Package ‘pMEM’

September 30, 2024

Type Package

Title Predictive Moran's Eigenvector Maps

Version 0.1-1

Date 2024-09-26

Encoding UTF-8

Description Calculation of Predictive Moran's eigenvector maps (pMEM), as defined by Guénard and Legendre (In Press) ``Spatially-explicit predictions using spatial eigenvector maps'' <doi:10.5281/zenodo.13356457>. Methods in Ecology and Evolution. This method enables scientists to predict the values of spatially-structured environmental variables. Multiple types of pMEM are defined, each one implemented on the basis of spatial weighting function taking a range parameter, and sometimes also a shape parameter. The code's modular nature enables programers to implement new pMEM by defining new spatial weighting functions.

Depends R (>= 3.5.0), sf

Suggests knitr, xfun, magrittr, glmnet

Imports Rcpp (>= 1.0.11)

License GPL-3

LazyLoad yes

NeedsCompilation yes

LinkingTo Rcpp

VignetteBuilder knitr

Repository CRAN

RoxygenNote 7.3.1

Author Guillaume Guénard [aut, cre] (<`https://orcid.org/0000-0003-0761-3072`>), Pierre Legendre [ctb] (<`https://orcid.org/0000-0002-3838-3305`>)

Maintainer Guillaume Guénard <`guillaume.guenard@umontreal.ca`>

Date/Publication 2024-09-30 10:20:04 UTC

Contents

`genDistMetric` *Distance Metric Function Generator*

Description

Function `genDistMetric` generates a distance metric function, which calculate pairwise distance values on the basis of given arguments.

Usage

```
genDistMetric(delta, theta = 0)
```

Arguments

`delta`	Optional: the asymmetry parameter of the distance metric.
`theta`	The influence angle of the distance metric (default: 0).

Details

When argument `delta` is omitted, the returned function calculates the Euclidean distance, whereas when is it provided with a value, the returned function calculates a complex-values distance metric whose modulus is the Euclidean distance and argument is related with `delta`. For one-dimensional data (transects), the argument is ±`delta`, with negative value for every second object located before the first, and positive values for every second object located after the first. For two-dimensional data, the value of the argument is the cosine of the angular difference between the angle of the line traversing the two points, defined in the direction going from the first to the second point, and the influence angle. In any case, the argument of the distance metric from a point A to a point B has the opposite sign as that of the distance metric from point B to point A. Therefore, the pairwise distance matrix is Hermitian and, as such, has eigenvalues that are strictly real-valued.

It is noteworthy that `genDistMetric` does not calculate the distances directly, as is the most common workflow, but generate a function that calculate the metric on the basis of the specified parameters (arguments `delta` and `theta`). The values of these parameters are embedded together with distance metric function in the returned object's namespace and can only be changed by generating a new function.

Value

A two-argument function (x and y) calculating the distances between the rows of x and the rows of y. When y is omitted, the pairwise distances between the rows of x are calculated instead.

Author(s)

Guillaume Guénard [aut, cre] (ORCID: <https://orcid.org/0000-0003-0761-3072>), Pierre Legendre [ctb] (ORCID: <https://orcid.org/0000-0002-3838-3305>)

Examples

```
 ## A five point equidistant transect:
n <- 5
x <- (n - 1)*seq(0, 1, length.out=n)

## The symmetric (Euclidean metric) function is obtained by calling
## the generator function with not arguments as follows:
mSym <- genDistMetric()

## The pairwise symmetric metric between the rows of x:
mSym(x)

## A second set of points in the same range as the previous one, but at a
## distance of 0.05 from one another:
xx <- (n - 1)*seq(0, 1, 0.05)

## The same metrix, but between x and xx:
mSym(x,xx)

## The asymmetric function with a delta of 0.2:
mAsy <- genDistMetric(0.2)

## The pairwise asymmetric metric between the rows of x:
mAsy(x)

## The same metrix, but between x and xx:
mAsy(x,xx)
```

genDWF *Distance Weighting Function Generator*

Description

Function genDWF generates a distance weighting function on the basis of given arguments.

Usage

```
genDWF(
  fun = c("linear", "power", "hyperbolic", "spherical", "exponential", "Gaussian",
    "hole_effect"),
  range,
  shape = 1
)
```

Arguments

`fun`	The function describing the kind of distance weighting: one of 'linear', 'power', 'hyperbolic', 'spherical', 'exponential', 'Gaussian', 'hole_effect', or an unambiguous abbreviation of one of them.
`range`	A single numeric value giving the range of the distance weighting function (see details).
`shape`	A single (numeric) shape parameter used by functions 'power' or 'hyperbolic' (ignored by the other functions).

Details

All distance weighting function return the value 1 (or 1+0i) for a distance or 0 (or 0+0i). For functions 'linear', 'power', 'hyperbolic', and 'spherical', argument `range` corresponds to the distance above which weights have a constant value of 0 (or 0+0i). Functions 'exponential', 'Gaussian', and 'hole_effect' have no definite value beyond `d > range`, but collapse asymptotically toward 0 either monotonically ('exponential' and 'Gaussian'), or following dampened oscillations about the value 0 ('hole_effect').

Value

A single-argument function (d) transforming the distances into weights. This function returns a matrix when the distances are provided as a matrix and a numeric vector when the distances are provided as a numeric vector.

Author(s)

Guillaume Guénard [aut, cre] (ORCID: <https://orcid.org/0000-0003-0761-3072>), Pierre Legendre [ctb] (ORCID: <https://orcid.org/0000-0002-3838-3305>)

Examples

```
 ## Show examples of distance weighting functions (real-valued)

## Custom display function for this example (real-values):
plotDWF <- function(d, w, label, ylim = c(0,1)) {
  plot(x = d, y = w[,1L], type = "l", ylim = ylim, las = 1L,
       xlab = "Distance", ylab = "", cex.axis = 2, cex.lab=2, lwd = 2)
  lines(x = d, y = w[,2L], col = "red", lwd = 2)
  lines(x = d, y = w[,3L], col = "blue", lwd = 2)
  text(x = 2.5, y = 0.8, label = label, adj = 0, cex = 2)
}

## A set of distances from which to show the corresponding weights:
d <- seq(0,5,0.001)

## Graphical parameters for all the figures:
tmp <- par(no.readonly = TRUE)
par(mar=c(5.1,5.1,0.6,0.6))

## Shapes of the seven distance weighting functions implemented in this
## package for real-valued distances.

## The linear function:

cbind(
  genDWF(fun = "linear", range = 1)(d),
  genDWF(fun = "linear", range = 0.5)(d),
  genDWF(fun = "linear", range = 2)(d)
) -> w

plotDWF(d, w, label="Linear")

## The power function:

cbind(
  genDWF(fun = "power", range = 1, shape = 1)(d),
  genDWF(fun = "power", range = 2, shape = 0.5)(d),
  genDWF(fun = "power", range = 3, shape = 0.5)(d)
) -> w

plotDWF(d, w, label="Power")

## The hyperbolic function:

cbind(
  genDWF(fun = "hyperbolic", range = 1, shape = 1)(d),
  genDWF(fun = "hyperbolic", range = 2, shape = 0.5)(d),
  genDWF(fun = "hyperbolic", range = 0.5, shape = 2)(d)
) -> w

plotDWF(d, w, label="Hyperbolic")

## The spherical function:

cbind(
  genDWF(fun = "spherical", range = 1)(d),
  genDWF(fun = "spherical", range = 0.5)(d),
  genDWF(fun = "spherical", range = 2)(d)
) -> w

plotDWF(d, w, label="Spherical")

## The exponential function:
```

```
cbind(
  genDWF(fun = "exponential", range = 1)(d),
  genDWF(fun = "exponential", range = 0.5)(d),
  genDWF(fun = "exponential", range = 2)(d)
) -> w

plotDWF(d, w, label="Exponential")

## The Gaussian function:

cbind(
  genDWF(fun = "Gaussian", range = 1)(d),
  genDWF(fun = "Gaussian", range = 0.5)(d),
  genDWF(fun = "Gaussian", range = 2)(d)
) -> w

plotDWF(d, w, label="Gaussian")

## The "hole effect" (cardinal sine) function:

cbind(
  genDWF(fun = "hole_effect", range = 1)(d),
  genDWF(fun = "hole_effect", range = 0.5)(d),
  genDWF(fun = "hole_effect", range = 2)(d)
) -> w

plotDWF(d, w, label="Hole effect", ylim=c(-0.2,1))


## Custom display function for this example (complex-values):
plotDWFcplx <- function(d, w, label, ylim) {
  plot(x = Mod(d), y = Re(w[,1L]), type = "l", ylim = ylim, las = 1L,
  xlab = "Distance", ylab = "", cex.axis = 2, cex.lab=2, lwd = 2, lty = 2L)
  lines(x = Mod(d), y = Im(w[,1L]), lwd = 2, lty=3L)
  lines(x = Mod(d), y = Re(w[,2L]), col = "red", lwd = 2, lty = 2L)
  lines(x = Mod(d), y = Im(w[,2L]), col = "red", lwd = 2, lty = 3L)
  lines(x = Mod(d), y = Re(w[,3L]), col = "blue", lwd = 2, lty = 2L)
  lines(x = Mod(d), y = Im(w[,3L]), col = "blue", lwd = 2, lty = 3L)
  text(x = 2.5, y = 0.8, label = label, adj = 0, cex = 2)
  invisible(NULL)
}

## Generated the asymmetric distance metrics for a one-dimensional transect
## and a delta of pi/8 (0.39...):
dd <- complex(modulus=seq(0,5,0.001), argument = pi/8)

## Shapes of the seven distance weighting functions implemented in this
## package for complex-valued distances.

## The linear function:

cbind(
  genDWF(fun = "linear", range = 1)(dd),
  genDWF(fun = "linear", range = 0.5)(dd),
  genDWF(fun = "linear", range = 2)(dd)
) -> ww

plotDWFcplx(dd, ww, label="Linear", ylim=c(-0.4,1))

## The power function:

cbind(
  genDWF(fun = "power", range = 1, shape = 1)(dd),
  genDWF(fun = "power", range = 2, shape = 0.5)(dd),
  genDWF(fun = "power", range = 3, shape = 0.5)(dd)
) -> ww

plotDWFcplx(dd, ww, label="Power", ylim=c(-0.4,1))

## The hyperbolic down function:

cbind(
  genDWF(fun = "hyperbolic", range = 1, shape = 1)(dd),
  genDWF(fun = "hyperbolic", range = 2, shape = 0.5)(dd),
  genDWF(fun = "hyperbolic", range = 0.5, shape = 2)(dd)
) -> ww

plotDWFcplx(dd, ww, label="Hyperbolic", ylim=c(-0.4,1))

## The spherical function:

cbind(
  genDWF(fun = "spherical", range = 1)(dd),
  genDWF(fun = "spherical", range = 0.5)(dd),
  genDWF(fun = "spherical", range = 2)(dd)
) -> ww

plotDWFcplx(dd, ww, label="Spherical", ylim=c(-0.4,1))

## The exponential function:

cbind(
  genDWF(fun = "exponential", range = 1)(dd),
  genDWF(fun = "exponential", range = 0.5)(dd),
  genDWF(fun = "exponential", range = 2)(dd)
) -> ww

plotDWFcplx(dd, ww, label="Exponential", ylim=c(-0.4,1))

## The Gaussian function:

cbind(
  genDWF(fun = "Gaussian", range = 1)(dd),
  genDWF(fun = "Gaussian", range = 0.5)(dd),
  genDWF(fun = "Gaussian", range = 2)(dd)
```

```
) -> ww

plotDWFcplx(dd, ww, label="Gaussian", ylim=c(-0.4,1))

## The "hole effect" (cardinal sine) function:

cbind(
  genDWF(fun = "hole_effect", range = 1)(dd),
  genDWF(fun = "hole_effect", range = 0.5)(dd),
  genDWF(fun = "hole_effect", range = 2)(dd)
) -> ww

plotDWFcplx(dd, ww, label="Hole effect", ylim=c(-0.3,1.1))

## Restore previous graphical parameters:
par(tmp)
```

---

geoMite — *Borcard's Oribatid Mite Data Set - Geographic Information System Version -*

Description

Oribatid mite community data in a peat bog surrounding Lac Geai, QC, Canada

Usage

```
data(geoMite)
```

Format

A list with five `sf` data frames:

core A data frame with 70 rows (peat cores) containing point geometries and 46 fields containing values of environmental variables at the locations of the cores as well as the number of individuals of one of 35 Oribatid species observed in the cores (see details).

water A data frame with three rows containing polygon geometries and a single field: a `factor` variable named "Type" and specifying whether the polygon represents open water (value == "Water") or flooded areas (value == "Flooded") at the time of sampling.

substrate A data frame with 13 rows containing polygon geometries and seven fields. The first field is a `factor` that specifies one of six substrate classes (see details) and the remaining six fields are binary variables for each of these six substrate classes that take the value 1 when the polygon is of the class being represented by the that variable and otherwise take the value 0.

shrub A data frame with four rows containing polygon geometries and a single field: an `ordered` variable named "Type" and specifying whether the polygon represents areas with no shrub (value == "None"), a few shrubs (value == "Few"), or many shrubs (value == "Many").

topo A data frame with four rows containing polygon geometries and a single field: a `factor` variable named "Type" and specifying the type of peat micro-topography. There are two such types: "Blanket" (flat area) and "Hummock" (raised bumps).

Details

Fields of the point geometry (`geoMite$core`) are:

SubsDens Substrate density (g/L).

WatrCont Water content of the peat (g/L)

Substrate-prefixed Six binary variables describing the substrate(s) from which the peat core samples were collected. Further details are given below.

Shrub A three-level `ordered` factor describing the presence and abundance of shrubs (mainly Ericaceae) on the peat surface.

Topo A two-level factor describing the microtopography of the peat mat.

Flooded A binary variable specifying whether the area in which the core was sampled was flooded at the time of sampling.

Species-prefixed Counts of one of 35 Oribatid species identified purely on the basis of their morphology.

The types of substrates are described as follows:

Sphagn1 Sphagnum magellanicum (with a majority of S. rubellum).

Sphagn2 Sphagnum rubellum.

Sphagn3 Sphagnum nemoreum (with a minority of S. angustifolium).

Sphagn4 Sphagnum rubellum and S. magellanicum in equal parts.

Litter Ligneous litter.

Barepeat Bare peat.

These types are not mutually exclusive categories: cores were sometimes taken at the boundary between two or more substrate types and thus belong to many of these categories.

As stated earlier, identification of the Oribatid species was carried out solely on the basis of their morphology as little is known on the ecology of these small animals.

Geometries in geoMite$water, geoMite$substrate, geoMite$shrub, and geoMite$topo were generated by outlining images from Fig. 1 in Borcard et al. (1994) using a square grid with a resolution of about 10 mm. Because of the inaccuracy to the available printed document the actual resolution is probably inferior (i.e., 10 cm in both the x and y direction).

Orientation: the X coordinates corresponds to distances going from the edge of the water to the edge of the forest. The Y coordinates correspond the distances along the lake's shore.

Author(s)

Daniel Borcard, <daniel.borcard@umontreal.ca> and Pierre Legendre <pierre.legendre@umontreal.ca>

References

Borcard, D. and Legendre, P. 1994. Environmental Control and Spatial Structure in Ecological Communities: An Example Using Oribatid Mites (Acari, Oribatei). Environ. Ecol. Stat. 1(1): 37-61 doi:10.1007/BF00714196

Borcard, D., Legendre, P., and Drapeau, P. 1992. Partialling out the spatial component of ecological variation. Ecology, 73, 1045-1055. doi:10.2307/1940179

Borcard, D.; Legendre, P.; and Gillet, F. 2018. Numerical Ecology with R (2nd Edition) Sprigner, Cham, Switzerland. doi:10.1007/9783319714042

See Also

Data set `oribatid` from package `ade4`, which is another version of this data set.

Examples

```
data(geoMite)

attach(geoMite)

## Color definitions:
col <- list()
col[["substrate"]] <- c(Sphagn1 = "#00ff00", Sphagn2 = "#fffb00",
                        Sphagn3 = "#774b00", Sphagn4 = "#ff8400",
                        Litter = "#ee00ff", Barepeat = "#ff0004")
col[["water"]] <- c(Water = "#008cff", Flooded = "#ffffff00",
                    core = "#000000ff")
col[["shrub"]] <- c(None = "#dfdfdf", Few = "#a7a7a7", Many = "#5c5c5c")
col[["topo"]] <- c(Blanket = "#74cd00", Hummock = "#bc9d00")

## Graphical paramters:
p <- par(no.readonly = TRUE)
par(mar=c(0,0,1,0), mfrow=c(1L,4L))

## Substrate:
plot(st_geometry(substrate), col=col[["substrate"]][substrate$Type],
     main="Substrate")
plot(st_geometry(water[1L,]), col=col[["water"]][water[1L,]$Type], add=TRUE)
plot(st_geometry(water[-1L,]), col=col[["water"]][water[-1L,]$Type], lty=3L,
     add=TRUE)
plot(st_geometry(core), pch = 21L, bg = "black", add=TRUE)

## Shrubs:
plot(st_geometry(shrub), col = col[["shrub"]][shrub$Type], main="Shrubs")
plot(st_geometry(water[1L,]), col=col[["water"]][water[1L,]$Type], add=TRUE)
plot(st_geometry(water[-1L,]), col=col[["water"]][water[-1L,]$Type], lty=3L,
     add=TRUE)
plot(st_geometry(core), pch = 21L, bg = "black", add=TRUE)

## Topograghy:
plot(st_geometry(topo), col = col[["topo"]][topo$Type], main="Topography")
plot(st_geometry(water[1L,]), col=col[["water"]][water[1L,]$Type], add=TRUE)
plot(st_geometry(water[-1L,]), col=col[["water"]][water[-1L,]$Type], lty=3L,
     add=TRUE)
plot(st_geometry(core), pch = 21L, bg = "black", add=TRUE)

## Legends:
plot(NA, xlim=c(0,1), ylim=c(0,1), axes = FALSE)
legend(x=0, y=0.9, pch=22L, pt.cex = 2.5, pt.bg=col[["substrate"]],
       box.lwd = 0, legend=names(col[["substrate"]]), title="Substrate")
legend(x=-0.025, y=0.6, pch=c(22L,NA,21L), pt.cex = c(2.5,NA,1),
       pt.bg=col[["water"]], box.lwd = 0, lty = c(0L,3L,NA),
       legend=c("Open water","Flooded area","Peat core"))
legend(x=0, y=0.4, pch=22L, pt.cex = 2.5, pt.bg=col[["shrub"]], box.lwd = 0,
       legend=names(col[["shrub"]]), title="Shrubs")
legend(x=0, y=0.2, pch=22L, pt.cex = 2.5, pt.bg=col[["topo"]], box.lwd = 0,
       legend=names(col[["topo"]]), title="Topography")

### Display the species counts

## Get the species names:
unlist(
  lapply(
    strsplit(colnames(core),".",fixed=TRUE),
    function(x) if(x[1L] == "Species") x[2L] else NULL
  )
) -> spnms

## See the maximum counts for all the species
apply(st_drop_geometry(core[,paste("Species",spnms,sep=".")]),2L,max)

## Species selection to display:
sel <- c("Brachysp","Hoplcfpa","Oppinova","Limncfci","Limncfru")

## Range of counts to display:
rng <- log1p(c(0,1000))

colmap <- grey(seq(1,0,length.out=256L))

## Update the graphical parameters for this example
par(mar=c(0,0,2,0), mfrow=c(1L,length(sel) + 1L))

## Display each species in the selection over the substrate map
for(sp in sel) {
  plot(st_geometry(substrate), col=col[["substrate"]][substrate$Type],
       main=sp)
  plot(st_geometry(core), pch=21L, add = TRUE, cex=1.5,
       bg=colmap[1 + 255*log1p(core[[paste("Species",sp,sep=".")]])/rng[2L]])
}

## Display the colour chart for the species counts:
par(mar=c(2,7,3,1))
image(z=matrix(seq(0,log1p(1000),length.out=256L),1L,256L), col=colmap,
      xaxt="n", yaxt="n", y=seq(0,log1p(1000),length.out=256L), xlab="",
      cex.lab = 1.5,
      ylab=expression(paste("Counts by species (",ind~core^{-1},")")))
axis(2L, at=log1p(c(0,1,3,10,30,100,300,1000)), cex.axis = 1.5,
     label=c(0,1,3,10,30,100,300,1000))

## Restore graphical parameters:
par(p)
```

getMinMSE *Simple Orthogonal Term Selection Regression*

Description

A simple orthogonal term selection regression function for minimizing out of the sample mean squared error (MSE).

Usage

```
getMinMSE(U, y, Up, yy, complete = TRUE)
```

Arguments

`U`	A matrix of spatial eigenvectors to be used as training data.
`y`	A numeric vector containing a single response variable to be used as training labels.
`Up`	A numeric matrix of spatial eigenvector scores to be used as testing data.
`yy`	A numeric vector containing a single response variable to be used as testing labels.
`complete`	A boolean specifying whether to return the complete data of the selection procedure (`complete=TRUE`; the default) or only the resulting mean square error and beta threshold (`complete=FALSE`).

Details

This function allows one to calculate a simple model, involving only the spatial eigenvectors and a single response variable. The coefficients are estimated on a training data set; the ones that are retained are chosen on the basis of minimizing the mean squared error on the testing data set. As such, both a training and a testing data set are mandatory for this procedure to be carried on. The procedure goes as follows:

1. The regression coefficients are calculated as the cross-product `b = t(U)y` and are sorted in decreasing order of their absolute values.
2. The mean of the training labels is calculated, then the residuals training labels are calculated, and the null MSE is calculated from the testing labels.
3. For each regression coefficient, the partial predicted value is calculated and subtracted from the testing labels, and the new MSE value is calculated.

4. The minimum MSE value is identified.
5. The regression coefficients are standardized ans squared and the results are returned.

For this procedure, the training data must be are orthonormal, a condition met design by spatial eigenvectors.

Value

If `complete = TRUE`, a list with the following members:

betasq The squared standardized regression coefficients.

nullmse The null MSE value: the mean squared out of the sample error using only the mean of the training labels as the prediction.

mse The mean squared error of each incremental model.

ord The order of the squared standardized regression coefficients.

wh The index of the model with the smallest mean squared error. The value 0 means that the smallest MSE is the null MSE.

If `complete = FALSE` a two element list with the following members:

betasq The squared standardized regression coefficient associated with the minimum means squared error value.

mse The minimum means squared error value.

Author(s)

Guillaume Guénard [aut, cre] (ORCID: <https://orcid.org/0000-0003-0761-3072>), Pierre Legendre [ctb] (ORCID: <https://orcid.org/0000-0002-3838-3305>)

Maintainer: Guillaume Guénard <guillaume.guenard@umontreal.ca>

Examples

```
## Loading the 'salmon' dataset
data("salmon")
seq(1,nrow(salmon),3) -> test      # Indices of the testing set.
(1:nrow(salmon))[-test] -> train   # Indices of the training set.

## A set of locations located 1 m apart:
xx <- seq(min(salmon$Position) - 20, max(salmon$Position) + 20, 1)

## Lists to contain the results:
mseRes <- list()
sel <- list()
lm <- list()
prd <- list()

## Generate the spatial eigenfunctions:
genSEF(
  x = salmon$Position[train],
  m = genDistMetric(),
  f = genDWF("Gaussian",40)
) -> sefTrain

## Spatially-explicit modelling of the channel depth:

## Calculate the minimum MSE model:
getMinMSE(
  U = as.matrix(sefTrain),
  y = salmon$Depth[train],
  Up = predict(sefTrain, salmon$Position[test]),
  yy = salmon$Depth[test]
) -> mseRes[["Depth"]]

## This is the coefficient of prediction:
1 - mseRes$Depth$mse[mseRes$Depth$wh]/mseRes$Depth$nullmse

## Storing graphical parameters:
tmp <- par(no.readonly = TRUE)

## Changing the graphical margins:
par(mar=c(4,4,2,2))

## Plot of the MSE values:
plot(mseRes$Depth$mse, type="l", ylab="MSE", xlab="order", axes=FALSE,
     ylim=c(0.005,0.025))
points(x=1:length(mseRes$Depth$mse), y=mseRes$Depth$mse, pch=21, bg="black")
axis(1)
axis(2, las=1)
abline(h=mseRes$Depth$nullmse, lty=3)  # Dotted line: the null MSE

## A list of the selected spatial eigenfunctions:
sel[["Depth"]] <- sort(mseRes$Depth$ord[1:mseRes$Depth$wh])

## A linear model build using the selected spatial eigenfunctions:
lm(
  formula = y~.,
  data = cbind(
    y = salmon$Depth[train],
    as.data.frame(sefTrain, wh=sel$Depth)
  )
) -> lm[["Depth"]]

## Calculating predictions of depth at each 1 m intervals:
predict(
  lm$Depth,
  newdata = as.data.frame(
    predict(
      object = sefTrain,
      newdata = xx,
      wh = sel$Depth
    )
  )
) -> prd[["Depth"]]
```

```
## Plot of the predicted depth (solid line), and observed depth for the
## training set (black markers) and testing set (red markers):
plot(x=xx, y=prd$Depth, type="l", ylim=range(salmon$Depth, prd$Depth), las=1,
     ylab="Depth (m)", xlab="Location along the transect (m)")
points(x = salmon$Position[train], y = salmon$Depth[train], pch=21,
       bg="black")
points(x = salmon$Position[test], y = salmon$Depth[test], pch=21, bg="red")

## Spatially-explicit modelling of the water velocity:

## Calculate the minimum MSE model:
getMinMSE(
  U = as.matrix(sefTrain),
  y = salmon$Velocity[train],
  Up = predict(sefTrain, salmon$Position[test]),
  yy = salmon$Velocity[test]
) -> mseRes[["Velocity"]]

## This is the coefficient of prediction:
1 - mseRes$Velocity$mse[mseRes$Velocity$wh]/mseRes$Velocity$nullmse

## Plot of the MSE values:
plot(mseRes$Velocity$mse, type="l", ylab="MSE", xlab="order", axes=FALSE,
     ylim=c(0.010,0.030))
points(x=1:length(mseRes$Velocity$mse), y=mseRes$Velocity$mse, pch=21,
       bg="black")
axis(1)
axis(2, las=1)
abline(h=mseRes$Velocity$nullmse, lty=3)

## A list of the selected spatial eigenfunctions:
sel[["Velocity"]] <- sort(mseRes$Velocity$ord[1:mseRes$Velocity$wh])

## A linear model build using the selected spatial eigenfunctions:
lm(
  formula = y~.,
  data = cbind(
    y = salmon$Velocity[train],
    as.data.frame(sefTrain, wh=sel$Velocity)
  )
) -> lm[["Velocity"]]

## Calculating predictions of velocity at each 1 m intervals:
predict(
  lm$Velocity,
  newdata = as.data.frame(
    predict(
      object = sefTrain,
      newdata = xx,
      wh = sel$Velocity
    )
  )
```

```
) -> prd[["Velocity"]]

## Plot of the predicted velocity (solid line), and observed velocity for the
## training set (black markers) and testing set (red markers):
plot(x=xx, y=prd$Velocity, type="l",
     ylim=range(salmon$Velocity, prd$Velocity),
     las=1, ylab="Velocity (m/s)", xlab="Location along the transect (m)")
points(x = salmon$Position[train], y = salmon$Velocity[train], pch=21,
       bg="black")
points(x = salmon$Position[test], y = salmon$Velocity[test], pch=21,
       bg="red")

## Spatially-explicit modelling of the mean substrate size (D50):

## Calculate the minimum MSE model:
getMinMSE(
  U = as.matrix(sefTrain),
  y = salmon$Substrate[train],
  Up = predict(sefTrain, salmon$Position[test]),
  yy = salmon$Substrate[test]
) -> mseRes[["Substrate"]]

## This is the coefficient of prediction:
1 - mseRes$Substrate$mse[mseRes$Substrate$wh]/mseRes$Substrate$nullmse

## Plot of the MSE values:
plot(mseRes$Substrate$mse, type="l", ylab="MSE", xlab="order", axes=FALSE,
     ylim=c(1000,6000))
points(x=1:length(mseRes$Substrate$mse), y=mseRes$Substrate$mse, pch=21,
       bg="black")
axis(1)
axis(2, las=1)
abline(h=mseRes$Substrate$nullmse, lty=3)

## A list of the selected spatial eigenfunctions:
sel[["Substrate"]] <- sort(mseRes$Substrate$ord[1:mseRes$Substrate$wh])

## A linear model build using the selected spatial eigenfunctions:
lm(
  formula = y~.,
  data = cbind(
    y = salmon$Substrate[train],
    as.data.frame(sefTrain, wh=sel$Substrate)
  )
) -> lm[["Substrate"]]

## Calculating predictions of D50 at each 1 m intervals:
predict(
  lm$Substrate,
  newdata = as.data.frame(
    predict(
      object = sefTrain,
      newdata = xx,
      wh = sel$Substrate
    )
  )
) -> prd[["Substrate"]]

## Plot of the predicted D50 (solid line), and observed D50 for the training
## set (black markers) and testing set (red markers):
plot(x=xx, y=prd$Substrate, type="l",
     ylim=range(salmon$Substrate, prd$Substrate), las=1, ylab="D50 (mm)",
     xlab="Location along the transect (m)")
points(x = salmon$Position[train], y = salmon$Substrate[train], pch=21,
       bg="black")
points(x = salmon$Position[test], y = salmon$Substrate[test], pch=21,
       bg="red")

## Spatially-explicit modelling of Atlantic salmon parr abundance using
## x=channel depth + water velocity + D50 + pMEM:

## Requires suggested package glmnet to perform elasticnet regression:
library(glmnet)

## Calculation of the elastic net model (cross-validated):
cv.glmnet(
  y = salmon$Abundance[train],
  x = cbind(
    Depth = salmon$Depth[train],
    Velocity = salmon$Velocity[train],
    Substrate = salmon$Substrate[train],
    as.matrix(sefTrain)
  ),
  family = "poisson"
) -> cvglm

## Calculating predictions for the test data:
predict(
  cvglm,
  newx = cbind(
    Depth = salmon$Depth[test],
    Velocity = salmon$Velocity[test],
    Substrate = salmon$Substrate[test],
    predict(sefTrain, salmon$Position[test])
  ),
  s="lambda.min",
  type = "response"
) -> yhatTest

## Calculating predictions for the transect (1 m seperated data):
predict(
  cvglm,
  newx = cbind(
    Depth = prd$Depth,
    Velocity = prd$Velocity,
    Substrate = prd$Substrate,
    predict(sefTrain, xx)
  ),
  s = "lambda.min",
  type = "response"
) -> yhatTransect

## Plot of the predicted Atlantic salmon parr abundance (solid line, with the
## depth, velocity, and D50 also predicted using spatially-explicit
## submodels), the observed abundances for the training set (black markers),
## the observed abundances for the testing set (red markers), and the
## predicted abundances for the testing set calculated on the basis of
## observed depth, velocity, and D50:
plot(x=xx, y=yhatTransect, type="l",
     ylim=range(salmon$Abundance,yhatTransect), las=1,
     ylab="Abundance (fish)", xlab="Location along the transect (m)")
points(x=salmon$Position[train], y=salmon$Abundance[train], pch=21,
       bg="black")
points(x=salmon$Position[test], y=salmon$Abundance[test], pch=21, bg="red")
points(x=salmon$Position[test], y=yhatTest, pch=21, bg="green")

## Restoring previous graphical parameters:
par(tmp)
```

`salmon` *The St. Marguerite River Altantic Salmon Parr Transect*

Description

Juvenile Atlantic salmon (parr) density in a 1520m transect of the St. Marguerite River, Québec, Canada.

Usage

```
data(salmon)
```

Format

A 76 rows by 5 columns `data.frame`.

Details

Contains (1) the 76 sampling site positions along a 1520 m river segment beginning at a location called ‘Bardsville‘ (Lat: 48°23’01.59” N ; Long: 70°12’10.05” W), (2) the number of parr (young salmon, ages I+ and II+) observed at the sampling sites, (3) the mean water depths (m), (4) the mean current velocity (m/s), and (5) the mean substrate size (mm). Sampling took place on July 7, 2002, in the 76 sites, each 20 m long. The ‘Bardsville‘ river segment is located in the upper portion of Sainte-Marguerite River, Quebec, Canada.

Source

Daniel Boisclair, Département de sciences biologiques, Université de Montréal, Montréal, Québec, Canada.

References

Guénard, G., Legendre, P., Boisclair, D., and Bilodeau, M. 2010. Multiscale codependence analysis: an integrated approach to analyse relationships across scales. Ecology 91: 2952-2964

See Also

Bouchard, J. and Boisclair, D. 2008. The relative importance of local, lateral, and longitudinal variables on the development of habitat quality models for a river. Can. J. Fish. Aquat. Sci. 65: 61-73

Examples

```
data(salmon)
summary(salmon)
```

`SEMap-class` *Class and Methods for Predictive Moran's Eigenvector Maps (pMEM)*

Description

Generator function, class, and methods to handle predictive Moran's eigenvector maps (pMEM).

Usage

```
genSEF(x, m, f, tol = .Machine$double.eps^0.5)

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

## S3 method for class 'SEMap'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'SEMap'
as.matrix(x, ...)

## S3 method for class 'SEMap'
predict(object, newdata, ...)
```

Arguments

`x`	a set of coordinates to be given to the distance metric function (argument `m` below) to obtain the distance metric (`genSEF`) or an `SEMap-class` object (methods).
`m`	a distance metric function, such as one of those returned by `genDistMetric`.
`f`	a distance weighting function, such as one of those returned by `genDWF`.
`tol`	a tolerance threshold for absolute eigenvalues, below which to discard spatial eigenfunctions.
`...`	further arguments to be passed to other functions or methods.
`row.names`	`NULL` or a character vector giving the row names for the data frame. Missing values are not allowed.
`optional`	logical. If `TRUE`, setting row names and converting column names (to syntactic names: see `make.names`) is optional. See **base** `as.data.frame` for further details on this argument.
`object`	an `SEMap-class` object.
`newdata`	a set of new coordinates from which to calculate pMEM predictor scores.

Format

A `SEMap-class` object contains:

show A printing function.

getIMoran A function (with no argument) returning the Moran's I coefficients associated with the spatial eigenfunctions.

getSEF A function that return the spatial eigenvectors. It has an argument `wh` which allows one to specify a selection of the eigenvectors that are to be returned.

getLambda A function that returns the eigenvalues.

getPredictor A function that calculate the spatial eigenfunction values for arbitrary locations about the sampling points. The coordinates of these locations are given as a vector or matrix through argument `xx`. It also has an argument `wh` which allows one to specify a selection of the eigenfunctions that are to be returned.

Details

Predictive Moran's Eigenvector Maps (pMEM) allows one to model the spatial variability of an environmental variable and use the resulting model for making prediction at any location on and around the sampling points. They originate from coordinates in one or more dimensions, which are used to calculate distances. The distances are obtained from the coordinates using a function given through argument `m` (see `genDistMetric` for further details). The distances are then transformed to weights using a spatial weighting function given as argument `f` (see `genDWF` for implementations of spatial weighting function). The resulting weights are row- and column-centred to the value 0 before being submitted to an eigenvalue decomposition. Eigenvectors associated to eigenvalues whose absolute value are above the threshold value set through argument `tol` are retained as part of the resulting eigenvector map.

In a standard workflow, a model is built for the locations where values of the response variable are known using the eigenvectors (or a subset thereof). This model may be build using any model building approach using descriptors. The scores obtained for new coordinates from method `predict` are used given to the model for making predictions.

The function can handle real-valued as well as complex-valued distance metrics. The latter is useful to represent asymmetric (i.e., directed) spatial processes.

Value

genSEF	a `SEMap-class` object.
print.SEMap	`NULL` (invisibly).
as.data.frame.SEMap	A `data.frame` with the spatial eigenvectors.
as.matrix.SEMap	A matrix with the spatial eigenvectors.
predict.SEMap	A matrix with the spatial eigenfunction values

Functions

- `genSEF()`: Predictive Moran's Eigenvector Map (pMEM) Generation
  Generates a predictive spatial eigenvector map (a SEMap-class object).
- `print(SEMap)`: Print SEMap-class
  A print method for `SEMap-class` objects.
- `as.data.frame(SEMap)`: An `as.data.frame` Method for `SEMap-class` Objects
  A method to extract the spatial eigenvectors from an `SEMap-class` object as a data frame.
- `as.matrix(SEMap)`: An `as.matrix` Method for `SEMap-class` Objects
  A method to extract the spatial eigenvectors from an `SEMap-class` object as a matrix.
- `predict(SEMap)`: A `predict` Method for `SEMap-class` Objects
  A method to obtain predictions from an `SEMap-class` object.

Author(s)

Guillaume Guénard [aut, cre] (ORCID: <https://orcid.org/0000-0003-0761-3072>), Pierre Legendre [ctb] (ORCID: <https://orcid.org/0000-0002-3838-3305>)

Maintainer: Guillaume Guénard <guillaume.guenard@umontreal.ca>

Examples

```
## Store graphical parameters:
tmp <- par(no.readonly = TRUE)
par(las=1)

## Case 1: one-dimensional symmetrical

n <- 11
x <- (n - 1)*seq(0, 1, length.out=n)
xx <- (n - 1)*seq(0, 1, 0.01)
```

```
sefSym <- genSEF(x, genDistMetric(), genDWF("Gaussian",3))

plot(y = predict(sefSym, xx, wh=1), x = xx, type = "l", ylab = "PMEM_1",
     xlab = "x")
points(y = as.matrix(sefSym, wh=1), x = x)

plot(y = predict(sefSym, xx, wh=2), x = xx, type = "l", ylab = "PMEM_2",
     xlab = "x")
points(y = as.matrix(sefSym, wh=2), x = x)

plot(y = predict(sefSym, xx, wh=5), x = xx, type = "l", ylab = "PMEM_5",
     xlab = "x")
points(y = as.matrix(sefSym, wh=5), x = x)

## Case 2: one-dimensional asymmetrical (each has a real and imaginary parts)

sefAsy <- genSEF(x, genDistMetric(delta = pi/8), genDWF("Gaussian",3))

plot(y = Re(predict(sefAsy, xx, wh=1)), x = xx, type = "l", ylab = "PMEM_1",
     xlab = "x", ylim=c(-0.35,0.35))
lines(y = Im(predict(sefAsy, xx, wh=1)), x = xx, col="red")
points(y = Re(as.matrix(sefAsy, wh=1)), x = x)
points(y = Im(as.matrix(sefAsy, wh=1)), x = x, col="red")

plot(y = Re(predict(sefAsy, xx, wh=2)), x = xx, type = "l", ylab = "PMEM_2",
     xlab = "x", ylim=c(-0.45,0.35))
lines(y = Im(predict(sefAsy, xx, wh=2)), x = xx, col="red")
points(y = Re(as.matrix(sefAsy, wh=2)), x = x)
points(y = Im(as.matrix(sefAsy, wh=2)), x = x, col="red")

plot(y = Re(predict(sefAsy, xx, wh=5)), x = xx, type = "l", ylab = "PMEM_5",
     xlab = "x", ylim=c(-0.45,0.35))
lines(y = Im(predict(sefAsy, xx, wh=5)), x = xx, col="red")
points(y = Re(as.matrix(sefAsy, wh=5)), x = x)
points(y = Im(as.matrix(sefAsy, wh=5)), x = x, col="red")

## A function to display combinations of the real and imaginary parts:
plotAsy <- function(object, xx, wh, a, ylim) {
  pp <- predict(object, xx, wh=wh)
  plot(y = cos(a)*Re(pp) + sin(a)*Im(pp), x = xx, type = "l",
       ylab = "PMEM_5", xlab = "x", ylim=ylim, col="green")
  invisible(NULL)
}

## Display combinations at an angle of 45° (pMEM_5):
plotAsy(sefAsy, xx, 5, pi/4, ylim=c(-0.45,0.45))

## Display combinations for other angles:
for(i in 0:15) {
  plotAsy(sefAsy, xx, 5, i*pi/8, ylim=c(-0.45,0.45))
  if(is.null(locator(1))) break
}
```

```
## Case 3: two-dimensional symmetrical

cbind(
  x = c(-0.5,0.5,-1,0,1,-0.5,0.5),
  y = c(rep(sqrt(3)/2,2L),rep(0,3L),rep(-sqrt(3)/2,2L))
) -> x2

seq(min(x2[,1L]) - 0.3, max(x2[,1L]) + 0.3, 0.05) -> xx
seq(min(x2[,2L]) - 0.3, max(x2[,2L]) + 0.3, 0.05) -> yy

list(
  x = xx,
  y = yy,
  coords = cbind(
    x = rep(xx, length(yy)),
    y = rep(yy, each = length(xx))
  )
) -> ss

cc <- seq(0,1,0.01)
cc <- c(rgb(cc,cc,1),rgb(1,1-cc,1-cc))

sefSym2D <- genSEF(x2, genDistMetric(), genDWF("Gaussian",3))

scr <- predict(sefSym2D, ss$coords)

par(mfrow = c(2,3), mar=0.5*c(1,1,1,1))

for(i in 1L:6) {
  image(z=matrix(scr[,i],length(ss$x),length(ss$y)), x=ss$x, y=ss$y, asp=1,
        zlim=max(abs(scr[,i]))*c(-1,1), col=cc, axes=FALSE)
  points(x = x2[,1L], y = x2[,2L])
}

## Case 4: two-dimensional asymmetrical

sefAsy2D0 <- genSEF(x2, genDistMetric(delta=pi/8), genDWF("Gaussian",1))
## Note: default influence angle is 0 (with respect to the abscissa)

## A function to display combinations of the real and imaginary parts (2D):
plotAsy2 <- function(object, ss, a) {
  pp <- predict(object, ss$coords)
  for(i in 1:6) {
    z <- cos(a)*Re(pp[,i]) + sin(a)*Im(pp[,i])
    image(z=matrix(z,length(ss$x),length(ss$y)), x=ss$x, y=ss$y, asp=1,
          zlim=max(abs(z))*c(-1,1), col=cc, axes=FALSE)
  }
  invisible(NULL)
}

## Display combinations at an angle of 22°:
plotAsy2(sefAsy2D0, ss, pi/8)
```

```
## Display combinations at other angles:
for(i in 0:23) {
  plotAsy2(sefAsy2D0, ss, i*pi/12)
  if(is.null(locator(1))) break
}

## With an influence of +45° (with respect to the abscissa)
sefAsy2D1 <- genSEF(x2, genDistMetric(delta=pi/8, theta = pi/4),
                    genDWF("Gaussian",1))

for(i in 0:23) {
  plotAsy2(sefAsy2D1, ss, i*pi/12)
  if(is.null(locator(1))) break
}

## With an influence of +90° (with respect to the abscissa)
sefAsy2D2 <- genSEF(x2, genDistMetric(delta=pi/8, theta = pi/2),
                    genDWF("Gaussian",1))

for(i in 0:23) {
  plotAsy2(sefAsy2D2, ss, i*pi/12)
  if(is.null(locator(1))) break
}

## With an influence of -45° (with respect to the abscissa)
sefAsy2D3 <- genSEF(x2, genDistMetric(delta=pi/8, theta = -pi/4),
                    genDWF("Gaussian",1))

for(i in 0:23) {
  plotAsy2(sefAsy2D3, ss, i*pi/12)
  if(is.null(locator(1))) break
}

## Reverting to initial graphical parameters:
par(tmp)
```

Index