Package 'pMEM'

May 9, 2026

Type Package

Title Predictive Moran's Eigenvector Maps

Version 1.0-1

Date 2026-03-06

Encoding UTF-8

Description Calculate Predictive Moran's Eigenvector Maps (pMEM) for spatially-explicit prediction of environmental variables, as defined by Guénard and Legendre (2024) <doi:10.1111/2041-210X.14413>. pMEM extends classical MEM by enabling interpolation and prediction at unsampled locations using spatial weighting functions parameterized by range (and optionally shape). The package implements multiple pMEM types (e.g., exponential, Gaussian, linear) and features a modular architecture that allows programmers to define custom weighting functions. Designed for ecologists, geographers, and spatial analysts working with spatially-structured data.

Depends R (>= 4.1.0), sf

Suggests glmnet, knitr, magrittr, rmarkdown, xfun

Imports Rcpp (>= 1.0.11)

License GPL-3

LazyLoad yes

NeedsCompilation yes

LinkingTo Rcpp

VignetteBuilder knitr

Repository CRAN

RoxygenNote 7.3.3

Author 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@gmail.com>

Date/Publication 2026-03-08 23:00:02 UTC

Contents

`pMEM-package` *pMEM: Predictive Moran's Eigenvector Maps for Spatial Modeling*

Description

`pMEM` implements Predictive Moran's Eigenvector Maps, a method for spatially-explicit prediction of environmental variables using eigen- decomposition of distance-based spatial weighting matrices.

Details

Package Overview: pMEM extends classical Moran's Eigenvector Maps (MEM) by:

- Enabling prediction at unsampled locations via `predict()` methods
- Supporting tunable distance weighting functions (exponential, Gaussian, power, linear, spherical, hyperbolic, hole_effect)
- Optimizing spatial scale parameters via cross-validated model selection
- Integrating with regression frameworks (lm, glm, glmnet)
- Providing fast C++ backend via Rcpp for efficient eigenvector selection

Main Functions: Core workflow functions:

- `genDistMetric`: Generate distance metric functions (symmetric Euclidean or asymmetric complex-valued)
- `genDWF`: Generate distance weighting functions (7 built-in types with tunable parameters)
- `genSEF`: Generate Spatial EigenMap (SEMap) objects
- `getMinMSE`: Fast eigenvector selection via cross-validated MSE
- `predict.SEMap`: Interpolate eigenfunctions at new locations

SEMap Class and Methods: The `SEMap-class` encapsulates spatial eigenfunctions with methods:

- `print()`: Display object summary (sites, components, eigenvalues)
- `as.matrix()`: Extract eigenvectors as numeric/complex matrix
- `as.data.frame()`: Extract eigenvectors as data frame
- `predict()`: Calculate eigenfunction scores at new coordinates

Included Datasets: Two spatial ecological datasets for examples and testing:

- `salmon`: Atlantic salmon parr abundance and habitat variables along a 1520 m river transect (76 sites, 5 variables)
- `geoMite`: Oribatid mite community data with GIS layers from a peat bog (70 cores, 35 species, 5 polygon layers)

Typical Workflow: Standard spatial modeling workflow with pMEM:

1. Load coordinates and response variable(s)
2. Generate distance metric: `m <- genDistMetric()`
3. Generate weighting function: `f <- genDWF("Gaussian", range = 50)`
4. Create SEMap object: `sef <- genSEF(x, m, f)`
5. Split data into training/testing sets
6. Select optimal eigenvectors: `res <- getMinMSE(U, y, Up, yy)`
7. Fit model with selected eigenvectors: `lm(y ~ ., data = sef[, sel])`
8. Predict at new locations: `predict(sef, newdata, wh = sel)`

Complex-Valued (Asymmetric) Metrics: pMEM supports directional spatial processes via complex-valued distance metrics:

- Use `genDistMetric(delta)` with non-zero `delta` parameter
- For 1D transects: `delta` sets phase shift for upstream/downstream
- For 2D data: `theta` sets influence angle (direction of flow)
- Eigenfunctions have real and imaginary parts representing directionality
- Useful for rivers, wind patterns, ocean currents, and other directed processes

Performance Notes:

- `getMinMSE()` is implemented in C++ via Rcpp (~50× faster than pure R)
- Memory-efficient handling of large eigenvector matrices
- Parallelization-ready architecture for future extensions

References: Key methodological references:

- Guénard, G. and Legendre, P. (2024). Spatially-explicit predictions using spatial eigenvector maps. *Methods in Ecology and Evolution*. <doi:10.1111/2041-210X.14413>
- Dray, S., Legendre, P., and Peres-Neto, P.R. (2006). Spatial modelling: A comprehensive framework for Principal Coordinate Analysis of Neighbour Matrices (PCNM). *Ecological Modelling*, 196: 483–493. <doi:10.1016/j.ecolmodel.2006.02.015>
- Moran, P.A.P. (1948). The interpretation of statistical maps. *Journal of the Royal Statistical Society B*, 10: 243–251. <doi:10.1111/j.2517-6161.1948.tb00012.x>

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

```
## Quick start: model channel depth along a river transect
data("salmon", package = "pMEM")

## Split into training/testing sets:
set.seed(123)
test <- sample(nrow(salmon), 25)
train <- setdiff(seq_len(nrow(salmon)), test)

## Generate spatial eigenfunctions:
sef <- genSEF(
  x = salmon$Position[train],
  m = genDistMetric(),
  f = genDWF("Gaussian", range = 50)
)

## Select optimal eigenvectors via cross-validated MSE:
res <- getMinMSE(
  U = as.matrix(sef),
  y = salmon$Depth[train],
  Up = predict(sef, salmon$Position[test]),
  yy = salmon$Depth[test],
  complete = FALSE
)

## Coefficient of prediction (R² on test set):
1 - res$mse / var(salmon$Depth[test])
#> [1] 0.8108639

## View SEMap object summary:
sef
#> A SEMap-class object
#> --------------------
#> Number of sites: 51
#> Directional: no
#> Number of components: 50
#> Eigenvalues: 3.76546,3.54744,3.43502,...,0.00066,0.00012
#> --------------------

## More examples:
vignette("Using_pMEM", package = "pMEM")
```

`genDistMetric` *Generate a Distance Metric Function*

Description

`genDistMetric` returns a function that calculates pairwise distances between rows of coordinate matrices, with optional asymmetry parameters.

Usage

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

Arguments

`delta`	Optional asymmetry parameter. When provided, the returned distance metric becomes complex-valued, with modulus equal to the Euclidean distance and argument determined by `delta`. For 1D data, the argument is ±`delta`; for 2D data, it is the cosine of the angular difference relative to `theta`.
`theta`	Influence angle (in radians) for 2D asymmetric metrics. Used only when `delta` is provided. Default is `0`.

Details

Symmetric vs. Asymmetric Metrics: When `delta` is missing (default), the returned function computes the standard Euclidean distance. When `delta` is provided, the metric becomes complex-valued: its modulus equals the Euclidean distance, and its argument is determined by `delta` as follows:

- **1D data (transects):** The argument is `+delta` for pairs where the second point is downstream (higher coordinate) and `-delta` otherwise.
- **2D data:** The argument is the cosine of the angular difference between the bearing from point A to B and the influence angle `theta`.

In both cases, the distance from A $\rightarrow$ B has the opposite argument sign as B $\rightarrow$ A, ensuring the pairwise distance matrix is Hermitian with strictly real eigenvalues.

Function Generator Pattern: `genDistMetric` uses a function-generator pattern: it returns a closure that embeds the specified `delta` and `theta` values in its environment. To change these parameters, call `genDistMetric` again with new arguments.

Value

A function with signature `function(x, y)` that returns a numeric matrix of pairwise distances. When `y` is omitted, the function returns a square symmetric (or Hermitian) matrix of distances among rows of `x`. For asymmetric metrics (`delta` provided), the matrix is Hermitian with strictly real eigenvalues.

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)

## Symmetric (Euclidean) metric function:
mSym <- genDistMetric()
```

```
## The pairwise symmetric metric between the rows of x:
mSym(x)
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    0    1    2    3    4
#> [2,]    1    0    1    2    3
#> [3,]    2    1    0    1    2
#> [4,]    3    2    1    0    1
#> [5,]    4    3    2    1    0

## Distances between x and a finer grid xx:
xx <- (n - 1)*seq(0, 1, 0.05)
mSym(x, xx)  # Rectangular matrix: 5 rows × 21 columns

## Asymmetric metric with delta = 0.2:
mAsy <- genDistMetric(delta = 0.2)
mAsy(x)  # Hermitian matrix (complex values)
Mod(mAsy(x))  # Modulus equals Euclidean distances

## Asymmetric distances between x and xx:
mAsy(x, xx)
```

---

genDWF *Generate a Distance Weighting Function*

Description

genDWF returns a function that transforms pairwise distances into spatial weights, with support for both real and complex-valued distances.

Usage

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

Arguments

fun	Distance weighting function type. One of "linear", "power", "hyperbolic", "spherical", "exponential", "Gaussian", or "hole_effect". Partial matching is supported (e.g., "exp" for "exponential").
range	Range parameter (positive numeric). For "linear", "power", "hyperbolic", and "spherical", this is the distance beyond which weights equal 0. For "exponential", "Gaussian", and "hole_effect", this is the scale parameter controlling decay rate.

shape — Shape parameter (positive numeric). Used only for "power" and "hyperbolic" functions; ignored otherwise. Default is 1.

Details

All functions return 1 (or 1+0i) at distance 0. Behavior beyond varies.

Complex-Valued Distances: When distances are complex (from asymmetric metrics via genDistMetric(delta)), the weighting functions operate on the modulus while preserving phase information. This enables directional spatial modeling for transects or flow-influenced systems.

Function Generator Pattern: genDWF returns a closure embedding fun, range, and shape in its environment. To change parameters, call genDWF again with new arguments.

Value

A function with signature function(d) that transforms distances into weights. Returns a numeric vector for vector input, or a numeric/complex matrix for matrix input. For complex-valued distances (from asymmetric metrics), returns complex-valued weights.

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 set of distances from which to show the corresponding weights:
d <- seq(0, 5, 0.1)

## The linear function with different ranges:
w <- cbind(
  genDWF(fun = "linear", range = 1)(d),
  genDWF(fun = "linear", range = 0.5)(d),
  genDWF(fun = "linear", range = 2)(d)
)
dim(w)  # 51 × 3 matrix
#> [1] 51    3

## The exponential function:
w <- genDWF(fun = "exponential", range = 1)(d)
w[1:5]  # First 5 weights (starts at 1, decays)
#> [1] 1.0000000 0.9048374 0.8187308 0.7408182 0.6703200

## The hole_effect function (can go negative):
w <- genDWF(fun = "hole_effect", range = 1)(d)
range(w)  # Includes negative values
#> [1] -0.2162362  1.0000000

## Complex-valued distances (asymmetric metrics) with a delta of pi/8:
d <- complex(modulus = seq(0, 5, 0.1), argument = pi/8)
w <- genDWF(fun = "Gaussian", range = 1)(d)
```

```
Mod(w[1:5])  # Modulus of complex weights
#> 1.0000000 0.9929539 0.9721120 0.9383431 0.8930282
Arg(w[1:5])  # Phase preserved from input distances
#> 0.000000000 -0.007071068 -0.028284271 -0.063639610 -0.113137085
```

`geoMite` *Borcard's Oribatid Mite Data (GIS Version)*

Description

Spatially-referenced oribatid mite community and environmental data from a peat bog surrounding Lac Geai, Québec, Canada.

Format

Data Description: 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).

Coordinate Reference System: All geometries use a local Cartesian coordinate system with units in centimeters, oriented such that X increases from water edge to forest edge, and Y increases along the shoreline. No projected CRS is assigned; treat as a local grid for spatial modeling.

Details

Environmental Variables: Fields in `geoMite$core`:

- **SubsDens**: Substrate density (g/L)
- **WatrCont**: Water content of peat (g/L)
- **Substrate-***: Six binary indicators for substrate type (see substrate classification below)
- **Shrub**: Ordered factor (None < Few < Many) for shrub cover

- **Topo**: Factor with levels "Blanket" (flat) and "Hummock" (raised)
- **Flooded**: Binary indicator of flooding at time of sampling
- **Species-***: Counts of 35 oribatid species (morphological identification)

Substrate Classification: Substrate types (non-exclusive; cores may span multiple types):

- **Sphagn1**: *Sphagnum magellanicum* (majority) + *S. rubellum*
- **Sphagn2**: *Sphagnum rubellum*
- **Sphagn3**: *Sphagnum nemoreum* (+ minority *S. angustifolium*)
- **Sphagn4**: *S. rubellum* and *S. magellanicum* in equal parts
- **Litter**: Ligneous plant litter
- **Barepeat**: Exposed peat without vegetation

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

Data Provenance: Polygon geometries were digitized from Figure 1 in Borcard et al. (1994) using a 10 mm grid. Due to print resolution limits, effective positional accuracy is approximately 10 cm in both X and Y directions.

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.

Taxonomic Resolution: The 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.

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) Springer, 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)

## Quick summary of species counts:
species_cols <- grep("^Species\\.", names(geoMite$core), value = TRUE)

## Maximum count per species
sapply(st_drop_geometry(geoMite$core[species_cols]), max)
#> Species.Brachysp Species.Hoplcfpa  Species.Oppinova ...
#>               42                8                37 ...

## Simple map of core locations over substrate:
if(requireNamespace("sf", quietly = TRUE)) {
  plot(st_geometry(geoMite$substrate), col = "lightgreen", main = "Peat Cores")
  plot(st_geometry(geoMite$core), pch = 21, bg = "black", add = TRUE)
}

## Not run:

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

## Shrubs:
with(
  geoMite,
  {
    plot(st_geometry(shrub), col = col[["shrub"]][shrub$Type], main="Shrubs")
```

```
    plot(st_geometry(water[1,]), col=col[["water"]][water[1,]$Type],
         add=TRUE)
    plot(st_geometry(water[-1,]), col=col[["water"]][water[-1,]$Type], lty=3,
         add=TRUE)
    plot(st_geometry(core), pch = 21, bg = "black", add=TRUE)
  }
)

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

## Legends:
with(
  geoMite,
  {
    plot(NA, xlim=c(0,1), ylim=c(0,1), axes = FALSE)
    legend(x=0, y=0.9, pch=22, 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(22,NA,21), 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=22, pt.cex=2.5, pt.bg=col[["shrub"]], box.lwd=0,
           legend=names(col[["shrub"]]), title="Shrubs")
    legend(x=0, y=0.2, pch=22, 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(geoMite$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(geoMite$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))

with(
  geoMite,
  {
    ## 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),1,256), col=colmap,
        xaxt="n", yaxt="n", y=seq(0,log1p(1000),length.out=256), xlab="",
        cex.lab=1.5,
        ylab=expression(paste("Counts by species (",ind~core^{-1},")")))
  axis(2, 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)

## End(Not run)
```

`getMinMSE` *Orthogonal Term Selection via Cross-Validated MSE*

Description

`getMinMSE` performs forward selection of orthogonal predictors by minimizing out-of-sample mean squared error (MSE) on a testing dataset.

Usage

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

Arguments

U	Training matrix of orthogonal predictors (n_train × p), where n_train is the number of training observations and p is the number of predictors (e.g., spatial eigenvectors). Must be orthonormal (columns have unit norm and are mutually orthogonal).
y	Training response vector (length n_train).
Up	Testing matrix of orthogonal predictors (n_test × p), evaluated at testing locations.
yy	Testing response vector (length n_test).
complete	If TRUE (default), return full selection results (all MSE values, coefficient order, etc.). If FALSE, return only the minimum MSE and associated coefficient threshold.

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.

Algorithm: The selection procedure proceeds as follows:

1. Compute regression coefficients: b = t(U) %*% y (valid because U is orthonormal).
2. Sort coefficients by decreasing absolute value; store order in ord.
3. Compute null MSE: variance of yy around mean(y).
4. For each predictor (in sorted order), add its partial prediction to the model and compute out-of-sample MSE on yy.
5. Identify the model with minimum MSE; return corresponding coefficient threshold and MSE.

Orthonormality Requirement: Predictors in U must be orthonormal (columns have unit norm and are mutually orthogonal). This condition is met by design for spatial eigenvectors produced by genSEF(), but users must ensure it for custom predictors.

Performance: This function is implemented in C++ via Rcpp for efficiency. For p = 50 predictors and n = 100 observations, it is approximately 50× faster than a pure R implementation.

Value

If complete = TRUE, a list with the following elements:

betasq Squared standardized regression coefficients (length p).

nullmse Null model MSE: variance of testing responses around their mean.

mse MSE values for incremental models (length p + 1); mse[1] is the null MSE.

ord Indices of predictors sorted by decreasing absolute coefficient value.

wh Index of the best model (1-based). Value 1 means the null model is best; value k + 1 means the first k predictors from ord are selected.

If complete = FALSE, a list with two elements:

betasq Squared standardized coefficient at the optimal model.

mse Minimum MSE value achieved.

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

```
## 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 built 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")

## Prediction of the velocity, substrate, and using them to predict the parr
## density.
## Not run:

## 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 separated 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 median substrate grain size:
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")


## End(Not run)
## Restoring previous graphical parameters:
```

```
par(tmp)
```

`salmon` *Sainte-Marguerite River Atlantic Salmon Parr Data*

Description

Spatially-referenced observations of juvenile Atlantic salmon (parr) density and habitat variables along a 1520 m transect in the Sainte-Marguerite River, Québec, Canada.

Format

A `data.frame` with 76 rows (sampling sites) and 5 columns:

Position Numeric: distance along transect from `Bardsville` (m).

Abundance Integer: count of Atlantic salmon parr observed per 20 m site.

Depth Numeric: mean water depth (m) at thalweg.

Velocity Numeric: mean current velocity (m/s) at thalweg.

Substrate Numeric: mean substrate grain size (mm, D50) at thalweg.

Details

Sampling Design: Data were collected on July 7, 2002, along a 1520 m river segment starting at `Bardsville` (48°23'01.59" N, 70°12'10.05" W). The transect was divided into 76 contiguous 20 m sites. At each site, snorkelers recorded parr abundance while moving upstream in a zigzag pattern to minimize disturbance.

Habitat Measurements: Environmental variables were measured at the thalweg (channel centerline) at the midpoint of each 20 m site (i.e., 10 m from each boundary):

- **Depth**: Mean water depth (m)
- **Velocity**: Mean current velocity (m/s)
- **Substrate**: Mean grain size (mm, D50 metric)

Coordinate System: `Position` is a local Cartesian coordinate (meters) increasing upstream from the `Bardsville` reference point. No projected CRS is assigned; treat as a 1D transect for spatial modeling.

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 <doi:10.1890/09-0460.1>

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 <doi:10.1139/f07-140>

Examples

```
data(salmon)

## Quick summary:
summary(salmon)
#>     Position        Abundance            Depth         Velocity       Substrate
#> Min.   :1280   Min.   : 0.000   Min.   :0.3600   Min.   :0.1900   Min.   : 51.0
#> 1st Qu.:1655   1st Qu.: 0.000   1st Qu.:0.5075   1st Qu.:0.4700   1st Qu.:159.5
#> Median :2030   Median : 2.000   Median :0.6050   Median :0.5750   Median :220.0
#> Mean   :2030   Mean   : 2.092   Mean   :0.6184   Mean   :0.5868   Mean   :213.9
#> 3rd Qu.:2405   3rd Qu.: 3.000   3rd Qu.:0.6900   3rd Qu.:0.6725   3rd Qu.:265.2
#> Max.   :2780   Max.   :14.000   Max.   :1.0600   Max.   :1.0700   Max.   :381.0

## Simple plot of parr abundance along the transect:
if(interactive()) {
  plot(salmon$Position, salmon$Abundance, type = "b", pch = 21, bg = "black",
       xlab = "Position along transect (m)", ylab = "Parr abundance")
}
```

SEMap-class *SEMap: Spatial EigenMap Class for Predictive Moran's Eigenvector Maps*

Description

SEMap objects store spatial eigenfunctions derived from distance-based weighting matrices, enabling spatially-explicit prediction at sampled and unsampled locations.

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	For genSEF: a numeric matrix or vector of coordinates (n × d, where n is number of sites and d is dimensions). For methods: an SEMap-class object.
m	A distance metric function (e.g., from genDistMetric). Must accept two arguments (x, y) and return a numeric or complex matrix of pairwise distances.
f	A distance weighting function (e.g., from genDWF). Must accept one argument (distances) and return weights of the same type.
tol	Tolerance threshold (positive numeric). Eigenvectors with absolute eigenvalues below this value are discarded. Default is .Machine$double.eps^0.5 (~1e-8).
...	Additional arguments passed to methods (e.g., wh for selecting specific eigenvectors).
row.names, optional	Passed to as.data.frame; see as.data.frame for details.
object	An SEMap-class object (for methods).
newdata	A numeric matrix or vector of coordinates for prediction (n_new × d).

Format

An SEMap-class object is a list with class attribute "SEMap" containing the following components:

show Function to print object summary.

getIMoran Function returning Moran's I coefficients for each eigenfunction.

getSEF Function returning eigenvectors; accepts wh argument to select specific columns.

getLambda Function returning eigenvalues.

getPredictor Function computing eigenfunction scores at new locations; accepts xx (coordinates) and wh (selection).

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.

Algorithm: pMEM eigenfunctions are computed as follows:

1. Compute pairwise distances from coordinates using m(x, x).
2. Transform distances to weights using f(d).
3. Double-center the weight matrix (row and column means subtracted).
4. Perform eigenvalue decomposition on the centered matrix.
5. Retain eigenvectors with absolute eigenvalues above tol.

Prediction at New Locations: The predict method calculates eigenfunction scores at unsampled locations by:

1. Computing distances from training sites to new locations.
2. Transforming to weights and re-centering using training site centers.
3. Projecting onto the eigenvector basis via matrix multiplication.

Complex-Valued Eigenfunctions: When using asymmetric distance metrics (via `genDistMetric(delta)`), eigenfunctions are complex-valued. The real and imaginary parts represent directional spatial patterns (e.g., upstream vs. downstream in rivers).

Modeling Workflow: Standard workflow:

1. Generate `SEMap` object with `genSEF()`.
2. Extract eigenvectors with `as.matrix()` or `as.data.frame()`.
3. Select optimal subset using `getMinMSE` or other criteria.
4. Fit model (e.g., `lm()`, `glm()`) with selected eigenvectors.
5. Predict at new locations using `predict(SEMap, newdata)`.

Value

`genSEF` An `SEMap`-class object containing eigenfunctions, eigenvalues, and prediction methods.

`print.SEMap` `NULL` (invisibly); prints summary to console.

`as.data.frame.SEMap` A `data.frame` with eigenvectors as columns.

`as.matrix.SEMap` A numeric/complex matrix of eigenvectors.

`predict.SEMap` A matrix of eigenfunction scores at `newdata` locations (`n_new` × `k`, where `k` is the number of eigenvectors).

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>)

Examples

```
## Case 1: one-dimensional symmetrical

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

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

sef <- genSEF(x, genDistMetric(), genDWF("Gaussian",3))
sef
#> A SEMap-class object
#> --------------------
#> Number of sites: 11
#> Directional: no
#> Number of components: 10
#> Eigenvalues: 3.28700,1.98782,0.79880,...,0.00005,0.00000
#> --------------------

## Extract eigenvectors:
dim(as.matrix(sef))  # 11 × 10 matrix
#> [1] 11 10

## Predict at new locations:
xx <- (n - 1)*seq(0, 1, 0.01)
pred <- predict(sef, xx, wh=1:3)
dim(pred)  # 101 × 3 matrix
#> [1] 101   3

## Quick plot of the first eigenfunction:
if(interactive()) {
  plot(xx, pred[, 1], type="l", xlab="Position", ylab="pMEM_1")
  points(x, as.matrix(sef)[, 1], pch=21, bg="black")
}

## Not run:

## The Second eigenfunction:
plot(y = predict(sef, xx, wh=2), x=xx, type="l", ylab="PMEM_2", xlab="x")
points(y=as.matrix(sef, wh=2), x=x)

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

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

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

## First asymmetric eigenfunction:
plot(y = Re(predict(sef, xx, wh=1)), x=xx, type="l", ylab="PMEM_1", xlab="x",
     ylim=c(-0.35,0.35))
```

```
lines(y = Im(predict(sef, xx, wh=1)), x=xx, col="red")
points(y=Re(as.matrix(sef, wh=1)), x=x)
points(y=Im(as.matrix(sef, wh=1)), x=x, col="red")

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

## Fifth asymmetric eigenfunction:
plot(y = Re(predict(sef, xx, wh=5)), x=xx, type="l", ylab="PMEM_5", xlab="x",
     ylim=c(-0.45,0.35))
lines(y = Im(predict(sef, xx, wh=5)), x=xx, col="red")
points(y = Re(as.matrix(sef, wh=5)), x=x)
points(y = Im(as.matrix(sef, 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(sef, xx, 5, pi/4, ylim=c(-0.45,0.45))

## Display combinations for other angles:
for(i in 0:15)
  plotAsy(sef, xx, 5, i*pi/8, ylim=c(-0.45,0.45))

## 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))

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

scr <- predict(sef, 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

sef <- 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(sef, ss, pi/8)

## Combinations at other angles:
for(i in 0:23)
  plotAsy2(sef, ss, i*pi/12)

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

## Combinations at other angles:
for(i in 0:23)
  plotAsy2(sef, ss, i*pi/12)

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

## Combinations at other angles:
for(i in 0:23)
  plotAsy2(sef, ss, i*pi/12)
```

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

## Combinations at other angles:
for(i in 0:23)
  plotAsy2(sef, ss, i*pi/12)


## End(Not run)

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

Index