, y12) = c(1, 0, 2)
# XtX + lambda*penmat = diag(c(7, 1, 7)/6
# so coef(fd1.3.5) = solve(XtX + lambda*penmat, Xt.y)
#                  = c(6/7, 0, 12/7)
##
## linear spline fit
##
# 3 bases, order 2 = degree 1
b2.3 <- create.bspline.basis(norder=2, breaks=c(0, .5, 1))
# interpolate the values 0, 2, 1
fd2.3 <- Data2fd(c(0,2,1), basisobj=b2.3, lambda=0)
#  display the coefficients
round(fd2.3$coefs, 4)
# plot the results
op <- par(mfrow=c(2,1))
plot(b2.3, main='bases')
plot(fd2.3, main='fit')
par(op)
# apply some smoothing
fd2.3. <- Data2fd(c(0,2,1), basisobj=b2.3, lambda=1)
op <- par(mfrow=c(2,1))
plot(b2.3, main='bases')
plot(fd2.3., main='fit', ylim=c(0,2))
par(op)
all.equal(
 unclass(fd2.3)[-1],
 unclass(fd2.3.)[-1])
##** CONCLUSION:
```

```
##** The only differences between fd2.3 and fd2.3.
##** are the coefficients, as we would expect.

##
## quadratic spline fit
##
# 4 bases, order 3 = degree 2 = continuous, bounded, locally quadratic
b3.4 <- create.bspline.basis(norder=3, breaks=c(0, .5, 1))
# fit values c(0,4,2,3) without interpolation
fd3.4 <- Data2fd(c(0,4,2,3), basisobj=b3.4, lambda=0)
round(fd3.4$coefs, 4)
op <- par(mfrow=c(2,1))
plot(b3.4)
plot(fd3.4)
points(c(0,1/3,2/3,1), c(0,4,2,3))
par(op)
#  try smoothing
fd3.4. <- Data2fd(c(0,4,2,3), basisobj=b3.4, lambda=1)
round(fd3.4.$coef, 4)
op <- par(mfrow=c(2,1))
plot(b3.4)
plot(fd3.4., ylim=c(0,4))
points(seq(0,1,len=4), c(0,4,2,3))
par(op)
##
##  Two simple Fourier examples
##
gaitbasis3 <- create.fourier.basis(nbasis=5)
gaitfd3    <- Data2fd(gait, basisobj=gaitbasis3)
# plotfit.fd(gait, seq(0,1,len=20), gaitfd3)
#    set up the fourier basis
daybasis <- create.fourier.basis(c(0, 365), nbasis=65)
#  Make temperature fd object
#  Temperature data are in 12 by 365 matrix tempav
#    See analyses of weather data.
tempfd <- Data2fd(CanadianWeather$dailyAv[,,"Temperature.C"],
                  day.5, daybasis)
#  plot the temperature curves
par(mfrow=c(1,1))
plot(tempfd)
##
## argvals of class Date and POSIXct
##
#  These classes of time can generate very large numbers when converted to
#  numeric vectors.  For basis systems such as polynomials or splines,
#  severe rounding error issues can arise if the time interval for the
#  data is very large.  To offset this, it is best to normalize the
#  numeric version of the data before analyzing them.
#  Date class time unit is one day, divide by 365.25.
invasion1 <- as.Date('1775-09-04')
invasion2 <- as.Date('1812-07-12')
earlyUS.Canada <- as.numeric(c(invasion1, invasion2))/365.25
BspInvasion <- create.bspline.basis(earlyUS.Canada)
```

```
earlyYears  <- seq(invasion1, invasion2, length.out=7)
earlyQuad   <- (as.numeric(earlyYears-invasion1)/365.25)^2
earlyYears  <- as.numeric(earlyYears)/365.25
fitQuad <- Data2fd(earlyYears, earlyQuad, BspInvasion)
# POSIXct: time unit is one second, divide by 365.25*24*60*60
rescale     <- 365.25*24*60*60
AmRev.ct    <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev.ct   <- create.bspline.basis(as.numeric(AmRev.ct)/rescale)
AmRevYrs.ct <- seq(AmRev.ct[1], AmRev.ct[2], length.out=14)
AmRevLin.ct <- as.numeric(AmRevYrs.ct-AmRev.ct[1])
AmRevYrs.ct <- as.numeric(AmRevYrs.ct)/rescale
AmRevLin.ct <- as.numeric(AmRevLin.ct)/rescale
fitLin.ct   <- Data2fd(AmRevYrs.ct, AmRevLin.ct, BspRev.ct)
```

`dateAccessories` *Numeric and character vectors to facilitate working with dates*

Description

Numeric and character vectors to simplify functional data computations and plotting involving dates.

Format

- dayOfYear a numeric vector = 1:365 with names 'jan01' to 'dec31'.
- dayOfYearShifted a numeric vector = c(182:365, 1:181) with names 'jul01' to 'jun30'.
- day.5 a numeric vector = dayOfYear-0.5 = 0.5, 1.5, ..., 364.5
- daysPerMonth a numeric vector of the days in each month (ignoring leap years) with names = month.abb
- monthEnd a numeric vector of cumsum(daysPerMonth) with names = month.abb
- monthEnd.5 a numeric vector of the middle of the last day of each month with names = month.abb = c(Jan=30.5, Feb=58.5, ..., Dec=364.5)
- monthBegin.5 a numeric vector of the middle of the first day of each month with names - month.abb = c(Jan=0.5, Feb=31.5, ..., Dec=334.5)
- monthMid a numeric vector of the middle of the month = (monthBegin.5 + monthEnd.5)/2
- monthLetters A character vector of c("j", "F", "m", "A", "M", "J", "J", "A", "S", "O", "N", "D"), with 'month.abb' as the names.
- weeks a numeric vector of length 53 marking 52 periods of approximately 7 days each throughout the year = c(0, 365/52, ..., 365)

Details

Miscellaneous vectors often used in 'fda' scripts.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York, pp. 5, 47-53.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York

See Also

`axisIntervals` `month.abb`

Examples

```
daybasis65 <- create.fourier.basis(c(0, 365), 65)
daytempfd <- with(CanadianWeather, smooth.basisPar(day.5,
    dailyAv[,,"Temperature.C"], daybasis65)$fd )
plot(daytempfd, axes=FALSE)
axisIntervals(1)
# axisIntervals by default uses
# monthBegin.5, monthEnd.5, monthMid, and month.abb
axis(2)
```

---

density.fd *Compute a Probability Density Function*

Description

Like the regular S-PLUS function `density`, this function computes a probability density function for a sample of values of a random variable. However, in this case the density function is defined by a functional parameter object `WfdParobj` along with a normalizing constant `C`.

The density function $p(x)$ has the form `p(x) = C exp[W(x)]` where function $W(x)$ is defined by the functional data object `WfdParobj`.

Usage

```
## S3 method for class 'fd'
density(x, WfdParobj, conv=0.0001, iterlim=20,
           active=1:nbasis, dbglev=1, ...)
```

Arguments

`x`	a set observations, which may be one of two forms: 1. a vector of observations x_i 2. a two-column matrix, with the observations x_i in the first column, and frequencies f_i in the second. The first option corresponds to all $f_i = 1$.

WfdParobj	a functional parameter object specifying the initial value, basis object, roughness penalty and smoothing parameter defining function $W(t).$
conv	a positive constant defining the convergence criterion.
iterlim	the maximum number of iterations allowed.
active	a logical vector of length equal to the number of coefficients defining Wfdobj. If an entry is TRUE, the corresponding coefficient is estimated, and if FALSE, it is held at the value defining the argument Wfdobj. Normally the first coefficient is set to 0 and not estimated, since it is assumed that $W(0) = 0$.
dbglev	either 0, 1, or 2. This controls the amount information printed out on each iteration, with 0 implying no output, 1 intermediate output level, and 2 full output. If levels 1 and 2 are used, it is helpful to turn off the output buffering option in S-PLUS.
...	Other arguments to match the generic function 'density'

Details

The goal of the function is provide a smooth density function estimate that approaches some target density by an amount that is controlled by the linear differential operator Lfdobj and the penalty parameter. For example, if the second derivative of $W(t)$ is penalized heavily, this will force the function to approach a straight line, which in turn will force the density function itself to be nearly normal or Gaussian. Similarly, to each textbook density function there corresponds a $W(t)$, and to each of these in turn their corresponds a linear differential operator that will, when apply to $W(t)$, produce zero as a result. To plot the density function or to evaluate it, evaluate Wfdobj, exponentiate the resulting vector, and then divide by the normalizing constant C.

Value

a named list of length 4 containing:

Wfdobj	a functional data object defining function $W(x)$ that that optimizes the fit to the data of the monotone function that it defines.
C	the normalizing constant.
Flist	a named list containing three results for the final converged solution: (1) **f**: the optimal function value being minimized, (2) **grad**: the gradient vector at the optimal solution, and (3) **norm**: the norm of the gradient vector at the optimal solution.
iternum	the number of iterations.
iterhist	a iternum+1 by 5 matrix containing the iteration history.

See Also

intensity.fd

Examples

```
#  set up range for density
rangeval <- c(-3,3)
#  set up some standard normal data
x <- rnorm(50)
#  make sure values within the range
x[x < -3] <- -2.99
x[x >  3] <-  2.99
#  set up basis for W(x)
basisobj <- create.bspline.basis(rangeval, 11)
#  set up initial value for Wfdobj
Wfd0 <- fd(matrix(0,11,1), basisobj)
WfdParobj <- fdPar(Wfd0)
#  estimate density
denslist <- density.fd(x, WfdParobj)
#  plot density
xval <- seq(-3,3,.2)
wval <- eval.fd(xval, denslist$Wfdobj)
pval <- exp(wval)/denslist$C
plot(xval, pval, type="l", ylim=c(0,0.4))
points(x,rep(0,50))
```

---

deriv.fd *Compute a Derivative of a Functional Data Object*

Description

A derivative of a functional data object, or the result of applying a linear differential operator to a functional data object, is then converted to a functional data object. This is intended for situations where a derivative is to be manipulated as a functional data object rather than simply evaluated.

Usage

```
## S3 method for class 'fd'
deriv(expr, Lfdobj=int2Lfd(1), ...)
```

Arguments

expr	a functional data object. It is assumed that the basis for representing the object can support the order of derivative to be computed. For B-spline bases, this means that the order of the spline must be at least one larger than the order of the derivative to be computed.
Lfdobj	either a positive integer or a linear differential operator object.
...	Other arguments to match generic for 'deriv'

Details

Typically, a derivative has more high frequency variation or detail than the function itself. The basis defining the function is used, and therefore this must have enough basis functions to represent the variation in the derivative satisfactorily.

Value

a functional data object for the derivative

See Also

getbasismatrix, eval.basis deriv

Examples

```
#  Estimate the acceleration functions for growth curves
#  See the analyses of the growth data.
#  Set up the ages of height measurements for Berkeley data
age <- c( seq(1, 2, 0.25), seq(3, 8, 1), seq(8.5, 18, 0.5))
#  Range of observations
rng <- c(1,18)
#  Set up a B-spline basis of order 6 with knots at ages
knots  <- age
norder <- 6
nbasis <- length(knots) + norder - 2
hgtbasis <- create.bspline.basis(rng, nbasis, norder, knots)
#  Set up a functional parameter object for estimating
#  growth curves.  The 4th derivative is penalyzed to
#  ensure a smooth 2nd derivative or acceleration.
Lfdobj <- 4
lambda <- 10^(-0.5)   #  This value known in advance.
growfdPar <- fdPar(hgtbasis, Lfdobj, lambda)
#  Smooth the data.  The data for the boys and girls
#  are in matrices hgtm and hgtf, respectively.
hgtmfd <- smooth.basis(age, growth$hgtm, growfdPar)$fd
hgtffd <- smooth.basis(age, growth$hgtf, growfdPar)$fd
#  Compute the acceleration functions
accmfd <- deriv.fd(hgtmfd, 2)
accffd <- deriv.fd(hgtffd, 2)
#  Plot the two sets of curves
par(mfrow=c(2,1))
plot(accmfd)
plot(accffd)
```

`df.residual.fRegress` *Degrees of Freedom for Residuals from a Functional Regression*

Description

Effective degrees of freedom for residuals, being the trace of the idempotent `hat` matrix transforming observations into residuals from the fit.

Usage

```
## S3 method for class 'fRegress'
df.residual(object, ...)
```

Arguments

`object`	Object of class inheriting from `fRegress`
`...`	additional arguments for other methods

Details

1. Determine N = number of observations
2. df.model <- object\$df
3. df.residual <- (N - df.model)
4. Add attributes

Value

The numeric value of the residual degrees-of-freedom extracted from `object` with the following attributes:

`nobs`	number of observations
`df.model`	effective degrees of freedom for the model, being the trace of the idempotent linear projection operator transforming the observations into their predictions per the model. This includes the intercept, so the 'degrees of freedom for the model' for many standard purposes that compare with a model with an estimated mean will be 1 less than this number.

Author(s)

Spencer Graves

References

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York. Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome (2001) *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer, New York.

See Also

fRegress df.residual

---

df2lambda *Convert Degrees of Freedom to a Smoothing Parameter Value*

Description

The degree of roughness of an estimated function is controlled by a smoothing parameter $lambda$ that directly multiplies the penalty. However, it can be difficult to interpret or choose this value, and it is often easier to determine the roughness by choosing a value that is equivalent of the degrees of freedom used by the smoothing procedure. This function converts a degrees of freedom value into a multiplier $lambda$.

Usage

```
df2lambda(argvals, basisobj, wtvec=rep(1, n), Lfdobj=0,
          df=nbasis)
```

Arguments

argvals	a vector containing argument values associated with the values to be smoothed.
basisobj	a basis function object.
wtvec	a vector of weights for the data to be smoothed.
Lfdobj	either a nonnegative integer or a linear differential operator object.
df	the degrees of freedom to be converted.

Details

The conversion requires a one-dimensional optimization and may be therefore computationally intensive.

Value

a positive smoothing parameter value $lambda$

See Also

lambda2df, lambda2gcv

Examples

```
#  Smooth growth curves using a specified value of
#  degrees of freedom.
#  Set up the ages of height measurements for Berkeley data
age <- c( seq(1, 2, 0.25), seq(3, 8, 1), seq(8.5, 18, 0.5))
#  Range of observations
rng <- c(1,18)
#  Set up a B-spline basis of order 6 with knots at ages
knots  <- age
norder <- 6
nbasis <- length(knots) + norder - 2
hgtbasis <- create.bspline.basis(rng, nbasis, norder, knots)
#  Find the smoothing parameter equivalent to 12
#  degrees of freedom
lambda <- df2lambda(age, hgtbasis, df=12)
#  Set up a functional parameter object for estimating
#  growth curves.  The 4th derivative is penalyzed to
#  ensure a smooth 2nd derivative or acceleration.
Lfdobj <- 4
growfdPar <- fdPar(hgtbasis, Lfdobj, lambda)
#  Smooth the data.  The data for the girls are in matrix
#  hgtf.
hgtffd <- smooth.basis(age, growth$hgtf, growfdPar)$fd
#  Plot the curves
plot(hgtffd)
```

`dirs` *Get subdirectories*

Description

If you want only subfolders and no files, use `dirs`. With `recursive` = FALSE, `dir` returns both folders and files. With `recursive` = TRUE, it returns only files.

Usage

```
dirs(path='.', pattern=NULL, exclude=NULL, all.files=FALSE,
     full.names=FALSE, recursive=FALSE, ignore.case=FALSE)
```

Arguments

`path`, `all.files`, `full.names`, `recursive`, `ignore.case`
: as for `dir`

`pattern`, `exclude`
: optional regular expressions of filenames to include or exclude, respectively.

Details

1. mainDir <- dir(...) without recurse
2. Use `file.info` to restrict mainDir to only directories.
3. If !recursive, return the restricted mainDir. Else, if length(mainDir) > 0, create dirList to hold the results of the recursion and call `dirs` for each component of mainDir. Then `unlist` and return the result.

Value

A character vector of the desired subdirectories.

Author(s)

Spencer Graves

See Also

`dir`, `file.info`

`Eigen` *Eigenanalysis preserving dimnames*

Description

Compute eigenvalues and vectors, assigning names to the eigenvalues and dimnames to the eigenvectors.

Usage

```
Eigen(x, symmetric, only.values = FALSE, valuenames)
```

Arguments

`x`	a square matrix whose spectral decomposition is to be computed.
`symmetric`	logical: If TRUE, the matrix is assumed to be symmetric (or Hermitian if complex) and only its lower triangle (diagonal included) is used. If 'symmetric' is not specified, the matrix is inspected for symmetry.
`only.values`	if 'TRUE', only the eigenvalues are computed and returned, otherwise both eigenvalues and eigenvectors are returned.
`valuenames`	character vector of length nrow(x) or a character string that can be extended to that length by appending 1:nrow(x). The default depends on symmetric and whether `rownames` == `colnames`: If `rownames` == `colnames` and symmetric = TRUE (either specified or determined by inspection), the default is "paste('ev', 1:nrow(x), sep='')". Otherwise, the default is colnames(x) unless this is NULL.

Details

1. Check 'symmetric'

2. ev <- eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE); see `eigen` for more details.

3. rNames = rownames(x); if this is NULL, rNames = if(symmetric) paste('x', 1:nrow(x), sep='') else paste('xcol', 1:nrow(x)).

4. Parse 'valuenames', assign to names(ev[['values']]).

5. dimnames(ev[['vectors']]) <- list(rNames, valuenames)

NOTE: This naming convention is fairly obvious if 'x' is symmetric. Otherwise, dimensional analysis suggests problems with almost any naming convention. To see this, consider the following simple example:

$$X < -matrix(1:4, 2, dimnames = list(LETTERS[1:2], letters[3:4]))$$

	c	d
A	1	3
B	2	4

$$X.inv < -solve(X)$$

	A	B
c	-2	1.5
d	1	-0.5

One way of interpreting this is to assume that colnames are really reciprocals of the units. Thus, in this example, X[1,1] is in units of 'A/c' and X.inv[1,1] is in units of 'c/A'. This would make any matrix with the same row and column names potentially dimensionless. Since eigenvalues are essentially the diagonal of a diagonal matrix, this would mean that eigenvalues are dimensionless, and their names are merely placeholders.

Value

a list with components values and (if only.values = FALSE) vectors, as described in `eigen`.

Author(s)

Spencer Graves

See Also

`eigen`, `svd` `qr` `chol`

Examples

```
X <- matrix(1:4, 2, dimnames=list(LETTERS[1:2], letters[3:4]))
eigen(X)
Eigen(X)
Eigen(X, valuenames='eigval')

Y <- matrix(1:4, 2, dimnames=list(letters[5:6], letters[5:6]))
Eigen(Y)

Eigen(Y, symmetric=TRUE)
# only the lower triangle is used;
# the upper triangle is ignored.
```

---

eigen.pda *Stability Analysis for Principle Differential Analysis*

Description

Performs a stability analysis of the result of pda.fd, returning the real and imaginary parts of the eigenfunctions associated with the linear differential operator.

Usage

```
eigen.pda(pdaList,plotresult=TRUE,npts=501,...)
```

Arguments

pdaList	a list object returned by pda.fd.
plotresult	should the result be plotted? Default is TRUE
npts	number of points to use for plotting.
...	other arguments for 'plot'.

Details

Conducts an eigen decomposition of the linear differential equation implied by the result of pda.fd. Imaginary eigenvalues indicate instantaneous oscillatory behavior. Positive real eigenvalues indicate exponential increase, negative real eigenvalues correspond to exponential decay. If the principle differential analysis also included the estimation of a forcing function, the limiting stable points are also tracked.

Value

Returns a list with elements

argvals	The evaluation points of the coefficient functions.
eigvals	The corresponding eigenvalues at each time.
limvals	The stable points of the system at each time.

See Also

pda.fd plot.pda.fd pda.overlay

Examples

```
#  A pda analysis of the handwriting data

# reduce the size to reduce the compute time for the example
ni <- 281
indx <- seq(1, 1401, length=ni)
fdaarray = handwrit[indx,,]
fdatime  <- seq(0, 2.3, len=ni)

#  basis for coordinates

fdarange <- c(0, 2.3)
breaks = seq(0,2.3,length.out=116)
norder = 6
fdabasis = create.bspline.basis(fdarange,norder=norder,breaks=breaks)

#  parameter object for coordinates

fdaPar = fdPar(fdabasis,int2Lfd(4),1e-8)

#  coordinate functions and a list tontaining them

Xfd = smooth.basis(fdatime, fdaarray[,,1], fdaPar)$fd
Yfd = smooth.basis(fdatime, fdaarray[,,2], fdaPar)$fd

xfdlist = list(Xfd, Yfd)

#  basis and parameter object for weight functions

fdabasis2 = create.bspline.basis(fdarange,norder=norder,nbasis=31)
fdafd2    = fd(matrix(0,31,2),fdabasis2)
pdaPar    = fdPar(fdafd2,1,1e-8)

pdaParlist = list(pdaPar, pdaPar)

bwtlist = list( list(pdaParlist,pdaParlist), list(pdaParlist,pdaParlist) )
```

`ElectricDemand` *Predicting electricity demand in Adelaide from temperature*

Description

The data sets used in this demonstration analysis consist of half-hourly electricity demands from Sunday to Saturday in Adelaide, Australia, between July 6, 1976 and March 31, 2007. Also provided in the same format and times are the temperatures in degrees Celsius at the Adelaide airport. The shapes of the demand curves for each day resemble those for temperature, and the goal is to see how well a concurrent functional regress model fit by function `fRegress` can fit the demand curves.

Format

There is a data object for each day of the week, and in each object, the member y, such as `mondaydemand$y`, is a 48 by 508 matrix of electricity demand values, one for each of 48 half-hourly points, and for each of 508 weeks.

Details

The demonstration is designed to show how to use the main functional regression function `fRegress` in order to fit a functional dependent variable (electricity demand) from an obviously important input or covariate functional variable (temperature). In the texts cited, this model is referred to as a concurrent functional regression because the model assumes that changes in the input functional variable directly and immediately change the dependent functional variable.

Also illustrated is the estimation of pointwise confidence intervals for the regression coefficient functions and for the fitting functions that approximate electricity demand. These steps involve functons `fRegress.stderr` and `predict.fRegress`.

The data are supplied through the CRAN system by package `fds`, and further information as well as many other datasets are to be found there.

Source

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

Ramsay, James O., Hooker, Giles and Graves, Spencer (2009) Functional Data Analysis with R and Matlab, Springer, New York.

`eval.basis` *Values of Basis Functions or their Derivatives*

Description

A set of basis functions are evaluated at a vector of argument values. If a linear differential object is provided, the values are the result of applying the the operator to each basis function.

Usage

```
eval.basis(evalarg, basisobj, Lfdobj=0, returnMatrix=FALSE)
## S3 method for class 'basisfd'
predict(object, newdata=NULL, Lfdobj=0,
                          returnMatrix=FALSE, ...)
```

Arguments

`evalarg, newdata`	a vector of argument values at which the basis functiona is to be evaluated.
`basisobj`	a basis object defining basis functions whose values are to be computed.
`Lfdobj`	either a nonnegative integer or a linear differential. operator object.
`object`	an object of class `basisfd`
`...`	optional arguments for `predict`, not currently used
`returnMatrix`	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Details

If a linear differential operator object is supplied, the basis must be such that the highest order derivative can be computed. If a B-spline basis is used, for example, its order must be one larger than the highest order of derivative required.

Value

a matrix of basis function values with rows corresponding to argument values and columns to basis functions.

`predict.basisfd` is a convenience wrapper for `eval.basis`.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York

See Also

getbasismatrix, eval.fd, plot.basisfd

Examples

```
##
## 1.  B-splines
##
# The simplest basis currently available:
# a single step function
bspl1.1 <- create.bspline.basis(norder=1, breaks=0:1)
eval.bspl1.1 <- eval.basis(seq(0, 1, .2), bspl1.1)

# check
eval.bspl1.1. <- matrix(rep(1, 6), 6,
       dimnames=list(NULL, 'bspl') )

all.equal(eval.bspl1.1, eval.bspl1.1.)


# The second simplest basis:
# 2 step functions, [0, .5], [.5, 1]
bspl1.2 <- create.bspline.basis(norder=1, breaks=c(0,.5, 1))
eval.bspl1.2 <- eval.basis(seq(0, 1, .2), bspl1.2)

# Second order B-splines (degree 1:  linear splines)
bspl2.3 <- create.bspline.basis(norder=2, breaks=c(0,.5, 1))
eval.bspl2.3 <- eval.basis(seq(0, 1, .1), bspl2.3)
# 3 bases:  order 2 = degree 1 = linear
# (1) line from (0,1) down to (0.5, 0), 0 after
# (2) line from (0,0) up to (0.5, 1), then down to (1,0)
# (3) 0 to (0.5, 0) then up to (1,1).

##
## 2.  Fourier
##
# The false Fourier series with 1 basis function
falseFourierBasis <- create.fourier.basis(nbasis=1)
eval.fFB <- eval.basis(seq(0, 1, .2), falseFourierBasis)

# Simplest real Fourier basis with 3 basis functions
fourier3 <- create.fourier.basis()
eval.fourier3 <- eval.basis(seq(0, 1, .2), fourier3)

# 3 basis functions on [0, 365]
fourier3.365 <- create.fourier.basis(c(0, 365))
eval.F3.365 <- eval.basis(day.5, fourier3.365)

matplot(eval.F3.365, type="l")

# The next simplest Fourier basis (5  basis functions)
fourier5 <- create.fourier.basis(nbasis=5)
```

```
eval.F5 <- eval.basis(seq(0, 1, .1), fourier5)
matplot(eval.F5, type="l")

# A more complicated example
dayrng <- c(0, 365)

nbasis <- 51
norder <- 6

weatherBasis <- create.fourier.basis(dayrng, nbasis)
basisMat <- eval.basis(day.5, weatherBasis)

matplot(basisMat[, 1:5], type="l")

##
## 3.  predict.basisfd
##
basisMat. <- predict(weatherBasis, day.5)

all.equal(basisMat, basisMat.)


##
## 4.  Date and POSIXct
##
# Date
July4.1776 <- as.Date('1776-07-04')
Apr30.1789 <- as.Date('1789-04-30')
AmRev <- c(July4.1776, Apr30.1789)
BspRevolution <- create.bspline.basis(AmRev)
AmRevYears <- seq(July4.1776, Apr30.1789, length.out=14)
AmRevBases <- predict(BspRevolution, AmRevYears)
matplot(AmRevYears, AmRevBases, type='b')
# Image is correct, but
# matplot does not recogize the Date class of x

# POSIXct
AmRev.ct <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev.ct <- create.bspline.basis(AmRev.ct)
AmRevYrs.ct <- seq(AmRev.ct[1], AmRev.ct[2], length.out=14)
AmRevBas.ct <- predict(BspRev.ct, AmRevYrs.ct)
matplot(AmRevYrs.ct, AmRevBas.ct, type='b')
# Image is correct, but
# matplot does not recognize the POSIXct class of x
```

`eval.bifd` *Values a Two-argument Functional Data Object*

Description

A vector of argument values for the first argument `s` of the functional data object to be evaluated.

Usage

```
eval.bifd(sevalarg, tevalarg, bifd, sLfdobj=0, tLfdobj=0)
```

Arguments

sevalarg	a vector of argument values for the first argument s of the functional data object to be evaluated.
tevalarg	a vector of argument values for the second argument t of the functional data object to be evaluated.
bifd	a two-argument functional data object.
sLfdobj	either a nonnegative integer or a linear differential operator object. If present, the derivative or the value of applying the operator to the object as a function of the first argument s is evaluated rather than the functions themselves.
tLfdobj	either a nonnegative integer or a linear differential operator object. If present, the derivative or the value of applying the operator to the object as a function of the second argument t is evaluated rather than the functions themselves.

Value

an array of 2, 3, or 4 dimensions containing the function values. The first dimension corresponds to the argument values in sevalarg, the second to argument values in tevalarg, the third if present to replications, and the fourth if present to functions.

Examples

```
# every-other-day basis to save test time
daybasis   <- create.fourier.basis(c(0,365), 183)
harmLcoef  <- c(0,(2*pi/365)^2,0)
harmLfd    <- vec2Lfd(harmLcoef, c(0,365))
templambda <- 1.0
tempfdPar  <- fdPar(daybasis, harmLfd, lambda=1)
tempfd     <- smooth.basis(day.5,
          CanadianWeather$dailyAv[,,"Temperature.C"], tempfdPar)$fd
#    define the variance-covariance bivariate fd object
tempvarbifd <- var.fd(tempfd)
#    evaluate the variance-covariance surface and plot
weektime    <- seq(0,365,len=53)
tempvarmat  <- eval.bifd(weektime,weektime,tempvarbifd)
#    make a perspective plot of the variance function
persp(tempvarmat)
```

---

eval.fd *Values of a Functional Data Object*

Description

Evaluate a functional data object at specified argument values, or evaluate a derivative or the result of applying a linear differential operator to the functional object.

Usage

```
eval.fd(evalarg, fdobj, Lfdobj=0, returnMatrix=FALSE)
## S3 method for class 'fd'
predict(object, newdata=NULL, Lfdobj=0, returnMatrix=FALSE,
                    ...)
## S3 method for class 'fdPar'
predict(object, newdata=NULL, Lfdobj=0,
                    returnMatrix=FALSE, ...)
## S3 method for class 'fdSmooth'
predict(object, newdata=NULL, Lfdobj=0,
                    returnMatrix=FALSE, ...)
## S3 method for class 'fdSmooth'
fitted(object, returnMatrix=FALSE, ...)
## S3 method for class 'fdSmooth'
residuals(object, returnMatrix=FALSE, ...)
```

Arguments

evalarg, newdata	a vector or matrix of argument values at which the functional data object is to be evaluated. If a matrix with more than one column, the number of columns must match ncol(dfobj[['coefs']]).
fdobj	a functional data object to be evaluated.
Lfdobj	either a nonnegative integer or a linear differential operator object. If present, the derivative or the value of applying the operator is evaluated rather than the functions themselves.
object	an object of class fd
returnMatrix	logical: Should a 2-dimensional array to be returned using a special class from the Matrix package if appropriate?
...	optional arguments for predict, not currently used

Details

eval.fd evaluates Lfdobj of fdobj at evalarg.

predict.fd is a convenience wrapper for eval.fd. If newdata is NULL and fdobj[['basis']][['type']] is bspline, newdata = unique(knots(fdojb,interior=FALSE)); otherwise, newdata = fdobj[['basis']][['rangeval

predict.fdSmooth, fitted.fdSmooth and residuals.fdSmooth are other wrappers for eval.fd.

Value

an array of 2 or 3 dimensions containing the function values. The first dimension corresponds to the argument values in evalarg, the second to replications, and the third if present to functions.

Author(s)

Soren Hosgaard wrote an initial version of predict.fdSmooth, fitted.fdSmooth, and residuals.fdSmooth.

See Also

getbasismatrix, eval.bifd, eval.penalty, eval.monfd, eval.posfd

Examples

```
##
## eval.fd
##
#    set up the fourier basis
daybasis <- create.fourier.basis(c(0, 365), nbasis=65)
#  Make temperature fd object
#  Temperature data are in 12 by 365 matrix tempav
#  See analyses of weather data.
#  Set up sampling points at mid days
#  Convert the data to a functional data object
tempfd <- smooth.basis(day.5,  CanadianWeather$dailyAv[,,"Temperature.C"],
                       daybasis)$fd
#   set up the harmonic acceleration operator
Lbasis  <- create.constant.basis(c(0, 365))
Lcoef   <- matrix(c(0,(2*pi/365)^2,0),1,3)
bfdobj  <- fd(Lcoef,Lbasis)
bwtlist <- fd2list(bfdobj)
harmaccelLfd <- Lfd(3, bwtlist)
#   evaluate the value of the harmonic acceleration
#   operator at the sampling points
Ltempmat <- eval.fd(day.5, tempfd, harmaccelLfd)

#  Confirm that it still works with
#  evalarg = a matrix with only one column
#  when fdobj[['coefs']] is a matrix with multiple columns

Ltempmat. <- eval.fd(matrix(day.5, ncol=1), tempfd, harmaccelLfd)
#  confirm that the two answers are the same


all.equal(Ltempmat, Ltempmat.)


#  Plot the values of this operator
matplot(day.5, Ltempmat, type="l")

##
## predict.fd
##
predict(tempfd) # end points only at 35 locations
str(predict(tempfd, day.5)) # 365 x 35 matrix
str(predict(tempfd, day.5, harmaccelLfd))

# cublic splie with knots at 0, .5, 1
bspl3 <- create.bspline.basis(c(0, .5, 1))
plot(bspl3) # 5 bases
fd.bspl3 <- fd(c(0, 0, 1, 0, 0), bspl3)
```

```
pred3 <- predict(fd.bspl3)

pred3. <- matrix(c(0, .5, 0), 3)
dimnames(pred3.) <- list(NULL, 'reps 1')

all.equal(pred3, pred3.)


pred.2 <- predict(fd.bspl3, c(.2, .8))

pred.2. <- matrix(.176, 2, 1)
dimnames(pred.2.) <- list(NULL, 'reps 1')

all.equal(pred.2, pred.2.)


##
## predict.fdSmooth
##
lipSm9 <- smooth.basisPar(liptime, lip, lambda=1e-9)$fd
plot(lipSm9)

##
## with evalarg of class Date and POSIXct
##
# Date
July4.1776 <- as.Date('1776-07-04')
Apr30.1789 <- as.Date('1789-04-30')
AmRev <- c(July4.1776, Apr30.1789)
BspRevolution <- create.bspline.basis(AmRev)

AmRevYears <- seq(July4.1776, Apr30.1789, length.out=14)
(AmRevLinear <- as.numeric(AmRevYears-July4.1776))
fitLin <- smooth.basis(AmRevYears, AmRevLinear, BspRevolution)
AmPred <- predict(fitLin, AmRevYears)

# POSIXct
AmRev.ct <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev.ct <- create.bspline.basis(AmRev.ct)
AmRevYrs.ct <- seq(AmRev.ct[1], AmRev.ct[2], length.out=14)
(AmRevLin.ct <- as.numeric(AmRevYrs.ct-AmRev.ct[2]))
fitLin.ct <- smooth.basis(AmRevYrs.ct, AmRevLin.ct, BspRev.ct)
AmPred.ct <- predict(fitLin.ct, AmRevYrs.ct)
```

---

eval.monfd *Values of a Monotone Functional Data Object*

Description

Evaluate a monotone functional data object at specified argument values, or evaluate a derivative of the functional object.

Usage

```
eval.monfd(evalarg, Wfdobj, Lfdobj=int2Lfd(0), returnMatrix=FALSE)
## S3 method for class 'monfd'
predict(object, newdata=NULL, Lfdobj=0, returnMatrix=FALSE, ...)
## S3 method for class 'monfd'
fitted(object, ...)
## S3 method for class 'monfd'
residuals(object, ...)
```

Arguments

`evalarg, newdata`	a vector of argument values at which the functional data object is to be evaluated.
`Wfdobj`	an object of class `fd` that defines the monotone function to be evaluated. Only univariate functions are permitted.
`Lfdobj`	a nonnegative integer specifying a derivative to be evaluated. At this time of writing, permissible derivative values are 0, 1, 2, or 3. A linear differential operator is not allowed.
`object`	an object of class `monfd` that defines the monotone function to be evaluated. Only univariate functions are permitted.
`returnMatrix`	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.
`...`	optional arguments required by `predict`; not currently used.

Details

A monotone function data object $h(t)$ is defined by $h(t) = [D^{-1} exp Wfdobj](t)$. In this equation, the operator D^{-1} means taking the indefinite integral of the function to which it applies. Note that this equation implies that the monotone function has a value of zero at the lower limit of the arguments. To actually fit monotone data, it will usually be necessary to estimate an intercept and a regression coefficient to be applied to $h(t)$, usually with the least squares regression function `lsfit`. The function `Wfdobj` that defines the monotone function is usually estimated by monotone smoothing function `smooth.monotone`.

`eval.monfd` only computes the standardized monotone form. `predict.monfd` computes the scaled version using `with(object,beta[1] + beta[2]*eval.monfd(...))` if Lfdobj = 0 or beta[2]*eval.monfd(...) if Lfdobj > 0.

Value

a matrix containing the monotone function values. The first dimension corresponds to the argument values in `evalarg` and the second to replications.

See Also

eval.fd, smooth.monotone eval.posfd

Examples

```
#  Estimate the acceleration functions for growth curves
#  See the analyses of the growth data.
#  Set up the ages of height measurements for Berkeley data
age <- c( seq(1, 2, 0.25), seq(3, 8, 1), seq(8.5, 18, 0.5))
#  Range of observations
rng <- c(1,18)
#  First set up a basis for monotone smooth
#  We use b-spline basis functions of order 6
#  Knots are positioned at the ages of observation.
norder <- 6
nage   <- length(age)
nbasis <- nage + norder - 2
wbasis <- create.bspline.basis(rng, nbasis, norder, age)
#  starting values for coefficient
cvec0 <- matrix(0,nbasis,1)
Wfd0  <- fd(cvec0, wbasis)
#  set up functional parameter object
Lfdobj    <- 3          #  penalize curvature of acceleration
lambda    <- 10^(-0.5)  #  smoothing parameter
growfdPar <- fdPar(Wfd0, Lfdobj, lambda)
#  Smooth the data for the first girl
hgt1 <- growth$hgtf[,1]
#   set conv = 0.1 and iterlim=1 to reduce the compute time
#   required for this test on CRAN;
#   We would not do this normally.
result <- smooth.monotone(age, hgt1, growfdPar, conv=0.1,
                          iterlim=1)
#  Extract the functional data object and regression
#  coefficients
Wfd  <- result$Wfdobj
beta <- result$beta
#  Evaluate the fitted height curve over a fine mesh
agefine <- seq(1,18,len=60)
hgtfine <- beta[1] + beta[2]*eval.monfd(agefine, Wfd)
#  Plot the data and the curve
plot(age, hgt1, type="p")
lines(agefine, hgtfine)
#  Evaluate the acceleration curve
accfine <- beta[2]*eval.monfd(agefine, Wfd, 2)
#  Plot the acceleration curve
plot(agefine, accfine, type="l")
lines(c(1,18),c(0,0),lty=4)

##
## using predict.monfd
##
```

```
hgtfit <- with(result, beta[1]+beta[2]*eval.monfd(argvals, Wfdobj))
hgtfit. <- fitted(result)

all.equal(hgtfit, hgtfit.)


accfine. <- predict(result, agefine, Lfdobj=2)

all.equal(accfine, accfine.)


growthResid <- resid(result)

all.equal(growthResid, with(result, y-hgtfit.))
```

`eval.penalty` *Evaluate a Basis Penalty Matrix*

Description

A basis roughness penalty matrix is the matrix containing the possible inner products of pairs of basis functions. These inner products are typically defined in terms of the value of a derivative or of a linear differential operator applied to the basis function. The basis penalty matrix plays an important role in the computation of functions whose roughness is controlled by a roughness penalty.

Usage

```
eval.penalty(basisobj, Lfdobj=int2Lfd(0), rng=rangeval)
```

Arguments

`basisobj`	a basis object.
`Lfdobj`	either a nonnegative integer defining an order of a derivative or a linear differential operator.
`rng`	a vector of length 2 defining a restricted range. Optionally, the inner products can be computed over a range of argument values that lies within the interval covered by the basis function definition.

Details

The inner product can be computed exactly for many types of bases if m is an integer. These include B-spline, fourier, exponential, monomial, polynomial and power bases. In other cases, and for noninteger operators, the inner products are computed by an iterative numerical integration method called Richard extrapolation using the trapezoidal rule.

If the penalty matrix must be evaluated repeatedly, computation can be greatly speeded up by avoiding the use of this function, and instead using quadrature points and weights defined by Simpson's rule.

Value

a square symmetric matrix whose order is equal to the number of basis functions defined by the basis function object `basisobj` . If `Lfdobj` is m or a linear differential operator of order m, the rank of the matrix should be at least approximately equal to its order minus m.

See Also

`getbasispenalty`, `eval.basis`,

`eval.posfd` *Evaluate a Positive Functional Data Object*

Description

Evaluate a positive functional data object at specified argument values, or evaluate a derivative of the functional object.

Usage

```
eval.posfd(evalarg, Wfdobj, Lfdobj=int2Lfd(0))
## S3 method for class 'posfd'
predict(object, newdata=NULL, Lfdobj=0, ...)
## S3 method for class 'posfd'
fitted(object, ...)
## S3 method for class 'posfd'
residuals(object, ...)
```

Arguments

`evalarg, newdata`
: a vector of argument values at which the functional data object is to be evaluated.

`Wfdobj`
: a functional data object that defines the positive function to be evaluated. Only univariate functions are permitted.

`Lfdobj`
: a nonnegative integer specifying a derivative to be evaluated. At this time of writing, permissible derivative values are 0, 1 or 2. A linear differential operator is not allowed.

`object`
: an object of class `posfd` that defines the positive function to be evaluated. Only univariate functions are permitted.

`...`
: optional arguments required by `predict`; not currently used.

Details

A positive function data object $h(t)$ is defined by $h(t) =[exp Wfd](t)$. The function `Wfdobj` that defines the positive function is usually estimated by positive smoothing function `smooth.pos`

Value

a matrix containing the positive function values. The first dimension corresponds to the argument values in `evalarg` and the second to replications.

See Also

`eval.fd`, `eval.monfd`

Examples

```
harmaccelLfd <- vec2Lfd(c(0,(2*pi/365)^2,0), c(0, 365))
smallbasis   <- create.fourier.basis(c(0, 365), 65)
index        <- (1:35)[CanadianWeather$place == "Vancouver"]
VanPrec      <- CanadianWeather$dailyAv[,index, "Precipitation.mm"]
lambda       <- 1e4
dayfdPar     <- fdPar(smallbasis, harmaccelLfd, lambda)
VanPrecPos   <- smooth.pos(day.5, VanPrec, dayfdPar)
#  compute fitted values using eval.posfd()
VanPrecPosFit1 <- eval.posfd(day.5, VanPrecPos$Wfdobj)
#  compute fitted values using predict()
VanPrecPosFit2 <- predict(VanPrecPos, day.5)

all.equal(VanPrecPosFit1, VanPrecPosFit2)

#  compute fitted values using fitted()
VanPrecPosFit3 <- fitted(VanPrecPos)
#  compute residuals
VanPrecRes <- resid(VanPrecPos)

all.equal(VanPrecRes, VanPrecPos$y-VanPrecPosFit3)
```

---

evaldiag.bifd *Evaluate the Diagonal of a Bivariate Functional Data Object*

Description

Bivariate function data objects are functions of two arguments, $f(s,t)$. It can be useful to evaluate the function for argument values satisfying $s=t$, such as evaluating the univariate variance function given the bivariate function that defines the variance-covariance function or surface. A linear differential operator can be applied to function $f(s,t)$ considered as a univariate function of either object holding the other object fixed.

Usage

```
evaldiag.bifd(evalarg, bifdobj, sLfd=int2Lfd(0), tLfd=int2Lfd(0))
```

Arguments

evalarg	a vector of values of $s = t$.
bifdobj	a bivariate functional data object of the bifd class.
sLfd	either a nonnegative integer or a linear differential operator object.
tLfd	either a nonnegative integer or a linear differential operator object.

Value

a vector or matrix of diagonal function values.

See Also

var.fd, eval.bifd

---

expon *Exponential Basis Function Values*

Description

Evaluates a set of exponential basis functions, or a derivative of these functions, at a set of arguments.

Usage

```
expon(x, ratevec=1, nderiv=0)
```

Arguments

x	a vector of values at which the basis functions are to be evaluated.
ratevec	a vector of rate or time constants defining the exponential functions. That is, if a is the value of an element of this vector, then the corresponding basis function is $exp(at)$. The number of basis functions is equal to the length of ratevec.
nderiv	a nonnegative integer specifying an order of derivative to be computed. The default is 0, or the basis function value.

Details

There are no restrictions on the rate constants.

Value

a matrix of basis function values with rows corresponding to argument values and columns to basis functions.

See Also

exponpen

`exponentiate.fd` *Powers of a functional data ('fd') object*

Description

Exponentiate a functional data object where feasible.

Usage

```
## S3 method for class 'fd'
e1 ^ e2
exponentiate.fd(e1, e2, tolint=.Machine$double.eps^0.75,
  basisobj=e1$basis,
  tolfd=sqrt(.Machine$double.eps)*
          sqrt(sum(e1$coefs^2)+.Machine$double.eps)^abs(e2),
  maxbasis=NULL, npoints=NULL)
```

Arguments

`e1`	object of class 'fd'.
`e2`	a numeric vector of length 1.
`basisobj`	reference basis
`tolint`	if abs(e2-round(e2))<tolint, we assume e2 is an integer. This simplifies the algorithm.
`tolfd`	the maximum error allowed in the difference between the direct computation `eval.fd(e1)^e2` and the computed representation.
`maxbasis`	The maximum number of basis functions in growing `referencebasis` to achieve a fit within `tolfd`. Default = 2*nbasis12+1 where nbasis12 = nbasis of e1^floor(e2).
`npoints`	The number of points at which to compute `eval.fd(e1)^e2` and the computed representation to evaluate the adequacy of the representation. Default = 2*maxbasis-1. For a max Fourier basis, this samples the highest frequency at all its extrema and zeros.

Details

If e1 has a B-spline basis, this uses the B-spline algorithm.

Otherwise it throws an error unless it finds one of the following special cases:

- e2 = 0 Return an `fd` object with a constant basis that is everywhere 1
- e2 is a positive integer to within tolint Multiply e1 by itself e2 times
- e2 is positive and e1 has a Fourier basis e120 <- e1^floor(e2)
  outBasis <- e120$basis
  rng <- outBasis$rangeval
  Time <- seq(rng[1], rng[2], npoints)
  e1.2 <- predict(e1, Time)^e2
  fd1.2 <- smooth.basis(Time, e1.2, outBasis)$
  d1.2 <- (e1.2 - predict(fd1.2, Time))
  if(all(abs(d1.2)<tolfd))return(fd1.2)
  Else if(outBasis$nbasis<maxbasis) increase the size of outBasis and try again.
  Else write a warning with the max(abs(d1.2)) and return fd1.2.

Value

A function data object approximating the desired power.

See Also

`arithmetic.fd` `basisfd`, `basisfd.product`

Examples

```
##
## sin^2
##

basis3 <- create.fourier.basis(nbasis=3)
plot(basis3)
# max = sqrt(2), so
# integral of the square of each basis function (from 0 to 1) is 1
integrate(function(x)sin(2*pi*x)^2, 0, 1) # = 0.5

# sin(theta)
fdsin <- fd(c(0,sqrt(0.5),0), basis3)
plot(fdsin)

fdsin2 <- fdsin^2

# check
fdsinsin <- fdsin*fdsin
# sin^2(pi*time) = 0.5*(1-cos(2*pi*theta) basic trig identity
plot(fdsinsin) # good
```

```
all.equal(fdsin2, fdsinsin)
```

---

exponpen *Exponential Penalty Matrix*

Description

Computes the matrix defining the roughness penalty for functions expressed in terms of an exponential basis.

Usage

```
exponpen(basisobj, Lfdobj=int2Lfd(2))
```

Arguments

basisobj	an exponential basis object.
Lfdobj	either a nonnegative integer or a linear differential operator object.

Details

A roughness penalty for a function $x(t)$ is defined by integrating the square of either the derivative of $ x(t) $ or, more generally, the result of applying a linear differential operator L to it. The most common roughness penalty is the integral of the square of the second derivative, and this is the default. To apply this roughness penalty, the matrix of inner products of the basis functions (possibly after applying the linear differential operator to them) defining this function is necessary. This function just calls the roughness penalty evaluation function specific to the basis involved.

Value

a symmetric matrix of order equal to the number of basis functions defined by the exponential basis object. Each element is the inner product of two exponential basis functions after applying the derivative or linear differential operator defined by Lfdobj.

See Also

expon, eval.penalty, getbasispenalty

Examples

```
#  set up an exponential basis with 3 basis functions
ratevec  <- c(0, -1, -5)
basisobj <- create.exponential.basis(c(0,1),3,ratevec)
#  compute the 3 by 3 matrix of inner products of
#  second derivatives
penmat <- exponpen(basisobj)
```

---

fbplot *Functional Boxplots*

Description

Produces functional boxplots or enhanced functional boxplots of the given functional data. It can also be used to carry out functional data ordering based on band depth.

Usage

```
fbplot(fit, x = NULL, method = "MBD", depth = NULL, plot = TRUE,
 prob = 0.5, color = 6, outliercol = 2, barcol = 4,
 fullout=FALSE, factor=1.5,xlim=c(1,nrow(fit)),
 ylim=c(min(fit)-.5*diff(range(fit)),max(fit)+.5*diff(range(fit))),...)
## S3 method for class 'fd'
boxplot(x, z=NULL, ...)
## S3 method for class 'fdPar'
boxplot(x, z=NULL, ...)
## S3 method for class 'fdSmooth'
boxplot(x, z=NULL, ...)
```

Arguments

fit	a p-by-n functional data matrix where n is the number of curves, and p is defined below.
x	For fbplot, x is the x coordinates of curves. Defaults to 1:p where p is the number of x coordinates. For boxplot.fd, boxplot.fdPar and boxplot.fdSmooth, x is an object of class fd, fdPar or fdSmooth, respectively.
z	The coordinate of the curves, labeled x for fdplot. For boxplot.fd, boxplot.fdPar and boxplot.fdSmooth, this cannot be x, because that would clash with the generic boxplot(x,...) standard.
method	the method to be used to compute band depth. Can be one of "BD2", "MBD" or "Both" with a default of "MBD". See also details.
depth	a vector giving band depths of curves. If missing, band depth computation is conducted.
plot	logical. If TRUE (the default) then a functional boxplot is produced. If not, band depth and outliers are returned.
prob	a vector giving the probabilities of central regions in a decreasing order, then an enhanced functional boxplot is produced. Defaults to be 0.5 and a functional boxplot is plotted.
color	a vector giving the colors of central regions from light to dark for an enhanced functional boxplot. Defaults to be magenta for a functional boxplot.
outliercol	color of outlying curves. Defaults to be red.
barcol	color of bars in a functional boxplot. Defaults to be blue.
fullout	logical for plotting outlying curves. If FALSE (the default) then only the part outside the box is plotted. If TRUE, complete outlying curves are plotted.
factor	the constant factor to inflate the middle box and determine fences for outliers. Defaults to be 1.5 as in a classical boxplot.
xlim	x-axis limits
ylim	y-axis limits
...	For fbplot, optional arguments for plot. For boxplot.fd, boxplot.fdPar, or boxplot.fdSmooth, optional arguments for fbplot.

Details

For functional data, the band depth (BD) or modified band depth (MBD) allows for ordering a sample of curves from the center outwards and, thus, introduces a measure to define functional quantiles and the centrality or outlyingness of an observation. A smaller rank is associated with a more central position with respect to the sample curves. BD usually provides many ties (curves have the same depth values), but MBD does not. "BD2" uses two curves to determine a band. The method "Both" uses "BD2" first and then uses "MBD" to break ties. The method "Both" uses BD2 first and then uses MBD to break ties. The computation is carried out by the fast algorithm proposed by Sun et. al. (2012).

Value

`depth`	band depths of given curves.
`outpoint`	column indices of detected outliers.

Author(s)

Ying Sun <sunwards@stat.osu.edu>

Marc G. Genton <marc.genton@kaust.edu.sa>

References

Sun, Y., Genton, M. G. and Nychka, D. (2012), "Exact fast computation of band depth for large functional datasets: How quickly can one million curves be ranked?" Stat, 1, 68-74.

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

Lopez-Pintado, S. and Romo, J. (2009), "On the concept of depth for functional data," Journal of the American Statistical Association, 104, 718-734.

Examples

```
##
## 1.  generate 50 random curves with some covariance structure
##     model 1 without outliers
##
cov.fun=function(d,k,c,mu){
        k*exp(-c*d^mu)
}
n=50
p=30
t=seq(0,1,len=p)
d=dist(t,upper=TRUE,diag=TRUE)
d.matrix=as.matrix(d)
#covariance function in time
t.cov=cov.fun(d.matrix,1,1,1)
# Cholesky Decomposition
L=chol(t.cov)
mu=4*t
e=matrix(rnorm(n*p),p,n)
```

```
ydata  = mu+t(L)%*%e

#functional boxplot
fbplot(ydata,method='MBD',ylim=c(-11,15))

# The same using boxplot.fd
boxplot.fd(ydata, method='MBD', ylim=c(-11, 15))

# same with default ylim
boxplot.fd(ydata)

##
## 2.  as an fd object
##
T      = dim(ydata)[1]
time   = seq(0,T,len=T)
ybasis = create.bspline.basis(c(0,T), 23)
Yfd    = smooth.basis(time, ydata, ybasis)$fd
boxplot(Yfd)

##
## 3.  as an fdPar object
##
Ypar <- fdPar(Yfd)
boxplot(Ypar)

##
## 4.  Smoothed version
##
Ysmooth <- smooth.fdPar(Yfd)
boxplot(Ysmooth)

##
## 5.  model 2 with outliers
##
#magnitude
k=6
#randomly introduce outliers
C=rbinom(n,1,0.1)
s=2*rbinom(n,1,0.5)-1
cs.m=matrix(C*s,p,n,byrow=TRUE)

e=matrix(rnorm(n*p),p,n)
y=mu+t(L)%*%e+k*cs.m

#functional boxplot
fbplot(y,method='MBD',ylim=c(-11,15))
```

---

fd2list *Convert a univariate functional data object to a list*

Description

Convert a univariate functional data object to a list for input to Lfd.

Usage

```
fd2list(fdobj)
```

Arguments

fdobj	a univariate functional data object.

Value

a list as required for the second argument of Lfd.

See Also

Lfd

Examples

```
Lbasis  = create.constant.basis(c(0,365));  #  create a constant basis
Lcoef   = matrix(c(0,(2*pi/365)^2,0),1,3)   #  set up three coefficients
wfdobj  = fd(Lcoef,Lbasis)      # define an FD object for weight functions
wfdlist = fd2list(wfdobj)       # convert the FD object to a cell object
harmaccelLfd = Lfd(3, wfdlist)  #  define the operator object
```

---

fdlabels *Extract plot labels and names for replicates and variables*

Description

Extract plot labels and, if available, names for each replicate and variable

Usage

```
fdlabels(fdnames, nrep, nvar)
```

Arguments

fdnames	a list of length 3 with xlabel, casenames, and ylabels.
nrep	integer number of cases or observations
nvar	integer number of variables

Details

xlabel <- if(length(fdnames[[1]])>1) names(fdnames)[1] else fdnames[[1]]

ylabel <- if(length(fdnames[[3]])>1) names(fdnames)[3] else fdnames[[3]]

casenames <- if(length(fdnames[[2]])== nrep)fdnames[[2]] else NULL

varnames <- if(length(fdnames[[3]])==nvar)fdnames[[3]] else NULL

Value

A list of xlabel, ylabel, casenames, and varnames

Author(s)

Jim Ramsay

See Also

plot.fd

---

fdPar *Define a Functional Parameter Object*

Description

Functional parameter objects are used as arguments to functions that estimate functional parameters, such as smoothing functions like smooth.basis. A functional parameter object is a functional data object with additional slots specifying a roughness penalty, a smoothing parameter and whether or not the functional parameter is to be estimated or held fixed. Functional parameter objects are used as arguments to functions that estimate functional parameters.

Usage

```
fdPar(fdobj=NULL, Lfdobj=NULL, lambda=0, estimate=TRUE,
      penmat=NULL)
```

Arguments

fdobj	a functional data object, functional basis object, a functional parameter object or a matrix. If it a matrix, it is replaced by fd(fdobj). If class(fdobj) == 'basisfd', it is converted to an object of class fd with a coefficient matrix consisting of a single column of zeros.
Lfdobj	either a nonnegative integer or a linear differential operator object. If NULL, Lfdobj depends on fdobj[['basis']][['type']]: • bspline Lfdobj <- int2Lfd(max(0, norder-2)), where norder = norder(fdobj).

- fourier Lfdobj = a harmonic acceleration operator:
  `Lfdobj <-vec2Lfd(c(0,(2*pi/diff(rng))^2,0),rng)`
  where rng = fdobj[['basis']][['rangeval']].
- anything elseLfdobj <- int2Lfd(0)

`lambda` a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter.

`estimate` not currently used.

`penmat` a roughness penalty matrix. Including this can eliminate the need to compute this matrix over and over again in some types of calculations.

Details

Functional parameters are often needed to specify initial values for iteratively refined estimates, as is the case in functions `register.fd` and `smooth.monotone`.

Often a list of functional parameters must be supplied to a function as an argument, and it may be that some of these parameters are considered known and must remain fixed during the analysis. This is the case for functions `fRegress` and `pda.fd`, for example.

Value

a functional parameter object (i.e., an object of class `fdPar`), which is a list with the following components:

`fd` a functional data object (i.e., with class `fd`)

`Lfd` a linear differential operator object (i.e., with class `Lfd`)

`lambda` a nonnegative real number

`estimate` not currently used

`penmat` either NULL or a square, symmetric matrix with penmat[i, j] = integral over fd[['basis']][['rangeval']] of basis[i]*basis[j]

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York

See Also

`cca.fd`, `density.fd`, `fRegress`, `intensity.fd`, `pca.fd`, `smooth.fdPar`, `smooth.basis`, `smooth.monotone`, `int2Lfd`

Examples

```
##
## Simple example
##
#  set up range for density
rangeval <- c(-3,3)
#  set up some standard normal data
x <- rnorm(50)
#  make sure values within the range
x[x < -3] <- -2.99
x[x >  3] <-  2.99
#  set up basis for W(x)
basisobj <- create.bspline.basis(rangeval, 11)
#  set up initial value for Wfdobj
Wfd0 <- fd(matrix(0,11,1), basisobj)
WfdParobj <- fdPar(Wfd0)

WfdP3 <- fdPar(seq(-3, 3, length=11))

##
##  smooth the Canadian daily temperature data
##
#    set up the fourier basis
nbasis   <- 365
dayrange <- c(0,365)
daybasis <- create.fourier.basis(dayrange, nbasis)
dayperiod <- 365
harmaccelLfd <- vec2Lfd(c(0,(2*pi/365)^2,0), dayrange)
#  Make temperature fd object
#  Temperature data are in 12 by 365 matrix tempav
#    See analyses of weather data.
#  Set up sampling points at mid days
daytime  <- (1:365)-0.5
#  Convert the data to a functional data object
daybasis65 <- create.fourier.basis(dayrange, nbasis, dayperiod)
templambda <- 1e1
tempfdPar  <- fdPar(fdobj=daybasis65, Lfdobj=harmaccelLfd,
                    lambda=templambda)

#FIXME
#tempfd <- smooth.basis(CanadianWeather$tempav, daytime, tempfdPar)$fd
#  Set up the harmonic acceleration operator
Lbasis  <- create.constant.basis(dayrange);
Lcoef   <- matrix(c(0,(2*pi/365)^2,0),1,3)
bfdobj  <- fd(Lcoef,Lbasis)
bwtlist <- fd2list(bfdobj)
harmaccelLfd <- Lfd(3, bwtlist)
#  Define the functional parameter object for
#  smoothing the temperature data
lambda   <- 0.01  #  minimum GCV estimate
#tempPar <- fdPar(daybasis365, harmaccelLfd, lambda)
#  smooth the data
```

```
#tempfd <- smooth.basis(daytime, CanadialWeather$tempav, tempPar)$fd
#  plot the temperature curves
#plot(tempfd)

##
## with rangeval of class Date and POSIXct
##
```

`fourier` *Fourier Basis Function Values*

Description

Evaluates a set of Fourier basis functions, or a derivative of these functions, at a set of arguments.

Usage

```
fourier(x, nbasis=n, period=span, nderiv=0)
```

Arguments

`x`	a vector of argument values at which the Fourier basis functions are to be evaluated.
`nbasis`	the number of basis functions in the Fourier basis. The first basis function is the constant function, followed by sets of sine/cosine pairs. Normally the number of basis functions will be an odd. The default number is the number of argument values.
`period`	the width of an interval over which all sine/cosine basis functions repeat themselves. The default is the difference between the largest and smallest argument values.
`nderiv`	the derivative to be evaluated. The derivative must not exceed the order. The default derivative is 0, meaning that the basis functions themselves are evaluated.

Value

a matrix of function values. The number of rows equals the number of arguments, and the number of columns equals the number of basis functions.

See Also

`fourierpen`

Examples

```
#  set up a set of 11 argument values
x <- seq(0,1,0.1)
names(x) <- paste("x", 0:10, sep="")
#  compute values for five Fourier basis functions
#  with the default period (1) and derivative (0)
(basismat <- fourier(x, 5))

# Create a false Fourier basis, i.e., nbasis = 1
# = a constant function
fourier(x, 1)
```

`fourierpen` *Fourier Penalty Matrix*

Description

Computes the matrix defining the roughness penalty for functions expressed in terms of a Fourier basis.

Usage

```
fourierpen(basisobj, Lfdobj=int2Lfd(2))
```

Arguments

`basisobj`	a Fourier basis object.
`Lfdobj`	either a nonnegative integer or a linear differential operator object.

Details

A roughness penalty for a function $x(t)$ is defined by integrating the square of either the derivative of $ x(t) $ or, more generally, the result of applying a linear differential operator L to it. The most common roughness penalty is the integral of the square of the second derivative, and this is the default. To apply this roughness penalty, the matrix of inner products of the basis functions (possibly after applying the linear differential operator to them) defining this function is necessary. This function just calls the roughness penalty evaluation function specific to the basis involved.

Value

a symmetric matrix of order equal to the number of basis functions defined by the Fourier basis object. Each element is the inner product of two Fourier basis functions after applying the derivative or linear differential operator defined by Lfdobj.

See Also

`fourier`, `eval.penalty`, `getbasispenalty`

Examples

```
#  set up a Fourier basis with 13 basis functions
#  and and period 1.0.
basisobj <- create.fourier.basis(c(0,1),13)
#  compute the 13 by 13 matrix of inner products
#  of second derivatives
penmat <- fourierpen(basisobj)
```

`Fperm.fd` *Permutation F-test for functional linear regression.*

Description

`Fperm.fd` creates a null distribution for a test of no effect in functional linear regression. It makes generic use of `fRegress` and permutes the `yfdPar` input.

Usage

```
Fperm.fd(yfdPar, xfdlist, betalist, wt=NULL, nperm=200,
         argvals=NULL, q=0.05, plotres=TRUE, ...)
```

Arguments

`yfdPar`	the dependent variable object. It may be an object of three possible classes: • vector if the dependent variable is scalar. • fd a functional data object if the dependent variable is functional. • fdPar a functional parameter object if the dependent variable is functional, and if it is necessary to smooth the prediction of the dependent variable.
`xfdlist`	a list of length equal to the number of independent variables. Members of this list are the independent variables. They be objects of either of these two classes: • a vector if the independent dependent variable is scalar. • a functional data object if the dependent variable is functional. In either case, the object must have the same number of replications as the dependent variable object. That is, if it is a scalar, it must be of the same length as the dependent variable, and if it is functional, it must have the same number of replications as the dependent variable.
`betalist`	a list of length equal to the number of independent variables. Members of this list define the regression functions to be estimated. They are functional parameter objects. Note that even if corresponding independent variable is scalar, its regression coefficient will be functional if the dependent variable is functional. Each of these functional parameter objects defines a single functional data object, that is, with only one replication.
`wt`	weights for weighted least squares, defaults to all 1.

nperm	number of permutations to use in creating the null distribution.
argvals	If yfdPar is a fd object, the points at which to evaluate the point-wise F-statistic.
q	Critical upper-tail quantile of the null distribution to compare to the observed F-statistic.
plotres	Argument to plot a visual display of the null distribution displaying the qth quantile and observed F-statistic.
...	Additional plotting arguments that can be used with plot.

Details

An F-statistic is calculated as the ratio of residual variance to predicted variance. The observed F-statistic is returned along with the permutation distribution.

If yfdPar is a fd object, the maximal value of the pointwise F-statistic is calculated. The pointwise F-statistics are also returned.

The default of setting q = 0.95 is, by now, fairly standard. The default nperm = 200 may be small, depending on the amount of computing time available.

If argvals is not specified and yfdPar is a fd object, it defaults to 101 equally-spaced points on the range of yfdPar.

Value

A list with the following components:

pval	the observed p-value of the permutation test.
qval	the qth quantile of the null distribution.
Fobs	the observed maximal F-statistic.
Fnull	a vector of length nperm giving the observed values of the permutation distribution.
Fvals	the pointwise values of the observed F-statistic.
Fnullvals	the pointwise values of of the permutation observations.
pvals.pts	pointwise p-values of the F-statistic.
qvals.pts	pointwise qth quantiles of the null distribution
fRegressList	the result of fRegress on the observed data
argvals	argument values for evaluating the F-statistic if yfdPar is a functional data object.

Side Effects

a plot of the functional observations

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

See Also

fRegress, Fstat.fd

Examples

```
##
## 1.  yfdPar = vector
##
annualprec <- log10(apply(
    CanadianWeather$dailyAv[,,"Precipitation.mm"], 2,sum))

#  set up a smaller basis using only 40 Fourier basis functions
#  to save some computation time

smallnbasis <- 40
smallbasis  <- create.fourier.basis(c(0, 365), smallnbasis)
tempfd      <- smooth.basis(day.5, CanadianWeather$dailyAv[,,"Temperature.C"],
                    smallbasis)$fd
constantfd <- fd(matrix(1,1,35), create.constant.basis(c(0, 365)))

xfdlist <- vector("list",2)
xfdlist[[1]] <- constantfd
xfdlist[[2]] <- tempfd[1:35]

betalist   <- vector("list",2)
#  set up the first regression function as a constant
betabasis1 <- create.constant.basis(c(0, 365))
betafd1    <- fd(0, betabasis1)
betafdPar1 <- fdPar(betafd1)
betalist[[1]] <- betafdPar1

nbetabasis  <- 35
betabasis2  <- create.fourier.basis(c(0, 365), nbetabasis)
betafd2     <- fd(matrix(0,nbetabasis,1), betabasis2)

lambda        <- 10^12.5
harmaccelLfd365 <- vec2Lfd(c(0,(2*pi/365)^2,0), c(0, 365))
betafdPar2    <- fdPar(betafd2, harmaccelLfd365, lambda)
betalist[[2]] <- betafdPar2

# Should use the default nperm = 200
# but use 10 to save test time for illustration
F.res2 = Fperm.fd(annualprec, xfdlist, betalist, nperm=10)

##
## 2.  yfdpar = Functional data object (class fd)
##
# The very simplest example is the equivalent of the permutation
# t-test on the growth data.

# First set up a basis system to hold the smooths
```

```
## Not run:

Knots  <- growth$age
norder <- 6
nbasis <- length(Knots) + norder - 2
hgtbasis <- create.bspline.basis(range(Knots), nbasis, norder, Knots)

# Now smooth with a fourth-derivative penalty and a very small smoothing
# parameter

Lfdobj <- 4
lambda <- 1e-2
growfdPar <- fdPar(hgtbasis, Lfdobj, lambda)

hgtfd <- smooth.basis(growth$age,
                      cbind(growth$hgtm,growth$hgtf),growfdPar)$fd

# Now set up factors for fRegress:

cbasis = create.constant.basis(range(Knots))

maleind = c(rep(1,ncol(growth$hgtm)),rep(0,ncol(growth$hgtf)))

constfd = fd( matrix(1,1,length(maleind)),cbasis)
maleindfd = fd( matrix(maleind,1,length(maleind)),cbasis)

xfdlist = list(constfd,maleindfd)

# The fdPar object for the coefficients and call Fperm.fd

betalist = list(fdPar(hgtbasis,2,1e-6),fdPar(hgtbasis,2,1e-6))

# Should use nperm = 200 or so,
# but use 10 to save test time
Fres = Fperm.fd(hgtfd,xfdlist,betalist,nperm=10)

## End(Not run)
```

---

fRegress *Functional Regression Analysis*

Description

This function carries out a functional regression analysis, where either the dependent variable or one or more independent variables are functional. Non-functional variables may be used on either side of the equation. In a simple problem where there is a single scalar independent covariate with values $z_i, i = 1, \ldots, N$ and a single functional covariate with values $x_i(t)$, the two versions of the model fit by fRegress are the *scalar* dependent variable model

$$y_i = \beta_1 z_i + \int x_i(t)\beta_2(t)\,dt + e_i$$

and the *concurrent* functional dependent variable model

$$y_i(t) = \beta_1(t)z_i + \beta_2(t)x_i(t) + e_i(t).$$

In these models, the final term e_i or $e_i(t)$ is a residual, lack of fit or error term.

In the concurrent functional linear model for a functional dependent variable, all functional variables are all evaluated at a common time or argument value t. That is, the fit is defined in terms of the behavior of all variables at a fixed time, or in terms of "now" behavior.

All regression coefficient functions $\beta_j(t)$ are considered to be functional. In the case of a scalar dependent variable, the regression coefficient for a scalar covariate is converted to a functional variable with a constant basis. All regression coefficient functions can be forced to be *smooth* through the use of roughness penalties, and consequently are specified in the argument list as functional parameter objects.

Usage

```
fRegress(y, ...)
## S3 method for class 'fd'
fRegress(y, xfdlist, betalist, wt=NULL,
                     y2cMap=NULL, SigmaE=NULL, returnMatrix=FALSE,
                        method=c('fRegress', 'model'), sep='.', ...)
## S3 method for class 'double'
fRegress(y, xfdlist, betalist, wt=NULL,
                     y2cMap=NULL, SigmaE=NULL, returnMatrix=FALSE, ...)
## S3 method for class 'formula'
fRegress(y, data=NULL, betalist=NULL, wt=NULL,
                 y2cMap=NULL, SigmaE=NULL,
                 method=c('fRegress', 'model'), sep='.', ...)
## S3 method for class 'character'
fRegress(y, data=NULL, betalist=NULL, wt=NULL,
                 y2cMap=NULL, SigmaE=NULL,
                 method=c('fRegress', 'model'), sep='.', ...)
```

Arguments

y	the dependent variable object. It may be an object of five possible classes or attributes:

- character or formula a `formula` object or a `character` object that can be coerced into a `formula` providing a symbolic description of the model to be fitted satisfying the following rules:
  The left hand side, `formula` `y`, must be either a numeric vector or a univariate object of class `fd`.
  All objects named on the right hand side must be either `numeric` or `fd` (functional data). The number of replications of `fd` object(s) must match

each other and the number of observations of numeric objects named, as well as the number of replications of the dependent variable object. The right hand side of this formula is translated into xfdlist, then passed to another method for fitting (unless method = 'model'). Multivariate independent variables are allowed in a formula and are split into univariate independent variables in the resulting xfdlist. Similarly, categorical independent variables with k levels are translated into k-1 contrasts in xfdlist. Any smoothing information is passed to the corresponding component of betalist.

- numeric a numeric vector object or a matrix object if the dependent variable is numeric or a matrix.
- fd a functional data object or an fdPar object if the dependent variable is functional.

data — an optional list or data.frame containing names of objects identified in the formula or character y.

xfdlist — a list of length equal to the number of independent variables (including any intercept). Members of this list are the independent variables. They can be objects of either of these two classes:

- scalar a numeric vector if the independent variable is scalar.
- fd a (univariate) functional data object.

In either case, the object must have the same number of replications as the dependent variable object. That is, if it is a scalar, it must be of the same length as the dependent variable, and if it is functional, it must have the same number of replications as the dependent variable. (Only univariate independent variables are currently allowed in xfdlist.)

betalist — For the fd, fdPar, and numeric methods, betalist must be a list of length equal to length(xfdlist). Members of this list are functional parameter objects (class fdPar) defining the regression functions to be estimated. Even if a corresponding independent variable is scalar, its regression coefficient must be functional if the dependent variable is functional. (If the dependent variable is a scalar, the coefficients of scalar independent variables, including the intercept, must be constants, but the coefficients of functional independent variables must be functional.) Each of these functional parameter objects defines a single functional data object, that is, with only one replication.

For the formula and character methods, betalist can be either a list, as for the other methods, or NULL, in which case a list is created. If betalist is created, it will use the bases from the corresponding component of xfdlist if it is function or from the response variable. Smoothing information (arguments Lfdobj, lambda, estimate, and penmat of function fdPar) will come from the corresponding component of xfdlist if it is of class fdPar (or for scalar independent variables from the response variable if it is of class fdPar) or from optional ... arguments if the reference variable is not of class fdPar.

wt — weights for weighted least squares

y2cMap — the matrix mapping from the vector of observed values to the coefficients for the dependent variable. This is output by function smooth.basis. If this is supplied, confidence limits are computed, otherwise not.

SigmaE	Estimate of the covariances among the residuals. This can only be estimated after a preliminary analysis with fRegress.
method	a character string matching either fRegress for functional regression estimation or mode to create the argument lists for functional regression estimation without running it.
sep	separator for creating names for multiple variables for fRegress.fdPar or fRegress.numeric created from single variables on the right hand side of the formula y. This happens with multidimensional fd objects as well as with categorical variables.
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.
...	optional arguments

Details

Alternative forms of functional regression can be categorized with traditional least squares using the following 2 x 2 table:

	explanatory	variable
response	scalar	function
scalar	lm	fRegress.numeric
function	fRegress.fd or fRegress.fdPar	fRegress.fd or fRegress.fdPar or linmod

For fRegress.numeric, the numeric response is assumed to be the sum of integrals of xfd * beta for all functional xfd terms.

fRegress.fd or .fdPar produces a concurrent regression with each beta being also a (univariate) function.

linmod predicts a functional response from a convolution integral, estimating a bivariate regression function.

In the computation of regression function estimates in fRegress, all independent variables are treated as if they are functional. If argument xfdlist contains one or more vectors, these are converted to functional data objects having the constant basis with coefficients equal to the elements of the vector.

Needless to say, if all the variables in the model are scalar, do NOT use this function. Instead, use either lm or lsfit.

These functions provide a partial implementation of Ramsay and Silverman (2005, chapters 12-20).

Value

These functions return either a standard fRegress fit object or or a model specification:

The fRegress fit object case:

A list of class fRegress with the following components:

- y: The first argument in the call to fRegress. This argument is coerced to class fd in fda version 5.1.9. Prior versions of the package converted it to an fdPar, but the extra structures in that class were not used in any of the fRegress codes.
- xfdlist: The second argument in the call to fRegress.
- betalist: The third argument in the call to fRegress.
- betaestlist: A list of length equal to the number of independent variables and with members having the same functional parameter structure as the corresponding members of betalist. These are the estimated regression coefficient functions.
- yhatfdobj: A functional parameter object (class fdPar) if the dependent variable is functional or a vector if the dependent variable is scalar. This is the set of predicted by the functional regression model for the dependent variable.
- Cmatinv: A matrix containing the inverse of the coefficient matrix for the linear equations that define the solution to the regression problem. This matrix is required for function fRegress.stderr that estimates confidence regions for the regression coefficient function estimates.
- wt: The vector of weights input or inferred.
If class(y) is numeric, the fRegress object also includes:
- df: The equivalent degrees of freedom for the fit.
- OCV the leave-one-out cross validation score for the model.
- gcv: The generalized cross validation score.
If class(y) is fd or fdPar, the fRegress object returned also includes 5 other components:
- y2cMap: An input y2cMap.
- SigmaE: An input SigmaE.
- betastderrlist: An fd object estimating the standard errors of betaestlist.
- bvar: A covariance matrix for regression coefficient estimates.
- c2bMap: A mapping matrix that maps variation in Cmat to variation in regression coefficients.

The model specification object case:

The fRegress.formula and fRegress.character functions translate the formula into the argument list required by fRegress.fdPar or fRegress.numeric. With the default value 'fRegress' for the argument method, this list is then used to call the appropriate other fRegress function.

Alternatively, to see how the formula is translated, use the alternative 'model' value for the argument method. In that case, the function returns a list with the arguments otherwise passed to these other functions plus the following additional components:

- xfdlist0: A list of the objects named on the right hand side of formula. This will differ from xfdlist for any categorical or multivariate right hand side object.
- type: the type component of any fd object on the right hand side of formula.
- nbasis: A vector containing the nbasis components of variables named in formula having such components.

- xVars: An integer vector with all the variable names on the right hand side of `formula` containing the corresponding number of variables in `xfdlist`. This can exceed 1 for any multivariate object on the right hand side of class either `numeric` or `fd` as well as any categorical variable.

Author(s)

J. O. Ramsay, Giles Hooker, and Spencer Graves

References

Ramsay, James O., Hooker, Giles, and Graves, Spencer (2009) *Functional Data Analysis in R and Matlab*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

See Also

`fRegress.formula`, `fRegress.double`, `fRegress.stderr`, `fRegress.CV`, `linmod`

Examples

```
###
###
###  ---------  vector response with functional explanatory variable  ---------
###
###

#  data are in Canadian Weather object
#  print the names of the data
print(names(CanadianWeather))
#  set up log10 of annual precipitation for 35 weather stations
annualprec <- log10(apply(CanadianWeather$dailyAv[,,"Precipitation.mm"], 2,sum))
# The simplest 'fRegress' call is singular with more bases
# than observations, so we use only 25 basis functions, for this example
smallbasis  <- create.fourier.basis(c(0, 365), 25)
# The covariate is the temperature curve for each station.
tempfd <- smooth.basis(day.5,
          CanadianWeather$dailyAv[,,"Temperature.C"], smallbasis)$fd

##
## formula interface:  specify the model by a formula, the method
## fRegress.formula automatically sets up the regression coefficient functions,
## a constant function for the intercept, and a higher dimensional function
## for the inner product with temperature
##

precip.Temp1 <- fRegress(annualprec ~ tempfd)
#  the output is a list with class name fRegress, display names
names(precip.Temp1)
#  the vector of fits to the data is object  precip.Temp1$yfdPar,
```

```
#  but since the dependent variable is a vector, so is the fit
annualprec.fit1 <- precip.Temp1$yhatfdobj
#  plot the data and the fit
plot(annualprec.fit1, annualprec, type="p", pch="o")
lines(annualprec.fit1, annualprec.fit1, lty=2)
#  print root mean squared error
RMSE <- sqrt(mean((annualprec-annualprec.fit1)^2))
print(paste("RMSE =",RMSE))
#  plot the estimated regression function
plot(precip.Temp1$betaestlist[[2]])
#  This isn't helpful either, the coefficient function is too
#  complicated to interpret.
#  display the number of basis functions used:
print(precip.Temp1$betaestlist[[2]]$fd$basis$nbasis)
#  25 basis functions to fit 35 values, no wonder we over-fit the data

##
## Get the default setup and modify it
## the "model" value of the method argument causes the analysis
## to produce a list vector of arguments for calling the
## fRegress function
##

precip.Temp.mdl1 <- fRegress(annualprec ~ tempfd, method="model")
# First confirm we get the same answer as above by calling
# function fRegress() with these arguments:
precip.Temp.m <- do.call('fRegress', precip.Temp.mdl1)

all.equal(precip.Temp.m, precip.Temp1)


#  set up a smaller basis for beta2 than for temperature so that we
#  get a more parsimonious fit to the data

nbetabasis2 <- 21  #  not much less, but we add some roughness penalization
betabasis2  <- create.fourier.basis(c(0, 365), nbetabasis2)
betafd2     <- fd(rep(0, nbetabasis2), betabasis2)
# add smoothing
betafdPar2  <- fdPar(betafd2, lambda=10)

# replace the regress coefficient function with this fdPar object
precip.Temp.mdl2 <- precip.Temp.mdl1
precip.Temp.mdl2[['betalist']][['tempfd']] <- betafdPar2

# Now do re-fit the data
precip.Temp2 <- do.call('fRegress', precip.Temp.mdl2)

# Compare the two fits:
#  degrees of freedom
precip.Temp1[['df']] # 26
precip.Temp2[['df']]  # 8
#  root-mean-squared errors:
RMSE1 <- sqrt(mean(with(precip.Temp1, (yhatfdobj-yfdobj)^2)))
```

```
RMSE2 <- sqrt(mean(with(precip.Temp2, (yhatfdobj-yfdobj)^2)))
#  display further results for the more parsimonious model
annualprec.fit2 <- precip.Temp2$yhatfdobj
plot(annualprec.fit2, annualprec, type="p", pch="o")
lines(annualprec.fit2, annualprec.fit2, lty=2)
#  plot the estimated regression function
plot(precip.Temp2$betaestlist[[2]])
#  now we see that it is primarily the temperatures in the
#  early winter that provide the fit to log precipitation by temperature

##
## Manual construction of xfdlist and betalist
##

xfdlist <- list(const=rep(1, 35), tempfd=tempfd)

# The intercept must be constant for a scalar response
betabasis1 <- create.constant.basis(c(0, 365))
betafd1    <- fd(0, betabasis1)
betafdPar1 <- fdPar(betafd1)

betafd2     <- create.bspline.basis(c(0, 365),7)
# convert to an fdPar object
betafdPar2  <- fdPar(betafd2)

betalist <- list(const=betafdPar1, tempfd=betafdPar2)

precip.Temp3   <- fRegress(annualprec, xfdlist, betalist)
annualprec.fit3 <- precip.Temp3$yhatfdobj
#  plot the data and the fit
plot(annualprec.fit3, annualprec, type="p", pch="o")
lines(annualprec.fit3, annualprec.fit3)
plot(precip.Temp3$betaestlist[[2]])

###
###
### --------  functional response with vector explanatory variables  ----------
###
###

##
## simplest:  formula interface
##

daybasis65 <- create.fourier.basis(rangeval=c(0, 365), nbasis=65,
                 axes=list('axesIntervals'))
Temp.fd <- with(CanadianWeather, smooth.basisPar(day.5,
                dailyAv[,,'Temperature.C'], daybasis65)$fd)
TempRgn.f <- fRegress(Temp.fd ~ region, CanadianWeather)

##
## Get the default setup and possibly modify it
##
```

```
TempRgn.mdl <- fRegress(Temp.fd ~ region, CanadianWeather, method='m')

# make desired modifications here
# then run

TempRgn.m <- do.call('fRegress', TempRgn.mdl)

# no change, so match the first run

all.equal(TempRgn.m, TempRgn.f)


##
## More detailed set up
##

region.contrasts <- model.matrix(~factor(CanadianWeather$region))
rgnContr3 <- region.contrasts
dim(rgnContr3) <- c(1, 35, 4)
dimnames(rgnContr3) <- list('', CanadianWeather$place, c('const',
   paste('region', c('Atlantic', 'Continental', 'Pacific'), sep='.')) )

const365 <- create.constant.basis(c(0, 365))
region.fd.Atlantic <- fd(matrix(rgnContr3[,,2], 1), const365)
# str(region.fd.Atlantic)
region.fd.Continental <- fd(matrix(rgnContr3[,,3], 1), const365)
region.fd.Pacific <- fd(matrix(rgnContr3[,,4], 1), const365)
region.fdlist <- list(const=rep(1, 35),
     region.Atlantic=region.fd.Atlantic,
     region.Continental=region.fd.Continental,
     region.Pacific=region.fd.Pacific)
# str(TempRgn.mdl$betalist)

beta1 <- with(Temp.fd, fd(basisobj=basis, fdnames=fdnames))
beta0 <- fdPar(beta1)
betalist <- list(const=beta0, region.Atlantic=beta0,
             region.Continental=beta0, region.Pacific=beta0)

TempRgn <- fRegress(Temp.fd, region.fdlist, betalist)


all.equal(TempRgn, TempRgn.f)


###
###
### --------  functional response with functional explanatory variable  -------
###
###

##
##  predict knee angle from hip angle;  from demo('gait', package='fda')
```

```
##
## formula interface
##
gaittime  <- as.matrix((1:20)/21)
gaitrange <- c(0,20)
gaitbasis <- create.fourier.basis(gaitrange, nbasis=21)
harmaccelLfd <- vec2Lfd(c(0, (2*pi/20)^2, 0), rangeval=gaitrange)
gaitfd <- smooth.basisPar(gaittime, gait, gaitbasis,
                          Lfdobj=harmaccelLfd, lambda=1e-2)$fd
hipfd  <- gaitfd[,1]
kneefd <- gaitfd[,2]

knee.hip.f <- fRegress(kneefd ~ hipfd)

##
## manual set-up
##

#  set up the list of covariate objects
const  <- rep(1, dim(kneefd$coef)[2])
xfdlist  <- list(const=const, hipfd=hipfd)

beta0 <- with(kneefd, fd(basisobj=basis, fdnames=fdnames))
beta1 <- with(hipfd, fd(basisobj=basis, fdnames=fdnames))

betalist  <- list(const=fdPar(beta0), hipfd=fdPar(beta1))

fRegressout <- fRegress(kneefd, xfdlist, betalist)


all.equal(fRegressout, knee.hip.f)
```

`fRegress.CV` *Computes Cross-validated Error Sum of Integrated Squared Errors for a Functional Regression Model*

Description

For a functional regression model, a cross-validated error sum of squares is computed. For a functional dependent variable this is the sum of integrated squared errors. For a scalar response, this function has been superseded by the OCV and gcv elements returned by `fRegress`. This function aids the choice of smoothing parameters in this model using the cross-validated error sum of squares criterion.

Usage

```
#fRegress.CV(y, xfdlist, betalist, wt=NULL, CVobs=1:N,
```

```
#            returnMatrix=FALSE, ...)

#NOTE:  The following is required by CRAN rules that
# function names like "as.numeric" must follow the documentation
# standards for S3 generics, even when they are not.
# Please ignore the following line:
## S3 method for class 'CV'
fRegress(y, xfdlist, betalist, wt=NULL, CVobs=1:N,
            returnMatrix=FALSE, ...)
```

Arguments

y	the dependent variable object.
xfdlist	a list whose members are functional parameter objects specifying functional independent variables. Some of these may also be vectors specifying scalar independent variables.
betalist	a list containing functional parameter objects specifying the regression functions and their level of smoothing.
wt	weights for weighted least squares. Defaults to all 1's.
CVobs	Indices of observations to be deleted. Defaults to 1:N.
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.
...	optional arguments not used by fRegress.CV but needed for superficial compatibility with fRegress methods.

Value

A list containing

SSE.CV	The sum of squared errors, or integrated squared errors
errfd.cv	Either a vector or a functional data object giving the cross-validated errors

See Also

fRegress, fRegress.stderr

Examples

```
#See the analyses of the Canadian daily weather data.
```

`fRegress.stderr` *Compute Standard errors of Coefficient Functions Estimated by Functional Regression Analysis*

Description

Function `fRegress` carries out a functional regression analysis of the concurrent kind, and estimates a regression coefficient function corresponding to each independent variable, whether it is scalar or functional. This function uses the list that is output by `fRegress` to provide standard error functions for each regression function. These standard error functions are pointwise, meaning that sampling standard deviation functions only are computed, and not sampling covariances.

Usage

```
## S3 method for class 'stderr'
fRegress(y, y2cMap, SigmaE, returnMatrix=FALSE, ...)
```

Arguments

`y`	the named list that is returned from a call to function `fRegress`, where it is referred to as fRegressList. (R syntax requires that the first argument of any function beginning with `fRegress.` must begin with `y`.)
`y2cMap`	a matrix that contains the linear transformation that takes the raw data values into the coefficients defining a smooth functional data object. Typically, this matrix is returned from a call to function `smooth.basis` that generates the dependent variable objects. If the dependent variable is scalar, this matrix is an identity matrix of order equal to the length of the vector.
`SigmaE`	either a matrix or a bivariate functional data object according to whether the dependent variable is scalar or functional, respectively. This object has a number of replications equal to the length of the dependent variable object. It contains an estimate of the variance-covariance matrix or function for the residuals.
`returnMatrix`	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.
`...`	optional arguments not used by `fRegress.stderr` but needed for superficial compatibility with `fRegress` methods.

Value

a named list of length 3 containing:

`betastderrlist`	a list object of length the number of independent variables. Each member contains a functional parameter object for the standard error of a regression function.
`bvar`	a symmetric matrix containing sampling variances and covariances for the matrix of regression coefficients for the regression functions. These are stored column-wise in defining BVARIANCE.
`c2bMap`	a matrix containing the mapping from response variable coefficients to coefficients for regression coefficients.

See Also

fRegress, fRegress.CV

Examples

```
#See the weather data analyses in the file daily.ssc for
#examples of the use of function fRegress.stderr.
```

`Fstat.fd` *F-statistic for functional linear regression.*

Description

Fstat.fd calculates a pointwise F-statistic for functional linear regression.

Usage

```
Fstat.fd(y,yhat,argvals=NULL)
```

Arguments

`y`	the dependent variable object. It may be: • a vector if the dependent variable is scalar. • a functional data object if the dependent variable is functional.
`yhat`	The predicted values corresponding to `y`. It must be of the same class.
`argvals`	If `yfdPar` is a functional data object, the points at which to evaluate the pointwise F-statistic.

Details

An F-statistic is calculated as the ratio of residual variance to predicted variance.

If `argvals` is not specified and `yfdPar` is a `fd` object, it defaults to 101 equally-spaced points on the range of `yfdPar`.

Value

A list with components

`F`	the calculated pointwise F-statistics.
`argvals`	argument values for evaluating the F-statistic if `yfdPar` is a functional data object.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

See Also

fRegress Fstat.fd

---

gait *Hip and knee angle while walking*

Description

Hip and knee angle in degrees through a 20 point movement cycle for 39 boys

Format

An array of dim c(20, 39, 2) giving the "Hip Angle" and "Knee Angle" for 39 repetitions of a 20 point gait cycle.

Details

The components of dimnames(gait) are as follows:

[[1]] standardized gait time = seq(from=0.025, to=0.975, by=0.05)

[[2]] subject ID = "boy1", "boy2", ..., "boy39"

[[3]] gait variable = "Hip Angle" or "Knee Angle"

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

Examples

```
plot(gait[,1, 1], gait[, 1, 2], type="b")
```

`geigen` *Generalized eigenanalysis*

Description

Find matrices L and M to maximize

tr(L'AM) / sqrt(tr(L'BL) tr(M'CM'))

where A = a p x q matrix, B = p x p symmetric, positive definite matrix, B = q x q symmetric positive definite matrix, L = p x s matrix, and M = q x s matrix, where s = the number of non-zero generalized eigenvalues of A.

Usage

```
geigen(Amat, Bmat, Cmat)
```

Arguments

`Amat`	a numeric matrix
`Bmat`	a symmetric, positive definite matrix with dimension = number of rows of A
`Cmat`	a symmetric, positive definite matrix with dimension = number of columns of A

Value

list(values, Lmat, Mmat)

See Also

`eigen`

Examples

```
A <- matrix(1:6, 2)
B <- matrix(c(2, 1, 1, 2), 2)
C <- diag(1:3)
ABC <- geigen(A, B, C)
```

getbasismatrix *Values of Basis Functions or their Derivatives*

Description

Evaluate a set of basis functions or their derivatives at a set of argument values.

Usage

```
getbasismatrix(evalarg, basisobj, nderiv=0, returnMatrix=FALSE)
```

Arguments

evalarg	a vector of arguments values.
basisobj	a basis object.
nderiv	a nonnegative integer specifying the derivative to be evaluated.
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Value

a matrix of basis function or derivative values. Rows correspond to argument values and columns to basis functions.

See Also

eval.fd

Examples

```
##
## Minimal example:  a B-spline of order 1, i.e., a step function
## with 0 interior knots:
##
bspl1.1 <- create.bspline.basis(norder=1, breaks=0:1)
m <- getbasismatrix(seq(0, 1, .2), bspl1.1)

# check
m. <- matrix(rep(1, 6), 6,
    dimnames=list(NULL, 'bspl') )

all.equal(m, m.)


##
## Date and POSIXct
##
# Date
```

```
July4.1776    <- as.Date('1776-07-04')
Apr30.1789    <- as.Date('1789-04-30')
AmRev         <- c(July4.1776, Apr30.1789)
BspRevolution <- create.bspline.basis(AmRev)
AmRevYears    <- as.numeric(seq(July4.1776, Apr30.1789, length.out=14))
AmRevMatrix   <- getbasismatrix(AmRevYears, BspRevolution)
matplot(AmRevYears, AmRevMatrix, type='b')

# POSIXct
AmRev.ct    <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev.ct   <- create.bspline.basis(AmRev.ct)
AmRevYrs.ct <- as.numeric(seq(AmRev.ct[1], AmRev.ct[2], length.out=14))
AmRevMat.ct <- getbasismatrix(AmRevYrs.ct, BspRev.ct)
matplot(AmRevYrs.ct, AmRevMat.ct, type='b')
```

`getbasispenalty` *Evaluate a Roughness Penalty Matrix*

Description

A basis roughness penalty matrix is the matrix containing the possible inner products of pairs of basis functions. These inner products are typically defined in terms of the value of a derivative or of a linear differential operator applied to the basis function. The basis penalty matrix plays an important role in the computation of functions whose roughness is controlled by a roughness penalty.

Usage

```
getbasispenalty(basisobj, Lfdobj=NULL)
```

Arguments

`basisobj`	a basis object.
`Lfdobj`	

Details

A roughness penalty for a function $x(t)$ is defined by integrating the square of either the derivative of $ x(t) $ or, more generally, the result of applying a linear differential operator L to it. The most common roughness penalty is the integral of the square of the second derivative, and this is the default. To apply this roughness penalty, the matrix of inner products of the basis functions defining this function is necessary. This function just calls the roughness penalty evaluation function specific to the basis involved.

Value

a symmetric matrix of order equal to the number of basis functions defined by the B-spline basis object. Each element is the inner product of two B-spline basis functions after taking the derivative.

See Also

eval.penalty

Examples

```
#  set up a B-spline basis of order 4 with 13 basis functions
#  and knots at 0.0, 0.1,..., 0.9, 1.0.
basisobj <- create.bspline.basis(c(0,1),13)
#  compute the 13 by 13 matrix of inner products of second derivatives
penmat <- getbasispenalty(basisobj)
#  set up a Fourier basis with 13 basis functions
#  and and period 1.0.
basisobj <- create.fourier.basis(c(0,1),13)
#  compute the 13 by 13 matrix of inner products of second derivatives
penmat <- getbasispenalty(basisobj)
```

---

getbasisrange *Extract the range from a basis object*

Description

Extracts the 'range' component from basis object 'basisobj'.

Usage

```
getbasisrange(basisobj)
```

Arguments

basisobj	a functional basis object

Value

a numeric vector of length 2

growth *Berkeley Growth Study data*

Description

A list containing the heights of 39 boys and 54 girls from age 1 to 18 and the ages at which they were collected.

Format

This list contains the following components:

hgtm a 31 by 39 numeric matrix giving the heights in centimeters of 39 boys at 31 ages.

hgtf a 31 by 54 numeric matrix giving the heights in centimeters of 54 girls at 31 ages.

age a numeric vector of length 31 giving the ages at which the heights were measured.

Details

The ages are not equally spaced.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York, ch. 6.

Tuddenham, R. D., and Snyder, M. M. (1954) "Physical growth of California boys and girls from birth to age 18", *University of California Publications in Child Development*, 1, 183-364.

Examples

```
with(growth, matplot(age, hgtf[, 1:10], type="b"))
```

handwrit *Cursive handwriting samples*

Description

20 cursive samples of 1401 (x, y,) coordinates for writing "fda"

Usage

```
handwrit
handwritTime
```

Format

- handwrit An array of dimensions (1401, 20, 2) giving 1401 pairs of (x, y) coordinates for each of 20 replicates of cursively writing "fda"
- handwritTime seq(0, 2300, length=1401) = sampling times

Details

These data are the X-Y coordinates of 20 replications of writing the script "fda". The subject was Jim Ramsay. Each replication is represented by 1401 coordinate values. The scripts have been extensively pre-processed. They have been adjusted to a common length that corresponds to 2.3 seconds or 2300 milliseconds, and they have already been registered so that important features in each script are aligned.

This analysis is designed to illustrate techniques for working with functional data having rather high frequency variation and represented by thousands of data points per record. Comments along the way explain the choices of analysis that were made.

The final result of the analysis is a third order linear differential equation for each coordinate forced by a constant and by time. The equations are able to reconstruct the scripts to a fairly high level of accuracy, and are also able to accommodate a substantial amount of the variation in the observed scripts across replications. by contrast, a second order equation was found to be completely inadequate.

An interesting surprise in the results is the role placed by a 120 millisecond cycle such that sharp features such as cusps correspond closely to this period. This 110-120 msec cycle seems is usually seen in human movement data involving rapid movements, such as speech, juggling and so on.

These 20 records have already been normalized to a common time interval of 2300 milliseconds and have beeen also registered so that prominent features occur at the same times across replications. Time will be measured in (approximate) milliseconds and space in meters. The data will require a small amount of smoothing, since an error of 0.5 mm is characteristic of the OPTOTRAK 3D measurement system used to collect the data.

Milliseconds were chosen as a time scale in order to make the ratio of the time unit to the inter-knot interval not too far from one. Otherwise, smoothing parameter values may be extremely small or extremely large.

The basis functions will be B-splines, with a spline placed at each knot. One may question whether so many basis functions are required, but this decision is found to be essential for stable derivative estimation up to the third order at and near the boundaries.

Order 7 was used to get a smooth third derivative, which requires penalizing the size of the 5th derivative, which in turn requires an order of at least 7. This implies norder + no. of interior knots = 1399 + 7 = 1406 basis functions.

The smoothing parameter value 1e8 was chosen to obtain a fitting error of about 0.5 mm, the known error level in the OPTOTRACK equipment.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Examples

```
plot(handwrit[, 1, 1], handwrit[, 1, 2], type="l")
```

`infantGrowth` *Tibia Length for One Baby*

Description

Measurement of the length of the tibia for the first 40 days of life for one infant.

Usage

```
data(infantGrowth)
```

Format

A matrix with three columns:

- dayage in days
- tibiaLength The average of five measurements of tibia length in millimeters
- sd.length The standard deviation of five measurements of tibia length in millimeters

Source

Hermanussen, M., Thiel, C., von Bueren, E., de los Angeles Rol. de Lama, M., Perez Romero, A., Arizonavarreta Ruiz, C., Burmeister, J., Tresguerras, J. A. F. (1998) Micro and macro perspectives in auxology: Findings and considerations upon the variability of short term and individual growth and the stability of population derived parameters, *Annals of Human Biology*, 25: 359-395.

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Examples

```
data(infantGrowth)
plot(tibiaLength~day, infantGrowth)
```

inprod *Inner products of Functional Data Objects.*

Description

Computes a matrix of inner products for each pairing of a replicate for the first argument with a replicate for the second argument. This is perhaps the most important function in the functional data library. Hardly any analysis fails to use inner products in some way, and many employ multiple inner products. While in certain cases these may be computed exactly, this is a more general function that approximates the inner product approximately when required. The inner product is defined by two derivatives or linear differential operators that are applied to the first two arguments. The range used to compute the inner product may be contained within the range over which the functions are defined. A weight functional data object may also be used to define weights for the inner product.

Usage

```
inprod(fdobj1, fdobj2,
       Lfdobj1=int2Lfd(0), Lfdobj2=int2Lfd(0), rng = range1, wtfd = 0)
```

Arguments

fdobj1	a functional data object or a basis object. If the object is of the basis class, it is converted to a functional data object by using the identity matrix as the coefficient matrix.
fdobj2	a functional data object or a basis object. If the object is of the basis class, it is converted to a functional data object by using the identity matrix as the coefficient matrix.
Lfdobj1	either a nonnegative integer specifying the derivative of the first argument to be used, or a linear differential operator object to be applied to the first argument.
Lfdobj2	either a nonnegative integer specifying the derivative of the second argument to be used, or a linear differential operator object to be applied to the second argument.
rng	a vector of length 2 defining a restricted range contained within the range over which the arguments are defined.
wtfd	a univariate functional data object with a single replicate defining weights to be used in computing the inner product.

Details

The approximation method is Richardson extrapolation using numerical integration by the trapezoidal rule. At each iteration, the number of values at which the functions are evaluated is doubled, and a polynomial extrapolation method is used to estimate the converged integral values as well as an error tolerance. Convergence is declared when the relative error falls below EPS for all products. The extrapolation method generally saves at least one and often two iterations relative to un-extrapolated trapezoidal integration. Functional data analyses will seldom need to use inprod

directly, but code developers should be aware of its pivotal role. Future work may require more sophisticated and specialized numerical integration methods. `inprod` computes the definite integral, but some functions such as `smooth.monotone` and `register.fd` also need to compute indefinite integrals. These use the same approximation scheme, but usually require more accuracy, and hence more iterations. When one or both arguments are basis objects, they are converted to functional data objects using identity matrices as the coefficient matrices. `inprod` is only called when there is no faster or exact method available. In cases where there is, it has been found that the approximation is good to about four to five significant digits, which is sufficient for most applications. Perhaps surprisingly, in the case of B-splines, the exact method is not appreciably faster, but of course is more accurate. `inprod` calls function `eval.fd` perhaps thousands of times, so high efficiency for this function and the functions that it calls is important.

Value

a matrix of inner products. The number of rows is the number of functions or basis functions in argument `fd1`, and the number of columns is the same thing for argument `fd2`.

References

Press, et, al, $Numerical Recipes$.

See Also

`eval.penalty`,

`inprod.bspline` *Compute Inner Products B-spline Expansions.*

Description

Computes the matrix of inner products when both functions are represented by B-spline expansions and when both derivatives are integers. This function is called by function `inprod`, and is not normally used directly.

Usage

```
inprod.bspline(fdobj1, fdobj2=fdobj1, nderiv1=0, nderiv2=0)
```

Arguments

`fdobj1`	a functional data object having a B-spline basis function expansion.
`fdobj2`	a second functional data object with a B-spline basis function expansion. By default, this is the same as the first argument.
`nderiv1`	a nonnegative integer specifying the derivative for the first argument.
`nderiv2`	a nonnegative integer specifying the derivative for the second argument.

Value

a matrix of inner products with number of rows equal to the number of replications of the first argument and number of columns equal to the number of replications of the second object.

`int2Lfd` *Convert Integer to Linear Differential Operator*

Description

This function turns an integer specifying an order of a derivative into the equivalent linear differential operator object. It is also useful for checking that an object is of the "Lfd" class.

Usage

```
int2Lfd(m=0)
```

Arguments

`m`	either a nonnegative integer or a linear differential operator object.

Details

Smoothing is achieved by penalizing the integral of the square of the derivative of order `m` over `rangeval`:

m = 0 penalizes the squared difference from 0 of the function

1 = penalize the square of the slope or velocity

2 = penalize the squared acceleration

3 = penalize the squared rate of change of acceleration

4 = penalize the squared curvature of acceleration?

Value

a linear differential operator object of the "Lfd" class that is equivalent to the integer argument.

Examples

```
# Lfd to penalize the squared acceleration
# typical for smoothing a cubic spline (order 4)
int2Lfd(2)

# Lfd to penalize the curvature of acceleration
# used with splines of order 6
# when it is desired to study velocity and acceleration
int2Lfd(4)
```

intensity.fd *Intensity Function for Point Process*

Description

The intensity mu of a series of event times that obey a homogeneous Poisson process is the mean number of events per unit time. When this event rate varies over time, the process is said to be nonhomogeneous, and $mu(t)$, and is estimated by this function `intensity.fd`.

Usage

```
intensity.fd(x, WfdParobj, conv=0.0001, iterlim=20,
             dbglev=1, returnMatrix=FALSE)
```

Arguments

`x`	a vector containing a strictly increasing series of event times. These event times assume that the the events begin to be observed at time 0, and therefore are times since the beginning of observation.
`WfdParobj`	a functional parameter object estimating the log-intensity function $W(t) = log[mu(t)]$. Because the intensity function $mu(t)$ is necessarily positive, it is represented by `mu(x) = exp[W(x)]`.
`conv`	a convergence criterion, required because the estimation process is iterative.
`iterlim`	maximum number of iterations that are allowed.
`dbglev`	either 0, 1, or 2. This controls the amount information printed out on each iteration, with 0 implying no output, 1 intermediate output level, and 2 full output. If levels 1 and 2 are used, turn off the output buffering option.
`returnMatrix`	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Details

The intensity function $I(t)$ is almost the same thing as a probability density function $p(t)$ estimated by function `densify.fd`. The only difference is the absence of the normalizing constant C that a density function requires in order to have a unit integral. The goal of the function is provide a smooth intensity function estimate that approaches some target intensity by an amount that is controlled by the linear differential operator `Lfdobj` and the penalty parameter in argument `WfdPar`. For example, if the first derivative of $W(t)$ is penalized heavily, this will force the function to approach a constant, which in turn will force the estimated Poisson process itself to be nearly homogeneous. To plot the intensity function or to evaluate it, evaluate `Wfdobj`, exponentiate the resulting vector.

Value

a named list of length 4 containing:

`Wfdobj`	a functional data object defining function $W(x)$ that that optimizes the fit to the data of the monotone function that it defines.
`Flist`	a named list containing three results for the final converged solution: (1) **f**: the optimal function value being minimized, (2) **grad**: the gradient vector at the optimal solution, and (3) **norm**: the norm of the gradient vector at the optimal solution.
`iternum`	the number of iterations.
`iterhist`	a `iternum`+1 by 5 matrix containing the iteration history.

See Also

`density.fd`

Examples

```
#  Generate 101 Poisson-distributed event times with
#  intensity or rate two events per unit time
N  <- 101
mu <- 2
#  generate 101 uniform deviates
uvec <- runif(rep(0,N))
#  convert to 101 exponential waiting times
wvec <- -log(1-uvec)/mu
#  accumulate to get event times
tvec <- cumsum(wvec)
tmax <- max(tvec)
#  set up an order 4 B-spline basis over [0,tmax] with
#  21 equally spaced knots
tbasis <- create.bspline.basis(c(0,tmax), 23)
#  set up a functional parameter object for W(t),
#  the log intensity function.  The first derivative
#  is penalized in order to smooth toward a constant
lambda <- 10
Wfd0 <- fd(matrix(0,23,1),tbasis)
WfdParobj <- fdPar(Wfd0, 1, lambda)
#  estimate the intensity function
Wfdobj <- intensity.fd(tvec, WfdParobj)$Wfdobj
#  get intensity function values at 0 and event times
events <- c(0,tvec)
intenvec <- exp(eval.fd(events,Wfdobj))
#  plot intensity function
plot(events, intenvec, type="b")
lines(c(0,tmax),c(mu,mu),lty=4)
```

is.basis *Confirm Object is Class "Basisfd"*

Description

Check that an argument is a basis object.

Usage

```
is.basis(basisobj)
```

Arguments

basisobj	an object to be checked.

Value

a logical value: TRUE if the class is correct, FALSE otherwise.

See Also

is.fd, is.fdPar, is.Lfd

is.eqbasis *Confirm that two objects of class "Basisfd" are identical*

Description

Check all the slots of two basis objects to see that they are identical.

Usage

```
is.eqbasis(basisobj1, basisobj2)
```

Arguments

basisobj1	The first basis object to be checked for being identical to the second.
basisobj2	The second basis object to be checked for being identical to the first.

Value

a logical value: TRUE if the two basis objects are identical, FALSE otherwise.

See Also

basisfd,

is.fd *Confirm Object has Class "fd"*

Description

Check that an argument is a functional data object.

Usage

```
is.fd(fdobj)
```

Arguments

fdobj	an object to be checked.

Value

a logical value: TRUE if the class is correct, FALSE otherwise.

See Also

is.basis, is.fdPar, is.Lfd

is.fdPar *Confirm Object has Class "fdPar"*

Description

Check that an argument is a functional parameter object.

Usage

```
is.fdPar(fdParobj)
```

Arguments

fdParobj	an object to be checked.

Value

a logical value: TRUE if the class is correct, FALSE otherwise.

See Also

is.basis, is.fd, is.Lfd

is.fdSmooth *Confirm Object has Class "fdSmooth"*

Description

Check that an argument is a functional parameter object.

Usage

```
is.fdSmooth(fdSmoothobj)
```

Arguments

fdSmoothobj	an object to be checked.

Value

a logical value: TRUE if the class is correct, FALSE otherwise.

See Also

is.basis, is.fd, is.Lfd, is.fdPar

is.Lfd *Confirm Object has Class "Lfd"*

Description

Check that an argument is a linear differential operator object.

Usage

```
is.Lfd(Lfdobj)
```

Arguments

Lfdobj	an object to be checked.

Value

a logical value: TRUE if the class is correct, FALSE otherwise.

See Also

is.basis, is.fd, is.fdPar

knots.fd *Extract the knots from a function basis or data object*

Description

Extract either all or only the interior knots from an object of class basisfd, fd, or fdSmooth.

Usage

```
## S3 method for class 'fd'
knots(Fn, interior=TRUE, ...)
## S3 method for class 'fdSmooth'
knots(Fn, interior=TRUE, ...)
## S3 method for class 'basisfd'
knots(Fn, interior=TRUE, ...)
```

Arguments

Fn	an object of class basisfd or containing such an object
interior	logical: if TRUE, Fn[["params"]] are returned. Else, nord <- norder(Fn); rng <- Fn[['rangeval']]; return c(rep(rng[1], nord), Fn[["params"]], rep(rng[2], nord))
...	ignored

Details

The interior knots of a bspline basis are stored in the params component. The remaining knots are in the rangeval component, with multiplicity norder(Fn).

Value

Numeric vector. If 'interior' is TRUE, this is the params component of the bspline basis. Otherwise, params is bracketed by rep(rangeval, norder(basisfd).

Author(s)

Spencer Graves

See Also

fd, create.bspline.basis,

lambda2df *Convert Smoothing Parameter to Degrees of Freedom*

Description

The degree of roughness of an estimated function is controlled by a smoothing parameter $lambda$ that directly multiplies the penalty. However, it can be difficult to interpret or choose this value, and it is often easier to determine the roughness by choosing a value that is equivalent of the degrees of freedom used by the smoothing procedure. This function converts a multipler $lambda$ into a degrees of freedom value.

Usage

```
lambda2df(argvals, basisobj, wtvec=rep(1, n), Lfdobj=NULL, lambda=0)
```

Arguments

argvals	a vector containing the argument values used in the smooth of the data.
basisobj	the basis object used in the smoothing of the data.
wtvec	the weight vector, if any, that was used in the smoothing of the data.
Lfdobj	the linear differential operator object used to defining the roughness penalty employed in smoothing the data.
lambda	the smoothing parameter to be converted.

Value

the equivalent degrees of freedom value.

See Also

df2lambda

lambda2gcv *Compute GCV Criterion*

Description

The generalized cross-validation or GCV criterion is often used to select an appropriate smoothing parameter value, by finding the smoothing parameter that minimizes GCV. This function locates that value.

Usage

```
lambda2gcv(log10lambda, argvals, y, fdParobj, wtvec=rep(1,length(argvals)))
```

Arguments

log10lambda	the logarithm (base 10) of the smoothing parameter
argvals	a vector of argument values.
y	the data to be smoothed.
fdParobj	a functional parameter object defining the smooth.
wtvec	a weight vector used in the smoothing.

Details

Currently, lambda2gcv

Value

1. $fdParobj[['lambda']] < -10^log10lambda$
2. smoothlist <- smooth.basks(argvals, y, fdParobj, wtvec)
3. return(smoothlist[['gcv']])

See Also

smooth.basis fdPar

---

landmark.reg.expData *Experiment data for landmark registration and alignment*

Description

A data.frame object obtaining flow cytometry data with 5 samples subject to landmark alignment.

Usage

```
sampleData
```

Format

data Flow cytometry fluorescent intensity data.

which Indicating the sample name of the data.

Author(s)

C. J. Wong <cwon2@fhcrc.org>

See Also

landmarkreg

Examples

```
data(landmark.reg.expData)
head(sampleData)
```

`landmarkreg` *Landmark Registration of Functional Observations*

Description

It is common to see that among a set of functions certain prominent features such peaks and valleys, called $landmarks$, do not occur at the same times, or other argument values. This is called $phase variation$, and it can be essential to align these features before proceeding with further functional data analyses. This function uses the timings of these features to align or register the curves. The registration involves estimating a nonlinear transformation of the argument continuum for each functional observation. This transformation is called a warping function. It must be strictly increasing and smooth.

Usage

```
landmarkreg(fdobj, ximarks, x0marks=xmeanmarks,
            WfdPar, monwrd=FALSE, ylambda=1e-10)
```

Arguments

`fdobj`	a functional data object containing the curves to be registered.
`ximarks`	a matrix containing the timings or argument values associated with the landmarks for the observations in `fd` to be registered. The number of rows N equals the number of observations, and the number of columns NL equals the number of landmarks. These landmark times must be in the interior of the interval over which the functions are defined.
`x0marks`	a vector of length NL of times of landmarks for target curve. If not supplied, the mean of the landmark times in `ximarks` is used.
`WfdPar`	a functional parameter object defining the warping functions that transform time in order to register the curves.
`monwrd`	A logical value: if `TRUE`, the warping function is estimated using a monotone smoothing method; otherwise, a regular smoothing method is used, which is not guaranteed to give strictly monotonic warping functions.
`ylambda`	Smoothing parameter controlling the smoothness of the registered functions. It can happen with high dimensional bases that local wiggles can appear in the registered curves or their derivatives that are not seen in the unregistered versions. In this case, this parameter should be increased to the point where they disappear.

Details

It is essential that the location of every landmark be clearly defined in each of the curves as well as the template function. If this is not the case, consider using the continuous registration function `register.fd`. Although requiring that a monotone smoother be used to estimate the warping functions is safer, it adds considerably to the computatation time since monotone smoothing is itself an iterative process. It is usually better to try an initial registration with this feature to see if there are any failures of monotonicity. Moreover, monotonicity failures can usually be cured by increasing the smoothing parameter defining `WfdPar`. Not much curvature is usually required in the warping functions, so a rather low power basis, usually B-splines, is suitable for defining the functional parameter argument `WfdPar`. A registration with a few prominent landmarks is often a good preliminary to using the more sophisticated but more lengthy process in `register.fd`.

Value

a named list of length 2 with components:

`fdreg`	a functional data object for the registered curves.
`warpfd`	a functional data object for the warping functions.

See Also

`register.fd`, `smooth.morph`

Examples

```
#See the analysis for the lip data in the examples.

## setting parameters
library(lattice)
data(landmark.reg.expData) ## containing an simple object called sampleData

# Preferred:
# eps <- .Machine$double.eps
# to reduce compute time:
eps <- 1000*.Machine$double.eps
from <- -1.0187
to <- 9.4551
# Preferred:
# nb <- 201
# to reduce compute time:
nb <- 31
nbreaks <- 11
## assign landmarks
landmark <- matrix(c(0.4999, 0.657, 0.8141, 0.5523, 0.5523,
                     3.3279, 3.066, 3.0137, 3.2231, 3.2231),
                   ncol=2)
wbasis <- create.bspline.basis(rangeval=c(from, to),
                       norder=4, breaks=seq(from, to, len=nbreaks))
Wfd0   <- fd(matrix(0,wbasis$nbasis,1),wbasis)
WfdPar <- fdPar(Wfd0, 1, 1e-4)
## get the density of the data
```

```
x <- split(sampleData, factor(sampleData$which))
# to save time, reduce the number of curves from 5 to 3
k <- 3
densY <- sapply(x[1:k], function(z){
            r <- range(z[, 1])
            z <- z[, 1]
            z <- z[z>r[1]+eps & z<r[2]-eps]
            density(z, from=from, to=to, n=nb, na.rm=TRUE)$y
        })

argvals <- seq(from, to, len=nb)
fdobj   <- smooth.basis(argvals, densY, wbasis,
                  fdnames = c("x", "samples", "density"))$fd

regDens   <- landmarkreg(fdobj, landmark[1:k,], WfdPar=WfdPar, monwrd=TRUE)

warpfdobj <- regDens$warpfd
warpedX   <- as.matrix(eval.fd(warpfdobj, argvals))
matplot(argvals, warpedX, type="l")
funs      <- apply(warpedX, 2, approxfun, argvals)
```

Lfd *Define a Linear Differential Operator Object*

Description

A linear differential operator of order m is defined, usually to specify a roughness penalty.

Usage

```
Lfd(nderiv=0, bwtlist=vector("list", 0))
```

Arguments

nderiv	a nonnegative integer specifying the order m of the highest order derivative in the operator
bwtlist	a list of length m. Each member contains a functional data object that acts as a weight function for a derivative. The first member weights the function, the second the first derivative, and so on up to order $m-1$.

Details

To check that an object is of this class, use functions `is.Lfd` or `int2Lfd`.

Linear differential operator objects are often used to define roughness penalties for smoothing towards a "hypersmooth" function that is annihilated by the operator. For example, the harmonic acceleration operator used in the analysis of the Canadian daily weather data annihilates linear combinations of $1, sin(2 pi t/365)$ and $cos(2 pi t/365)$, and the larger the smoothing parameter, the closer the smooth function will be to a function of this shape.

Function pda.fd estimates a linear differential operator object that comes as close as possible to annihilating a functional data object.

A linear differential operator of order m is a linear combination of the derivatives of a functional data object up to order m. The derivatives of orders 0, 1, ..., $m-1$ can each be multiplied by a weight function $b(t)$ that may or may not vary with argument t.

If the notation D^j is taken to mean "take the derivative of order j", then a linear differental operator L applied to function x has the expression

$Lx(t) = b_0(t) x(t) + b_1(t)Dx(t) + ... + b_{m-1}(t) D^{m-1} x(t) + D^mx(t)$

There are print, summary, and plot methods for objects of class Lfd.

Value

a linear differential operator object

See Also

int2Lfd, vec2Lfd, fdPar, pda.fd plot.Lfd

Examples

```
#  Set up the harmonic acceleration operator
dayrange  <- c(0,365)
Lbasis  <- create.constant.basis(dayrange,
                 axes=list("axesIntervals"))
Lcoef   <- matrix(c(0,(2*pi/365)^2,0),1,3)
bfdobj  <- fd(Lcoef,Lbasis)
bwtlist <- fd2list(bfdobj)
harmaccelLfd <- Lfd(3, bwtlist)
```

---

lines.fd *Add Lines from Functional Data to a Plot*

Description

Lines defined by functional observations are added to an existing plot.

Usage

```
## S3 method for class 'fd'
lines(x, Lfdobj=int2Lfd(0), nx=201, ...)
## S3 method for class 'fdSmooth'
lines(x, Lfdobj=int2Lfd(0), nx=201, ...)
```

Arguments

x	a univariate functional data object to be evaluated at nx points over xlim and added as a line to an existing plot.
Lfdobj	either a nonnegative integer or a linear differential operator object. If present, the derivative or the value of applying the operator is evaluated rather than the functions themselves.
nx	Number of points within xlim at which to evaluate x for plotting.
...	additional arguments such as axis titles and so forth that can be used in plotting programs called by lines.fd or lines.fdSmooth.

Side Effects

Lines added to an existing plot.

See Also

plot.fd, plotfit.fd

Examples

```
##
## plot a fit with 3 levels of smoothing
##
x <- seq(-1,1,0.02)
y <- x + 3*exp(-6*x^2) + sin(1:101)/2
# sin not rnorm to make it easier to compare
# results across platforms

result4.0  <- smooth.basisPar(argvals=x, y=y, lambda=1)
result4.m4 <- smooth.basisPar(argvals=x, y=y, lambda=1e-4)

plot(x, y)
lines(result4.0$fd)
lines(result4.m4$fd,      col='blue')
lines.fdSmooth(result4.0, col='red')

plot(x, y, xlim=c(0.5, 1))
lines.fdSmooth(result4.0)
lines.fdSmooth(result4.m4, col='blue')
# no visible difference from the default?
```

---

linmod *Fit Fully Functional Linear Model*

Description

A functional dependent variable $y_i(t)$ is approximated by a single functional covariate $x_i(s)$ plus an intercept function $\alpha(t)$, and the covariate can affect the dependent variable for all values of its argument. The equation for the model is

$$y_i(t) = \beta_0(t) + \int \beta_1(s,t)x_i(s)ds + e_i(t)$$

for $i = 1, ..., N$. The regression function $\beta_1(s,t)$ is a bivariate function. The final term $e_i(t)$ is a residual, lack of fit or error term. There is no need for values s and t to be on the same continuum.

Usage

```
linmod(xfdobj, yfdobj, betaList, wtvec=NULL)
```

Arguments

xfdobj	a functional data object for the covariate
yfdobj	a functional data object for the dependent variable
betaList	a list object of length 2. The first element is a functional parameter object specifying a basis and a roughness penalty for the intercept term. The second element is a bivariate functional parameter object for the bivariate regression function.
wtvec	a vector of weights for each observation. Its default value is NULL, in which case the weights are assumed to be 1.

Value

a named list of length 3 with the following entries:

beta0estfd	the intercept functional data object.
beta1estbifd	a bivariate functional data object for the regression function.
yhatfdobj	a functional data object for the approximation to the dependent variable defined by the linear model, if the dependent variable is functional. Otherwise the matrix of approximate values.

Source

Ramsay, James O., Hooker, Giles, and Graves, Spencer (2009) *Functional Data Analysis in R and Matlab*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York

See Also

bifdPar, fRegress

Examples

```
#See the prediction of precipitation using temperature as
#the independent variable in the analysis of the daily weather
#data, and the analysis of the Swedish mortality data.
```

`lip` *Lip motion*

Description

51 measurements of the position of the lower lip every 7 milliseconds for 20 repitions of the syllable 'bob'.

Usage

```
lip
lipmarks
liptime
```

Format

- lip a matrix of dimension c(51, 20) giving the position of the lower lip every 7 milliseconds for 350 milliseconds.
- lipmarks a matrix of dimension c(20, 2) giving the positions of the 'leftElbow' and 'rightElbow' in each of the 20 repetitions of the syllable 'bob'.
- liptime time in seconds from the start = seq(0, 0.35, 51) = every 7 milliseconds.

Details

These are rather simple data, involving the movement of the lower lip while saying "bob". There are 20 replications and 51 sampling points. The data are used to illustrate two techniques: landmark registration and principal differental analysis. Principal differential analysis estimates a linear differential equation that can be used to describe not only the observed curves, but also a certain number of their derivatives. For a rather more elaborate example of principal differential analysis, see the handwriting data.

See the `lip demo`.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York, sections 19.2 and 19.3.

Examples

```
#  See the lip demo.
```

matplot *Plot Columns of Matrices*

Description

Plot the columns of one matrix against the columns of another.

Usage

```
matplot(x, ...)
## Default S3 method:
matplot(x, y, type = "p", lty = 1:5, lwd = 1,
    lend = par("lend"), pch = NULL, col = 1:6, cex = NULL, bg = NA,
    xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL, ..., add = FALSE,
    verbose = getOption("verbose"))
## S3 method for class 'Date'
matplot(x, y, type = "p", lty = 1:5, lwd = 1,
    lend = par("lend"), pch = NULL, col = 1:6, cex = NULL, bg = NA,
    xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL, ..., add = FALSE,
    verbose = getOption("verbose"))
## S3 method for class 'POSIXct'
matplot(x, y, type = "p", lty = 1:5, lwd = 1,
    lend = par("lend"), pch = NULL, col = 1:6, cex = NULL, bg = NA,
    xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL, ..., add = FALSE,
    verbose = getOption("verbose"))
```

Arguments

x, y	vectors or matrices of data for plotting. The number of rows should match. If one of them are missing, the other is taken as 'y' and an 'x' vector of '1:n' is used. Missing values ('NA's) are allowed.
type	character string (length 1 vector) or vector of 1-character strings indicating the type of plot for each column of 'y', see 'plot' for all possible 'type's. The first character of 'type' defines the first plot, the second character the second, etc. Characters in 'type' are cycled through; e.g., '"pl"' alternately plots points and lines.
lty, lwd, lend	vector of line types, widths, and end styles. The first element is for the first column, the second element for the second column, etc., even if lines are not plotted for all columns. Line types will be used cyclically until all plots are drawn.
pch	character string or vector of 1-characters or integers for plotting characters, see 'points'. The first character is the plotting-character for the first plot, the second for the second, etc. The default is the digits (1 through 9, 0) then the lowercase and uppercase letters.
col	vector of colors. Colors are used cyclically.

cex	vector of character expansion sizes, used cyclically. This works as a multiple of 'par("cex")'. 'NULL' is equivalent to '1.0'.
bg	vector of background (fill) colors for the open plot symbols given by 'pch=21:25' as in 'points'. The default 'NA' corresponds to the one of the underlying function 'plot.xy'.
xlab, ylab	titles for x and y axes, as in 'plot'.
xlim, ylim	ranges of x and y axes, as in 'plot'.
...	Graphical parameters (see 'par') and any further arguments of 'plot', typically 'plot.default', may also be supplied as arguments to this function. Hence, the high-level graphics control arguments described under 'par' and the arguments to 'title' may be supplied to this function.
add	logical. If 'TRUE', plots are added to current one, using 'points' and 'lines'.
verbose	logical. If 'TRUE', write one line of what is done.

Details

Note that for multivariate data, a suitable array must first be defined using the `par` function.

`matplot.default` calls `matplot`. The other methods are needed, because the default methods ignore the `Date` or `POSIXct` character of `x`, labeling the horizontal axis as numbers, thereby placing it on the user to translate the numbers of days or seconds since the start of the epoch into dates (and possibly times for `POSIXct x`).

Side Effects

a plot of the functional observations

See Also

`matplot`, `plot`, `points`, `lines`, `matrix`, `par`

Examples

```
##
## matplot.Date, matplot.POSIXct
##
# Date
invasion1 <- as.Date('1775-09-04')
invasion2 <- as.Date('1812-07-12')
earlyUS.Canada <- c(invasion1, invasion2)
Y <- matrix(1:4, 2, 2)
matplot(earlyUS.Canada, Y)

# POSIXct
AmRev.ct <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
matplot(AmRev.ct, Y)

##
## matplot.default (copied from matplot{graphics})
##
```

```
matplot((-4:5)^2, main = "Quadratic") # almost identical to plot(*)
sines <- outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
matplot(sines, pch = 1:4, type = "o", col = rainbow(ncol(sines)))
matplot(sines, type = "b", pch = 21:23, col = 2:5, bg = 2:5,
            main = "matplot(...., pch = 21:23, bg = 2:5)")

x <- 0:50/50
matplot(x, outer(x, 1:8, function(x, k) sin(k*pi * x)),
            ylim = c(-2,2), type = "plobcsSh",
            main= "matplot(,type = \"plobcsSh\" )")
## pch & type =  vector of 1-chars :
matplot(x, outer(x, 1:4, function(x, k) sin(k*pi * x)),
            pch = letters[1:4], type = c("b","p","o"))

lends <- c("round","butt","square")
matplot(matrix(1:12, 4), type="c", lty=1, lwd=10, lend=lends)
text(cbind(2.5, 2*c(1,3,5)-.4), lends, col= 1:3, cex = 1.5)

table(iris$Species) # is data.frame with 'Species' factor
iS <- iris$Species == "setosa"
iV <- iris$Species == "versicolor"
op <- par(bg = "bisque")
matplot(c(1, 8), c(0, 4.5), type= "n", xlab = "Length", ylab = "Width",
            main = "Petal and Sepal Dimensions in Iris Blossoms")
matpoints(iris[iS,c(1,3)], iris[iS,c(2,4)], pch = "sS", col = c(2,4))
matpoints(iris[iV,c(1,3)], iris[iV,c(2,4)], pch = "vV", col = c(2,4))
legend(1, 4, c("    Setosa Petals", "    Setosa Sepals",
                    "Versicolor Petals", "Versicolor Sepals"),
           pch = "sSvV", col = rep(c(2,4), 2))

nam.var <- colnames(iris)[-5]
nam.spec <- as.character(iris[1+50*0:2, "Species"])
iris.S <- array(NA, dim = c(50,4,3),
                    dimnames = list(NULL, nam.var, nam.spec))
for(i in 1:3) iris.S[,,i] <- data.matrix(iris[1:50+50*(i-1), -5])

matplot(iris.S[,"Petal.Length",], iris.S[,"Petal.Width",], pch="SCV",
            col = rainbow(3, start = .8, end = .1),
            sub = paste(c("S", "C", "V"), dimnames(iris.S)[[3]],
                        sep = "=", collapse= ",  "),
            main = "Fisher's Iris Data")
par(op)
```

`mean.fd` *Mean of Functional Data*

Description

Evaluate the mean of a set of functions in a functional data object.

Usage

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

Arguments

x	a functional data object.
...	Other arguments to match the generic function for 'mean'

Value

a functional data object with a single replication that contains the mean of the functions in the object fd.

See Also

stddev.fd, var.fd, sum.fd, center.fd mean

Examples

```
##
## 1.  univeriate:  lip motion
##
liptime  <- seq(0,1,.02)
liprange <- c(0,1)

#  -------------  create the fd object -----------------
#       use 31 order 6 splines so we can look at acceleration

nbasis <- 51
norder <- 6
lipbasis <- create.bspline.basis(liprange, nbasis, norder)

#  ------------  apply some light smoothing to this object  -------

lipLfdobj   <- int2Lfd(4)
lipLambda   <- 1e-12
lipfdPar <- fdPar(lipbasis, lipLfdobj, lipLambda)

lipfd <- smooth.basis(liptime, lip, lipfdPar)$fd
names(lipfd$fdnames) = c("Normalized time", "Replications", "mm")

lipmeanfd <- mean.fd(lipfd)
plot(lipmeanfd)

##
## 2.  Trivariate:  CanadianWeather
##
dayrng <- c(0, 365)

nbasis <- 51
```

```
norder <- 6

weatherBasis <- create.fourier.basis(dayrng, nbasis)

weather.fd <- smooth.basis(day.5, CanadianWeather$dailyAv,
                           weatherBasis)$fd

str(weather.fd.mean <- mean.fd(weather.fd))
```

`melanoma` *melanoma 1936-1972*

Description

These data from the Connecticut Tumor Registry present age-adjusted numbers of melanoma skin-cancer incidences per 100,000 people in Connectict for the years from 1936 to 1972.

Format

A data frame with 37 observations on the following 2 variables.

year Years 1936 to 1972.

incidence Rate of melanoma cancer per 100,000 population.

Details

This is a copy of the 'melanoma' dataset in the 'lattice' package. It is unrelated to the object of the same name in the 'boot' package.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

See Also

`melanoma` `melanoma`

Examples

```
plot(melanoma[, -1], type="b")
```

monfn *Evaluates a monotone function*

Description

Evaluates a monotone function

Usage

```
monfn(argvals, Wfdobj, basislist=vector("list", JMAX), returnMatrix=FALSE)
```

Arguments

argvals	A numerical vector at which function and derivative are evaluated.
Wfdobj	A functional data object.
basislist	A list containing values of basis functions.
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Details

This function evaluates a strictly monotone function of the form

$$h(x) = [D^{-1}exp(Wfdobj)](x),$$

where D^{-1} means taking the indefinite integral. The interval over which the integration takes places is defined in the basis object in Wfdobj.

Value

A numerical vector or matrix containing the values the warping function h.

See Also

landmarkreg

Examples

```
## basically this example resembles part of landmarkreg.R that uses monfn.R to
## estimate the warping function.

## Specify the curve subject to be registered
n=21
tbreaks = seq(0, 2*pi, len=n)
xval <- sin(tbreaks)
rangeval <- range(tbreaks)
```

```
## Establish a B-spline basis for the curve
wbasis <- create.bspline.basis(rangeval=rangeval, breaks=tbreaks)
Wfd0   <- fd(matrix(0,wbasis$nbasis,1),wbasis)
WfdPar <- fdPar(Wfd0, 1, 1e-4)
fdObj  <- smooth.basis(tbreaks, xval, WfdPar)$fd

## Set the mean landmark times. Note that the objective of the warping
## function is to transform the curve such that the landmarks of the curve
## occur at the designated mean landmark times.

## Specify the mean landmark times: tbreak[8]=2.2 and tbreaks[13]=3.76
meanmarks <- c(rangeval[1], tbreaks[8], tbreaks[13], rangeval[2])
## Specify landmark locations of the curve: tbreaks[6] and tbreaks[16]
cmarks <- c(rangeval[1], tbreaks[6], tbreaks[16], rangeval[2])

## Establish a B-basis object for the warping function
Wfd = smooth.morph(x=meanmarks, y=cmarks, WfdPar=WfdPar)$Wfdobj

## Estimate the warping function
h = monfn(tbreaks, Wfd)

## scale using a linear equation h such that h(0)=0 and h(END)=END
b <- (rangeval[2]-rangeval[1])/ (h[n]-h[1])
a <- rangeval[1] - b*h[1]
h <- a + b*h
plot(tbreaks, h, xlab="Time", ylab="Transformed time", type="b")
```

`monomial` *Evaluate Monomial Basis*

Description

Computes the values of the powers of argument t.

Usage

```
monomial(evalarg, exponents=1, nderiv=0, argtrans=c(0,1))
```

Arguments

`evalarg`	a vector of argument values.
`exponents`	a vector of nonnegative integer values specifying the powers to be computed.
`nderiv`	a nonnegative integer specifying the order of derivative to be evaluated.
`argtrans`	Linearly transform an argument before constructing a basis. The first element is the shift in value and the second the scale factor.

Value

a matrix of values of basis functions. Rows correspond to argument values and columns to basis functions.

See Also

power, expon, fourier, polyg, bsplineS

Examples

```
# set up a monomial basis for the first five powers
nbasis   <- 5
basisobj <- create.monomial.basis(c(-1,1),nbasis)
#  evaluate the basis
tval <- seq(-1,1,0.1)
basismat <- monomial(tval, 1:basisobj$nbasis)
```

---

monomialpen *Evaluate Monomial Roughness Penalty Matrix*

Description

The roughness penalty matrix is the set of inner products of all pairs of a derivative of integer powers of the argument.

Usage

```
monomialpen(basisobj, Lfdobj=int2Lfd(2),
            rng=basisobj$rangeval)
```

Arguments

basisobj	a monomial basis object.
Lfdobj	either a nonnegative integer specifying an order of derivative or a linear differential operator object.
rng	the inner product may be computed over a range that is contained within the range defined in the basis object. This is a vector or length two defining the range.

Value

a symmetric matrix of order equal to the number of monomial basis functions.

See Also

exponpen, fourierpen, bsplinepen, polygpen

Examples

```
##
## set up a monomial basis for the first five powers
##
nbasis   <- 5
basisobj <- create.monomial.basis(c(-1,1),nbasis)
#  evaluate the rougness penalty matrix for the
#  second derivative.
penmat <- monomialpen(basisobj, 2)

##
## with rng of class Date and POSIXct
##
# Date
invasion1 <- as.Date('1775-09-04')
invasion2 <- as.Date('1812-07-12')
earlyUS.Canada <- c(invasion1, invasion2)
BspInvade1 <- create.monomial.basis(earlyUS.Canada)
invadmat <- monomialpen(BspInvade1)

# POSIXct
AmRev.ct <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
BspRev1.ct <- create.monomial.basis(AmRev.ct)
revmat <- monomialpen(BspRev1.ct)
```

MontrealTemp *Montreal Daily Temperature*

Description

Temperature in degrees Celsius in Montreal each day from 1961 through 1994

Usage

```
data(MontrealTemp)
```

Format

A numeric array with dimnames = list(1961:1994, names(dayOfYear)).

References

Ramsay, James O., Hooker, Giles, and Graves, Spencer B. (2009) *Functional Data Analysis with R and Matlab*, Springer, New York (esp. Section 4.3)

See Also

CanadianWeather monthAccessories

Examples

```
data(MontrealTemp)

JanuaryThaw <- t(MontrealTemp[, 16:47])
```

`nondurables` *Nondurable goods index*

Description

US nondurable goods index time series, January 1919 to January 2000.

Format

An object of class 'ts'.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York, ch. 3.

Examples

```
plot(nondurables, log="y")
```

`norder` *Order of a B-spline*

Description

norder = number of basis functions minus the number of interior knots.

Usage

```
norder(x, ...)
## S3 method for class 'fd'
norder(x, ...)
## S3 method for class 'basisfd'
norder(x, ...)
## Default S3 method:
norder(x, ...)
```

```
#norder.bspline(x, ...)

#NOTE:  The following is required by CRAN rules that
# function names like "as.numeric" must follow the documentation
# standards for S3 generics, even when they are not.
# Please ignore the following line:
## S3 method for class 'bspline'
norder(x, ...)
```

Arguments

x	Either a basisfd object or an object containing a basisfd object as a component.
...	optional arguments currently unused

Details

norder throws an error of basisfd[['type']] != 'bspline'.

Value

An integer giving the order of the B-spline.

Author(s)

Spencer Graves

See Also

create.bspline.basis

Examples

```
bspl1.1 <- create.bspline.basis(norder=1, breaks=0:1)

stopifnot(norder(bspl1.1)==1)

stopifnot(norder(fd(0, basisobj=bspl1.1))==1)

stopifnot(norder(fd(rep(0,4)))==4)

stopifnot(norder(list(fd(rep(0,4))))==4)
## Not run:
norder(list(list(fd(rep(0,4)))))
Error in norder.default(list(list(fd(rep(0, 4))))) :
  input is not a 'basisfd' object and does not have a 'basisfd'
component.

## End(Not run)

stopifnot(norder(create.bspline.basis(norder=1, breaks=c(0,.5, 1))) == 1)
```

```
stopifnot(norder(create.bspline.basis(norder=2, breaks=c(0,.5, 1))) == 2)

# Defaut B-spline basis:  Cubic spline:  degree 3, order 4,
# 21 breaks, 19 interior knots.
stopifnot(norder(create.bspline.basis()) == 4)

## Not run:
norder(create.fourier.basis(c(0,12) ))
Error in norder.bspline(x) :
  object x is of type = fourier;  'norder' is only defined for type = 'bsline'

## End(Not run)
```

`objAndNames` *Add names to an object*

Description

Add names to an object from 'preferred' if available and 'default' if not.

Usage

```
objAndNames(object, preferred, default)
```

Arguments

`object`	an object of some type to which names must be added. If length(dim(object))>0 add 'dimnames', else add 'names'.
`preferred`	A list to check first for names to add to 'object'.
`default`	A list to check for names to add to 'object' if appropriate names are not found in 'preferred'.

Details

1. If length(dim(object))<1, names(object) are taken from 'preferred' if they are not NULL and have the correct length, else try 'default'.

2. Else for(lvl in 1:length(dim(object))) take dimnames[[lvl]] from 'preferred[[i]]' if they are not NULL and have the correct length, else try 'default[[lvl]].

Value

An object of the same class and structure as 'object' but with either names or dimnames added or changed.

Author(s)

Spencer Graves

See Also

bifd

Examples

```
# The following should NOT check 'anything' here
tst1 <- objAndNames(1:2, list(letters[1:2], LETTERS[1:2]), anything)
all.equal(tst1, c(a=1, b=2))

# The following should return 'object unchanged
tst2 <- objAndNames(1:2, NULL, list(letters))
all.equal(tst2, 1:2)


tst3 <- objAndNames(1:2, list("a", 2), list(letters[1:2]))
all.equal(tst3, c(a=1, b=2) )

# The following checks a matrix / array
tst4 <- array(1:6, dim=c(2,3))
tst4a <- tst4
dimnames(tst4a) <- list(letters[1:2], LETTERS[1:3])
tst4b <- objAndNames(tst4,
       list(letters[1:2], LETTERS[1:3]), anything)
all.equal(tst4b, tst4a)

tst4c <- objAndNames(tst4, NULL,
       list(letters[1:2], LETTERS[1:3]) )
all.equal(tst4c, tst4a)
```

---

odesolv *Numerical Solution mth Order Differential Equation System*

Description

The system of differential equations is linear, with possibly time-varying coefficient functions. The numerical solution is computed with the Runge-Kutta method.

Usage

```
odesolv(bwtlist, ystart=diag(rep(1,norder)),
        h0=width/100, hmin=width*1e-10, hmax=width*0.5,
        EPS=1e-4, MAXSTP=1000)
```

Arguments

bwtlist	a list whose members are functional parameter objects defining the weight functions for the linear differential equation.

ystart	a vector of initial values for the equations. These are the values at time 0 of the solution and its first m - 1 derivatives.
h0	a positive initial step size.
hmin	the minimum allowable step size.
hmax	the maximum allowable step size.
EPS	a convergence criterion.
MAXSTP	the maximum number of steps allowed.

Details

This function is required to compute a set of solutions of an estimated linear differential equation in order compute a fit to the data that solves the equation. Such a fit will be a linear combinations of m independent solutions.

Value

a named list of length 2 containing

tp	a vector of time values at which the system is evaluated
yp	a matrix of variable values corresponding to tp.

See Also

pda.fd. For new applications, users are encouraged to consider deSolve. The deSolve package provides general solvers for ordinary and partial differential equations, as well as differential algebraic equations and delay differential equations.

Examples

```
#See the analyses of the lip data.
```

---

pca.fd *Functional Principal Components Analysis*

Description

Functional Principal components analysis aims to display types of variation across a sample of functions. Principal components analysis is an exploratory data analysis that tends to be an early part of many projects. These modes of variation are called $principal components$ or $harmonics.$ This function computes these harmonics, the eigenvalues that indicate how important each mode of variation, and harmonic scores for individual functions. If the functions are multivariate, these harmonics are combined into a composite function that summarizes joint variation among the several functions that make up a multivariate functional observation.

Usage

```
pca.fd(fdobj, nharm = 2, harmfdPar=fdPar(fdobj),
       centerfns = TRUE)
```

Arguments

fdobj	a functional data object.
nharm	the number of harmonics or principal components to compute.
harmfdPar	a functional parameter object that defines the harmonic or principal component functions to be estimated.
centerfns	a logical value: if TRUE, subtract the mean function from each function before computing principal components.

Value

an object of class "pca.fd" with these named entries:

harmonics	a functional data object for the harmonics or eigenfunctions
values	the complete set of eigenvalues
scores	s matrix of scores on the principal components or harmonics
varprop	a vector giving the proportion of variance explained by each eigenfunction
meanfd	a functional data object giving the mean function

See Also

cca.fd, pda.fd

Examples

```
#  carry out a PCA of temperature
#  penalize harmonic acceleration, use varimax rotation

daybasis65 <- create.fourier.basis(c(0, 365), nbasis=65, period=365)

harmaccelLfd <- vec2Lfd(c(0,(2*pi/365)^2,0), c(0, 365))
harmfdPar     <- fdPar(daybasis65, harmaccelLfd, lambda=1e5)
daytempfd <- smooth.basis(day.5, CanadianWeather$dailyAv[,,"Temperature.C"],
                  daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd

daytemppcaobj <- pca.fd(daytempfd, nharm=4, harmfdPar)
daytemppcaVarmx <- varmx.pca.fd(daytemppcaobj)
#  plot harmonics
op <- par(mfrow=c(2,2))
plot.pca.fd(daytemppcaobj, cex.main=0.9)

plot.pca.fd(daytemppcaVarmx, cex.main=0.9)
par(op)
```

```
plot(daytemppcaobj$harmonics)
plot(daytemppcaVarmx$harmonics)
```

---

pcaPACE *Estimate the functional principal components*

Description

Carries out a functional PCA with regularization from the estimate of the covariance surface

Usage

```
pcaPACE(covestimate, nharm, harmfdPar, cross)
```

Arguments

covestimate	a list with the two named entries "cov.estimate" and "meanfd"
nharm	the number of harmonics or principal components to compute.
harmfdPar	a functional parameter object that defines the harmonic or principal component functions to be estimated.
cross	a logical value: if TRUE, take into account the cross covariance for estimating the eigen functions.

Value

an object of class "pca.fd" with these named entries:

harmonics	a functional data object for the harmonics or eigenfunctions
values	the complete set of eigenvalues
scores	NULL. Use "scoresPACE" for estimating the pca scores
varprop	a vector giving the proportion of variance explained by each eigenfunction
meanfd	a functional data object giving the mean function

pda.fd *Principal Differential Analysis*

Description

Principal differential analysis (PDA) estimates a system of n linear differential equations that define functions that fit the data and their derivatives. There is an equation in the system for each variable.

Each equation has on its right side the highest order derivative that is used, and the order of this derivative, $m_j, j = 1, ..., n$ can vary over equations.

On the left side of equation is a linear combination of all the variables and all the derivatives of these variables up to order one less than the order m_j of the highest derivative.

In addition, the right side may contain linear combinations of forcing functions as well, with the number of forcing functions varying over equations.

The linear combinations are defined by weighting functions multiplying each variable, derivative, and forcing function in the equation. These weighting functions may be constant or vary over time. They are each represented by a functional parameter object, specifying a basis for an expansion of a coefficient, a linear differential operator for smoothing purposes, a smoothing parameter value, and a logical variable indicating whether the function is to be estimated, or kept fixed.

Usage

```
pda.fd(xfdlist, bwtlist=NULL,
       awtlist=NULL, ufdlist=NULL, nfine=501)
```

Arguments

xfdlist	a list whose members are functional data objects representing each variable in the system of differential equations. Each of these objects contain one or more curves to be represented by the corresponding differential equation. The length of the list is equal to the number of differential equations. The number N of replications must be the same for each member functional data object.
bwtlist	this argument contains the weight coefficients that multiply, in the right side of each equation, all the variables in the system, and all their derivatives, where the derivatives are used up to one less than the order of the variable. This argument has, in general, a three-level structure, defined by a three-level hierarchy of list objects. At the top level, the argument is a single list of length equal to the number of variables. Each component of this list is itself a list At the second level, each component of the top level list is itself a list, also of length equal to the number of variables. At the third and bottom level, each component of a second level list is a list of length equal to the number of orders of derivatives appearing on the right side of the equation, including the variable itself, a derivative of order 0. If m indicates the order of the equation, that is the order of the derivative on the left side, then this list is length m.

The components in the third level lists are functional parameter objects defining estimates for weight functions. For a first order equation, for example, $m = 1$ and the single component in each list contains a weight function for the variable. Since each equation has a term involving each variable in the system, a system of first order equations will have n^2 at the third level of this structure.

There MUST be a component for each weight function, even if the corresponding term does not appear in the equation. In the case of a missing term, the corresponding component can be NULL, and it will be treated as a coefficient fixed at 0.

However, in the case of a single differential equation, bwtlist can be given a simpler structure, since in this case only m coefficients are required. Therefore, for a single equation, bwtlist can be a list of length m with each component containing a functional parameter object for the corresponding derivative.

awtlist — a two-level list containing weight functions for forcing functions.

In addition to terms in each of the equations involving terms corresponding to each derivative of each variable in the system, each equation can also have a contribution from one or more exogenous variables, often called *forcing functions.*

This argument defines the weights multiplying these forcing functions, and is a list of length n, the number of variables. Each component of this is is in turn a list, each component of which contains a functional parameter object defining a weight function for a forcing function. If there are no forcing functions for an equation, this list can be NULL. If none of the equations involve forcing functions, awtlist can be NULL, which is its default value if it is not in the argument list.

ufdlist — a two-level list containing forcing functions. This list structure is identical to that for awtlist, the only difference being that the components in the second level contain functional data objects for the forcing functions, rather than functional parameter objects.

nfine — a number of values for a fine mesh. The estimation of the differential equation involves discrete numerical quadrature estimates of integrals, and these require that functions be evaluated at a fine mesh of values of the argument. This argument defines the number to use. The default value of 501 is reset to five times the largest number of basis functions used to represent any variable in the system, if this number is larger.

Value

an object of class pda.fd, being a list with the following components:

bwtlist — a list array of the same dimensions as the corresponding argument, containing the estimated or fixed weight functions defining the system of linear differential equations.

resfdlist — a list of length equal to the number of variables or equations. Each members is a functional data object giving the residual functions or forcing functions defined as the left side of the equation (the derivative of order m of a variable) minus the linear fit on the right side. The number of replicates for each residual functional data object is the same as that for the variables.

awtlist — a list of the same dimensions as the corresponding argument. Each member is an estimated or fixed weighting function for a forcing function.

See Also

pca.fd, cca.fd

Examples

```
#See analyses of daily weather data for examples.
##
##  set up objects for examples
##

#  set up basis objects
#  constant basis object  for estimating weight functions
cbasis = create.constant.basis(c(0,1))
#  monomial basis: {1,t}  for estimating weight functions
mbasis = create.monomial.basis(c(0,1),2)
#  quartic spline basis with 54 basis functions for
#    defining functions to be analyzed
xbasis = create.bspline.basis(c(0,1),24,5)
#  set up functional parameter objects for weight bases
cfd0   = fd(0,cbasis)
cfdPar = fdPar(cfd0)
mfd0   = fd(matrix(0,2,1),mbasis)
mfdPar = fdPar(mfd0)

#  fine mesh for plotting functions
#  sampling points over [0,1]
tvec = seq(0,1,len=101)

##
##  Example 1:  a single first order constant coefficient unforced equation
##     Dx = -4*x  for  x(t) = exp(-4t)

beta    = 4
xvec    = exp(-beta*tvec)
xfd     = smooth.basis(tvec, xvec, xbasis)$fd
xfdlist = list(xfd)
bwtlist = list(cfdPar)
#  perform the principal differential analysis
result = pda.fd(xfdlist, bwtlist)
#  display weight coefficient for variable
bwtlistout = result$bwtlist
bwtfd      = bwtlistout[[1]]$fd
par(mfrow=c(1,1))
plot(bwtfd)
title("Weight coefficient for variable")
print(round(bwtfd$coefs,3))
#  display residual functions
reslist    = result$resfdlist
plot(reslist[[1]])
```

```
title("Residual function")
##
##  Example 2:  a single first order varying coefficient unforced equation
##     Dx(t) = -t*x(t) or x(t) = exp(-t^2/2)
bvec    = tvec
xvec    = exp(-tvec^2/2)
xfd     = smooth.basis(tvec, xvec, xbasis)$fd
xfdlist = list(xfd)
bwtlist = list(mfdPar)
#  perform the principal differential analysis
result = pda.fd(xfdlist, bwtlist)
#  display weight coefficient for variable
bwtlistout = result$bwtlist
bwtfd      = bwtlistout[[1]]$fd
par(mfrow=c(1,1))
plot(bwtfd)
title("Weight coefficient for variable")
print(round(bwtfd$coefs,3))
#  display residual function
reslist    = result$resfdlist
plot(reslist[[1]])
title("Residual function")
##
##  Example 3:  a single second order constant coefficient unforced equation
##     Dx(t) = -(2*pi)^2*x(t) or x(t) = sin(2*pi*t)
##
xvec    = sin(2*pi*tvec)
xfd     = smooth.basis(tvec, xvec, xbasis)$fd
xfdlist = list(xfd)
bwtlist = list(cfdPar,cfdPar)
#  perform the principal differential analysis
result = pda.fd(xfdlist, bwtlist)
#  display weight coefficients
bwtlistout = result$bwtlist
bwtfd1     = bwtlistout[[1]]$fd
bwtfd2     = bwtlistout[[2]]$fd
par(mfrow=c(2,1))
plot(bwtfd1)
title("Weight coefficient for variable")
plot(bwtfd2)
title("Weight coefficient for derivative of variable")
print(round(c(bwtfd1$coefs, bwtfd2$coefs),3))
print(bwtfd2$coefs)
#  display residual function
reslist    = result$resfdlist
par(mfrow=c(1,1))
plot(reslist[[1]])
title("Residual function")
##
##  Example 4:  two first order constant coefficient unforced equations
##     Dx1(t) = x2(t) and Dx2(t) = -x1(t)
##   equivalent to  x1(t) = sin(2*pi*t)
##
```

```
xvec1     = sin(2*pi*tvec)
xvec2     = 2*pi*cos(2*pi*tvec)
xfd1      = smooth.basis(tvec, xvec1, xbasis)$fd
xfd2      = smooth.basis(tvec, xvec2, xbasis)$fd
xfdlist   = list(xfd1,xfd2)
bwtlist   = list(
                list(
                     list(cfdPar),
                     list(cfdPar)
                    ),
                list(
                     list(cfdPar),
                     list(cfdPar)
                    )
               )
#  perform the principal differential analysis
result = pda.fd(xfdlist, bwtlist)
#  display weight coefficients
bwtlistout = result$bwtlist
bwtfd11    = bwtlistout[[1]][[1]][[1]]$fd
bwtfd21    = bwtlistout[[2]][[1]][[1]]$fd
bwtfd12    = bwtlistout[[1]][[2]][[1]]$fd
bwtfd22    = bwtlistout[[2]][[2]][[1]]$fd
par(mfrow=c(2,2))
plot(bwtfd11)
title("Weight for variable 1 in equation 1")
plot(bwtfd21)
title("Weight for variable 2 in equation 1")
plot(bwtfd12)
title("Weight for variable 1 in equation 2")
plot(bwtfd22)
title("Weight for variable 2 in equation 2")
print(round(bwtfd11$coefs,3))
print(round(bwtfd21$coefs,3))
print(round(bwtfd12$coefs,3))
print(round(bwtfd22$coefs,3))
#  display residual functions
reslist = result$resfdlist
par(mfrow=c(2,1))
plot(reslist[[1]])
title("Residual function for variable 1")
plot(reslist[[2]])
title("Residual function for variable 2")
##
##  Example 5:  a single first order constant coefficient equation
##     Dx = -4*x  for  x(t) = exp(-4t) forced by u(t) = 2
##
beta    = 4
alpha   = 2
xvec0   = exp(-beta*tvec)
intv    = (exp(beta*tvec) - 1)/beta
xvec    = xvec0*(1 + alpha*intv)
xfd     = smooth.basis(tvec, xvec, xbasis)$fd
```

```
xfdlist = list(xfd)
bwtlist = list(cfdPar)
awtlist = list(cfdPar)
ufdlist = list(fd(1,cbasis))
#  perform the principal differential analysis
result = pda.fd(xfdlist, bwtlist, awtlist, ufdlist)
#  display weight coefficients
bwtlistout = result$bwtlist
bwtfd      = bwtlistout[[1]]$fd
awtlistout = result$awtlist
awtfd      = awtlistout[[1]]$fd
par(mfrow=c(2,1))
plot(bwtfd)
title("Weight for variable")
plot(awtfd)
title("Weight for forcing function")
#  display residual function
reslist = result$resfdlist
par(mfrow=c(1,1))
plot(reslist[[1]], ylab="residual")
title("Residual function")
##
##  Example 6:  two first order constant coefficient equations
##     Dx = -4*x    for  x(t) = exp(-4t)     forced by u(t) =  2
##     Dx = -4*t*x  for  x(t) = exp(-4t^2/2) forced by u(t) = -1
##
beta    = 4
xvec10  = exp(-beta*tvec)
alpha1  = 2
alpha2  = -1
xvec1   = xvec0 + alpha1*(1-xvec10)/beta
xvec20  = exp(-beta*tvec^2/2)
vvec    = exp(beta*tvec^2/2);
intv    = 0.01*(cumsum(vvec) - 0.5*vvec)
xvec2   = xvec20*(1 + alpha2*intv)
xfd1    = smooth.basis(tvec, xvec1, xbasis)$fd
xfd2    = smooth.basis(tvec, xvec2, xbasis)$fd
xfdlist = list(xfd1, xfd2)
bwtlist    = list(
                 list(
                      list(cfdPar),
                      list(cfdPar)
                     ),
                 list(
                      list(cfdPar),
                      list(mfdPar)
                     )
                )
awtlist = list(list(cfdPar), list(cfdPar))
ufdlist = list(list(fd(1,cbasis)), list(fd(1,cbasis)))

#  perform the principal differential analysis
result = pda.fd(xfdlist, bwtlist, awtlist, ufdlist)
```

```
# display weight functions for variables
bwtlistout = result$bwtlist
bwtfd11    = bwtlistout[[1]][[1]][[1]]$fd
bwtfd21    = bwtlistout[[2]][[1]][[1]]$fd
bwtfd12    = bwtlistout[[1]][[2]][[1]]$fd
bwtfd22    = bwtlistout[[2]][[2]][[1]]$fd
par(mfrow=c(2,2))
plot(bwtfd11)
title("weight on variable 1 in equation 1")
plot(bwtfd21)
title("weight on variable 2 in equation 1")
plot(bwtfd12)
title("weight on variable 1 in equation 2")
plot(bwtfd22)
title("weight on variable 2 in equation 2")
print(round(bwtfd11$coefs,3))
print(round(bwtfd21$coefs,3))
print(round(bwtfd12$coefs,3))
print(round(bwtfd22$coefs,3))
#  display weight functions for forcing functions
awtlistout = result$awtlist
awtfd1     = awtlistout[[1]][[1]]
awtfd2     = awtlistout[[2]][[1]]
par(mfrow=c(2,1))
plot(awtfd1)
title("weight on forcing function in equation 1")
plot(awtfd2)
title("weight on forcing function in equation 2")
#  display residual functions
reslist    = result$resfdlist
par(mfrow=c(2,1))
plot(reslist[[1]])
title("residual function for equation 1")
plot(reslist[[2]])
title("residual function for equation 2")
```

`pda.overlay` *Stability Analysis for Principle Differential Analysis*

Description

Overlays the results of a univariate, second-order principal differential analysis on a bifurcation diagram to demonstrate stability.

Usage

```
pda.overlay(pdaList,nfine=501,ncoarse=11,...)
```

Arguments

pdaList	a list object returned by pda.fd.
nfine	number of plotting points to use.
ncoarse	number of time markers to place along the plotted curve.
...	other arguments for 'plot'.

Details

Overlays a bivariate plot of the functional parameters in a univariate second-order principal differential analysis on a bifurcation diagram.

Value

None.

See Also

pda.fd plot.pda.fd eigen.pda

Examples

```
# This example looks at a principal differential analysis of the lip data
# in Ramsay and Silverman (2005).

# First smooth the data

lipfd <- smooth.basisPar(liptime, lip, 6, Lfdobj=int2Lfd(4),
                         lambda=1e-12)$fd
names(lipfd$fdnames) <- c("time(seconds)", "replications", "mm")

# Now we'll set up functional parameter objects for the beta coefficients.

lipbasis <- lipfd$basis
lipfd0   <- fd(matrix(0,lipbasis$nbasis,1),lipbasis)
lipfdPar <- fdPar(lipfd0,2,0)
bwtlist  <- list(lipfdPar,lipfdPar)
xfdlist  <- list(lipfd)

# Call pda

pdaList <- pda.fd(xfdlist, bwtlist)

# And plot the overlay

pda.overlay(pdaList,lwd=2,cex.lab=1.5,cex.axis=1.5)
```

phaseplanePlot *Phase-plane plot*

Description

Plot acceleration (or Ldfobj2) vs. velocity (or Lfdobj1) of a function data object.

Usage

```
phaseplanePlot(evalarg, fdobj, Lfdobj1=1, Lfdobj2=2,
        lty=c("longdash", "solid"),
      labels=list(evalarg=seq(evalarg[1], max(evalarg), length=13),
             labels=fda::monthLetters),
      abline=list(h=0, v=0, lty=2), xlab="Velocity",
      ylab="Acceleration", returnMatrix=FALSE, ...)
```

Arguments

evalarg	a vector of argument values at which the functional data object is to be evaluated.
	Defaults to a sequence of 181 points in the range specified by fdobj[["basis"]][["rangeval"]].
	If(length(evalarg) == 1)it is replaced by seq(evalarg[1], evalarg[1]+1, length=181).
	If(length(evalarg) == 2)it is replaced by seq(evalarg[1], evalarg[2], length=181).

fdobj	a functional data object to be evaluated.
Lfdobj1	either a nonnegative integer or a linear differential operator object. The points plotted on the horizontal axis are eval.fd(evalarg, fdobj, Lfdobj1). By default, this is the velocity.
Lfdobj2	either a nonnegative integer or a linear differential operator object. The points plotted on the vertical axis are eval.fd(evalarg, fdobj, Lfdobj2). By default, this is the acceleration.
lty	line types for the first and second halves of the plot.
labels	a list of length two: evalarg = a numeric vector of 'evalarg' values to be labeled. labels = a character vector of labels, replicated to the same length as labels[["evalarg"]] in case it's not of the same length.
abline	arguments to a call to abline.
xlab	x axis label
ylab	y axis label
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.
...	optional arguments passed to plot.

Value

Invisibly returns a matrix with two columns containg the points plotted.

See Also

plot, eval.fd plot.fd nondurables

Examples

```
goodsbasis <- create.bspline.basis(rangeval=c(1919,2000),
                                   nbasis=161, norder=8)
LfdobjNonDur <- int2Lfd(4)
argvals = seq(1919,2000,len=length(nondurables))
logNondurSm <- smooth.basisPar(argvals,
                y=log10(nondurables), fdobj=goodsbasis,
                Lfdobj=LfdobjNonDur, lambda=1e-11)
phaseplanePlot(1964, logNondurSm$fd)
```

---

pinch *pinch force data*

Description

151 measurements of pinch force during 20 replications with time from start of measurement.

Usage

```
pinch
pinchraw
pinchtime
```

Format

pinch, pinchraw Matrices of dimension c(151, 20) = 20 replications of measuring pinch force every 2 milliseconds for 300 milliseconds. The original data included 300 observations. pinchraw consists of the first 151 of the 300 observations. pinch selected 151 observations so the maximum of each curve occurred at 0.076 seconds.

pinchtime time in seconds from the start = seq(0, 0.3, 151) = every 2 milliseconds.

Details

Measurements every 2 milliseconds.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York, p. 13, Figure 1.11, pp. 22-23, Figure 2.2, and p. 144, Figure 7.13.

Examples

```
matplot (pinchtime, pinchraw, type="l", lty=1, cex=2,
         col=1, lwd=1,  xlab = "Seconds", ylab="Force (N)")
abline(h=2, lty=2)

matplot (pinchtime, pinch, type="l", lty=1, cex=2,
         col=1, lwd=1,  xlab = "Seconds", ylab="Force (N)")
abline(h=2, v=0.075, lty=2)
```

`plot.basisfd` *Plot a Basis Object*

Description

Plots all the basis functions.

Usage

```
## S3 method for class 'basisfd'
plot(x, knots=TRUE, axes=NULL, ...)
```

Arguments

`x`	a basis object
`knots`	logical: If TRUE and x[['type']] == 'bspline', the knot locations are plotted using vertical dotted, red lines. Ignored otherwise.
`axes`	Either a logical or a list or `NULL`. • logical whether axes should be drawn on the plot • list a list used to create custom `axes` used to create axes via `x$axes[[1]]` and `x$axes[-1]`. The primary example of this uses `list("axesIntervals",...)`, e.g., with `Fourier` bases to create `CanadianWeather` plots
`...`	additional plotting parameters passed to `matplot`.

Value

none

Side Effects

a plot of the basis functions

See Also

`plot.fd`

Examples

```
##
## 1.  b-spline
##
# set up the b-spline basis for the lip data, using 23 basis functions,
#   order 4 (cubic), and equally spaced knots.
#  There will be 23 - 4 = 19 interior knots at 0.05, ..., 0.95
lipbasis <- create.bspline.basis(c(0,1), 23)
# plot the basis functions
plot(lipbasis)

##
## 2.  Fourier basis
##
yearbasis3 <- create.fourier.basis(c(0,365),
                    axes=list("axesIntervals") )
#  plot the basis
plot(yearbasis3)

##
## 3.  With Date and POSIXct rangeval
##
# Date
July4.1776 <- as.Date('1776-07-04')
Apr30.1789 <- as.Date('1789-04-30')
AmRev <- c(July4.1776, Apr30.1789)
BspRevolution <- create.bspline.basis(AmRev)
plot(BspRevolution)

# POSIXct
July4.1776ct <- as.POSIXct1970('1776-07-04')
Apr30.1789ct <- as.POSIXct1970('1789-04-30')
AmRev.ct <- c(July4.1776ct, Apr30.1789ct)
BspRev.ct <- create.bspline.basis(AmRev.ct)
plot(BspRev.ct)
```

`plot.cca.fd` *Plot Functional Canonical Correlation Weight Functions*

Description

A canonical correlation analysis produces a series of pairs of functional data objects which, when used as weighting functions, successively maximize the corresponding canonical correlation between two functional data objects. Like functional principal component weight functions, successive weight within either side fo the pair are required to be orthogonal to all previous weight functions. Consequently, each successive canonical correlation will no larger than its predecessor, and more likely substantially smaller. This function plots an object of class `cca.fd` that results from the use of function `cca.fd`. Each pair of weight functions is plotted after a left mouse click indicating that you are ready for the next plot.

Usage

```
## S3 method for class 'cca.fd'
plot(x, cexval = 1, ...)
```

Arguments

x	an object of class cca.fd produced by an invocation of function cca.fd.R.
cexval	A number used to determine label sizes in the plots.
...	other arguments for 'plot'.

Details

Produces a plot of a pair of weight functions corresponding to each canonical correlation between two functional data objects.

Value

invisible(NULL)

See Also

cca.fd, pda.fd plot.pca.fd

Examples

```
#  Canonical correlation analysis of knee-hip curves

gaittime  <- (1:20)/21
gaitrange <- c(0,1)
gaitbasis <- create.fourier.basis(gaitrange,21)
lambda    <- 10^(-11.5)
harmaccelLfd <- vec2Lfd(c(0, 0, (2*pi)^2, 0))
gaitfdPar <- fdPar(gaitbasis, harmaccelLfd, lambda)
gaitfd    <- smooth.basis(gaittime, gait, gaitfdPar)$fd
ccafdPar  <- fdPar(gaitfd, harmaccelLfd, 1e-8)
ccafd0    <- cca.fd(gaitfd[,1], gaitfd[,2], ncan=3, ccafdPar, ccafdPar)
#  display the canonical correlations
round(ccafd0$ccacorr[1:6],3)
#  plot the unrotated canonical weight functions
plot.cca.fd(ccafd0)
#  compute a VARIMAX rotation of the canonical variables
ccafd1 <- varmx.cca.fd(ccafd0)
#  plot the rotated canonical weight functions
plot.cca.fd(ccafd1)
```

plot.fd *Plot a Functional Data Object*

Description

Functional data observations, or a derivative of them, are plotted. These may be either plotted simultaneously, as matplot does for multivariate data, or one by one with a mouse click to move from one plot to another. The function also accepts the other plot specification arguments that the regular plot does. Calling plot with an fdSmooth or an fdPar object plots its fd component.

Usage

```
## S3 method for class 'fd'
  plot(x, y, Lfdobj=0, href=TRUE, titles=NULL,
                    xlim=NULL, ylim=NULL, xlab=NULL,
                    ylab=NULL, ask=FALSE, nx=NULL, axes=NULL, ...)
## S3 method for class 'fdPar'
plot(x, y, Lfdobj=0, href=TRUE, titles=NULL,
                    xlim=NULL, ylim=NULL, xlab=NULL,
                    ylab=NULL, ask=FALSE, nx=NULL, axes=NULL, ...)
## S3 method for class 'fdSmooth'
plot(x, y, Lfdobj=0, href=TRUE, titles=NULL,
                    xlim=NULL, ylim=NULL, xlab=NULL,
                    ylab=NULL, ask=FALSE, nx=NULL, axes=NULL, ...)
```

Arguments

x	functional data object(s) to be plotted.
y	sequence of points at which to evaluate the functions 'x' and plot on the horizontal axis. Defaults to seq(rangex[1], rangex[2], length = nx). NOTE: This will be the values on the horizontal axis, NOT the vertical axis.
Lfdobj	either a nonnegative integer or a linear differential operator object. If present, the derivative or the value of applying the operator is plotted rather than the functions themselves.
href	a logical variable: If TRUE, add a horizontal reference line at 0.
titles	a vector of strings for identifying curves
xlab	a label for the horizontal axis.
ylab	a label for the vertical axis.
xlim	a vector of length 2 containing axis limits for the horizontal axis.
ylim	a vector of length 2 containing axis limits for the vertical axis.
ask	a logical value: If TRUE, each curve is shown separately, and the plot advances with a mouse click
nx	the number of points to use to define the plot. The default is usually enough, but for a highly variable function more may be required.

axes — Either a logical or a list or NULL.

- logical whether axes should be drawn on the plot
- list a list used to create custom axes used to create axes via x$axes[[1]] and x$axes[-1]. The primary example of this uses list("axesIntervals",...), e.g., with Fourier bases to create CanadianWeather plots

... — additional plotting arguments that can be used with function plot

Details

Note that for multivariate data, a suitable array must first be defined using the par function.

Value

'done'

Side Effects

a plot of the functional observations

See Also

lines.fd, plotfit.fd

Examples

```
##
## plot.fd
##

daybasis65 <- create.fourier.basis(c(0, 365), 65,
                    axes=list("axesIntervals"))
harmaccelLfd <- vec2Lfd(c(0,(2*pi/365)^2,0), c(0, 365))

harmfdPar     <- fdPar(daybasis65, harmaccelLfd, lambda=1e5)

daytempfd <- with(CanadianWeather, smooth.basis(day.5,
        dailyAv[,,"Temperature.C"], daybasis65)$fd)

#  plot all the temperature functions for the monthly weather data
plot(daytempfd, main="Temperature Functions")

## Not run:
# To plot one at a time:
# The following pauses to request page changes.
\dontshow{
# (Without 'dontrun', the package build process
# might encounter problems with the par(ask=TRUE)
# feature.)
}
plot(daytempfd, ask=TRUE)
```

```
## End(Not run)

##
## plot.fdSmooth
##
b3.4 <- create.bspline.basis(norder=3, breaks=c(0, .5, 1))
# 4 bases, order 3 = degree 2 =
# continuous, bounded, locally quadratic
fdPar3 <- fdPar(b3.4, lambda=1)



# Penalize excessive slope Lfdobj=1;
# (Can not smooth on second derivative Lfdobj=2 at it is discontinuous.)
fd3.4s0 <- smooth.basis(0:1, 0:1, fdPar3)

# using plot.fd directly
plot(fd3.4s0$fd)




##
## with Date and POSIXct argvals
##
# Date
invasion1 <- as.Date('1775-09-04')
invasion2 <- as.Date('1812-07-12')
earlyUS.Canada <- as.numeric(c(invasion1, invasion2))
BspInvasion    <- create.bspline.basis(earlyUS.Canada)

earlyUSyears <- seq(invasion1, invasion2, length.out=7)
earlyUScubic <- (as.numeric(earlyUSyears-invasion1)/365.24)^3
earlyUSyears <- as.numeric(earlyUSyears)
fitCubic     <- smooth.basis(earlyUSyears, earlyUScubic, BspInvasion)$fd
plot(fitCubic)

# POSIXct
AmRev.ct    <- as.POSIXct1970(c('1776-07-04', '1789-04-30'))
AmRevYrs.ct <- seq(AmRev.ct[1], AmRev.ct[2], length.out=14)
AmRevLin.ct <- as.numeric(AmRevYrs.ct-AmRev.ct[2])
AmRevYrs.ct <- as.numeric(AmRevYrs.ct)
BspRev.ct   <- create.bspline.basis(AmRev.ct)
fitLin.ct   <- smooth.basis(AmRevYrs.ct, AmRevLin.ct, BspRev.ct)$fd
plot(fitLin.ct)
```

---

plot.Lfd *Plot a Linear Differential Operator Object*

Description

Plot the coefficents of the terms of order 0 through m-1 of an object of class Lfd and length m.

Usage

```
## S3 method for class 'Lfd'
plot(x, axes=NULL, ...)
```

Arguments

x	a linear differential operator object to be plotted.
axes	Either a logical or a list or NULL passed to plot.fd. • logical whether axes should be drawn on the plot • list a list used to create custom axes used to create axes via x$axes[[1]] and x$axes[-1]. The primary example of this uses list("axesIntervals",...), e.g., with Fourier bases to create CanadianWeather plots
...	additional plotting arguments that can be used with function plot

Value

invisible(NULL)

Side Effects

a plot of the linear differential operator object.

See Also

Lfd, plot.fd

Examples

```
#  Set up the harmonic acceleration operator
dayrange  <- c(0,365)
Lbasis  <- create.constant.basis(dayrange,
                 axes=list("axesIntervals"))
Lcoef   <- matrix(c(0,(2*pi/365)^2,0),1,3)
bfdobj  <- fd(Lcoef,Lbasis)
bwtlist <- fd2list(bfdobj)
harmaccelLfd <- Lfd(3, bwtlist)
plot(harmaccelLfd)
```

plot.pca.fd *Plot Functional Principal Components*

Description

Display the types of variation across a sample of functions. Label with the eigenvalues that indicate the relative importance of each mode of variation.

Usage

```
#plot.pca.fd(x, nx = 128, pointplot = TRUE, harm = 0,
#                        expand = 0, cycle = FALSE, ...)

#NOTE:  The following is required by CRAN rules that
# function names like "as.numeric" must follow the documentation
# standards for S3 generics, even when they are not.
# Please ignore the following line:
## S3 method for class 'pca.fd'
plot(x, nx = 128, pointplot = TRUE, harm = 0,
                       expand = 0, cycle = FALSE, ...)
```

Arguments

x	a functional data object.
nx	Number of points to plot or vector (if length > 1) to use as evalarg in evaluating and plotting the functional principal components.
pointplot	logical: If TRUE, the harmonics / principal components are plotted as '+' and '-'. Otherwise lines are used.
harm	Harmonics / principal components to plot. If 0, plot all. If length(harm) > sum(par("mfrow")), the user advised, "Waiting to confirm page change..." and / or 'Click or hit ENTER for next page' for each page after the first.
expand	nonnegative real: If expand == 0 then effect of +/- 2 standard deviations of each pc are given otherwise the factor expand is used.
cycle	logical: If cycle=TRUE and there are 2 variables then a cycle plot will be drawn If the number of variables is anything else, cycle will be ignored.
...	other arguments for 'plot'.

Details

Produces one plot for each principal component / harmonic to be plotted.

Value

invisible(NULL)

See Also

cca.fd, pda.fd plot.pca.fd

Examples

```
#  carry out a PCA of temperature
#  penalize harmonic acceleration, use varimax rotation

daybasis65 <- create.fourier.basis(c(0, 365), nbasis=65, period=365)

harmaccelLfd <- vec2Lfd(c(0,(2*pi/365)^2,0), c(0, 365))
harmfdPar     <- fdPar(daybasis65, harmaccelLfd, lambda=1e5)
daytempfd <- smooth.basis(day.5, CanadianWeather$dailyAv[,,"Temperature.C"],
                     daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd

daytemppcaobj <- pca.fd(daytempfd, nharm=4, harmfdPar)
#  plot harmonics, asking before each new page after the first:
plot.pca.fd(daytemppcaobj)

# plot 4 on 1 page
op <- par(mfrow=c(2,2))
plot.pca.fd(daytemppcaobj, cex.main=0.9)
par(op)
```

---

plot.pda.fd *Plot Principle Differential Analysis Components*

Description

Plots the results of pda.fd, allows the user to group coefficient functions by variable, equation, derivative or combination of them.

Usage

```
## S3 method for class 'pda.fd'
plot(x,whichdim=1,npts=501,...)
```

Arguments

x	an object of class pda.fd.
whichdim	which dimension to use as grouping variables • 1 coefficients of each variable differential equation • 2 coefficient functions for each equation • 3 coefficients of derivatives of each variable

	whichdim should be an ordered vector of length between 1 and 3.
npts	number of points to use for plotting.
...	other arguments for 'plot'.

Details

Produces one plot for each coefficient function in a principle differential analysis.

Value

invisible(NULL)

See Also

pda.fd eigen.pda

Examples

```
#  A pda analysis of the handwriting data

# reduce the size to reduce the compute time for the example
ni <- 281
indx <- seq(1, 1401, length=ni)
fdaarray <- handwrit[indx,,]
fdatime  <- seq(0, 2.3, len=ni)

#  basis for coordinates

fdarange <- c(0, 2.3)
breaks   <- seq(0,2.3,length.out=116)
norder   <- 6
fdabasis <- create.bspline.basis(fdarange,norder=norder,breaks=breaks)

#  parameter object for coordinates

fdafd0 <- fd(matrix(0,fdabasis$nbasis,1), fdabasis)
fdaPar <- fdPar(fdafd0,int2Lfd(4),1e-8)

#  coordinate functions and a list tontaining them

Xfd <- smooth.basis(fdatime, fdaarray[,,1], fdaPar)$fd
Yfd <- smooth.basis(fdatime, fdaarray[,,2], fdaPar)$fd

xfdlist <- list(Xfd, Yfd)

#  basis and parameter object for weight functions

fdabasis2 <- create.bspline.basis(fdarange,norder=norder,nbasis=31)
fdafd0    <- fd(matrix(0,fdabasis2$nbasis,1), fdabasis2)
pdaPar    <- fdPar(fdafd0,1,1e-8)
```

```
pdaParlist <- list(pdaPar, pdaPar)

bwtlist <- list( list(pdaParlist,pdaParlist), list(pdaParlist,pdaParlist) )

#  do the second order pda

pdaList <- pda.fd(xfdlist, bwtlist)

# plot the results

plot(pdaList,whichdim=1)
plot(pdaList,whichdim=2)
plot(pdaList,whichdim=3)

plot(pdaList,whichdim=c(1,2))
plot(pdaList,whichdim=c(1,3))
plot(pdaList,whichdim=c(2,3))

plot(pdaList,whichdim=1:3)
```

`plotbeta` *Plot a functional parameter object with confidence limits*

Description

Plot a functional parameter object with confidence limits

Usage

```
plotbeta(betaestlist, betastderrlist, argvals=NULL, xlab="", ...)
```

Arguments

`betaestlist`	a list containing one or more functional parameter objects (class = fdPar) or functional data objects (class = fd).
`betastderrlist`	a list containing functional data objects for the standard errors of the objects in `betaestlist`.
`argvals`	a sequence of values at which to evaluate `betaestlist` and `betastderrlist`.
`xlab`	x axis label
`...`	additional plotting parameters passed to `plot`.

Value

none

Side Effects

a plot of the basis functions

See Also

plot.fd

---

plotfit *Plot a Functional Data Object With Data*

Description

Plot either functional data observations 'x' with a fit 'fdobj' or residuals from the fit.

This function is useful for assessing how well a functional data object fits the actual discrete data.

The default is to make one plot per functional observation with fit if residual is FALSE and superimposed lines if residual==TRUE.

With multiple plots, the system waits to confirm a desire to move to the next page unless ask==FALSE.

Usage

```
plotfit.fd(y, argvals, fdobj, rng = NULL, index = NULL,
      nfine = 101, residual = FALSE, sortwrd = FALSE, titles=NULL,
      ylim=NULL, ask=TRUE, type=c("p", "l")[1+residual],
      xlab=NULL, ylab=NULL, sub=NULL, col=1:9, lty=1:9, lwd=1,
      cex.pch=1, axes=NULL, ...)
plotfit.fdSmooth(y, argvals, fdSm, rng = NULL, index = NULL,
      nfine = 101, residual = FALSE, sortwrd = FALSE, titles=NULL,
      ylim=NULL, ask=TRUE, type=c("p", "l")[1+residual],
      xlab=NULL, ylab=NULL, sub=NULL, col=1:9, lty=1:9, lwd=1,
      cex.pch=1, axes=NULL, ...)
```

Arguments

y	a vector, matrix or array containing the discrete observations used to estimate the functional data object.
argvals	a vector containing the argument values corresponding to the first dimension of y.
fdobj	a functional data object estimated from the data.
fdSm	an object of class fdSmooth
rng	a vector of length 2 specifying the limits for the horizontal axis. This must be a subset of fdobj[['basis']][['rangeval']], which is the default.
index	a set of indices of functions if only a subset of the observations are to be plotted. Subsetting can also be achieved by subsetting y; see details, below.
nfine	the number of argument values used to define the plot of the functional data object. This may need to be increased if the functions have a great deal of fine detail.
residual	a logical variable: if TRUE, the residuals are plotted rather than the data and functional data object.

sortwrd
: a logical variable: if TRUE, the observations (i.e., second dimension of y) are sorted for plotting by the size of the sum of squared residuals.

titles
: a vector containing strings that are titles for each observation.

ylim
: a numeric vector of length 2 giving the y axis limits; see 'par'.

ask
: If TRUE and if 'y' has more levels than the max length of col, lty, lwd and cex.pch, the user must confirm page change before the next plot will be created.

type
: type of plot desired, as described with plot. If residual == FALSE, 'type' controls the representation for 'x', which will typically be 'p' to plot points but not lines; 'fdobj' will always plot as a line. If residual == TRUE, the default type == "l"; an alternative is "b" for both.

xlab
: x axis label.

ylab
: Character vector of y axis labels. If(residual), ylab defaults to 'Residuals', else to varnames derived from names(fdnames[[3]] or fdnames[[3]] or dimnames(y)[[3]].

sub
: subtitle under the x axis label. Defaults to the RMS residual from the smooth.

col, lty, lwd, cex.pch
: Numeric or character vectors specifying the color (col), line type (lty), line width (lwd) and size of plotted character symbols (cex.pch) of the data representation on the plot.

  If ask==TRUE, the length of the longest of these determines the number of levels of the array 'x' in each plot before asking the user to acknowledge a desire to change to the next page. Each of these is replicated to that length, so col[i] is used for x[,i] (if x is 2 dimensional), with line type and width controlled by lty[i] and lwd[i], respectively.

  If ask==FALSE, these are all replicated to length = the number of plots to be superimposed.

  For more information on alternative values for these paramters, see 'col', 'lty', 'lwd', or 'cex' with par.

axes
: Either a logical or a list or NULL.

  - logical whether axes should be drawn on the plot
  - list a list used to create custom axes used to create axes via x$axes[[1]] and x$axes[-1]). The primary example of this uses list("axesIntervals",...), e.g., with Fourier bases to create CanadianWeather plots

...
: additional arguments such as axis labels that may be used with other plot functions.

Details

plotfit plots discrete data along with a functional data object for fitting the data. It is designed to be used after something like smooth.fd, smooth.basis or smooth.fdPar to check the fit of the data offered by the fd object.

plotfit.fdSmooth calls plotfit for its 'fd' component.

The plot can be restricted to a subset of observations (i.e., second dimension of y) or variables (i.e., third dimension of y) by providing y with the dimnames for its second and third dimensions

matching a subset of the dimnames of fdobj[['coef']] (for `plotfit.fd` or fdSm[['fdobj']][['coef']] for `plotfit.fdSmooth`). If only one observation or variable is to be plotted, y must include 'drop = TRUE', as, e.g., y[, 2, 3, drop=TRUE]. If y or fdobj[['coef']] does not have dimnames on its second or third dimension, subsetting is achieved by taking the first few colums so the second or third dimensions match. This is achieved using checkDims3(y, fdobj[['coef']], defaultNames = fdobj[['fdnames']]]).

Value

A matrix of mean square deviations from predicted is returned invisibly. If fdobj[["coefs"]] is a 3-dimensional array, this is a matrix of dimensions equal to the last two dimensions of fdobj[["coefs"]]. This will typically be the case when x is also a 3-dimensional array with the last two dimensions matching those of fdobj[["coefs"]]. The second dimension is typically replications and the third different variables.

If x and fobj[["coefs"]] are vectors or 2-dimensional arrays, they are padded to three dimensions, and then MSE is computed as a matrix with the second dimension = 1; if x and fobj[["coefs"]] are vectors, the first dimension of the returned matrix will also be 1.

Side Effects

a plot of the the data 'x' with the function or the deviations between observed and predicted, depending on whether residual is FALSE or TRUE.

See Also

`plot`, `plot.fd`, `lines.fd`, `plot.fdSmooth`, `lines.fdSmooth`, `par`, `smooth.fd`, `smooth.basis`, `smooth.basisPar`, `checkDims3`

Examples

```
#  set up a Fourier basis for smoothing temperature data
daybasis65 <- create.fourier.basis(c(0, 365), 65,
                   axes=list("axesIntervals"))
#  smooth the average temperature data using function smooth.basis
Daytempfd <- with(CanadianWeather, smooth.basis(day.5,
         dailyAv[,,"Temperature.C"], daybasis65)$fd )
daytempfd <- with(CanadianWeather, smooth.basis(day.5,
         dailyAv[,,"Temperature.C"],
         daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd )
#  Plot the temperature data along with the fit to the data for the first
#  station, St. John's Newfoundland
# If you want only the fitted functions, use plot(daytempfd)
# To plot only a single fit vs. observations, use argument index
# to request which one you want.
with(CanadianWeather, plotfit.fd(dailyAv[, , "Temperature.C",
     drop=FALSE], argvals= day.5, daytempfd, index=1, titles=place) )
# Default ylab = daytempfd[['fdnames']]

with(CanadianWeather, plotfit.fd(dailyAv[, , "Temperature.C"],
     argvals= day.5, Daytempfd, index=1, titles=place) )
```

```
plot(daytempfd)

## Not run:
# plot all the weather stations, one by one after a click on the plot
# in response to a request.
# This example is within the "dontrun" environment to prevent the
# the R package checking process from pausing: without 'dontrun', the package
# build process might encounter problems with the par(ask=TRUE) feature.
with(CanadianWeather, plotfit.fd(dailyAv[,, "Temperature.C"], day.5,
     daytempfd, ask=TRUE) )

## End(Not run)
#  Now plot results for two weather stations.
op <- par(mfrow=c(2,1), xpd=NA, bty="n")
# xpd=NA:  clip lines to the device region,
#       not the plot or figure region
# bty="n":  Do not draw boxes around the plots.
ylim <- range(CanadianWeather$dailyAv[,,"Temperature.C"])
# Force the two plots to have the same scale
with(CanadianWeather, plotfit.fd(dailyAv[,,"Temperature.C"], day.5,
          daytempfd, index=2, titles=place, ylim=ylim) )
with(CanadianWeather, plotfit.fd(dailyAv[,,"Temperature.C"], day.5,
          daytempfd, index=35, titles=place, ylim=ylim) )
## Not run:
# Plot residuals with interactive display of stations one by one
par(op)
with(CanadianWeather, plotfit.fd(dailyAv[, , "Temperature.C"],
          day.5, daytempfd, residual=TRUE) )

## End(Not run)
#  The gait data are bivariate, and this code illustrates how plotfit.fd
#  deals with the plotting of two variables at the same time
#  First define normalized times and their range
gaittime  <- (0:19) + 0.5
gaitrange <- c(0,20)
#  Define the harmonic acceleration differential operator
harmaccelLfd <- vec2Lfd(c(0, (2*pi/20)^2, 0), rangeval=gaitrange)
#  Set up basis for representing gait data.
gaitbasis <- create.fourier.basis(gaitrange, nbasis=21)
#  Smooth the data
gaitfd <- smooth.basisPar(gaittime, gait, gaitbasis,
                          Lfdobj=harmaccelLfd, lambda=1e-2)$fd
#  Assign names to the data
names(gaitfd$fdnames) <- c("Normalized time", "Child", "Angle")
gaitfd$fdnames[[3]] <- c("Hip", "Knee")
## Not run:
#  plot each pair of curves interactively, two plots per page, the top
#  for hip angle, and the bottom for knee angle
plotfit.fd(gait, gaittime, gaitfd)
#  Plot the residuals, sorting cases by residual sum of squares summed over
#  both hip and knee angles.
#  The first series of 39 plots are for hip angle, two plots per page,
#  and the second 39 are for knee angle.  The plots are sorted by the
```

```
#  size of the total residual sum of squares, but RMS residual values
#  for specific angles are not all going to be in order.
plotfit.fd(gait, gaittime, gaitfd, residual=TRUE, sort=TRUE)

## End(Not run)
```

---

plotreg.fd *Plot the results of the registration of a set of curves*

Description

A function is said to be aligned or registered with a target function if its salient features, such as peaks, valleys and crossings of fixed thresholds, occur at about the same argument values as those of the target. Function `plotreg.fd` plots for each curve that is registered (1) the unregistered curve (blue dashed line), (2) the target curve (red dashed line) and (3) the registered curve (blue solid line). It also plots within the same graphics window the warping function $h(t)$ along with a dashed diagonal line as a comparison.

Usage

```
plotreg.fd(reglist)
```

Arguments

`reglist`	a named list that is output by a call to function `register.fd`. The members of `reglist` that are required are: `regfd` ... the registered functions, `Wfd` ... the functions W(t) defining the warping functions h(t), `yfd` ... the unregistered functions, and `y0fd` ... the target functions. If required objects are missing, REGLIST was probably generated by an older verson of REGISTER.FD, and the registration should be redone.

Value

a series of plots, each containing two side-by-side panels. Clicking on the R Graphics window advances to the next plot.

References

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York, ch. 6 & 7.

See Also

`smooth.monotone`, `smooth.morph` `register.fd`

Examples

```
## Not run:
#  register and plot the angular acceleration of the gait data
gaittime  <- seq(0.05, 0.95, 0.1)*20
gaitrange <- c(0,20)
#  set up a fourier basis object
gaitbasis <- create.fourier.basis(gaitrange, nbasis=21)
#  set up a functional parameter object penalizing harmonic acceleration
harmaccelLfd <- vec2Lfd(c(0, (2*pi/20)^2, 0), rangeval=gaitrange)
gaitfdPar    <- fdPar(gaitbasis, harmaccelLfd, 1e-2)
#  smooth the data
gaitfd <- smooth.basis(gaittime, gait, gaitfdPar)$fd
#  compute the angular acceleration functional data object
D2gaitfd      <- deriv.fd(gaitfd,2)
names(D2gaitfd$fdnames)[[3]] <- "Angular acceleration"
D2gaitfd$fdnames[[3]] <- c("Hip", "Knee")
#  compute the mean angular acceleration functional data object
D2gaitmeanfd  <- mean.fd(D2gaitfd)
names(D2gaitmeanfd$fdnames)[[3]] <- "Mean angular acceleration"
D2gaitmeanfd$fdnames[[3]] <- c("Hip", "Knee")
#  register the functions for the first 10 boys
#  argument periodic = TRUE causes register.fd to estimate a horizontal shift
#  for each curve, which is a possibility when the data are periodic
nBoys <- 2 # use only 2 boys to save test time.
#  set up the basis for the warping functions
nwbasis   <- 7
wbasis    <- create.bspline.basis(gaitrange,nwbasis,3)
Warpfd    <- fd(matrix(0,nwbasis,nBoys),wbasis)
WarpfdPar <- fdPar(Warpfd)
#  carry out the continuous registration
gaitreglist <- register.fd(D2gaitmeanfd, D2gaitfd[1:nBoys], WarpfdPar,
                           iterlim=4, periodic=TRUE)
# set iterlim=4 to reduce the compute time;
# this argument may not be needed in many applications.
#  plot the results
plotreg.fd(gaitreglist)
#  display horizonal shift values
print(round(gaitreglist$shift,1))

## End(Not run)
```

---

plotscores *Plot Principal Component Scores*

Description

The coefficients multiplying the harmonics or principal component functions are plotted as points.

Usage

```
plotscores(pcafd, scores=c(1, 2), xlab=NULL, ylab=NULL,
           loc=1, matplt2=FALSE, ...)
```

Arguments

pcafd	an object of the "pca.fd" class that is output by function pca.fd.
scores	the indices of the harmonics for which coefficients are plotted.
xlab	a label for the horizontal axis.
ylab	a label for the vertical axis.
loc	an integer: if loc >0, you can then click on the plot in loc places and you'll get plots of the functions with these values of the principal component coefficients.
matplt2	a logical value: if TRUE, the curves are plotted on the same plot; otherwise, they are plotted separately.
...	additional plotting arguments used in function plot.

Side Effects

a plot of scores

See Also

pca.fd

---

polyg *Polygonal Basis Function Values*

Description

Evaluates a set of polygonal basis functions, or a derivative of these functions, at a set of arguments.

Usage

```
polyg(x, argvals, nderiv=0)
```

Arguments

x	a vector of argument values at which the polygonal basis functions are to evaluated.
argvals	a strictly increasing set of argument values containing the range of x within it that defines the polygonal basis. The default is x itself.
nderiv	the order of derivative to be evaluated. The derivative must not exceed one. The default derivative is 0, meaning that the basis functions themselves are evaluated.

Value

a matrix of function values. The number of rows equals the number of arguments, and the number of columns equals the number of basis

See Also

create.polygonal.basis, polygpen

Examples

```
#  set up a set of 21 argument values
x <- seq(0,1,0.05)
#  set up a set of 11 argument values
argvals <- seq(0,1,0.1)
#  with the default period (1) and derivative (0)
basismat <- polyg(x, argvals)
#  plot the basis functions
matplot(x, basismat, type="l")
```

---

polygpen *Polygonal Penalty Matrix*

Description

Computes the matrix defining the roughness penalty for functions expressed in terms of a polygonal basis.

Usage

```
polygpen(basisobj, Lfdobj=int2Lfd(1))
```

Arguments

basisobj	a polygonal functional basis object.
Lfdobj	either an integer that is either 0 or 1, or a linear differential operator object of degree 0 or 1.

Details

a roughness penalty for a function $ x(t) $ is defined by integrating the square of either the derivative of $ x(t) $ or, more generally, the result of applying a linear differential operator $ L $ to it. The only roughness penalty possible aside from penalizing the size of the function itself is the integral of the square of the first derivative, and this is the default. To apply this roughness penalty, the matrix of inner products produced by this function is necessary.

Value

a symmetric matrix of order equal to the number of basis functions defined by the polygonal basis object. Each element is the inner product of two polygonal basis functions after applying the derivative or linear differential operator defined by Lfdobj.

See Also

create.polygonal.basis, polyg

Examples

```
#  set up a sequence of 11 argument values
argvals <- seq(0,1,0.1)
#  set up the polygonal basis
basisobj <- create.polygonal.basis(argvals)
#  compute the 11 by 11 penalty matrix

penmat <- polygpen(basisobj)
```

---

powerbasis *Power Basis Function Values*

Description

Evaluates a set of power basis functions, or a derivative of these functions, at a set of arguments.

Usage

```
powerbasis(x, exponents, nderiv=0)
```

Arguments

x	a vector of argument values at which the power basis functions are to evaluated. Since exponents may be negative, for example after differentiation, it is required that all argument values be positive.
exponents	a vector of exponents defining the power basis functions. If y is such a rate value, the corresponding basis function is x to the power y. The number of basis functions is equal to the number of exponents.
nderiv	the derivative to be evaluated. The derivative must not exceed the order. The default derivative is 0, meaning that the basis functions themselves are evaluated.

Value

a matrix of function values. The number of rows equals the number of arguments, and the number of columns equals the number of basis functions.

See Also

create.power.basis, powerpen

Examples

```
#  set up a set of 10 positive argument values.
x <- seq(0.1,1,0.1)
#  compute values for three power basis functions
exponents <- c(0, 1, 2)
#  evaluate the basis matrix
basismat <- powerbasis(x, exponents)
```

---

powerpen *Power Penalty Matrix*

Description

Computes the matrix defining the roughness penalty for functions expressed in terms of a power basis.

Usage

```
powerpen(basisobj, Lfdobj=int2Lfd(2))
```

Arguments

basisobj	a power basis object.
Lfdobj	either a nonnegative integer or a linear differential operator object.

Details

A roughness penalty for a function $ x(t) $ is defined by integrating the square of either the derivative of $ x(t) $ or, more generally, the result of applying a linear differential operator $ L $ to it. The most common roughness penalty is the integral of the square of the second derivative, and this is the default. To apply this roughness penalty, the matrix of inner products produced by this function is necessary.

Value

a symmetric matrix of order equal to the number of basis functions defined by the power basis object. Each element is the inner product of two power basis functions after applying the derivative or linear differential operator defined by Lfdobj.

See Also

create.power.basis, powerbasis

Examples

```
#  set up an power basis with 3 basis functions.
#  the powers are 0, 1, and 2.
basisobj <- create.power.basis(c(0,1),3,c(0,1,2))
#  compute the 3 by 3 matrix of inner products of second derivatives
#FIXME
#penmat <- powerpen(basisobj, 2)
```

---

ppBspline *Convert a B-spline function to piece-wise polynomial form*

Description

The B-spline basis functions of order `k = length(t) -1` defined by the knot sequence in argument `t` each consist of polynomial segments with the same order joined end-to-end over the successive gaps in the knot sequence. This function computes the `k` coefficients of these polynomial segments in the rows of the output matrix `coeff`, with each row corresponding to a B-spline basis function that is positive over the interval spanned by the values in `t`. The elements of the output vector `index` indicate where in the sequence `t` we find the knots. Note that we assume `t[1] < t[k+1]`, i.e. `t` is not a sequence of the same knot.

Usage

```
ppBspline(t)
```

Arguments

`t`	numeric vector = knot sequence of length norder+1 where norder = the order of the B-spline. The knot sequence must contain at least one gap.

Value

a list object containing components

`Coeff`	a matrix with rows corresponding to B-spline basis functions positive over the interval spanned by `t` and columns corresponding to the terms `1,x,x^2,...` in the polynomial representation.
`index`	indices indicating where in the sequence `t` the knots are to be found

See Also

`bsplineS`

Examples

```
ppBspline(1:5)
```

predict.fRegress *Predict method for Functional Regression*

Description

Model predictions for object of class fRegress.

Usage

```
## S3 method for class 'fRegress'
predict(object, newdata=NULL, se.fit = FALSE,
     interval = c("none", "confidence", "prediction"),
     level = 0.95, ...)
```

Arguments

object	Object of class inheriting from fRegress
newdata	Either NULL or a list matching object\$xfdlist. If(is.null(newdata)) predictions <- object\$yhatfdobj If newdata is a list, predictions = the sum of either newdata[i] * betaestfdlist[i] if object\$yfdobj has class fd or inprod(newdata[i], betaestfdlist[i]) if class(object\$yfdobj) = numeric.
se.fit	a switch indicating if standard errors of predictions are required NOTE: se.fit = TRUE is NOT IMPLEMENTED YET.
interval	type of prediction (response or model term) NOTE: Only "intervale = 'none'" has been implemented so far.
level	Tolerance/confidence level
...	additional arguments for other methods

Details

1. Without newdata, fit <- object\$yhatfdobj.

2. With newdata, if(class(object\$y) == 'numeric'), fit <- sum over i of inprod(betaestlist[i], newdata[i]). Otherwise, fit <- sum over i of betaestlist[i] * newdata[i].

3. If(se.fit | (interval != 'none')) compute se.fit, then return whatever is desired.

Value

The predictions produced by predict.fRegress are either a vector or a functional parameter (class fdPar) object, matching the class of object$y.

If interval is not "none", the predictions will be multivariate for object$y and the requested lwr and upr bounds. If object$y is a scalar, these predictions are returned as a matrix; otherwise, they are a multivariate functional parameter object (class fdPar).

If se.fit is TRUE, predict.fRegress returns a list with the following components:

fit	vector or matrix or univariate or multivariate functional parameter object depending on the value of interval and the class of object$y.
se.fit	standard error of predicted means

Author(s)

Spencer Graves

See Also

fRegress predict

Examples

```
##
## vector response with functional explanatory variable
##

annualprec <- log10(apply(CanadianWeather$dailyAv[,,
                  "Precipitation.mm"], 2,sum))
smallbasis  <- create.fourier.basis(c(0, 365), 25)
tempfd <- smooth.basis(day.5,
   CanadianWeather$dailyAv[,,"Temperature.C"], smallbasis)$fd
precip.Temp.f <- fRegress(annualprec ~ tempfd)

precip.Temp.p <- predict(precip.Temp.f)

# plot response vs. fitted
plot(annualprec, precip.Temp.p)
```

---

project.basis *Approximate Functional Data Using a Basis*

Description

A vector or matrix of discrete data is projected into the space spanned by the values of a set of basis functions. This amounts to a least squares regression of the data on to the values of the basis functions. A small penalty can be applied to deal with situations in which the number of basis functions exceeds the number of basis points. This function is not normally used in a functional data analysis to smooth data, since function smooth.basis is provided for that job.

Usage

```
project.basis(y, argvals, basisobj, penalize=FALSE)
```

Arguments

y	a vector or matrix of discrete data.
argvals	a vector containing the argument values correspond to the values in y.
basisobj	a basis object.
penalize	a logical variable. If TRUE, a small roughness penalty is applied to ensure that the linear equations defining the least squares solution are linearly independent or nonsingular.

Value

the matrix of coefficients defining the least squares approximation. This matrix has as many rows are there are basis functions, as many columns as there are curves, and if the data are multivariate, as many layers as there are functions.

See Also

smooth.basis

---

quadset *Quadrature points and weights for Simpson's rule*

Description

Set up quadrature points and weights for Simpson's rule.

Usage

```
quadset(nquad=5, basisobj=NULL, breaks)
```

Arguments

nquad	an odd integer at least 5 giving the number of evenly spaced Simpson's rule quadrature points to use over each interval (breaks[i], breaks[i+1]).
basisobj	the basis object that will contain the quadrature points and weights
breaks	optional interval boundaries. If this is provided, the first value must be the initial point of the interval over which the basis is defined, and the final value must be the end point. If this is not supplied, and 'basisobj' is of type 'bspline', the knots are used as these values.

Details

Set up quadrature points and weights for Simpson's rule and store information in the output 'basisobj'. Simpson's rule is used to integrate a function between successive values in vector 'breaks'. That is, over each interval (breaks[i], breaks[i+1]). Simpson's rule uses 'nquad' equally spaced quadrature points over this interval, starting with the the left boundary and ending with the right boundary. The quadrature weights are the values delta*c(1,4,2,4,2,4,..., 2,4,1) where 'delta' is the difference between successive quadrature points, that is, delta = (breaks[i-1]-breaks[i])/(nquad-1).

Value

If is.null(basisobj), quadset returns a 'quadvals' matrix with columns quadpts and quadwts. Otherwise, it returns basisobj with the two components set as follows:

quadvals	cbind(quadpts=quadpts, quadwts=quadwts)
value	a list with two components containing eval.basis(quadpts, basisobj, ival-1) for ival=1, 2.

See Also

create.bspline.basis eval.basis

Examples

```
(qs7.1 <- quadset(nquad=7, breaks=c(0, .3, 1)))
# cbind(quadpts=c(seq(0, 0.3, length=7),
#              seq(0.3, 1, length=7)),
#    quadwts=c((0.3/18)*c(1, 4, 2, 4, 2, 4, 1),
#              (0.7/18)*c(1, 4, 2, 4, 2, 4, 1) ) )

# The simplest basis currently available with this function:
bspl2.2 <- create.bspline.basis(norder=2, breaks=c(0,.5, 1))

bspl2.2a <- quadset(basisobj=bspl2.2)
bspl2.2a$quadvals
# cbind(quadpts=c((0:4)/8, .5+(0:4)/8),
#       quadwts=rep(c(1,4,2,4,1)/24, 2) )
bspl2.2a$values
# a list of length 2
# [[1]] = matrix of dimension c(10, 3) with the 3 basis
#      functions evaluated at the 10 quadrature points:
# values[[1]][, 1] = c(1, .75, .5, .25, rep(0, 6))
# values[[1]][, 2] = c(0, .25, .5, .75, 1, .75, .5, .25, 0)
# values[[1]][, 3] = values[10:1, 1]
#
# values[[2]] = matrix of dimension c(10, 3) with the
#     first derivative of values[[1]], being either
#    -2, 0, or 2.
```

---

reconsCurves *Reconstruct data curves using functional principal components*

Description

Reconstructs data curves as objects of class fd using functional principal components

Usage

```
reconsCurves(data, PC)
```

Arguments

data	a set of values of curves at discrete sampling points or argument values. If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If data is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications.
PC	an object of class pca.fd.

Value

an object of class fd.

ReginaPrecip *Regina Daily Precipitation*

Description

Temperature in millimeters in June in Regina, Saskatchewan, Canada, 1960 - 1993, omitting 16 missing values.

Usage

```
data(ReginaPrecip)
```

Format

A numeric vector of length 1006.

References

Ramsay, James O., Hooker, Giles, and Graves, Spencer B. (2009) *Functional Data Analysis with R and Matlab*, Springer, New York (esp. Section 5.4.3)

See Also

CanadianWeather MontrealTemp

Examples

```
data(ReginaPrecip)
hist(ReginaPrecip)
```

register.fd *Register Functional Data Objects Using a Continuous Criterion*

Description

A function is said to be aligned or registered with a target function if its salient features, such as peaks, valleys and crossings of fixed thresholds, occur at about the same argument values as those of the target. Function register.fd aligns these features by transforming or warping the argument domain of each function in a nonlinear but strictly order-preserving fashion. Multivariate functions may also be registered. If the domain is time, we say that this transformation transforms clock time to system time. The transformation itself is called a warping function.

Usage

```
register.fd(y0fd=NULL, yfd=NULL, WfdParobj=NULL,
            conv=1e-04, iterlim=20, dbglev=1, periodic=FALSE, crit=2)
```

Arguments

y0fd	a functional data object defining one or more target functions for registering the functions in argument yfd. If the functions to be registered are univariate, then y0fd may contain only a single function, or it may contain as many functions as are in yfd. If yfd contains multivariate functions, then y0fd may either as many functions as there are variables in yfd, or it may contain as many functions as are in yfd and these functions must then be multivariate and be of the same dimension as those in yfd. If yfd is supplied as a named argument and y0fd is not, then y0fd is computed inside the function to be the mean of the functions in yfd. If the function is called with a single unnamed argument, and there is no other argument that is named as y0fd then this unnamed argument is taken to be actually yfd rather than y0fd, and then also y0fd is computed to be the man of the functions supplied.
yfd	a functional data object defining the functions to be registered to target y0fd. The functions may be either univariate or multivariate. If yfd contains a single multivariate function is to be registered, it essential that the coefficient array for y0fd have class array, have three dimensions, and that its second dimension be of length 1.
WfdParobj	a functional parameter object containing either a single function or the same number of functions as are contained in yfd. The coefficients supply the initial values in the estimation of a functions $W(t)$ that defines the warping functions $h(t)$ that register the set of curves. WfdParobj also defines the roughness penalty and smoothing parameter used to control the roughness of $h(t)$. The basis used for this object must be a B-spline type, and the order of the B-spline basis must be at least 2 (piecewise linear). If WFDPAROBJ is not supplied, it is constructed from a bspline basis of order 2 with 2 basis functions; that is, a basis for piecewise linear functions. The smoothing parameter lambda for this default is 0.

conv	a criterion for convergence of the iterations.
iterlim	a limit on the number of iterations.
dbglev	either 0, 1, or 2. This controls the amount information printed out on each iteration, with 0 implying no output, 1 intermediate output level, and 2 full output. R normally postpones displaying these results until the entire computation is computed, an option that it calls "output buffering." Since the total computation time may be considerable, one may opt for turning this feature off by un-checking this box in the "Misc" menu item in the R Console.
periodic	a logical variable: if TRUE, the functions are considered to be periodic, in which case a constant can be added to all argument values after they are warped.
crit	an integer that is either 1 or 2 that indicates the nature of the continuous registration criterion that is used. If 1, the criterion is least squares, and if 2, the criterion is the minimum eigenvalue of a cross-product matrix. In general, criterion 2 is to be preferred.

Details

The warping function that smoothly and monotonically transforms the argument is defined by Wfd is the same as that defines the monotone smoothing function in for function smooth.monotone. See the help file for that function for further details.

Value

a named list of length 4 containing the following components:

regfd	A functional data object containing the registered functions.
warpfd	A functional data object containing the warping functions $h(t)$.
Wfd	A functional data object containing the functions $h W(t)$ that define the warping functions $h(t)$.
shift	If the functions are periodic, this is a vector of time shifts.
y0fd	The target function object y0fd.
yfd	The function object yfd containing the functions to be registered.

Source

Ramsay, James O., and Silverman, Bernard W. (2005), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York, ch. 6 & 7.

See Also

smooth.monotone, smooth.morph plotreg.fd register.newfd

Examples

```
#See the analyses of the growth data for examples.

##
## 1.  smooth the growth data for the Berkeley boys
##
# Specify smoothing weight
lambda.gr2.3 <- .03
# Specify what to smooth, namely the rate of change of curvature
Lfdobj.growth    <- 2
# Set up a B-spline basis for smoothing the discrete data
nage <- length(growth$age)
norder.growth <- 6
nbasis.growth <- nage + norder.growth - 2
rng.growth <- range(growth$age)
wbasis.growth <- create.bspline.basis(rangeval=rng.growth,
                   nbasis=nbasis.growth, norder=norder.growth,
                   breaks=growth$age)
# Smooth the data
# in afda-ch06.R, and register to individual smooths:
cvec0.growth <- matrix(0,nbasis.growth,1)
Wfd0.growth  <- fd(cvec0.growth, wbasis.growth)
growfdPar2.3 <- fdPar(Wfd0.growth, Lfdobj.growth, lambda.gr2.3)
hgtmfd.all   <- with(growth, smooth.basis(age, hgtm, growfdPar2.3)$fd)
# Register the growth velocity rather than the
# growth curves directly
smBv <- deriv.fd(hgtmfd.all, 1)

##
## 2.  Register the first 2 Berkeley boys using the default basis
##     for the warping function
##
# register.fd takes time, so we use only 2 curves as an illustration
# to minimize computing time in these examples
nBoys <- 2
#  Define the target function as the mean of the first nBoys records
smBv0 = mean.fd(smBv[1:nBoys])
#  Register these curves.  The default choice for the functional
#  parameter object WfdParObj is used.
smB.reg.0 <- register.fd(smBv0, smBv[1:nBoys])
#  plot each curve.  Click on the R Graphics window to show each plot.
#  The left panel contains:
#    -- the unregistered curve (dashed blue line)
#    -- the target function (dashed red line)
#    -- the registered curve (solid blue line)
#  The right panel contains:
#    -- the warping function h(t)
#    -- the linear function corresponding to no warping
plotreg.fd(smB.reg.0)
#  Notice that all the warping functions all have simple shapes
#  due to the use of the simplest possible basis
```

```
if (!CRAN()) {
##
## 3.  Define a more flexible basis for the warping functions
##
Wnbasis   <- 4
Wbasis    <- create.bspline.basis(rng.growth, Wnbasis)
Wfd0      <- fd(matrix(0,Wnbasis,1),Wbasis)
#  set up the functional parameter object using only
#      a light amount smoothing
WfdParobj <- fdPar(Wfd0, Lfdobj=2, lambda=0.01)
#  register the curves
smB.reg.1 <- register.fd(smBv0, smBv[1:nBoys], WfdParobj)
plotreg.fd(smB.reg.1)
#  Notice that now the warping functions can have more complex shapes

##
## 4.  Change the target to the mean of the registered functions ...
##     this should provide a better target for registration
##
smBv1 <- mean.fd(smB.reg.1$regfd)
#  plot the old and the new targets
par(mfrow=c(1,1),ask=FALSE)
plot(smBv1)
lines(smBv0, lty=2)
#  Notice how the new target (solid line) has sharper features and
#  a stronger pubertal growth spurt relative to the old target
#  (dashed line).  Now register to the new target
smB.reg.2 <- register.fd(smBv1, smBv[1:nBoys], WfdParobj)
plotreg.fd(smB.reg.2)
#  Plot the mean of these curves as well as the first and second targets
par(mfrow=c(1,1),ask=FALSE)
plot(mean.fd(smB.reg.2$regfd))
lines(smBv0, lty=2)
lines(smBv1, lty=3)
#  Notice that there is almost no improvement over the age of the
#  pubertal growth spurt, but some further detail added in the
#  pre-pubertal region.  Now register the previously registered
#  functions to the new target.
smB.reg.3 <- register.fd(smBv1, smB.reg.1$regfd, WfdParobj)
plotreg.fd(smB.reg.3)
#  Notice that the warping functions only deviate from the straight line
#  over the pre-pubertal region, and that there are some small adjustments
#  to the registered curves as well over the pre-pubertal region.
}

##
## 5.  register and plot the angular acceleration of the gait data
##
gaittime  <- as.matrix(0:19)+0.5
gaitrange <- c(0,20)
#  set up a fourier basis object
gaitbasis <- create.fourier.basis(gaitrange, nbasis=21)
# set up a functional parameter object penalizing harmonic acceleration
```

```
harmaccelLfd <- vec2Lfd(c(0, (2*pi/20)^2, 0), rangeval=gaitrange)
gaitfdPar    <- fdPar(gaitbasis, harmaccelLfd, 1e-2)
#  smooth the data
gaitfd <- smooth.basis(gaittime, gait, gaitfdPar)$fd
#  compute the angular acceleration functional data object
D2gaitfd    <- deriv.fd(gaitfd,2)
names(D2gaitfd$fdnames)[[3]] <- "Angular acceleration"
D2gaitfd$fdnames[[3]] <- c("Hip", "Knee")
#  compute the mean angular acceleration functional data object
D2gaitmeanfd  <- mean.fd(D2gaitfd)
names(D2gaitmeanfd$fdnames)[[3]] <- "Mean angular acceleration"
D2gaitmeanfd$fdnames[[3]] <- c("Hip", "Knee")
#  register the functions for the first 2 boys
#  argument periodic = TRUE causes register.fd to estimate a horizontal shift
#  for each curve, which is a possibility when the data are periodic
#  set up the basis for the warping functions
nwbasis   <- 4
wbasis    <- create.bspline.basis(gaitrange,nwbasis,3)
Warpfd    <- fd(matrix(0,nwbasis,nBoys),wbasis)
WarpfdPar <- fdPar(Warpfd)
#  register the functions
gaitreglist <- register.fd(D2gaitmeanfd, D2gaitfd[1:nBoys], WarpfdPar,
                           periodic=TRUE)
#  plot the results
plotreg.fd(gaitreglist)
#  display horizonal shift values
print(round(gaitreglist$shift,1))
```

`register.newfd` *Register Functional Data Objects with Pre-Computed Warping Functions*

Description

This function registers a new functional data object to pre-computed warping functions.

Usage

```
register.newfd(yfd, Wfd,type=c('direct','monotone','periodic'))
```

Arguments

`yfd`	a multivariate functional data object defining the functions to be registered with `Wfd`.
`Wfd`	a functional data object defining the registration functions to be used to register `yfd`. This can be the result of either `landmarkreg` or `register.fd`.
`type`	indicates the type of registration function.

- direct assumes `Wfd` is a direct definition of the registration functions. This is produced by `landmarkreg`.

- monotone assumes that `Wfd` defines a monotone functional data objected, up to shifting and scaling to make endpoints agree. This is produced by `register.fd`.
- periodic does shift registration for periodic functions. This is output from `register.fd` if `periodic=TRUE`.

Details

Only shift registration is considered for the periodic case.

Value

a functional data object defining the registered curves.

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York, ch. 6 & 7.

See Also

landmarkreg, register.fd

Examples

```
# Register the lip data with landmark registration, then register the first
# derivatives with the pre-computed warping functions.

# Lip data:
lipfd <- smooth.basisPar(liptime, lip, 6, Lfdobj=int2Lfd(4),
                         lambda=1e-12)$fd
names(lipfd$fdnames) <- c("time(seconds)", "replications", "mm")

# Landmark Registration:

lipmeanmarks <- apply(lipmarks,2,mean)

wnbasis <- 6
wnorder <- 4
wbreaks <- c(0,lipmeanmarks,0.35)

warpbasis <- create.bspline.basis(nbasis=wnbasis, norder=wnorder,
                                  breaks=wbreaks);
WfdPar    <- fdPar(fd(basisobj=warpbasis), 2, 1e-4)

lipreglist <- landmarkreg(lipfd, as.matrix(lipmarks), lipmeanmarks, WfdPar)

# And warp:
```

```
Dlipfd = deriv.fd(lipfd,Lfdobj=1)

Dlipregfd = register.newfd(Dlipfd,lipreglist$warpfd,type='direct')
```

---

scoresPACE *Estimates of functional Principal Component scores through PACE*

Description

Function scoresPACE estimates the functional Principal Component scores through Conditional Expectation (PACE)

Usage

```
scoresPACE(data, time, covestimate, PC)
```

Arguments

data	a matrix object or list – If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If y is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. – If it is a list, each element must be a matrix object, the rows correspond to argument values per individual. First column corresponds to time points and following columns to argument values per variable.
time	Array with time points where data was taken. length(time) == dim(data)[1]
covestimate	a list with the two named entries "cov.estimate" and "meanfd"
PC	an object of class "pca.fd"

Value

a matrix of scores with dimension nrow = nharm and ncol = ncol(data)

---

sd.fd *Standard Deviation of Functional Data*

Description

Evaluate the standard deviation of a set of functions in a functional data object.

Usage

```
sd.fd(fdobj)
std.fd(fdobj)
stdev.fd(fdobj)
stddev.fd(fdobj)
```

Arguments

fdobj	a functional data object.

Details

The multiple aliases are provided for compatibility with previous versions and with other languages. The name for the standard deviation function in R is 'sd'. Matlab uses 'std'. S-Plus and Microsoft Excal use 'stdev'. 'stddev' was used in a previous version of the 'fda' package and is retained for compatibility.

Value

a functional data object with a single replication that contains the standard deviation of the one or several functions in the object fdobj.

See Also

mean.fd, sum.fd, center.fd

Examples

```
liptime  <- seq(0,1,.02)
liprange <- c(0,1)

#  -------------  create the fd object -----------------
#       use 31 order 6 splines so we can look at acceleration

nbasis <- 51
norder <- 6
lipbasis <- create.bspline.basis(liprange, nbasis, norder)
lipbasis <- create.bspline.basis(liprange, nbasis, norder)

#  ------------  apply some light smoothing to this object  -------

Lfdobj   <- int2Lfd(4)
lambda   <- 1e-12
lipfdPar <- fdPar(lipbasis, Lfdobj, lambda)

lipfd <- smooth.basis(liptime, lip, lipfdPar)$fd
names(lipfd$fdnames) = c("Normalized time", "Replications", "mm")

lipstdfd <- sd.fd(lipfd)
plot(lipstdfd)
```

```
all.equal(lipstdfd, std.fd(lipfd))
all.equal(lipstdfd, stdev.fd(lipfd))
all.equal(lipstdfd, stddev.fd(lipfd))
```

`seabird` *Sea Bird Counts*

Description

Numbers of sightings of different species of seabirds by year 1986 - 2005 at E. Sitkalidak, Uganik, Uyak, and W. Sitkalidak by people affiliated with the Kodiak National Wildlife Refuge, Alaska.

Usage

```
data(seabird)
```

Format

A data frame with 3793 observations on the following 22 variables.

BAGO, BLSC, COME, COMU, CORM, HADU, HOGR, LOON, MAMU, OLDS, PIGU, RBME, RNGR, SUSC, WW integer count of the numbers of sightings of each species by transect by year

Year integer year, 1986 - 2005

Site Integer codes for `Bay`: 10 = Uyak, 20 = Uganik, 60 = E. Sitkalidak, 70 = W. Sitkalidak

Transect Integer code (101 - 749) for the specific plot of ground observed

Temp a numeric vector

ObservCond a factor with levels `Average`, `Excellent`, `Fair`, `Good`, and `Ideal`.

Bay a factor with levels `E. Sitkalidak` `Uganik` `Uyak` `W. Sitkalidak`

ObservCondFactor3 a factor with levels `ExcellentIdeal`, `FairAverage`, and `Good`. These combine levels from `ObservCond`.

Details

Data provided by the Kodiak National Wildlife Refuge

Source

Zwiefelhofer, D., Reynolds, J. H., and Keim, M. (2008) Population trends and annual density estimates for select wintering seabird species on Kodiak Island, Alaska, *U.S. Fish and Wildlife Service, Kodiak National Wildlife Refuge*, Technical Report, no. 08-00x

References

Ramsay, James O., Hooker, Giles, and Graves, Spencer B. (2009) *Functional Data Analysis with R and Matlab*, Springer, New York (esp. ch. 10)

Examples

```
data(seabird)
## maybe str(seabird) ; plot(seabird) ...
```

`smooth.basis` *Construct a functional data object by smoothing data using a roughness penalty*

Description

Discrete observations on one or more curves and for one more more variables are fit with a set of smooth curves, each defined by an expansion in terms of user-selected basis functions. The fitting criterion is weighted least squares, and smoothness can be defined in terms of a roughness penalty that is specified in a variety of ways.

Data smoothing requires at a bare minimum three elements: (1) a set of observed noisy values, (b) a set of argument values associated with these data, and (c) a specification of the basis function system used to define the curves. Typical basis functions systems are splines for nonperiodic curves, and fourier series for periodic curves.

Optionally, a set covariates may be also used to take account of various non-smooth contributions to the data. Smoothing without covariates is often called nonparametric regression, and with covariates is termed semiparametric regression.

Usage

```
smooth.basis(argvals=1:n, y, fdParobj, wtvec=NULL, fdnames=NULL,
             covariates=NULL, method="chol", dfscale=1, returnMatrix=FALSE)
```

Arguments

`argvals`	a set of argument values corresponding to the observations in array `y`. In most applications these values will be common to all curves and all variables, and therefore be defined as a vector or as a matrix with a single column. But it is possible that these argument values will vary from one curve to another, and in this case `argvals` will be input as a matrix with rows corresponding to observation points and columns corresponding to curves. Argument values can even vary from one variable to another, in which case they are input as an array with dimensions corresponding to observation points, curves and variables, respectively. Note, however, that the number of observation points per curve and per variable may NOT vary. If it does, then curves and variables must be smoothed individually rather than by a single call to this function. The default value for `argvals` are the integers 1 to `n`, where `n` is the size of the first dimension in argument `y`.
`y`	an set of values of curves at discrete sampling points or argument values. If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only

one variable per observation. If y is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. If y is a vector, only one replicate and variable are assumed. If the data come from a single replication but multiple vectors, such as data on coordinates for a single space curve, then be sure to coerce the data into an array object by using the as.array function with one as the central dimension length.

fdParobj — a functional parameter object, a functional data object or a functional basis object. In the simplest case, fdParobj may be a functional basis object with class "basisfd" defined previously by one of the "create" functions, and in this case, no roughness penalty is used. No roughness penalty is also the consequence of supplying a functional data object with class "fd", in which case the basis system used for smoothing is that defining this object. In these two simple cases, smooth.basis is essentially the same as function Data2fd, and this type of elementary smoothing is often called "regression smoothing." However, if the object is a functional parameter object with class "fdPar", then the linear differential operator object and the smoothing parameter in this object define the roughness penalty. For further details on how the roughness penalty is defined, see the help file for "fdPar". In general, better results can be obtained using a good roughness penalty than can be obtained by merely varying the number of basis functions in the expansion.

wtvec — typically a vector of length n that is the length of argvals containing weights for the values to be smoothed, However, it may also be a symmetric matrix of order n. If wtvec is a vector, all values must be positive, and if it is a symmetric matrix, this must be positive definite. Defaults to all weights equal to 1.

fdnames — a list of length 3 containing character vectors of names for the following:

- args name for each observation or point in time at which data are collected for each 'rep', unit or subject.
- reps name for each 'rep', unit or subject.
- fun name for each 'fun' or (response) variable measured repeatedly (per 'args') for each 'rep'.

covariates — The observed values in y are assumed to be primarily determined by the height of the curve being estimates, but from time to time certain values can also be influenced by other known variables. For example, multi-year sets of climate variables may be also determined by the presence of absence of an El Nino event, or a volcanic eruption. One or more of these covariates can be supplied as an n by p matrix, where p is the number of such covariates. When such covariates are available, the smoothing is called "semi-parametric." Matrices or arrays of regression coefficients are then estimated that define the impacts of each of these covariates for each curve and each variable.

method — by default the function uses the usual textbook equations for computing the coefficients of the basis function expansions. But, as in regression analysis, a price is paid in terms of rounding error for such computations since they involved cross-products of basis function values. Optionally, if method is set equal to the string "qr", the computation uses an algorithm based on the qr-decomposition which is more accurate, but will require substantially more computing time when n is

large, meaning more than 500 or so. The default is "chol", referring the Choleski decomposition of a symmetric positive definite matrix.

`dfscale` the generalized cross-validation or "gcv" criterion that is often used to determine the size of the smoothing parameter involves the subtraction of an measue of degrees of freedom from `n`. Chong Gu has argued that multiplying this degrees of freedom measure by a constant slightly greater than 1, such as 1.2, can produce better decisions about the level of smoothing to be used. The default value is, however, 1.0.

`returnMatrix` logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Details

A roughness penalty is a quantitative measure of the roughness of a curve that is designed to fit the data. For this function, this penalty consists of the product of two parts. The first is an approximate integral over the argument range of the square of a derivative of the curve. A typical choice of derivative order is 2, whose square is often called the local curvature of the function. Since a rough function has high curvature over most of the function's range, the integrated square of of the second derivative quantifies the total curvature of the function, and hence its roughness. The second factor is a positive constant called the bandwidth of smoothing parameter, and given the variable name `lambda` here.

In more sophisticated uses of `smooth.basis`, a derivative may be replaced by a linear combination of two or more order of derivative, with the coefficients of this combination themselves possibly varying over the argument range. Such a structure is called a "linear differential operator", and a clever choice of operator can result in much improved smoothing.

The rougnhness penalty is added to the weighted error sum of squares and the composite is minimized, usually in conjunction with a high dimensional basis expansion such as a spline function defined by placing a knot at every observation point. Consequently, the smoothing parameter controls the relative emphasis placed on fitting the data versus smoothness; when large, the fitted curve is more smooth, but the data fit worse, and when small, the fitted curve is more rough, but the data fit much better. Typically smoothing parameter `lambda` is manipulated on a logarithmic scale by, for example, defining it as a power of 10.

A good compromise `lambda` value can be difficult to define, and minimizing the generalized cross-validation or "gcv" criterion that is output by `smooth.basis` is a popular strategy for making this choice, although by no means foolproof. One may also explore `lambda` values for a few log units up and down from this minimizing value to see what the smoothing function and its derivatives look like. The function `plotfit.fd` was designed for this purpose.

The size of common logarithm of the minimizing value of `lambda` can vary widely, and spline functions depends critically on the typical spacing between knots. While there is typically a "natural" measurement scale for the argument, such as time in milliseconds, seconds, and so forth, it is better from a computational perspective to choose an argument scaling that gives knot spacings not too different from one.

An alternative to using `smooth.basis` is to first represent the data in a basis system with reasonably high resolution using `Data2fd`, and then smooth the resulting functional data object using function `smooth.fd`.

In warning and error messages, you may see reference to functions `smooth.basis1`, `smooth.basis2`, and `smooth.basis3`. These functions are defined within `smooth.basis`, and are not normally to be called by users.

The "qr" algorithm option defined by the "method" parameter will not normally be needed, but if a warning of a near singularity in the coefficient calculations appears, this choice may be a cure.

Value

an object of class `fdSmooth`, which is a named list of length 8 with the following components:

`fd`	a functional data object containing a smooth of the data.
`df`	a degrees of freedom measure of the smooth
`gcv`	the value of the generalized cross-validation or GCV criterion. If there are multiple curves, this is a vector of values, one per curve. If the smooth is multivariate, the result is a matrix of gcv values, with columns corresponding to variables. $$gcv = n * SSE/((n - df)^2)$$
`beta`	the regression coefficients associated with covariate variables. These are vector, matrix or array objects depending on whether there is a single curve, multiple curves or multiple curves and variables, respectively.
`SSE`	the error sums of squares. SSE is a vector or a matrix of the same size as GCV.
`penmat`	the penalty matrix.
`y2cMap`	the matrix mapping the data to the coefficients.
`argvals, y`	input arguments

See Also

`lambda2df`, `lambda2gcv`, `df2lambda`, `plot.fd`, `project.basis`, `smooth.fd`, `smooth.monotone`, `smooth.pos`, `smooth.basisPar` `Data2fd`,

Examples

```
##
######## Simulated data example 1: a simple regression smooth  ########
##
#  Warning:  In this and all simulated data examples, your results
#  probably won't be the same as we saw when we ran the example because
#  random numbers depend on the seed value in effect at the time of the
#  analysis.
#
#  Set up 51 observation points equally spaced between 0 and 1
n = 51
argvals = seq(0,1,len=n)
#  The true curve values are sine function values with period 1/2
x = sin(4*pi*argvals)
#  Add independent Gaussian errors with std. dev. 0.2 to the true values
sigerr = 0.2
```

```
y = x + rnorm(x)*sigerr
#  When we ran this code, we got these values of y (rounded to two
#  decimals):
y = c(0.27,  0.05,  0.58,  0.91,  1.07,  0.98,  0.54,  0.94,  1.13,  0.64,
      0.64,  0.60,  0.24,  0.15, -0.20, -0.63, -0.40, -1.22, -1.11, -0.76,
     -1.11, -0.69, -0.54, -0.50, -0.35, -0.15,  0.27,  0.35,  0.65,  0.75,
      0.75,  0.91,  1.04,  1.04,  1.04,  0.46,  0.30, -0.01, -0.19, -0.42,
     -0.63, -0.78, -1.01, -1.08, -0.91, -0.92, -0.72, -0.84, -0.38, -0.23,
      0.02)
#  Set up a B-spline basis system of order 4 (piecewise cubic) and with
#  knots at 0, 0.1, ..., 0.9 and 1.0, and plot the basis functions
nbasis = 13
basisobj = create.bspline.basis(c(0,1),nbasis)
plot(basisobj)
#  Smooth the data, outputting only the functional data object for the
#  fitted curve.  Note that in this simple case we can supply the basis
#  object as the "fdParobj" parameter
ys = smooth.basis(argvals=argvals, y=y, fdParobj=basisobj)
Ys = smooth.basis(argvals=argvals, y=y, fdParobj=basisobj,
                  returnMatrix=TRUE)
# Ys[[7]] = Ys$y2cMap is sparse;  everything else is the same


all.equal(ys[-7], Ys[-7])


xfd = ys$fd
Xfd = Ys$fd

#  Plot the curve along with the data
plotfit.fd(y, argvals, xfd)
#  Compute the root-mean-squared-error (RMSE) of the fit relative to the
#  truth
RMSE = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
print(RMSE)  #  We obtained 0.069
#  RMSE = 0.069 seems good relative to the standard error of 0.2.
#  Range through numbers of basis functions from 4 to 12 to see if we
#  can do better.  We want the best RMSE, but we also want the smallest
#  number of basis functions, which in this case is the degrees of
#  freedom for error (df).  Small df implies a stable estimate.
#  Note: 4 basis functions is as small as we can use without changing the
#  order of the spline.  Also display the gcv statistic to see what it
#  likes.
for (nbasis in 4:12) {
  basisobj = create.bspline.basis(c(0,1),nbasis)
  ys = smooth.basis(argvals, y, basisobj)
  xfd = ys$fd
  gcv = ys$gcv
  RMSE = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
# progress report:
#  cat(paste(nbasis,round(RMSE,3),round(gcv,3),"\n"))
}
#  We got RMSE = 0.062 for 10 basis functions as optimal, but gcv liked
```

```
#  almost the same thing, namely 9 basis functions.  Both RMSE and gcv
#  agreed emphatically that 7 or fewer basis functions was not enough.
#  Unlike RMSE, however, gcv does not depend on knowing the truth.
#  Plot the result for 10 basis functions along with "*" at the true
#  values
nbasis = 10
basisobj = create.bspline.basis(c(0,1),10)
xfd = smooth.basis(argvals, y, basisobj)$fd
plotfit.fd(y, argvals, xfd)
points(argvals,x, pch="*")
#  Homework:
#  Repeat all this with various values of sigerr and various values of n

##
####### Simulated data example 2: a roughness-penalized smooth  ########
##

#  A roughness penalty approach is more flexible, allowing continuous
#  control over smoothness and degrees of freedom, can be adapted to
#  known features in the curve, and will generally provide better RMSE
#  for given degrees of freedom.

#  It does require a bit more effort, though.
#  First, we define a little display function for showing how
#  df, gcv and RMSE depend on the log10 smoothing parameter
plotGCVRMSE.fd = function(lamlow, lamhi, lamdel, x, argvals, y,
            fdParobj, wtvec=NULL, fdnames=NULL, covariates=NULL)  {
  loglamvec = seq(lamlow, lamhi, lamdel)
  loglamout = matrix(0,length(loglamvec),4)
  m = 0
  for (loglambda in loglamvec) {
    m = m + 1
    loglamout[m,1] = loglambda
    fdParobj$lambda = 10^(loglambda)
    smoothlist = smooth.basis(argvals, y, fdParobj, wtvec=wtvec,
             fdnames=fdnames, covariates=covariates)
    xfd = smoothlist$fd   #  the curve smoothing the data
    loglamout[m,2] = smoothlist$df
    #  degrees of freedom in the smoothing curve
    loglamout[m,3] = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
    loglamout[m,4] = mean(smoothlist$gcv)  #  the mean of the N gcv values
  }
  cat("log10 lambda, deg. freedom, RMSE, gcv\n")
  for (i in 1:m) {
    cat(format(round(loglamout[i,],3)))
    cat("\n")
  }
  par(mfrow=c(3,1))
  plot(loglamvec, loglamout[,2], type="b")
  title("Degrees of freedom")
  plot(loglamvec, loglamout[,3], type="b")
  title("RMSE")
  plot(loglamvec, loglamout[,4], type="b")
```

```
  title("Mean gcv")
  return(loglamout)
}

#  Use the data that you used in Example 1, or run the following code:
n = 51
argvals = seq(0,1,len=n)
x = sin(4*pi*argvals)
sigerr = 0.2
err = matrix(rnorm(x),n,1)*sigerr
y = x + err
#  We now set up a basis system with a knot at every data point.
#  The number of basis functions will equal the number of interior knots
#  plus the order, which in this case is still 4.
#  49 interior knots + order 4 = 53 basis functions
nbasis = n + 2
basisobj = create.bspline.basis(c(0,1),nbasis)
#  Note that there are more basis functions than observed values.  A
#  basis like this is called "super-saturated."  We have to use a
#  positive smoothing parameter for it to work.  Set up an object of
#  class "fdPar" that penalizes the total squared second derivative,
#  using a smoothing parameter that is set here to 10^(-4.5).
lambda = 10^(-4.5)
fdParobj = fdPar(fdobj=basisobj, Lfdobj=2, lambda=lambda)
#  Smooth the data, outputting a list containing various quantities
smoothlist = smooth.basis(argvals, y, fdParobj)
xfd = smoothlist$fd   #  the curve smoothing the data
df  = smoothlist$df   #  the degrees of freedom in the smoothing curve
gcv = smoothlist$gcv  #  the value of the gcv statistic
RMSE = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
cat(round(c(df,RMSE,gcv),3),"\n")
plotfit.fd(y, argvals, xfd)
points(argvals,x, pch="*")
#  Repeat these analyses for a range of log10(lambda) values by running
#  the function plotGCVRMSE that we defined above.

loglamout = plotGCVRMSE.fd(-6, -3, 0.25, x, argvals, y, fdParobj)

#  When we ran this example, the optimal RMSE was 0.073, and was achieved
#  for log10(lambda) = -4.25 or lambda = 0.000056.  At this level of
#  smoothing, the degrees of freedom index was 10.6, a value close to
#  the 10 degrees of freedom that we saw for regression smoothing.  The
#  RMSE value is slightly higher than the regression analysis result, as
#  is the degrees of freedom associated with the optimal value.
#  Roughness penalty will, as we will see later, do better than
#  regression smoothing; but with slightly red faces we say, "That's
#  life with random data!"  The gcv statistic agreed with RMSE on the
#  optimal smoothing level, which is great because it does not need to
#  know the true values.  Note that gcv is emphatic about when there is
#  too much smoothing, but rather vague about when we have
#  under-smoothed the data.
#  Homework:
#  Compute average results taken across 100 sets of random data for each
```

```
#  level of smoothing parameter lambda, and for each number of basis
#  functions for regression smoothing.

##
##                        Simulated data example 3:
##           a roughness-penalized smooth of a sample of curves
##
n =  51   #  number of observations per curve
N = 100   #  number of curves
argvals = seq(0,1,len=n)
#  The true curve values are linear combinations of fourier function
#  values.
#  Set up the fourier basis
nfourierbasis = 13
fourierbasis = create.fourier.basis(c(0,1),nfourierbasis)
fourierbasismat = eval.basis(argvals,fourierbasis)
#  Set up some random coefficients, with declining contributions from
#  higher order basis functions
basiswt = matrix(exp(-(1:nfourierbasis)/3),nfourierbasis,N)
xcoef = matrix(rnorm(nfourierbasis*N),nfourierbasis,N)*basiswt
xfd = fd(xcoef, fourierbasis)
x   = eval.fd(argvals, xfd)
#  Add independent Gaussian noise to the data with std. dev. 0.2
sigerr = 0.2
y = x + matrix(rnorm(c(x)),n,N)*sigerr
#  Set up spline basis system
nbasis = n + 2
basisobj = create.bspline.basis(c(0,1),nbasis)
#  Set up roughness penalty with smoothing parameter 10^(-5)
lambda = 10^(-5)
fdParobj = fdPar(fdobj=basisobj, Lfdobj=2, lambda=lambda)
#  Smooth the data, outputting a list containing various quantities
smoothlist = smooth.basis(argvals, y, fdParobj)
xfd = smoothlist$fd   #  the curve smoothing the data
df  = smoothlist$df   #  the degrees of freedom in the smoothing curve
gcv = smoothlist$gcv  #  the value of the gcv statistic
RMSE = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
#  Display the results, note that a gcv value is returned for EACH curve,
#  and therefore that a mean gcv result is reported
cat(round(c(df,RMSE,mean(gcv)),3),"\n")
#  the fits are plotted interactively by plotfit.fd ... click to advance
#  plot
plotfit.fd(y, argvals, xfd)
#  Repeat these results for a range of log10(lambda) values
loglamout = plotGCVRMSE.fd(-6, -3, 0.25, x, argvals, y, fdParobj)
#  Our results were:
# log10 lambda, deg. freedom, RMSE, GCV
# -6.000 30.385  0.140  0.071
# -5.750 26.750  0.131  0.066
# -5.500 23.451  0.123  0.062
# -5.250 20.519  0.116  0.059
# -5.000 17.943  0.109  0.056
# -4.750 15.694  0.104  0.054
```

```
# -4.500 13.738  0.101  0.053
# -4.250 12.038  0.102  0.054
# -4.000 10.564  0.108  0.055
# -3.750  9.286  0.120  0.059
# -3.500  8.177  0.139  0.065
# -3.250  7.217  0.164  0.075
# -3.000  6.385  0.196  0.088
#  RMSE and gcv both indicate an optimal smoothing level of
#  log10(lambda) = -4.5 corresponding to 13.7 df.  When we repeated the
#  analysis using regression smoothing with 14 basis functions, we
#  obtained RMSE = 0.110, about 10 percent larger than the value of
#  0.101 in the roughness penalty result.  Smooth the data, outputting a
#  list containing various quantities
#  Homework:
#  Sine functions have a curvature that doesn't vary a great deal over
#  the range the curve.  Devise some test functions with sharp local
#  curvature, such as Gaussian densities with standard deviations that
#  are small relative to the range of the observations.  Compare
#  regression and roughness penalty smoothing in these situations.

if(!CRAN()){
##
####### Simulated data example 4: a roughness-penalized smooth  ########
##                           with correlated error
##
#  These three examples make GCV look pretty good as a basis for
#  selecting the smoothing parameter lambda.  BUT GCV is based an
#  assumption of independent errors, and in reality, functional data
#  often have autocorrelated errors, with an autocorrelation that is
#  usually positive among neighboring observations.  Positively
#  correlated random values tend to exhibit slowly varying values that
#  have long runs on one side or the other of their baseline, and
#  therefore can look like trend in the data that needs to be reflected
#  in the smooth curve.  This code sets up the error correlation matrix
#  for first-order autoregressive errors, or AR(1).
rho = 0.9
n = 51
argvals = seq(0,1,len=n)
x = sin(4*pi*argvals)
Rerr = matrix(0,n,n)
for (i in 1:n) {
  for (j in 1:n) Rerr[i,j] =  rho^abs(i-j)
}
#  Compute the Choleski factor of the correlation matrix
Lerr = chol(Rerr)
#  set up some data
#  Generate auto-correlated errors by multipling independent errors by
#  the transpose of the Choleski factor
sigerr = 0.2
err = as.vector(crossprod(Lerr,matrix(rnorm(x),n,1))*sigerr)
#  See the long-run errors that are genrated
plot(argvals, err)
y = x + err
```

```
#  Our values of y were:
y = c(-0.03, 0.36, 0.59, 0.71, 0.97,  1.2,  1.1, 0.96, 0.79, 0.68,
       0.56, 0.25, 0.01,-0.43,-0.69, -1,  -1.09,-1.29,-1.16,-1.09,
      -0.93, -0.9,-0.78,-0.47, -0.3,-0.01, 0.29, 0.47, 0.77, 0.85,
       0.87, 0.97,  0.9, 0.72, 0.48, 0.25,-0.17,-0.39,-0.47,-0.71,
      -1.07,-1.28,-1.33,-1.39,-1.45, -1.3,-1.25,-1.04,-0.82,-0.55, -0.2)
#  Retaining the above data, now set up a basis system with a knot at
#  every data point:  the number of basis functions will equal the
#  number of interior knots plus the order, which in this case is still
#  4.
#   19 interior knots + order 4 = 23 basis functions
nbasis = n + 2
basisobj = create.bspline.basis(c(0,1),nbasis)
fdParobj = fdPar(basisobj)
#  Smooth these results for a range of log10(lambda) values
loglamout = plotGCVRMSE.fd(-6, -3, 0.25, x, argvals, y, fdParobj)
#  Our results without weighting were:
# -6.000 30.385  0.261  0.004
# -5.750 26.750  0.260  0.005
# -5.500 23.451  0.259  0.005
# -5.250 20.519  0.258  0.005
# -5.000 17.943  0.256  0.005
# -4.750 15.694  0.255  0.006
# -4.500 13.738  0.252  0.006
# -4.250 12.038  0.249  0.007
# -4.000 10.564  0.246  0.010
# -3.750  9.286  0.244  0.015
# -3.500  8.177  0.248  0.028
# -3.250  7.217  0.267  0.055
# -3.000  6.385  0.310  0.102
#  Now GCV still is firm on the fact that log10(lambda) over -4 is
#  over-smoothing, but is quite unhelpful about what constitutes
#  undersmoothing.  In real data applications you will have to make the
#  final call.  Now set up a weight matrix equal to the inverse of the
#  correlation matrix
wtmat = solve(Rerr)
#  Smooth these results for a range of log10(lambda) values using the
#  weight matrix
loglamout = plotGCVRMSE.fd(-6, -3, 0.25, x, argvals, y, fdParobj,
                        wtvec=wtmat)
#  Our results with weighting were:
# -6.000 46.347  0.263  0.005
# -5.750 43.656  0.262  0.005
# -5.500 40.042  0.261  0.005
# -5.250 35.667  0.259  0.005
# -5.000 30.892  0.256  0.005
# -4.750 26.126  0.251  0.006
# -4.500 21.691  0.245  0.008
# -4.250 17.776  0.237  0.012
# -4.000 14.449  0.229  0.023
# -3.750 11.703  0.231  0.045
# -3.500  9.488  0.257  0.088
# -3.250  7.731  0.316  0.161
```

```
# -3.000  6.356  0.397  0.260
#  GCV is still not clear about what the right smoothing level is.
#  But, comparing the two results, we see an optimal RMSE without
#  smoothing of 0.244 at log10(lambda) = -3.75, and with smoothing 0.229
#  at log10(lambda) = -4.  Weighting improved the RMSE.  At
#  log10(lambda) = -4 the improvement is 9 percent.
#  Smooth the data with and without the weight matrix at log10(lambda) =
#  -4
fdParobj = fdPar(basisobj, 2, 10^(-4))
smoothlistnowt = smooth.basis(argvals, y, fdParobj)
fdobjnowt = smoothlistnowt$fd   #  the curve smoothing the data
df  = smoothlistnowt$df  # the degrees of freedom in the smoothing curve
GCV = smoothlistnowt$gcv  #  the value of the GCV statistic
RMSE = sqrt(mean((eval.fd(argvals, fdobjnowt) - x)^2))
cat(round(c(df,RMSE,GCV),3),"\n")
smoothlistwt = smooth.basis(argvals, y, fdParobj, wtvec=wtmat)
fdobjwt = smoothlistwt$fd   #  the curve smoothing the data
df  = smoothlistwt$df   #  the degrees of freedom in the smoothing curve
GCV = smoothlistwt$gcv  #  the value of the GCV statistic
RMSE = sqrt(mean((eval.fd(argvals, fdobjwt) - x)^2))
cat(round(c(df,RMSE,GCV),3),"\n")
par(mfrow=c(2,1))
plotfit.fd(y, argvals, fdobjnowt)
plotfit.fd(y, argvals, fdobjwt)
par(mfrow=c(1,1))
plot(fdobjnowt)
lines(fdobjwt,lty=2)
points(argvals, y)
#  Homework:
#  Repeat these analyses with various values of rho, perhaps using
#  multiple curves to get more stable indications of relative
#  performance.  Be sure to include some negative rho's.

##
######## Simulated data example 5: derivative estimation  ########
##
#  Functional data analyses often involve estimating derivatives.  Here
#  we repeat the analyses of Example 2, but this time focussing our
#  attention on the estimation of the first and second derivative.
n = 51
argvals = seq(0,1,len=n)
x   = sin(4*pi*argvals)
Dx  = 4*pi*cos(4*pi*argvals)
D2x = -(4*pi)^2*sin(4*pi*argvals)
sigerr = 0.2
y = x + rnorm(x)*sigerr
#  We now use order 6 splines so that we can control the curvature of
#  the second derivative, which therefore involves penalizing the
#  derivative of order four.
norder = 6
nbasis = n + norder - 2
basisobj = create.bspline.basis(c(0,1),nbasis,norder)
#  Note that there are more basis functions than observed values.  A
```

```
#  basis like this is called "super-saturated."  We have to use a
#  positive smoothing parameter for it to work.  Set up an object of
#  class "fdPar" that penalizes the total squared fourth derivative. The
#  smoothing parameter that is set here to 10^(-10), because the squared
#  fourth derivative is a much larger number than the squared second
#  derivative.
lambda = 10^(-10)
fdParobj = fdPar(fdobj=basisobj, Lfdobj=4, lambda=lambda)
#  Smooth the data, outputting a list containing various quantities
smoothlist = smooth.basis(argvals, y, fdParobj)
xfd = smoothlist$fd   #  the curve smoothing the data
df  = smoothlist$df   #  the degrees of freedom in the smoothing curve
gcv = smoothlist$gcv  #  the value of the gcv statistic
Dxhat  = eval.fd(argvals, xfd, Lfd=1)
D2xhat = eval.fd(argvals, xfd, Lfd=2)
RMSED  = sqrt(mean((Dxhat  - Dx )^2))
RMSED2 = sqrt(mean((D2xhat - D2x)^2))
cat(round(c(df,RMSED,RMSED2,gcv),3),"\n")
#  Four plots of the results row-wise: data fit, first derivative fit,
#  second derivative fit, second vs. first derivative fit
#  (phase-plane plot)
par(mfrow=c(2,2))
plotfit.fd(y, argvals, xfd)
plot(argvals, Dxhat, type="p", pch="o")
points(argvals, Dx, pch="*")
title("first derivative approximation")
plot(argvals, D2xhat, type="p", pch="o")
points(argvals, D2x, pch="*")
title("second derivative approximation")
plot(Dxhat, D2xhat, type="p", pch="o")
points(Dx, D2x, pch="*")
title("second against first derivative")
#  This illustrates an inevitable problem with spline basis functions;
#  because they are not periodic, they fail to capture derivative
#  information well at the ends of the interval.  The true phase-plane
#  plot is an ellipse, but the phase-plane plot of the estimated
#  derivatives here is only a rough approximtion, and breaks down at the
#  left boundary.
#  Homework:
#  Repeat these results with smaller standard errors.
#  Repeat these results, but this time use a fourier basis with no
#  roughness penalty, and find the number of basis functions that gives
#  the best result.  The right answer to this question is, of course, 3,
#  if we retain the constant term, even though it is here not needed.
#  Compare the smoothing parameter preferred by RMSE for a derivative to
#  that preferred by the RMSE for the function itself, and to that
#  preferred by gcv.

##                  Simulated data example 6:
##           a better roughness penalty for derivative estimation
##
#  We want to see if we can improve the spline fit.
#  We know from elementary calculus as well as the code above that
```

```
#  (4*pi)^2 sine(2*p*x) = -D2 sine(2*p*x), so that
#  Lx = D2x + (4*pi)^2 x is zero for a sine or a cosine curve.
#  We now penalize roughness using this "smart" roughness penalty
#  Here we set up a linear differential operator object that defines
#  this penalty
constbasis = create.constant.basis(c(0,1))
xcoef.fd  = fd((4*pi)^2, constbasis)
Dxcoef.fd = fd(0, constbasis)
bwtlist = vector("list", 2)
bwtlist[[1]] = xcoef.fd
bwtlist[[2]] = Dxcoef.fd
Lfdobj = Lfd(nderiv=2, bwtlist=bwtlist)
#  Now we use a much larger value of lambda to reflect our confidence
#  in power of calculus to solve problems!
lambda = 10^(0)
fdParobj = fdPar(fdobj=basisobj, Lfdobj=Lfdobj, lambda=lambda)
smoothlist = smooth.basis(argvals, y, fdParobj)
xfd = smoothlist$fd   #  the curve smoothing the data
df  = smoothlist$df   #  the degrees of freedom in the smoothing curve
gcv = smoothlist$gcv  #  the value of the gcv statistic
Dxhat  = eval.fd(argvals, xfd, Lfd=1)
D2xhat = eval.fd(argvals, xfd, Lfd=2)
RMSED  = sqrt(mean((Dxhat  - Dx )^2))
RMSED2 = sqrt(mean((D2xhat - D2x)^2))
cat(round(c(df,RMSED,RMSED2,gcv),3),"\n")
#  Four plots of the results row-wise: data fit, first derivative fit,
#  second derivative fit, second versus first derivative fit
#  (phase-plane plot)
par(mfrow=c(2,2))
plotfit.fd(y, argvals, xfd)
plot(argvals, Dxhat, type="p", pch="o")
points(argvals, Dx, pch="*")
title("first derivative approximation")
plot(argvals, D2xhat, type="p", pch="o")
points(argvals, D2x, pch="*")
title("second derivative approximation")
plot(Dxhat, D2xhat, type="p", pch="o")
points(Dx, D2x, pch="*")
title("second against first derivative")
#  The results are nearly perfect in spite of the fact that we are not using
#  periodic basis functions.  Notice, too, that we have used 2.03
#  degrees of freedom, which is close to what we would use for the ideal
#  fourier series basis with the constant term dropped.
#  Homework:
#  These results depended on us knowing the right period, of course.
#  The data would certainly allow us to estimate the period 1/2 closely,
#  but try various  other periods by replacing 1/2 by other values.
#  Alternatively, change x by adding a small amount of, say, linear trend.
#  How much trend do you have to add to seriously handicap the results?

##
######## Simulated data example 7: Using covariates  ########
##
```

```
#  Now we simulate data that are defined by a sine curve, but where the
#  the first 20 observed values are shifted upwards by 0.5, and the
#  second shifted downwards by -0.2.  The two covariates are indicator
#  or dummy variables, and the estimated regression coefficients will
#  indicate the shifts as estimated from the data.
n = 51
argvals = seq(0,1,len=n)
x = sin(4*pi*argvals)
sigerr = 0.2
y = x + rnorm(x)*sigerr
#  the n by p matrix of covariate values, p being here 2
p = 2
zmat = matrix(0,n,p)
zmat[ 1:11,1] = 1
zmat[11:20,2] = 1
#  The true values of the regression coefficients
beta0 = matrix(c(0.5,-0.2),p,1)
y = y + zmat
#  The same basis system and smoothing process as used in Example 2
nbasis = n + 2
basisobj = create.bspline.basis(c(0,1),nbasis)
lambda = 10^(-4)
fdParobj = fdPar(basisobj, 2, lambda)
#  Smooth the data, outputting a list containing various quantities
smoothlist = smooth.basis(argvals, y, fdParobj, covariates=zmat)
xfd  = smoothlist$fd   #  the curve smoothing the data
df   = smoothlist$df   #  the degrees of freedom in the smoothing curve
gcv  = smoothlist$gcv  #  the value of the gcv statistic
beta = smoothlist$beta  #  the regression coefficients
RMSE = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
cat(round(c(beta,df,RMSE,gcv),3),"\n")
par(mfrow=c(1,1))
plotfit.fd(y, argvals, xfd)
points(argvals,x, pch="*")
print(beta)
#  The recovery of the smooth curve is fine, as in Example 2.  The
#  shift of the first 10 observations was estimated to be 0.62 in our run,
#  and the shift of the second 20 was estimated to be -0.42.  These
#  estimates are based on only 10 observations, and these estimates are
#  therefore quite reasonable.
#  Repeat these analyses for a range of log10(lambda) values
loglamout = plotGCVRMSE.fd(-6, -3, 0.25, x, argvals, y, fdParobj,
                           covariates=zmat)
#  Homework:
#  Try an example where the covariate values are themselves are
#  generated by a smooth known curve.

##
##                     Simulated data example 8:
##          a roughness-penalized smooth of a sample of curves and
##                     variable observation points
##
n = 51   #  number of observations per curve
```

```
N = 100   #  number of curves
argvals = matrix(0,n,N)
for (i in 1:N) argvals[,i] = sort(runif(1:n))
#  The true curve values are linear combinations of fourier function
#  values.
#  Set up the fourier basis
nfourierbasis = 13
fourierbasis = create.fourier.basis(c(0,1),nfourierbasis)
#  Set up some random coefficients, with declining contributions from
#  higher order basis functions
basiswt = matrix(exp(-(1:nfourierbasis)/3),nfourierbasis,N)
xcoef = matrix(rnorm(nfourierbasis*N),nfourierbasis,N)*basiswt
xfd = fd(xcoef, fourierbasis)
x = matrix(0,n,N)
for (i in 1:N) x[,i] = eval.fd(argvals[,i], xfd[i])
#  Add independent Gaussian noise to the data with std. dev. 0.2
sigerr = 0.2
y = x + matrix(rnorm(c(x)),n,N)*sigerr
#  Set up spline basis system
nbasis = n + 2
basisobj = create.bspline.basis(c(0,1),nbasis)
#  Set up roughness penalty with smoothing parameter 10^(-5)
lambda = 10^(-5)
fdParobj = fdPar(fdobj=basisobj, Lfdobj=2, lambda=lambda)
#  Smooth the data, outputting a list containing various quantities
smoothlist = smooth.basis(argvals, y, fdParobj)
xfd = smoothlist$fd   #  the curve smoothing the data
df  = smoothlist$df   #  the degrees of freedom in the smoothing curve
gcv = smoothlist$gcv  #  the value of the gcv statistic
#RMSE = sqrt(mean((eval.fd(argvals, xfd) - x)^2))
eval.x <- eval.fd(argvals, xfd)
e.xfd <- (eval.x-x)
mean.e2 <- mean(e.xfd^2)

RMSE = sqrt(mean.e2)
#  Display the results, note that a gcv value is returned for EACH
#  curve, and therefore that a mean gcv result is reported
cat(round(c(df,RMSE,mean(gcv)),3),"\n")
#  Function plotfit.fd is not equipped to handle a matrix of argvals,
#  but can always be called within a loop to plot each curve in turn.
#  Although a call to function plotGCVRMSE.fd works, the computational
#  overhead is substantial, and we omit this here.

##
## Real data example 9.  gait
##
#  These data involve two variables in addition to multiple curves
gaittime  <- (1:20)/21
gaitrange <- c(0,1)
gaitbasis <- create.fourier.basis(gaitrange,21)
lambda    <- 10^(-11.5)
harmaccelLfd <- vec2Lfd(c(0, 0, (2*pi)^2, 0))
gaitfdPar <- fdPar(gaitbasis, harmaccelLfd, lambda)
```

```
gaitSmooth <- smooth.basis(gaittime, gait, gaitfdPar)
gaitfd <- gaitSmooth$fd
## Not run:
  # by default creates multiple plots, asking for a click between plots
 plotfit.fd(gait, gaittime, gaitfd)

## End(Not run)



}
#  end of if (!CRAN)
```

`smooth.basis.sparse` *Construct a functional data object by smoothing data using a roughness penalty*

Description

Makes it possible to perform smoothing with `smooth.basis` when the data has NAs.

Usage

```
smooth.basis.sparse(argvals, y, fdParobj, fdnames=NULL, covariates=NULL,
                    method="chol", dfscale=1 )
```

Arguments

`argvals`	a set of argument values corresponding to the observations in array `y`. In most applications these values will be common to all curves and all variables, and therefore be defined as a vector or as a matrix with a single column. But it is possible that these argument values will vary from one curve to another, and in this case `argvals` will be input as a matrix with rows corresponding to observation points and columns corresponding to curves. Argument values can even vary from one variable to another, in which case they are input as an array with dimensions corresponding to observation points, curves and variables, respectively. Note, however, that the number of observation points per curve and per variable may NOT vary. If it does, then curves and variables must be smoothed individually rather than by a single call to this function. The default value for `argvals` are the integers 1 to `n`, where `n` is the size of the first dimension in argument `y`.
`y`	an set of values of curves at discrete sampling points or argument values. If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If `y` is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. If `y` is a vector, only one replicate and variable

are assumed. If the data come from a single replication but multiple vectors, such as data on coordinates for a single space curve, then be sure to coerce the data into an array object by using the `as.array` function with one as the central dimension length.

`fdParobj` a functional parameter object, a functional data object or a functional basis object. In the simplest case, `fdParobj` may be a functional basis object with class "basisfd" defined previously by one of the "create" functions, and in this case, no roughness penalty is used. No roughness penalty is also the consequence of supplying a functional data object with class "fd", in which case the basis system used for smoothing is that defining this object. However, if the object is a functional parameter object with class "fdPar", then the linear differential operator object and the smoothing parameter in this object define the roughness penalty. For further details on how the roughness penalty is defined, see the help file for "fdPar". In general, better results can be obtained using a good roughness penalty than can be obtained by merely varying the number of basis functions in the expansion.

`fdnames` a list of length 3 containing character vectors of names for the following:

- args name for each observation or point in time at which data are collected for each 'rep', unit or subject.
- reps name for each 'rep', unit or subject.
- fun name for each 'fun' or (response) variable measured repeatedly (per 'args') for each 'rep'.

`covariates` The observed values in `y` are assumed to be primarily determined by the height of the curve being estimates, but from time to time certain values can also be influenced by other known variables. For example, multi-year sets of climate variables may be also determined by the presence of absence of an El Nino event, or a volcanic eruption. One or more of these covariates can be supplied as an `n` by `p` matrix, where `p` is the number of such covariates. When such covariates are available, the smoothing is called "semi-parametric." Matrices or arrays of regression coefficients are then estimated that define the impacts of each of these covariates for each curve and each variable.

`method` by default the function uses the usual textbook equations for computing the coefficients of the basis function expansions. But, as in regression analysis, a price is paid in terms of rounding error for such computations since they involved cross-products of basis function values. Optionally, if `method` is set equal to the string "qr", the computation uses an algorithm based on the qr-decomposition which is more accurate, but will require substantially more computing time when `n` is large, meaning more than 500 or so. The default is "chol", referring the Choleski decomposition of a symmetric positive definite matrix.

`dfscale` the generalized cross-validation or "gcv" criterion that is often used to determine the size of the smoothing parameter involves the subtraction of an measue of degrees of freedom from `n`. Chong Gu has argued that multiplying this degrees of freedom measure by a constant slightly greater than 1, such as 1.2, can produce better decisions about the level of smoothing to be used. The default value is, however, 1.0.

Value

an object of class fd.

See Also

smooth.basis

---

smooth.basisPar *Smooth Data Using a Directly Specified Roughness Penalty*

Description

Smooth (argvals, y) data with roughness penalty defined by the remaining arguments. This function acts as a wrapper for those who want to bypass the step of setting up a functional parameter object before invoking function smooth.basis. This function simply does this setup for the user. See the help file for functions smooth.basis and fdPar for further details, and more complete descriptions of the arguments.

Usage

```
smooth.basisPar(argvals, y, fdobj=NULL, Lfdobj=NULL,
      lambda=0, estimate=TRUE, penmat=NULL,
      wtvec=NULL, fdnames=NULL, covariates=NULL,
                        method="chol", dfscale=1)
```

Arguments

argvals	a set of argument values corresponding to the observations in array y. In most applications these values will be common to all curves and all variables, and therefore be defined as a vector or as a matrix with a single column. But it is possible that these argument values will vary from one curve to another, and in this case argvals will be input as a matrix with rows corresponding to observation points and columns corresponding to curves. Argument values can even vary from one variable to another, in which case they are input as an array with dimensions corresponding to observation points, curves and variables, respectively. Note, however, that the number of observation points per curve and per variable may NOT vary. If it does, then curves and variables must be smoothed individually rather than by a single call to this function. The default value for argvals are the integers 1 to n, where n is the size of the first dimension in argument y.
y	an set of values of curves at discrete sampling points or argument values. If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If y is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. If y is a vector, only one replicate and variable

are assumed. If the data come from a single replication but multiple vectors, such as data on coordinates for a single space curve, then be sure to coerce the data into an array object by using the `as.array` function with one as the central dimension length.

`fdobj` One of the following:

- fd a functional data object (class `fd`)
- basisfd a functional basis object (class `basisfd`), which is converted to a functional data object with the identity matrix as the coefficient matrix.
- fdPar a functional parameter object (class `fdPar`)
- integer a positive integer giving the order of a B-spline basis, which is further converted to a functional data object with the identity matrix as the coefficient matrix.
- matrix or arrayreplaced by fd(fdobj)
- NULL Defaults to fdobj = create.bspline.basis(argvals).

`Lfdobj` either a nonnegative integer or a linear differential operator object.

If `NULL`, Lfdobj depends on fdobj[['basis']][['type']]:

- bspline Lfdobj <- int2Lfd(max(0, norder-2)), where norder = norder(fdobj).
- fourier Lfdobj = a harmonic acceleration operator:
  `Lfdobj <-vec2Lfd(c(0,(2*pi/diff(rng))^2,0),rng)`
  where rng = fdobj[['basis']][['rangeval']].
- anything elseLfdobj <- int2Lfd(0)

`lambda` a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter.

`estimate` a logical value: if `TRUE`, the functional parameter is estimated, otherwise, it is held fixed.

`penmat` a roughness penalty matrix. Including this can eliminate the need to compute this matrix over and over again in some types of calculations.

`wtvec` typically a vector of length `n` that is the length of `argvals` containing weights for the values to be smoothed, However, it may also be a symmetric matrix of order `n`. If `wtvec` is a vector, all values must be positive, and if it is a symmetric matrix, this must be positive definite. Defaults to all weights equal to 1.

`fdnames` a list of length 3 containing character vectors of names for the following:

- args name for each observation or point in time at which data are collected for each 'rep', unit or subject.
- reps name for each 'rep', unit or subject.
- fun name for each 'fun' or (response) variable measured repeatedly (per 'args') for each 'rep'.

`covariates` the observed values in `y` are assumed to be primarily determined the the height of the curve being estimates, but from time to time certain values can also be influenced by other known variables. For example, multi-year sets of climate variables may be also determined by the presence of absence of an El Nino event, or a volcanic eruption. One or more of these covariates can be supplied as an `n` by `p` matrix, where `p` is the number of such covariates. When such

covariates are available, the smoothing is called "semi-parametric." Matrices or arrays of regression coefficients are then estimated that define the impacts of each of these covariates for each cueve and each variable.

`method` by default the function uses the usual textbook equations for computing the coefficients of the basis function expansions. But, as in regression analysis, a price is paid in terms of rounding error for such computations since they involved cross-products of basis function values. Optionally, if `method` is set equal to the string "qr", the computation uses an algorithm based on the qr-decomposition which is more accurate, but will require substantially more computing time when `n` is large, meaning more than 500 or so. The default is "chol", referring the Choleski decomposition of a symmetric positive definite matrix.

`dfscale` the generalized cross-validation or "gcv" criterion that is often used to determine the size of the smoothing parameter involves the subtraction of an measue of degrees of freedom from `n`. Chong Gu has argued that multiplying this degrees of freedom measure by a constant slightly greater than 1, such as 1.2, can produce better decisions about the level of smoothing to be used. The default value is, however, 1.0.

Details

1. if(is.null(fdobj))fdobj <- create.bspline.basis(argvals). Else if(is.integer(fdobj)) fdobj <- create.bspline.basis(argvals, norder = fdobj)

2. fdPar

3. smooth.basis

Value

The output of a call to `smooth.basis`, which is an object of class `fdSmooth`, being a list of length 8 with the following components:

`fd` a functional data object that smooths the data.

`df` a degrees of freedom measure of the smooth

`gcv` the value of the generalized cross-validation or GCV criterion. If there are multiple curves, this is a vector of values, one per curve. If the smooth is multivariate, the result is a matrix of gcv values, with columns corresponding to variables.

`SSE` the error sums of squares. SSE is a vector or a matrix of the same size as 'gcv'.

`penmat` the penalty matrix.

`y2cMap` the matrix mapping the data to the coefficients.

`argvals, y` input arguments

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

df2lambda, fdPar, lambda2df, lambda2gcv, plot.fd, project.basis, smooth.basis, smooth.fd, smooth.monotone, smooth.pos

Examples

```
#  see smooth.basis
```

---

smooth.bibasis *Smooth a discrete surface over a rectangular lattice*

Description

Estimate a smoothing function f(s, t) over a rectangular lattice

Usage

```
smooth.bibasis(sarg, targ, y, fdPars, fdPart, fdnames=NULL, returnMatrix=FALSE)
```

Arguments

sarg, targ	vectors of argument values for the first and second dimensions, respectively, of the surface function.
y	an array containing surface values measured with noise
fdPars, fdPart	functional parameter objects for sarg and targ, respectively
fdnames	a list of length 3 containing character vectors of names for sarg, targ, and the surface function f(s, t).
returnMatrix	logical: If TRUE, a two-dimensional is returned using a special class from the Matrix package.

Value

a list with the following components:

fdobj	a functional data object containing a smooth of the data.
df	a degrees of freedom measure of the smooth
gcv	the value of the generalized cross-validation or GCV criterion. If the function is univariate, GCV is a vector containing the error sum of squares for each function, and if the function is multivariate, GCV is a NVAR by NCURVES matrix.
coef	the coefficient matrix for the basis function expansion of the smoothing function
SSE	the error sums of squares. SSE is a vector or a matrix of the same size as GCV.
penmat	the penalty matrix.
y2cMap	the matrix mapping the data to the coefficients.

See Also

smooth.basis

`smooth.fd` *Smooth a Functional Data Object Using an Indirectly Specified Roughness Penalty*

Description

Smooth data already converted to a functional data object, fdobj, using criteria consolidated in a functional data parameter object, fdParobj. For example, data may have been converted to a functional data object using function `smooth.basis` using a fairly large set of basis functions. This 'fdobj' is then smoothed as specified in 'fdParobj'.

Usage

```
smooth.fd(fdobj, fdParobj)
```

Arguments

`fdobj`	a functional data object to be smoothed.
`fdParobj`	a functional parameter object. This object is defined by a roughness penalty in slot `Lfd` and a smoothing parameter lambda in slot `lambda`, and this information is used to further smooth argument `fdobj`.

Value

a functional data object.

See Also

`smooth.basis`,

Examples

```
#  Shows the effects of two levels of smoothing
#  where the size of the third derivative is penalized.
#  The null space contains quadratic functions.
x <- seq(-1,1,0.02)
y <- x + 3*exp(-6*x^2) + rnorm(rep(1,101))*0.2
#  set up a saturated B-spline basis
basisobj <- create.bspline.basis(c(-1,1),81)
#  convert to a functional data object that interpolates the data.
result <- smooth.basis(x, y, basisobj)
yfd  <- result$fd

#  set up a functional parameter object with smoothing
#  parameter 1e-6 and a penalty on the 3rd derivative.
yfdPar <- fdPar(yfd, 2, 1e-6)
yfd1 <- smooth.fd(yfd, yfdPar)
```

```
## Not run:
# FIXME: using 3rd derivative here gave error?????
yfdPar3 <- fdPar(yfd, 3, 1e-6)
yfd1.3 <- smooth.fd(yfd, yfdPar3)
#Error in bsplinepen(basisobj, Lfdobj, rng) :
# Penalty matrix cannot be evaluated
#  for derivative of order 3 for B-splines of order 4

## End(Not run)

#  set up a functional parameter object with smoothing
#  parameter 1 and a penalty on the 3rd derivative.
yfdPar <- fdPar(yfd, 2, 1)
yfd2 <- smooth.fd(yfd, yfdPar)
#  plot the data and smooth
plot(x,y)           # plot the data
lines(yfd1, lty=1)  #  add moderately penalized smooth
lines(yfd2, lty=3)  #  add heavily  penalized smooth
legend(-1,3,c("0.000001","1"),lty=c(1,3))
#  plot the data and smoothing using function plotfit.fd
plotfit.fd(y, x, yfd1)  # plot data and smooth
```

`smooth.fdPar` *Smooth a functional data object using a directly specified roughness penalty*

Description

Smooth data already converted to a functional data object, fdobj, using directly specified criteria.

Usage

```
smooth.fdPar(fdobj, Lfdobj=NULL, lambda=1e-4,
             estimate=TRUE, penmat=NULL)
```

Arguments

`fdobj`	a functional data object to be smoothed.
`Lfdobj`	either a nonnegative integer or a linear differential operator object. If `NULL`, Lfdobj depends on fdobj[['basis']][['type']]: • bspline Lfdobj <- int2Lfd(max(0, norder-2)), where norder = norder(fdobj). • fourier Lfdobj = a harmonic acceleration operator: `Lfdobj <-vec2Lfd(c(0,(2*pi/diff(rng))^2,0),rng)` where rng = fdobj[['basis']][['rangeval']]. • anything elseLfdobj <- int2Lfd(0)
`lambda`	a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter.

estimate	a logical value: if TRUE, the functional parameter is estimated, otherwise, it is held fixed.
penmat	a roughness penalty matrix. Including this can eliminate the need to compute this matrix over and over again in some types of calculations.

Details

1. fdPar
2. smooth.fd

Value

a functional data object.

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

smooth.fd, fdPar, smooth.basis, smooth.pos, smooth.morph

Examples

```
#  see smooth.basis
```

---

smooth.monotone *Monotone Smoothing of Data*

Description

When the discrete data that are observed reflect a smooth strictly increasing or strictly decreasing function, it is often desirable to smooth the data with a strictly monotone function, even though the data themselves may not be monotone due to observational error. An example is when data are collected on the size of a growing organism over time. This function computes such a smoothing function, but, unlike other smoothing functions, for only for one curve at a time. The smoothing function minimizes a weighted error sum of squares criterion. This minimization requires iteration, and therefore is more computationally intensive than normal smoothing.

The monotone smooth is beta[1]+beta[2]*integral(exp(Wfdobj)), where Wfdobj is a functional data object. Since exp(Wfdobj)>0, its integral is monotonically increasing.

Usage

```
smooth.monotone(argvals, y, WfdParobj, wtvec=rep(1,n),
                zmat=NULL, conv=.0001, iterlim=50,
                active=rep(TRUE,nbasis), dbglev=1)
```

Arguments

argvals	Argument value array of length N, where N is the number of observed curve values for each curve. It is assumed that that these argument values are common to all observed curves. If this is not the case, you will need to run this function inside one or more loops, smoothing each curve separately.
y	a vector of data values. This function can only smooth one set of data at a time. Function value array (the values to be fit). If the functional data are univariate, this array will be an N by NCURVE matrix, where N is the number of observed curve values for each curve and NCURVE is the number of curves observed. If the functional data are multivariate, this array will be an N by NCURVE by NVAR matrix, where NVAR the number of functions observed per case. For example, for the gait data, NVAR = 2, since we observe knee and hip angles.
WfdParobj	A functional parameter or fdPar object. This object contains the specifications for the functional data object to be estimated by smoothing the data. See comment lines in function fdPar for details. The functional data object WFD in WFDPAROBJ is used to initialize the optimization process. Its coefficient array contains the starting values for the iterative minimization of mean squared error.
wtvec	a vector of weights to be used in the smoothing.
zmat	a design matrix or a matrix of covariate values that also define the smooth of the data.
conv	a convergence criterion.
iterlim	the maximum number of iterations allowed in the minimization of error sum of squares.
active	a logical vector specifying which coefficients defining $W(t)$ are estimated. Normally, the first coefficient is fixed.
dbglev	either 0, 1, or 2. This controls the amount information printed out on each iteration, with 0 implying no output, 1 intermediate output level, and 2 full output. If either level 1 or 2 is specified, it can be helpful to turn off the output buffering feature of S-PLUS.

Details

The smoothing function $f(argvals)$ is determined by three objects that need to be estimated from the data:

- $W(argvals)$, a functional data object that is first exponentiated and then the result integrated. This is the heart of the monotone smooth. The closer $W(argvals)$ is to zero, the closer the monotone smooth becomes a straight line. The closer $W(argvals)$ becomes a constant, the more the monotone smoother becomes an exponential function. It is assumed that $W(0) = 0.$

- $b0$, an intercept term that determines the value of the smoothing function at $argvals = 0$.
- $b1$, a regression coefficient that determines the slope of the smoothing function at $argvals = 0$.

In addition, it is possible to have the intercept $b0$ depend in turn on the values of one or more covariates through the design matrix `Zmat` as follows: $b0 = Z c$. In this case, the single intercept coefficient is replaced by the regression coefficients in vector c multiplying the design matrix.

Value

an object of class `monfd`, which is a list with the following 5 components:

`Wfdobj`	a functional data object defining function $W(argvals)$ that optimizes the fit to the data of the monotone function that it defines.
`beta`	The regression coefficients b_0 and b_1 for each smoothed curve. If the curves are univariate and ... ZMAT is NULL, BETA is 2 by NCURVE. ... ZMAT has P columns, BETA is P+1 by NCURVE. If the curves are multivariate and ... ZMAT is NULL, BETA is 2 by NCURVE by NVAR. ... ZMAT has P columns, BETA is P+1 by NCURVE by NVAR.
`yhatfd`	A functional data object for the monotone curves that smooth the data. This object is constructed using the basis for WFDOBJ, and this basis may well be too simple to accommodate the curvature in the monotone function that Wfdobjnes. It may be necessary to discard this object and use a richer basis externally to smooth the values defined by beta[1] + beta[2]*eval.monfd(evalarg, Wfdobj).
`Flist`	a named list containing three results for the final converged solution: (1) **f**: the optimal function value being minimized, (2) **grad**: the gradient vector at the optimal solution, and (3) **norm**: the norm of the gradient vector at the optimal solution.
`y2cMap`	For each estimated curve (and variable if functions are multivariate, this is an N by NBASIS matrix containing a linear mapping from data to coefficients that can be used for computing point-wise confidence intervals. If NCURVE = NVAR = 1, a matrix is returned. Otherwise an NCURVE by NVAR list is returned, with each slot containing this mapping.
`argvals`	input `argvals`, possibly modified / clarified by `argcheck`.
`y`	input argument `y`, possibly modified / clarified by `ycheck`.

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

See Also

`smooth.basis`, `smooth.pos`, `smooth.morph`

Examples

```
#  Estimate the acceleration functions for growth curves
#  See the analyses of the growth data.
#  Set up the ages of height measurements for Berkeley data

age <- c( seq(1, 2, 0.25), seq(3, 8, 1), seq(8.5, 18, 0.5))
#  Range of observations
rng <- c(1,18)
#  First set up a basis for monotone smooth
#  We use b-spline basis functions of order 6
#  Knots are positioned at the ages of observation.
norder <- 6
nage   <- length(age)
nbasis <- nage + norder - 2
wbasis <- create.bspline.basis(rng, nbasis, norder, age)
#  starting values for coefficient
cvec0 <- matrix(0,nbasis,1)
Wfd0  <- fd(cvec0, wbasis)
#  set up functional parameter object
Lfdobj    <- 3          #  penalize curvature of acceleration
lambda    <- 10^(-0.5)  #  smoothing parameter
growfdPar <- fdPar(Wfd0, Lfdobj, lambda)
#  Set up wgt vector
wgt   <- rep(1,nage)
#  Smooth the data for the first girl
hgt1 = growth$hgtf[,1]

# conv=0.1 to reduce the compute time,
# required to reduce the test time on CRAN

# delete the test on CRAN because it takes too long

## Not run:
result <- smooth.monotone(age, hgt1, growfdPar, wgt,
                          conv=0.1)
#  Extract the functional data object and regression
#  coefficients
Wfd  <- result$Wfdobj
beta <- result$beta
#  Evaluate the fitted height curve over a fine mesh
agefine <- seq(1,18,len=73)
hgtfine <- beta[1] + beta[2]*eval.monfd(agefine, Wfd)
#  Plot the data and the curve
plot(age, hgt1, type="p")
lines(agefine, hgtfine)
#  Evaluate the acceleration curve
accfine <- beta[2]*eval.monfd(agefine, Wfd, 2)
#  Plot the acceleration curve
plot(agefine, accfine, type="l")
lines(c(1,18),c(0,0),lty=4)
```

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

`smooth.morph` *Estimates a Smooth Warping Function*

Description

This function is nearly identical to `smooth.monotone` but is intended to compute a smooth monotone transformation $h(t)$ of argument t such that $h(0) = 0$ and $h(TRUE) = TRUE$, where t is the upper limit of t. This function is used primarily to register curves.

Usage

```
smooth.morph(x, y, WfdPar, wt=rep(1,nobs),
             conv=.0001, iterlim=20, dbglev=0)
```

Arguments

`x`	a vector of argument values.
`y`	a vector of data values. This function can only smooth one set of data at a time.
`WfdPar`	a functional parameter object that provides an initial value for the coefficients defining function $W(t)$, and a roughness penalty on this function.
`wt`	a vector of weights to be used in the smoothing.
`conv`	a convergence criterion.
`iterlim`	the maximum number of iterations allowed in the minimization of error sum of squares.
`dbglev`	either 0, 1, or 2. This controls the amount information printed out on each iteration, with 0 implying no output, 1 intermediate output level, and 2 full output. If either level 1 or 2 is specified, it can be helpful to turn off the output buffering feature of S-PLUS.

Value

A named list of length 4 containing:

`Wfdobj`	a functional data object defining function $W(x)$ that that optimizes the fit to the data of the monotone function that it defines.
`Flist`	a named list containing three results for the final converged solution: (1) **f**: the optimal function value being minimized, (2) **grad**: the gradient vector at the optimal solution, and (3) **norm**: the norm of the gradient vector at the optimal solution.
`iternum`	the number of iterations.
`iterhist`	a by 5 matrix containing the iteration history.

See Also

`smooth.monotone`, `landmarkreg`, `register.fd`

smooth.pos *Smooth Data with a Positive Function*

Description

A set of data is smoothed with a functional data object that only takes positive values. For example, this function can be used to estimate a smooth variance function from a set of squared residuals. A function $W(t)$ is estimated such that that the smoothing function is $exp[W(t)]$.

Usage

```
smooth.pos(argvals, y, WfdParobj, wtvec=rep(1,n),
           conv=.0001, iterlim=50, dbglev=1)
```

Arguments

argvals	Argument value array of length N, where N is the number of observed curve values for each curve. It is assumed that that these argument values are common to all observed curves. If this is not the case, you will need to run this function inside one or more loops, smoothing each curve separately.
y	Function value array (the values to be fit). If the functional data are univariate, this array will be an N by NCURVE matrix, where N is the number of observed curve values for each curve and NCURVE is the number of curves observed. If the functional data are multivariate, this array will be an N by NCURVE by NVAR matrix, where NVAR the number of functions observed per case. For example, for the gait data, NVAR = 2, since we observe knee and hip angles.
WfdParobj	A functional parameter or fdPar object. This object contains the specifications for the functional data object to be estimated by smoothing the data. See comment lines in function fdPar for details. The functional data object WFD in WFDPAROBJ is used to initialize the optimization process. Its coefficient array contains the starting values for the iterative minimization of mean squared error.
wtvec	a vector of weights to be used in the smoothing.
conv	a convergence criterion.
iterlim	the maximum number of iterations allowed in the minimization of error sum of squares.
dbglev	either 0, 1, or 2. This controls the amount information printed out on each iteration, with 0 implying no output, 1 intermediate output level, and 2 full output. If either level 1 or 2 is specified, it can be helpful to turn off the output buffering feature of S-PLUS.

Value

an object of class posfd, being a list with 4 components:

Wfdobj	a functional data object defining function $W(x)$ that that optimizes the fit to the data of the positive function that it defines.

`Flist`	a named list containing three results for the final converged solution: (1) **f**: the optimal function value being minimized, (2) **grad**: the gradient vector at the optimal solution, and (3) **norm**: the norm of the gradient vector at the optimal solution.
`argvals, y`	the corresponding input arguments

See Also

`smooth.monotone`, `smooth.morph`

Examples

```
smallbasis  <- create.fourier.basis(c(0, 365), 65)
harmaccelLfd365 <- vec2Lfd(c(0,(2*pi/365)^2,0), c(0, 365))

index <- (1:35)[CanadianWeather$place == "Vancouver"]
VanPrec  <- CanadianWeather$dailyAv[,index, "Precipitation.mm"]

lambda    <- 1e4
dayfdPar <- fdPar(smallbasis, harmaccelLfd365, lambda)
smooth.pos(day.5, VanPrec, dayfdPar)
```

---

smooth.sparse.mean *Smooth the mean function of sparse data*

Description

Do a smoothing of the mean function for sparse data that is either given as a list or as a matrix with NAs. The smooth is done by basis expansion with the functional basis "type"; if !(lambda == 0) then the second derivative is penalized (int2Lfd(2)).

Usage

```
smooth.sparse.mean(data, time ,rng = c(0, 1), type = "", nbasis = NULL,
                   knots = NULL, norder = NULL, lambda = NULL)
```

Arguments

`data`	a matrix object or list – If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If y is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. – If it is a list, each element must be a matrix object, the rows correspond to argument values per individual. First column corresponds to time points and followins columns to argument values per variable.
`time`	Array with time points where data was taken. length(time) == ncol(data)

rng	an array of length 2 containing the lower and upper boundaries for the rangeval of argument values
type	Type of basisfd for smoothing the mean estimate function. "bspline", "fourier", "exp", "const" or "mon"
nbasis	An integer variable specifying the number of basis functions
knots	a vector specifying the break points if type == "bspline"
norder	an integer specifying the order of b-splines if type == "bspline"
lambda	a nonnegative real number specifying the amount of smoothing to be applied to the estimated functional parameter

Value

a functional data object containing a smooth of the mean.

---

sparse.list *Creates a list of sparse data from a matrix*

Description

Creates a list with sparse data from a matrix that has NAs. The length of the list will be equal to the number of columns in the data matrix (replications)

Usage

```
sparse.list(data,time)
```

Arguments

data	If the set is supplied as a matrix object, the rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If y is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications.
time	Time points where the observations where taken.

Value

a list with length `dim(data)[2]`. Each element of the list is a matrix with `ncol > 1`. The first column of each element corresponds to the point index per observation.

See Also

cca.fd, pda.fd

sparse.mat *Creates a matrix of sparse data with NAs out of a list*

Description

Creates a matrix or three dimensional array of sparse data with NAs from a list of sparse data. The number of columns of the matrix will be equal to the length of the list (replications)

Usage

```
sparse.mat(datalist)
```

Arguments

datalist	A list object. Each element must be a matrix object where the rows correspond to argument values per individual. First column corresponds to time points and the following columns to argument values per variable.

Value

a matrix or array with ncol = length(datalist). First dimension corresponds to point observations, second dimension corresponds to replications and third dimension corresponds to variables.

StatSciChinese *Statistical Science in Chinese*

Description

(x, y, z) coordinates of the location of the tip of a pen during fifty replications of writing 'Statistical Science' in simplified Chinese at 10 millisecond intervals

Usage

```
data(StatSciChinese)
```

Format

a 3-dimensional array of dimensions (601, 50, 3) containing 601 observations of (x, y, z) coordinates of the tip of a pen at 2.5 millisecond intervals for each of 50 repititions of writing 'Statistical Science' in simplified Chinese.

Details

Xiaochun Li wrote 'Statistical Science' in simplified Chinese 50 times. An infra-red detecting tablet was attached to the tip of the pen, and a wall-mounted set of three cameras recorded its position 400 times per second with an error level of about 0.5 millimeters. Each sample required about 6 seconds to produce, and for simplicity, time was normalized to this interval for all 50 records. The script requires 50 strokes, with an average time of 120 milliseconds per stroke. These raw data were shifted and rotated so the numbers more accurately reflected `x` and `y` coordinates relative to the drawn characters plus vertical distance from the paper.

References

Ramsay, James O. (2000) Functional Components of Variation in Handwriting, *Journal of the American Statistical Association*, 95, 9-15.

Examples

```
data(StatSciChinese)

i <- 3
StatSci1 <- StatSciChinese[, i, ]
# Where does the pen leave the paper?
plot(StatSci1[, 3], type='l')
thresh <- quantile(StatSci1[, 3], .8)
abline(h=thresh)

sel1 <- (StatSci1[, 3] < thresh)
StatSci1[!sel1, 1:2] <- NA
plot(StatSci1[, 1:2], type='l')

mark <- seq(1, 601, 12)
points(StatSci1[mark, 1], StatSci1[mark, 2])
```

---

sum.fd *Sum of Functional Data*

Description

Evaluate the sum of a set of functions in a functional data object.

Usage

```
## S3 method for class 'fd'
sum(..., na.rm)
```

Arguments

`...`	a functional data object to sum.
`na.rm`	Not used.

Value

a functional data object with a single replication that contains the sum of the functions in the object `fd`.

See Also

`mean.fd`, `std.fd`, `stddev.fd`, `center.fd`

`summary.basisfd` *Summarize a Functional Data Object*

Description

Provide a compact summary of the characteristics of a functional data object.

Usage

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

Arguments

`object`	a functional data object (i.e., of class 'basisfd').
`...`	Other arguments to match generic

Value

a displayed summary of the bivariate functional data object.

References

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

Ramsay, James O., and Silverman, Bernard W. (2002), *Applied Functional Data Analysis*, Springer, New York.

summary.bifd *Summarize a Bivariate Functional Data Object*

Description

Provide a compact summary of the characteristics of a bivariate functional data object.

Usage

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

Arguments

object	a bivariate functional data object.
...	Other arguments to match the generic function for 'summary'

Value

a displayed summary of the bivariate functional data object.

See Also

summary,

summary.fd *Summarize a Functional Data Object*

Description

Provide a compact summary of the characteristics of a functional data object.

Usage

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

Arguments

object	a functional data object.
...	Other arguments to match the generic for 'summary'

Value

a displayed summary of the functional data object.

See Also

summary,

---

summary.fdPar *Summarize a Functional Parameter Object*

Description

Provide a compact summary of the characteristics of a functional parameter object (i.e., class 'fdPar').

Usage

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

Arguments

object	a functional parameter object.
...	Other arguments to match the generic 'summary' function

Value

a displayed summary of the functional parameter object.

See Also

summary,

summary.Lfd *Summarize a Linear Differential Operator Object*

Description

Provide a compact summary of the characteristics of a linear differential operator object.

Usage

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

Arguments

object	a linear differential operator object.
...	Other arguments to match the generic 'summary' function

Value

a displayed summary of the linear differential operator object.

See Also

summary,

symsolve *solve(A, B) where A is symmetric*

Description

Solve A X = B for X where A is symmetric

Usage

```
symsolve(Asym, Bmat)
```

Arguments

Asym	a symmetric matrix
Bmat	a square matrix of dimensions matching Asym

Value

A square matrix of the same dimenions as Asym and Bmat

See Also

solve

Examples

```
A <- matrix(c(2,1,1,2), 2)
Ainv <- symsolve(A, diag(2))
```

---

tperm.fd *Permutation t-test for two groups of functional data objects.*

Description

tperm.fd creates a null distribution for a test of no difference between two groups of functional data objects.

Usage

```
tperm.fd(x1fd, x2fd, nperm=200, q=0.05, argvals=NULL, plotres=TRUE, ...)
```

Arguments

x1fd	a functional data object giving the first group of functional observations.
x2fd	a functional data object giving the second group of functional observations.
nperm	number of permutations to use in creating the null distribution.
q	Critical upper-tail quantile of the null distribution to compare to the observed t-statistic.
argvals	If yfdPar is a fd object, the points at which to evaluate the point-wise t-statistic.
plotres	Argument to plot a visual display of the null distribution displaying the 1-qth quantile and observed t-statistic.
...	Additional plotting arguments that can be used with plot.

Details

The usual t-statistic is calculated pointwise and the test based on the maximal value. If argvals is not specified, it defaults to 101 equally-spaced points on the range of yfdPar.

Value

A list with the following components:

pval	the observed p-value of the permutation test.
qval	the qth quantile of the null distribution.
Tobs	the observed maximal t-statistic.

Tnull	a vector of length nperm giving the observed values of the permutation distribution.
Tvals	the pointwise values of the observed t-statistic.
Tnullvals	the pointwise values of of the permutation observations.
pvals.pts	pointwise p-values of the t-statistic.
qvals.pts	pointwise qth quantiles of the null distribution
argvals	argument values for evaluating the F-statistic if yfdParis a functional data object.

Side Effects

a plot of the functional observations

Source

Ramsay, James O., and Silverman, Bernard W. (2006), *Functional Data Analysis, 2nd ed.*, Springer, New York.

See Also

fRegress Fstat.fd

Examples

```
# This tests the difference between boys and girls heights in the
# Berkeley growth data.

# First set up a basis system to hold the smooths

knots  <- growth$age
norder <- 6
nbasis <- length(knots) + norder - 2
hgtbasis <- create.bspline.basis(range(knots), nbasis, norder, knots)

# Now smooth with a fourth-derivative penalty and a very small smoothing
# parameter

Lfdobj <- 4
lambda <- 1e-2
growfdPar <- fdPar(hgtbasis, Lfdobj, lambda)

hgtmfd <- smooth.basis(growth$age, growth$hgtm, growfdPar)$fd
hgtffd <- smooth.basis(growth$age, growth$hgtf, growfdPar)$fd

# Call tperm.fd

tres <- tperm.fd(hgtmfd,hgtffd)
```

trapzmat *Approximate the functional inner product of two functional data objects using the trapezpoidal rule over a fine mesh of value.*

Description

The first two arguments are matrices of values of two functional data objects of the same fine mesh of argument values. The mesh is assumed to be equally spaced.

Usage

```
trapzmat(X,Y,delta=1,wt=rep(1,n))
```

Arguments

X	The first matrix of functional data object values over a fine mesh.
Y	The second matrix of functional data object values over the same fine mesh.
delta	The difference between adjacent argument values, assumed to be a constant.
wt	An optional vector of weights for the products of pairs of argument values, otherwise assumed to be all ones.

Value

A matrix of approximate inner products. The number of rows the number of columns of X and the number of columns is the number of columns of Y.

See Also

inprod, inprod.bspline

var.fd *Variance, Covariance, and Correlation Surfaces for Functional Data Object(s)*

Description

Compute variance, covariance, and / or correlation functions for functional data.

These are two-argument functions and therefore define surfaces. If only one functional data object is supplied, its variance or correlation function is computed. If two are supplied, the covariance or correlation function between them is computed.

Usage

```
var.fd(fdobj1, fdobj2=fdobj1)
```

Arguments

fdobj1, fdobj2	a functional data object.

Details

a two-argument or bivariate functional data object representing the variance, covariance or correlation surface for a single functional data object or the covariance between two functional data objects or between different variables in a multivariate functional data object.

Value

An list object of class bifd with the following components:

coefs	the coefficient array with dimensions fdobj1[["basis"]][["nbasis"]] by fdobj2[["basis"]][["nbasis"]] giving the coefficients of the covariance matrix in terms of the bases used by fdobj1 and fdobj2.
sbasis	fdobj1[["basis"]]
tbasis	fdobj2[["basis"]]
bifdnames	dimnames list for a 4-dimensional 'coefs' array. If length(dim(coefs)) is only 2 or 3, the last 2 or 1 component of bifdnames is not used with dimnames(coefs).

Examples below illustrate this structure in simple cases.

See Also

mean.fd, sd.fd, std.fd stdev.fd

Examples

```
##
## Example with 2 different bases
##
daybasis3 <- create.fourier.basis(c(0, 365))
daybasis5 <- create.fourier.basis(c(0, 365), 5)
tempfd3 <- with(CanadianWeather, smooth.basis(day.5,
       dailyAv[,,"Temperature.C"],
       daybasis3, fdnames=list("Day", "Station", "Deg C"))$fd )
precfd5 <- with(CanadianWeather, smooth.basis(day.5,
       dailyAv[,,"log10precip"],
       daybasis5, fdnames=list("Day", "Station", "Deg C"))$fd )

# Compare with structure described above under 'value':
str(tempPrecVar3.5 <- var.fd(tempfd3, precfd5))

##
## The following produces contour and perspective plots
##

# Evaluate at a 53 by 53 grid for plotting
```

```
daybasis65 <- create.fourier.basis(rangeval=c(0, 365), nbasis=65)

daytempfd <- with(CanadianWeather, smooth.basis(day.5,
       dailyAv[,,"Temperature.C"],
       daybasis65, fdnames=list("Day", "Station", "Deg C"))$fd )
str(tempvarbifd <- var.fd(daytempfd))

str(tempvarmat  <- eval.bifd(weeks,weeks,tempvarbifd))
# dim(tempvarmat)= c(53, 53)

op <- par(mfrow=c(1,2), pty="s")
#contour(tempvarmat, xlab="Days", ylab="Days")
contour(weeks, weeks, tempvarmat,
        xlab="Daily Average Temperature",
        ylab="Daily Average Temperature",
        main=paste("Variance function across locations\n",
          "for Canadian Anual Temperature Cycle"),
        cex.main=0.8, axes=FALSE)
axisIntervals(1, atTick1=seq(0, 365, length=5), atTick2=NA,
            atLabels=seq(1/8, 1, 1/4)*365,
            labels=paste("Q", 1:4) )
axisIntervals(2, atTick1=seq(0, 365, length=5), atTick2=NA,
            atLabels=seq(1/8, 1, 1/4)*365,
            labels=paste("Q", 1:4) )
persp(weeks, weeks, tempvarmat,
      xlab="Days", ylab="Days", zlab="Covariance")
mtext("Temperature Covariance", line=-4, outer=TRUE)
par(op)
```

`varmx` *Rotate a Matrix of Component Loadings using the VARIMAX Criterion*

Description

The matrix being rotated contains the values of the component functional data objects computed in either a principal components analysis or a canonical correlation analysis. The values are computed over a fine mesh of argument values.

Usage

```
varmx(amat, normalize=FALSE)
```

Arguments

`amat`	the matrix to be rotated. The number of rows is equal to the number of argument values `nx` used in a fine mesh. The number of columns is the number of components to be rotated.

normalize — either TRUE or FALSE. If TRUE, the columns of amat are normalized prior to computing the rotation matrix. However, this is seldom needed for functional data.

Details

The VARIMAX criterion is the variance of the squared component values. As this criterion is maximized with respect to a rotation of the space spanned by the columns of the matrix, the squared loadings tend more and more to be either near 0 or near 1, and this tends to help with the process of labelling or interpreting the rotated matrix.

Value

a square rotation matrix of order equal to the number of components that are rotated. A rotation matrix T has that property that $T'T = TT' = I$.

See Also

varmx.pca.fd, varmx.cca.fd

---

varmx.cca.fd *Rotation of Functional Canonical Components with VARIMAX*

Description

Results of canonical correlation analysis are often easier to interpret if they are rotated. Among the many possible ways in which this rotation can be defined, the VARIMAX criterion seems to give satisfactory results most of the time.

Usage

```
varmx.cca.fd(ccafd, nx=201)
```

Arguments

ccafd — an object of class "cca.fd" that is produced by function cca.fd.

nx — the number of points in a fine mesh of points that is required to approximate canonical variable functional data objects.

Value

a rotated version of argument cca.fd.

See Also

varmx, varmx.pca.fd

varmx.pca.fd *Rotation of Functional Principal Components with VARIMAX Criterion*

Description

Principal components are often easier to interpret if they are rotated. Among the many possible ways in which this rotation can be defined, the VARIMAX criterion seems to give satisfactory results most of the time.

Usage

```
varmx.pca.fd(pcafd, nharm=scoresd[2], nx=501)
```

Arguments

pcafd	an object of class pca.fd that is produced by function pca.fd.
nharm	the number of harmonics or principal components to be rotated.
nx	the number of argument values in a fine mesh used to define the harmonics to be rotated.

Value

a rotated principal components analysis object of class pca.fd.

See Also

varmx, varmx.cca.fd

vec2Lfd *Make a Linear Differential Operator Object from a Vector*

Description

A linear differential operator object of order m is constructed from the number in a vector of length m.

Usage

```
vec2Lfd(bwtvec, rangeval=c(0,1))
```

Arguments

bwtvec	a vector of coefficients to define the linear differential operator object
rangeval	a vector of length 2 specifying the range over which the operator is defined

Value

a linear differential operator object

See Also

int2Lfd, Lfd

Examples

```
#  define the harmonic acceleration operator used in the
#  analysis of the daily temperature data
Lcoef <- c(0,(2*pi/365)^2,0)
harmaccelLfd <- vec2Lfd(Lcoef, c(0,365))

hmat <- vec2Lfd(matrix(Lcoef, 1), c(0, 365))


all.equal(harmaccelLfd, hmat)
```

---

wtcheck *Check a vector of weights*

Description

Throws an error if wtvec is not n positive numbers, and return wtvec (stripped of any dim atrribute)

Usage

```
wtcheck(n, wtvec)
```

Arguments

n	the required length of wtvec
wtvec	an object to be checked

Value

a vector of n positive numbers

Examples

```
wtcheck(3, 1:3)

wtcheck(2, matrix(1:2, 2))
```

`zerofind` *Does the range of the input contain 0?*

Description

Returns TRUE if range of the argument includes 0 and FALSE if not.

Usage

```
zerofind(fmat)
```

Arguments

`fmat`	An object from which 'range' returns two numbers.

Value

A logical value TRUE or FALSE.

See Also

`range`

Examples

```
zerofind(1:5)
# FALSE
zerofind(0:3)
# TRUE
```

Index