Package ‘FRK’

April 1, 2020

Type Package

Title Fixed Rank Kriging

Version 0.2.2.1

Date 2018-07-11

Maintainer Andrew Zammit-Mangion <andrewzm@gmail.com>

VignetteBuilder knitr

Description Fixed Rank Kriging is a tool for spatial/spatio-temporal modelling and prediction with large datasets. The approach, discussed in Cressie and Johannesson (2008) <DOI:10.1111/j.1467-9868.2007.00633.x>, decomposes the field, and hence the covariance function, using a fixed set of n basis functions, where n is typically much smaller than the number of data points (or polygons) m. The method naturally allows for non-stationary, anisotropic covariance functions and the use of observations with varying support (with known error variance). The projected field is a key building block of the Spatial Random Effects (SRE) model, on which this package is based. The package FRK provides helper functions to model, fit, and predict using an SRE with relative ease.

BugReports http://github.com/andrewzm/FRK/issues

Depends R (>= 3.2.0)

Suggests covr, dggrids, gstat, INLA, knitr, mapproj, parallel, rgeos, spdep, splancs, testthat, verification

Imports digest, dplyr, ggplot2, grDevices, Hmisc (>= 4.1), Matrix, methods, plyr, Rcpp (>= 0.12.12), sp, spacetime, sparseinv, stats, utils

Additional_repositories https://andrewzm.github.io/dggrids-repo, https://inla.r-inla-download.org/R/stable

License GPL (>= 2)

NeedsCompilation yes

LazyData true

RoxygenNote 6.0.1

LinkingTo Rcpp

Author Andrew Zammit-Mangion [aut, cre]

Repository CRAN

Date/Publication 2020-04-01 05:42:27 UTC

R topics documented:

`FRK-package` *Fixed Rank Kriging*

Description

Fixed Rank Kriging is a tool for spatial/spatio-temporal modelling and prediction with large datasets. The approach, discussed in Cressie and Johannesson (2008), decomposes the field, and hence the covariance function, using a fixed set of n basis functions, where dimension n is typically much smaller than the number of data points (or polygons) m. The method naturally allows for non-stationary, anisotropic covariance functions and the use of observations with varying support (with known error variance). The dimension-reduced field is a key building block of the Spatial Random Effects (SRE) model, upon which this package is based. The package FRK provides helper functions to model, fit, and predict using an SRE with relative ease. Reference: Cressie, N. and Johannesson, G. (2008) <DOI:10.1111/j.1467-9868.2007.00633.x>.

`AIRS_05_2003` *AIRS data for May 2003*

Description

Mid-tropospheric CO2 measurements from the Atmospheric InfraRed Sounder (AIRS). The data are measurements between 60 degrees S and 90 degrees N at roughly 1:30 pm local time on 1 May through to 15 May 2003. (AIRS does not release data below 60 degrees S.)

Usage

```
AIRS_05_2003
```

Format

A data frame with 209631 rows and 7 variables:

year year of retrieval

month month of retrieval

day day of retrieval

lon longitude coordinate of retrieval

lat latitude coordinate of retrieval

co2avgret CO2 mole fraction retrieval in ppm

co2std standard error of CO2 retrieval in ppm

References

Chahine, M. et al. (2006). AIRS: Improving weather forecasting and providing new data on greenhouse gases. Bulletin of the American Meteorological Society 87, 911–26.

auto_basis *Automatic basis-function placement*

Description

Generate automatically a set of local basis functions in the domain, and automatically prune in regions of sparse data.

Usage

```
auto_basis(manifold = plane(), data, regular = 1, nres = 3, prune = 0,
  max_basis = NULL, subsamp = 10000, type = c("bisquare", "Gaussian",
  "exp", "Matern32"), isea3h_lo = 2, bndary = NULL,
  scale_aperture = ifelse(is(manifold, "sphere"), 1, 1.25), verbose = 0L,
  ...)
```

Arguments

`manifold`	object of class `manifold`, for example, `sphere` or `plane`
`data`	object of class `SpatialPointsDataFrame` or `SpatialPolygonsDataFrame` containing the data on which basis-function placement is based, or a list of these; see details
`regular`	an integer indicating the number of regularly-placed basis functions at the first resolution. In two dimensions, this dictates the smallest number of basis functions in a row or column at the coarsest resolution. If `regular=0`, an irregular grid is used, one that is based on the triangulation of the domain with increased mesh density in areas of high data density; see details
`nres`	the number of basis-function resolutions to use
`prune`	a threshold parameter that dictates when a basis function is considered irrelevent or unidentifiable, and thus removed; see details
`max_basis`	maximum number of basis functions. This overrides the parameter `nres`
`subsamp`	the maximum amount of data points to consider when carrying out basis-function placement: these data objects are randomly sampled from the full dataset. Keep this number fairly high (on the order of 10^5), otherwise fine-resolution basis functions may be spuriously removed
`type`	the type of basis functions to use; see details
`isea3h_lo`	if `manifold = sphere()`, this argument dictates which ISEA3H resolution is the coarsest one that should be used for the first resolution
`bndary`	a `matrix` containing points containing the boundary. If `regular == 0` this can be used to define a boundary in which irregularly-spaced basis functions are placed
`scale_aperture`	the aperture (in the case of the bisquare, but similar interpretation for other basis) width of the basis function is the minimum distance between all the basis function centroids multiplied by `scale_aperture`. Typically this ranges between 1 and 1.5 and is defaulted to 1 on the sphere and 1.25 on the other manifolds.

`verbose`	a logical variable indicating whether to output a summary of the basis functions created or not
`...`	unused

Details

This function automatically places basis functions within the domain of interest. If the domain is a plane or the real line, then the object `data` is used to establish the domain boundary.

The argument `type` can be either "Gaussian", in which case

$$\phi(u) = \exp\left(-\frac{\|u\|^2}{2\sigma^2}\right),$$

"bisquare", in which case

$$\phi(u) = \left(1 - \left(\frac{\|u\|}{R}\right)^2\right)^2 I(\|u\| < R),$$

"exp", in which case

$$\phi(u) = \exp\left(-\frac{\|u\|}{\tau}\right),$$

or "Matern32", in which case

$$\phi(u) = \left(1 + \frac{\sqrt{3}\|u\|}{\kappa}\right)\exp\left(-\frac{\sqrt{3}\|u\|}{\kappa}\right),$$

where the parameters σ, R, τ and κ are `scale` arguments.

If the manifold is the real line, the basis functions are placed regularly inside the domain, and the number of basis functions at the coarsest resolution is dictated by the integer parameter `regular` which has to be greater than zero. On the real line, each subsequent resolution has twice as many basis functions. The scale of the basis function is set based on the minimum distance between the centre locations following placement. The scale is equal to the minimum distance if the type of basis function is Gaussian, exponential, or Matern32, and is equal to 1.5 times this value if the function is bisquare.

If the manifold is a plane, and `regular > 0`, then basis functions are placed regularly within the bounding box of `data`, with the smallest number of basis functions in each row or column equal to the value of `regular` in the coarsest resolution (note, this is just the smallest number of basis functions). Subsequent resolutions have twice the number of basis functions in each row or column. If `regular = 0`, then the function `INLA::inla.nonconvex.hull` is used to construct a (non-convex) hull around the data. The buffer and smoothness of the hull is determined by the parameter `convex`. Once the domain boundary is found, `INLA::inla.mesh.2d` is used to construct a triangular mesh such that the node vertices coincide with data locations, subject to some minimum and maximum triangular-side-length constraints. The result is a mesh that is dense in regions of high data density and not dense in regions of sparse data. Even basis functions are irregularly placed, the scale is taken to be a function of the minimum distance between basis function centres, as detailed above. This may be changed in a future revision of the package.

If the manifold is the surface of a sphere, then basis functions are placed on the centroids of the discrete global grid (DGG), with the first basis resolution corresponding to the third resolution of the DGG (ISEA3H resolution 2, which yields 92 basis functions globally). It is not recommended to go above `nres == 3` (ISEA3H resolutions 2–4) for the whole sphere; `nres=3` yields a total of 1176 basis functions. Up to ISEA3H resolution 6 is available with `FRK`; for finer resolutions; please install `dggrids` from `https://github.com/andrewzm/dggrids` using `devtools`.

Basis functions that are not influenced by data points may hinder convergence of the EM algorithm when `K_type = ``unstructured''`, since the associated hidden states are, by and large, unidentifiable. We hence provide a means to automatically remove such basis functions through the parameter `prune`. The final set only contains basis functions for which the column sums in the associated matrix $S$ (which, recall, is the value/average of the basis functions at/over the data points/polygons) is greater than `prune`. If `prune == 0`, no basis functions are removed from the original design.

Examples

```
## Not run:
library(sp)
library(ggplot2)

### Create a synthetic dataset
set.seed(1)
d <- data.frame(lon = runif(n=1000,min = -179, max = 179),
                lat = runif(n=1000,min = -90, max = 90),
                z = rnorm(5000))
coordinates(d) <- ~lon + lat
proj4string(d)=CRS("+proj=longlat +ellps=sphere")

### Now create basis functions over sphere
G <- auto_basis(manifold = sphere(),data=d,
                nres = 2,prune=15,
                type = "bisquare",
                subsamp = 20000)

### Plot
\dontrun{show_basis(G,draw_world())}

## End(Not run)
```

`auto_BAUs` *Automatic BAU generation*

Description

This function calls the generic function `auto_BAU` (not exported) after a series of checks and is the easiest way to generate a set of Basic Areal Units (BAUs) on the manifold being used; see details.

Usage

```
auto_BAUs(manifold, type = NULL, cellsize = NULL, isea3h_res = NULL,
  data = NULL, nonconvex_hull = TRUE, convex = -0.05, tunit = NULL,
  xlims = NULL, ylims = NULL, ...)
```

Arguments

manifold	object of class manifold
type	either "grid" or "hex", indicating whether gridded or hexagonal BAUs should be used
cellsize	denotes size of gridcell when type = "grid". Needs to be of length 1 (square-grid case) or a vector of length dimensions(manifold) (rectangular-grid case)
isea3h_res	resolution number of the isea3h DGGRID cells for when type is "hex" and manifold is the surface of a sphere
data	object of class SpatialPointsDataFrame, SpatialPolygonsDataFrame, STIDF, or STFDF. Provision of data implies that the domain is bounded, and is thus necessary when the manifold is a real_line,plane, or STplane, but is not necessary when the manifold is the surface of a sphere
nonconvex_hull	flag indicating whether to use INLA to generate a non-convex hull. Otherwise a convex hull is used
convex	convex parameter used for smoothing an extended boundary when working on a bounded domain (that is, when the object data is supplied); see details
tunit	temporal unit when requiring space-time BAUs. Can be "secs", "mins", "hours", etc.
xlims	limits of the horizontal axis (overrides automatic selection)
ylims	limits of the vertical axis (overrides automatic selection)
...	currently unused

Details

auto_BAUs constructs a set of Basic Areal Units (BAUs) used both for data pre-processing and for prediction. As such, the BAUs need to be of sufficiently fine resolution so that inferences are not affected due to binning.

Two types of BAUs are supported by FRK: "hex" (hexagonal) and "grid" (rectangular). In order to have a "grid" set of BAUs, the user should specify a cellsize of length one, or of length equal to the dimensions of the manifold, that is, of length 1 for real_line and of length 2 for the surface of a sphere and plane. When a "hex" set of BAUs is desired, the first element of cellsize is used to determine the side length by dividing this value by approximately 2. The argument type is ignored with real_line and "hex" is not available for this manifold.

If the object data is provided, then automatic domain selection may be carried out by employing the INLA function inla.nonconvex.hull, which finds a (non-convex) hull surrounding the data points (or centroids of the data polygons). This domain is extended and smoothed using the parameter convex. The parameter convex should be negative, and a larger absolute value for convex results in a larger domain with smoother boundaries (note that INLA was not available on CRAN at the time of writing).

See Also

auto_basis for automatically constructing basis functions.

Examples

```
## First a 1D example
library(sp)
set.seed(1)
data <- data.frame(x = runif(10)*10, y = 0, z= runif(10)*10)
coordinates(data) <- ~x+y
Grid1D_df <- auto_BAUs(manifold = real_line(),
                       cellsize = 1,
                       data=data)
## Not run: spplot(Grid1D_df)

## Now a 2D example
data(meuse)
coordinates(meuse) = ~x+y # change into an sp object

## Grid BAUs
GridPols_df <- auto_BAUs(manifold = plane(),
                         cellsize = 200,
                         type = "grid",
                         data = meuse,
                         nonconvex_hull = 0)
## Not run: plot(GridPols_df)

## Hex BAUs
HexPols_df <- auto_BAUs(manifold = plane(),
                        cellsize = 200,
                        type = "hex",
                        data = meuse,
                        nonconvex_hull = 0)
## Not run: plot(HexPols_df)
```

Basis *Generic basis-function constructor*

Description

This function is meant to be used for manual construction of arbitrary basis functions. For 'local' basis functions, please use the function `local_basis` instead.

Usage

```
Basis(manifold, n, fn, pars, df)
```

Arguments

`manifold`	object of class `manifold`, for example, `sphere`
`n`	number of basis functions (should be an integer)

`fn` a list of functions, one for each basis function. Each function should be encapsulated within an environment in which the manifold and any other parameters required to evaluate the function are defined. The function itself takes a single input `s` which can be of class `numeric`, `matrix`, or `Matrix`, and returns a vector which contains the basis function evaluations at `s`.

`pars` A list containing a list of parameters for each function. For local basis functions these would correspond to location and scale parameters.

`df` A data frame containing one row per basis function, typically for providing informative summaries.

Details

This constructor checks that all the parameters are valid before constructing the basis functions using `new`. The requirement that every function is encapsulated is tedious, but necessary for FRK to work with a large range of basis functions in the future. Please see the example below which exemplifies the process of constructing linear basis functions from scratch using this function.

See Also

`auto_basis` for constructing basis functions automatically, `local_basis` for constructing 'local' basis functions, and `show_basis` for visualising basis functions.

Examples

```
## Construct two linear basis functions on [0, 1]
manifold <- real_line()
n <- 2
lin_basis_fn <- function(manifold, grad, intercept) {
   function(s) grad*s + intercept
}
pars <- list(list(grad = 1, intercept = 0),
             list(grad = -1, intercept = 1))
fn <- list(lin_basis_fn(manifold, 1, 0),
           lin_basis_fn(manifold, -1, 1))
df <- data.frame(n = 1:2, grad = c(1, -1), m = c(1, -1))
G <- Basis(manifold = manifold, n = n, fn = fn, pars = pars, df = df)
## Not run:
eval_basis(G, s = matrix(seq(0,1, by = 0.1), 11, 1))
## End(Not run)
```

`Basis_obj-class` *Basis functions*

Description

An object of class `Basis` contains the basis functions used to construct the matrix S in FRK. It contains five slots, described below.

Details

Basis functions are a central component of FRK, and the package is designed to work with user-defined specifications of these. For convenience, however, several functions are available to aid the user to construct a basis set for a given set of data points. Please see auto_basis for more details. The function local_basis helps the user construct a set of local basis functions (e.g., bisquare functions) from a collection of location and scale parameters.

Slots

- manifold an object of class manifold that contains information on the manifold and the distance measure used on the manifold. See manifold-class for more details
- n the number of basis functions in this set
- fn a list of length n, with each item the function of a specific basis function
- pars a list of parameters where the i-th item in the list contains the parameters of the i-th basis function, fn[[i]]
- df a data frame containing other attributes specific to each basis function (for example the geometric centre of the local basis function)

See Also

auto_basis for automatically constructing basis functions and show_basis for visualising basis functions.

BAUs_from_points *Creates pixels around points*

Description

Takes a SpatialPointsDataFrame and converts it into SpatialPolygonsDataFrame by constructing a tiny (within machine tolerance) BAU around each SpatialPoint.

Usage

```
BAUs_from_points(obj, offset = 1e-10)

## S4 method for signature 'SpatialPoints'
BAUs_from_points(obj, offset = 1e-10)

## S4 method for signature 'ST'
BAUs_from_points(obj, offset = 1e-10)
```

Arguments

obj	object of class SpatialPointsDataFrame
offset	edge size of the mini-BAU (default 1e-10)

Details

This function allows users to mimic standard geospatial analysis where BAUs are not used. Since FRK is built on the concept of a BAU, this function constructs tiny BAUs around the observation and prediction locations that can be subsequently passed on to the functions SRE and FRK. With BAUs_from_points, the user supplies both the data and prediction locations accompanied with covariates.

See Also

auto_BAUs for automatically constructing generic BAUs.

Examples

```
library(sp)
opts_FRK$set("parallel",0L)
df <- data.frame(x = rnorm(10),
                 y = rnorm(10))
coordinates(df) <- ~x+y
BAUs <- BAUs_from_points(df)
```

---

coef *Retrieve estimated regression coefficients*

Description

Takes a an object of class SRE and returns a numeric vector with the estimated regression coefficients.

Usage

```
coef(object, ...)

## S4 method for signature 'SRE'
coef(object, ...)
```

Arguments

object	object of class SRE
...	currently unused

See Also

SRE for more information on how to construct and fit an SRE model.

Examples

```
library(sp)
simdata <- SpatialPointsDataFrame(
               coords = matrix(runif(100), 50, 2),
                data = data.frame(z = rnorm(50)))
BAUs <- BAUs_from_points(SpatialPoints(simdata))
BAUs$fs <- 1
S <- SRE(f = z ~ 1 + coords.x1,
         basis = local_basis(plane()),
         BAUs = BAUs,
         data = list(simdata))
est_reg_coeff <- coef(S)
```

---

data.frame<- *Basis-function data frame object*

Description

Tools for retrieving and manipulating the data frame within the Basis objects. Use the assignment `data.frame()<-` with care; no checks are made to make sure the data frame conforms with the object. Only use if you know what you're doing.

Usage

```
data.frame(x) <- value

## S4 method for signature 'Basis'
x$name

## S4 replacement method for signature 'Basis'
x$name <- value

## S4 replacement method for signature 'Basis'
data.frame(x) <- value

## S4 replacement method for signature 'TensorP_Basis'
data.frame(x) <- value

## S3 method for class 'Basis'
as.data.frame(x, ...)

## S3 method for class 'TensorP_Basis'
as.data.frame(x, ...)
```

Arguments

`x`	the obect of class `Basis` we are assigning the new data to or retrieving data from

value	the new data being assigned to the Basis object
name	the field name to which values will be retrieved or assigned inside the Basis object's data frame
...	unused

Examples

```
G <- local_basis()
df <- data.frame(G)
print(df$res)
df$res <- 2
data.frame(G) <- df
```

---

df_to_SpatialPolygons *Convert data frame to SpatialPolygons*

Description

Convert data frame to SpatialPolygons object.

Usage

```
df_to_SpatialPolygons(df, keys, coords, proj)
```

Arguments

df	data frame containing polygon information, see details
keys	vector of variable names used to group rows belonging to the same polygon
coords	vector of variable names identifying the coordinate columns
proj	the projection of the `SpatialPolygons` object. Needs to be of class `CRS`

Details

Each row in the data frame `df` contains both coordinates and labels (or keys) that identify to which polygon the coordinates belong. This function groups the data frame according to `keys` and forms a `SpatialPolygons` object from the coordinates in each group. It is important that all rings are closed, that is, that the last row of each group is identical to the first row. Since `keys` can be of length greater than one, we identify each polygon with a new key by forming an MD5 hash made out of the respective `keys` variables that in themselves are unique (and therefore the hashed key is also unique). For lon-lat coordinates use `proj = CRS("+proj=longlat +ellps=sphere")`.

Examples

```
library(sp)
df <- data.frame(id = c(rep(1,4),rep(2,4)),
                 x = c(0,1,0,0,2,3,2,2),
                 y=c(0,0,1,0,0,1,1,0))
pols <- df_to_SpatialPolygons(df,"id",c("x","y"),CRS())
## Not run: plot(pols)
```

`dist-matrix` *Distance Matrix Computation from Two Matrices*

Description

This function extends `dist` to accept two arguments.

Usage

```
distR(x1, x2 = NULL)
```

Arguments

`x1`	matrix of size N1 x n
`x2`	matrix of size N2 x n

Details

Computes the distances between the coordinates in `x1` and the coordinates in `x2`. The matrices `x1` and `x2` do not need to have the same number of rows, but need to have the same number of columns (dimensions).

Value

Matrix of size N1 x N2

Examples

```
A <- matrix(rnorm(50),5,10)
D <- distR(A,A[-3,])
```

`distance` *Compute distance*

Description

Compute distance using object of class `measure` or `manifold`.

Usage

```
distance(d, x1, x2 = NULL)

## S4 method for signature 'measure'
distance(d, x1, x2 = NULL)

## S4 method for signature 'manifold'
distance(d, x1, x2 = NULL)
```

Arguments

`d`	object of class `measure` or `manifold`
`x1`	first coordinate
`x2`	second coordinate

See Also

`real_line`, `plane`, `sphere`, `STplane` and `STsphere` for constructing manifolds, and `distances` for the type of distances available.

Examples

```
distance(sphere(),matrix(0,1,2),matrix(10,1,2))
distance(plane(),matrix(0,1,2),matrix(10,1,2))
```

`distances` *Pre-configured distances*

Description

Useful objects of class `distance` included in package.

Usage

```
measure(dist, dim)

Euclid_dist(dim = 2L)

gc_dist(R = NULL)

gc_dist_time(R = NULL)
```

Arguments

`dist`	a function taking two arguments `x1,x2`
`dim`	the dimension of the manifold (e.g., 2 for a plane)
`R`	great-circle radius

Details

Initialises an object of class `measure` which contains a function `dist` used for computing the distance between two points. Currently the Euclidean distance and the great-circle distance are included with `FRK`.

Examples

```
M1 <- measure(distR,2)
D <- distance(M1,matrix(rnorm(10),5,2))
```

draw_world *Draw a map of the world with country boundaries.*

Description

Layers a ggplot2 map of the world over the current ggplot2 object.

Usage

```
draw_world(g = ggplot() + theme_bw() + xlab("") + ylab(""),
  inc_border = TRUE)
```

Arguments

g	initial ggplot object
inc_border	flag indicating whether a map border should be drawn or not; see details.

Details

This function uses ggplot2::map_data in order to create a world map. Since, by default, this creates lines crossing the world at the (-180,180) longitude boundary, function .homogenise_maps is used to split the polygons at this boundary into two. If inc_border is TRUE, then a border is drawn around the lon-lat space; this option is most useful for projections that do not yield rectangular plots (e.g., the sinusoidal global projection).

See Also

the help file for the dataset worldmap

Examples

```
## Not run:
library(ggplot2)
draw_world(g = ggplot())
## End(Not run)
```

eval_basis *Evaluate basis functions*

Description

Evaluate basis functions at points or average functions over polygons.

Usage

```
eval_basis(basis, s)

## S4 method for signature 'Basis,matrix'
eval_basis(basis, s)

## S4 method for signature 'Basis,SpatialPointsDataFrame'
eval_basis(basis, s)

## S4 method for signature 'Basis,SpatialPolygonsDataFrame'
eval_basis(basis, s)

## S4 method for signature 'Basis,STIDF'
eval_basis(basis, s)

## S4 method for signature 'TensorP_Basis,matrix'
eval_basis(basis, s)

## S4 method for signature 'TensorP_Basis,STIDF'
eval_basis(basis, s)

## S4 method for signature 'TensorP_Basis,STFDF'
eval_basis(basis, s)
```

Arguments

`basis`	object of class `Basis`
`s`	object of class `matrix`, `SpatialPointsDataFrame` or `SpatialPolygonsDataFrame` containing the spatial locations/footprints

Details

This function evaluates the basis functions at isolated points, or averages the basis functions over polygons, for computing the matrix S. The latter operation is carried out using Monte Carlo integration with 1000 samples per polygon. When using space-time basis functions, the object must contain a field `t` containing a numeric representation of the time, for example, containing the number of seconds, hours, or days since the first data point.

See Also

`auto_basis` for automatically constructing basis functions.

Examples

```
library(sp)

### Create a synthetic dataset
set.seed(1)
d <- data.frame(lon = runif(n=1000,min = -179, max = 179),
```

```
                 lat = runif(n=1000,min = -90, max = 90),
                 z = rnorm(5000))
coordinates(d) <- ~lon + lat
proj4string(d)=CRS("+proj=longlat")

### Now create basis functions on sphere
G <- auto_basis(manifold = sphere(),data=d,
                nres = 2,prune=15,
                type = "bisquare",
                subsamp = 20000)

### Now evaluate basis functions at origin
S <- eval_basis(G,matrix(c(0,0),1,2))
```

FRK *Construct SRE object, fit and predict*

Description

The Spatial Random Effects (SRE) model is the central object in FRK. The function FRK provides a wrapper for the construction and estimation of the SRE object from data, using the functions SRE (the object constructor) and SRE.fit (for fitting it to the data). Please see SRE-class for more details on the SRE object's properties and methods.

Usage

```
FRK(f, data, basis = NULL, BAUs = NULL, est_error = TRUE,
  average_in_BAU = TRUE, fs_model = "ind", vgm_model = NULL,
  K_type = "block-exponential", n_EM = 100, tol = 0.01, method = "EM",
  lambda = 0, print_lik = FALSE, ...)

SRE(f, data, basis, BAUs, est_error = TRUE, average_in_BAU = TRUE,
  fs_model = "ind", vgm_model = NULL, K_type = "block-exponential",
  normalise_basis = TRUE)

SRE.fit(SRE_model, n_EM = 100L, tol = 0.01, method = "EM", lambda = 0,
  print_lik = FALSE)

SRE.predict(SRE_model, obs_fs = FALSE, newdata = NULL, pred_polys = NULL,
  pred_time = NULL, covariances = FALSE)

## S4 method for signature 'SRE'
predict(object, newdata = NULL, obs_fs = FALSE,
  pred_polys = NULL, pred_time = NULL, covariances = FALSE)

loglik(SRE_model)
```

Arguments

`f`	`R` formula relating the dependent variable (or transformations thereof) to covariates
`data`	list of objects of class `SpatialPointsDataFrame`, `SpatialPolygonsDataFrame`, `STIDF`, or `STFDF`. If using space-time objects, the data frame must have another field, `t`, containing the time index of the data point
`basis`	object of class `Basis` (or `TensorP_Basis`)
`BAUs`	object of class `SpatialPolygonsDataFrame`, `SpatialPixelsDataFrame`, `STIDF`, or `STFDF`. The object's data frame must contain covariate information as well as a field `fs` describing the fine-scale variation up to a constant of proportionality. If the function `FRK` is used directly, then BAUs are created automatically, but only coordinates can then be used as covariates
`est_error`	flag indicating whether the measurement-error variance should be estimated from variogram techniques. If this is set to 0, then `data` must contain a field `std`. Measurement-error estimation is currently not implemented for spatio-temporal datasets
`average_in_BAU`	if `TRUE`, then multiple data points falling in the same BAU are averaged; the measurement error of the averaged data point is taken as the average of the individual measurement errors
`fs_model`	if "ind" then the fine-scale variation is independent at the BAU level. If "ICAR", then an ICAR model for the fine-scale variation is placed on the BAUs
`vgm_model`	an object of class `variogramModel` from the package `gstat` constructed using the function `vgm`. This object contains the variogram model that will be fit to the data. The nugget is taken as the measurement error when `est_error = TRUE`. If unspecified, the variogram used is `gstat::vgm(1,"Lin",d,1)`, where `d` is approximately one third of the maximum distance between any two data points
`K_type`	the parameterisation used for the `K` matrix. Currently this can be "unstructured" or "block-exponential" (default)
`n_EM`	maximum number of iterations for the EM algorithm
`tol`	convergence tolerance for the EM algorithm
`method`	parameter estimation method to employ. Currently only "EM" is supported
`lambda`	ridge-regression regularisation parameter for when `K` is unstructured (0 by default). Can be a single number, or a vector (one parameter for each resolution)
`print_lik`	flag indicating whether likelihood value should be printed or not after convergence of the EM estimation algorithm
`...`	other parameters passed on to `auto_basis` and `auto_BAUs` when calling the function `FRK`
`normalise_basis`	flag indicating whether to normalise the basis functions so that they reproduce a stochastic process with approximately constant variance spatially
`SRE_model`	object returned from the constructor `SRE()` containing all the parameters and information on the SRE model

obs_fs — flag indicating whether the fine-scale variation sits in the observation model (systematic error, Case 1) or in the process model (fine-scale process variation, Case 2, default)

newdata — object of class SpatialPoylgons indicating the regions over which prediction will be carried out. The BAUs are used if this option is not specified

pred_polys — deprecated. Please use newdata instead

pred_time — vector of time indices at which prediction will be carried out. All time points are used if this option is not specified

covariances — logical variable indicating whether prediction covariances should be returned or not. If set to TRUE, a maximum of 4000 prediction locations or polygons are allowed.

object — object of class SRE

Details

SRE() is the main function in the package: It constructs a spatial random effects model from the user-defined formula, data object, basis functions and a set of Basic Areal Units (BAUs). The function first takes each object in the list data and maps it to the BAUs – this entails binning the point-referenced data into the BAUs (and averaging within the BAU) if average_in_BAU = TRUE, and finding which BAUs are influenced by the polygon datasets. Following this, the incidence matrix Cmat is constructed, which appears in the observation model $Z = CY + C\delta + e$, where C is the incidence matrix and δ is systematic error at the BAU level.

The SRE model for the hidden process is given by $Y = T\alpha + S\eta + \xi$, where T are the covariates at the BAU level, α are the regression coefficients, S are the basis functions evaluated at the BAU level, η are the basis-function coefficients, and ξ is the fine scale variation (at the BAU level). The covariance matrix of ξ is diagonal, with its diagonal elements proportional to the field 'fs' in the BAUs (typically set to one). The constant of proportionality is estimated in the EM algorithm. All required matrices (S, T etc.) are initialised using sensible defaults and returned as part of the object, please see SRE-class for more details.

SRE.fit() takes an object of class SRE and estimates all unknown parameters, namely the covariance matrix K, the fine scale variance (σ^2_ξ or σ^2_δ, depending on whether Case 1 or Case 2 is chosen; see the vignette) and the regression parameters α. The only method currently implemented is the Expectation Maximisation (EM) algorithm, which the user configures through n_EM and tol. The log-likelihood (given in Section 2.2 of the vignette) is evaluated at each iteration at the current parameter estimate, and convergence is assumed to have been reached when this quantity stops changing by more than tol.

The actual computations for the E-step and M-step are relatively straightforward. The E-step contains an inverse of an $r \times r$ matrix, where r is the number of basis functions which should not exceed 2000. The M-step first updates the matrix K, which only depends on the sufficient statistics of the basis-function coefficients η. Then, the regression parameter α is updated and a simple optimisation routine (a line search) is used to update the fine-scale variance σ^2_δ or σ^2_ξ. If the fine-scale errors and measurement random errors are homoscedastic, then a closed-form solution is available for the update of σ^2_ξ or σ^2_δ. Irrespectively, since the udpates of α, and σ^2_δ or σ^2_ξ, are dependent, these two updates are iterated until the change in $\sigma^2_\cdot$ is no more than 0.1%. Information on the fitting (convergence etc.) can be extracted using info_fit(SRE_model).

The function FRK acts as a wrapper for the functions SRE and SRE.fit. An added advantage of using FRK directly is that it automatically generates BAUs and basis functions based on the data. Hence FRK can be called using only a list of data objects and an R formula, although the R formula can only contain space or time as covariates when BAUs are not explicitly supplied with the covariate data.

Once the parameters are fitted, the SRE object is passed onto the function predict() in order to carry out optimal predictions over the same BAUs used to construct the SRE model with SRE(). The first part of the prediction process is to construct the matrix S over the prediction polygons. This is made computationally efficient by treating the prediction over polygons as that of the prediction over a combination of BAUs. This will yield valid results only if the BAUs are relatively small. Once the matrix S is found, a standard Gaussian inversion (through conditioning) using the estimated parameters is used for prediction.

predict returns the BAUs, which are of class SpatialPolygonsDataFrame, SpatialPixelsDataFrame, or STFDF, with two added attributes, mu and var. These can then be easily plotted using spplot or ggplot2 (possibly in conjunction with SpatialPolygonsDataFrame_to_df) as shown in the package vignettes.

See Also

SRE-class for details on the SRE object internals, auto_basis for automatically constructing basis functions, and auto_BAUs for automatically constructing BAUs. See also the paper https://arxiv.org/abs/1705.08105 for details on code operation.

Examples

```
library(sp)

### Generate process and data
n <- 100
sim_process <- data.frame(x = seq(0.005,0.995,length=n))
sim_process$y <- 0
sim_process$proc <- sin(sim_process$x*10) + 0.3*rnorm(n)

sim_data <- sim_process[sample(1:n,50),]
sim_data$z <- sim_data$proc + 0.1*rnorm(50)
sim_data$std <- 0.1
coordinates(sim_data) = ~x + y # change into an sp object
grid_BAUs <- auto_BAUs(manifold=real_line(),data=sim_data,
                       nonconvex_hull=FALSE,cellsize = c(0.01),type="grid")
grid_BAUs$fs = 1

### Set up SRE model
G <- auto_basis(manifold = real_line(),
                data=sim_data,
                nres = 2,
                regular = 6,
                type = "bisquare",
                subsamp = 20000)
f <- z ~ 1
S <- SRE(f,list(sim_data),G,
         grid_BAUs,
         est_error = FALSE)
```

```
### Fit with 5 EM iterations so as not to take too much time
S <- SRE.fit(S,n_EM = 5,tol = 0.01,print_lik=TRUE)

### Check fit info


### Predict over BAUs
grid_BAUs <- predict(S)

### Plot
## Not run:
library(ggplot2)
X <- slot(grid_BAUs,"data")
X <- subset(X, x >= 0 & x <= 1)
 g1 <- LinePlotTheme() +
    geom_line(data=X,aes(x,y=mu)) +
    geom_errorbar(data=X,aes(x=x,ymax = mu + 2*sqrt(var), ymin= mu - 2*sqrt(var))) +
    geom_point(data = data.frame(sim_data),aes(x=x,y=z),size=3) +
    geom_line(data=sim_process,aes(x=x,y=proc),col="red")
 print(g1)
## End(Not run)
```

---

info_fit *Retrieve fit information for SRE model*

Description

Takes a an object of class SRE and returns a list containing all the relevant information on parameter estimation

Usage

```
info_fit(SRE_model)

## S4 method for signature 'SRE'
info_fit(SRE_model)
```

Arguments

SRE_model	object of class SRE

See Also

See SRE for more information on the SRE model and available fitting methods.

Examples

```
# See example in the help file for SRE
```

`initialize,manifold-method` *manifold*

Description

Manifold initialisation. This function should not be called directly as `manifold` is a virtual class.

Usage

```
## S4 method for signature 'manifold'
initialize(.Object)
```

Arguments

`.Object`	`manifold` object passed up from lower-level constructor

`isea3h` *ISEA Aperture 3 Hexagon (ISEA3H) Discrete Global Grid*

Description

The data used here were obtained from http://webpages.sou.edu/~sahrk/dgg/isea.old/gen/isea3h.html and represent ISEA discrete global grids (DGGRIDs) generated using the `DGGRID` software. The original .gen files were converted to a data frame using the function `dggrid_gen_to_df`, available with the `dggrids` package. Only resolutions 0–6 are supplied with `FRK` and note that resolution 0 of ISEA3H is equal to resolution 1 in `FRK`. For higher resolutions `dggrids` can be installed from `https://github.com/andrewzm/dggrids` using `devtools`.

Usage

```
isea3h
```

Format

A data frame with 284,208 rows and 5 variables:

id grid identification number within the given resolution

lon longitude coordinate

lat latitude coordinate

res DGGRID resolution (0 – 6)

centroid A 0-1 variable, indicating whether the point describes the centroid of the polygon, or whether it is a boundary point of the polygon

References

Sahr, K. (2008). Location coding on icosahedral aperture 3 hexagon discrete global grids. Computers, Environment and Urban Systems, 32, 174–187.

---

local_basis *Construct a set of local basis functions*

Description

Construct a set of local basis functions based on pre-specified location and scale parameters.

Usage

```
local_basis(manifold = sphere(), loc = matrix(c(1, 0), nrow = 1),
  scale = 1, type = c("bisquare", "Gaussian", "exp", "Matern32"))

radial_basis(manifold = sphere(), loc = matrix(c(1, 0), nrow = 1),
  scale = 1, type = c("bisquare", "Gaussian", "exp", "Matern32"))
```

Arguments

manifold	object of class `manifold`, for example, `sphere`
loc	a matrix of size `n` by `dimensions(manifold)` indicating centres of basis functions
scale	vector of length `n` containing the scale parameters of the basis functions; see details
type	either “bisquare”, “Gaussian”, “exp”, or “Matern32”

Details

This functions lays out local basis functions in a domain of interest based on pre-specified location and scale parameters. If `type` is “bisquare”, then

$$\phi(u) = \left(1 - \left(\frac{\|u\|}{R}\right)^2\right)^2 I(\|u\| < R),$$

and `scale` is given by R, the range of support of the bisquare function. If `type` is “Gaussian”, then

$$\phi(u) = \exp\left(-\frac{\|u\|^2}{2\sigma^2}\right),$$

and `scale` is given by σ, the standard deviation. If `type` is “exp”, then

$$\phi(u) = \exp\left(-\frac{\|u\|}{\tau}\right),$$

and `scale` is given by τ, the e-folding length. If `type` is “Matern32”, then

$$\phi(u) = \left(1 + \frac{\sqrt{3}\|u\|}{\kappa}\right) \exp\left(-\frac{\sqrt{3}\|u\|}{\kappa}\right),$$

and `scale` is given by κ, the function’s scale.

See Also

auto_basis for constructing basis functions automatically, and show_basis for visualising basis functions.

Examples

```
library(ggplot2)
G <-  local_basis(manifold = real_line(),
                  loc=matrix(1:10,10,1),
                  scale=rep(2,10),
                  type="bisquare")
## Not run: show_basis(G)
```

---

manifold *Retrieve manifold*

Description

Retrieve manifold from FRK object.

Usage

```
manifold(.Object)

## S4 method for signature 'Basis'
manifold(.Object)

## S4 method for signature 'TensorP_Basis'
manifold(.Object)
```

Arguments

.Object	FRK object

See Also

real_line, plane, sphere, STplane and STsphere for constructing manifolds.

Examples

```
G <-  local_basis(manifold = plane(),
                  loc=matrix(0,1,2),
                  scale=0.2,
                  type="bisquare")
manifold(G)
```

manifold-class *manifold*

Description

The class manifold is virtual; other manifold classes inherit from this class.

Details

A manifold object is characterised by a character variable type, which contains a description of the manifold, and a variable measure of type measure. A typical measure is the Euclidean distance.

FRK supports five manifolds; the real line (in one dimension), instantiated by using real_line(); the 2D plane, instantiated by using plane(); the 2D-sphere surface S2, instantiated by using sphere(); the R2 space-time manifold, instantiated by using STplane(), and the S2 space-time manifold, instantiated by using STsphere(). User-specific manifolds can also be specified, however helper functions that are manifold specific, such as auto_BAUs and auto_basis, only work with the pre-configured manifolds. Importantly, one can change the distance function used on the manifold to synthesise anisotropy or heterogeneity. See the vignette for one such example.

See Also

real_line, plane, sphere, STplane and STsphere for constructing manifolds.

measure-class *measure*

Description

Measure class used for defining measures used to compute distances between points in objects constructed with the FRK package.

Details

An object of class measure contains a distance function and a variable dim with the dimensions of the Riemannian manifold over which the distance is computed.

See Also

distance for computing a distance and distances for a list of implemented distance functions.

nbasis — *Number of basis functions*

Description

Retrieve the number of basis functions from Basis or SRE object.

Usage

```
nbasis(.Object)

## S4 method for signature 'Basis_obj'
nbasis(.Object)

## S4 method for signature 'SRE'
nbasis(.Object)
```

Arguments

.Object	object of class Basis or SRE

See Also

auto_basis for automatically constructing basis functions.

Examples

```
library(sp)
data(meuse)
coordinates(meuse) = ~x+y # change into an sp object
G <- auto_basis(manifold = plane(),
                data=meuse,
                nres = 2,
                regular=1,
                type = "Gaussian")
print(nbasis(G))
```

NOAA_df_1990 — *NOAA maximum temperature data for 1990–1993*

Description

Maximum temperature data obtained from the National Oceanic and Atmospheric Administration (NOAA) for a part of the USA between 1990 and 1993 (inclusive). See http://iridl.ldeo.columbia.edu/SOURCES/.NOAA/.NCDC/.DAILY/.FSOD/.

Usage

```
NOAA_df_1990
```

Format

A data frame with 196,253 rows and 8 variables:

year year of retrieval

month month of retrieval

day day of retrieval

z dependent variable

proc variable name (Tmax)

id station id

lon longitude coordinate of measurement station

lat latitude coordinate of measurement station

References

National Climatic Data Center, March 1993: Local Climatological Data. Environmental Information summary (C-2), NOAA-NCDC, Asheville, NC.

`nres` *Return the number of resolutions*

Description

Return the number of resolutions from a basis function object.

Usage

```
nres(b)

## S4 method for signature 'Basis'
nres(b)

## S4 method for signature 'TensorP_Basis'
nres(b)

## S4 method for signature 'SRE'
nres(b)
```

Arguments

`b`	object of class `Basis` or `SRE`

See Also

auto_basis for automatically constructing basis functions and show_basis for visualising basis functions.

Examples

```
library(sp)
set.seed(1)
d <- data.frame(lon = runif(n=1000,min = -179, max = 179),
                lat = runif(n=1000,min = -90, max = 90),
                z = rnorm(5000))
coordinates(d) <- ~lon + lat
proj4string(d)=CRS("+proj=longlat")

### Now create basis functions on sphere
G <- auto_basis(manifold = sphere(),data=d,
                nres = 2,prune=15,
                type = "bisquare",
                subsamp = 20000)
nres(G)
```

---

opts_FRK *FRK options*

Description

The main options list for the FRK package.

Usage

```
opts_FRK
```

Format

List of 2

- $ set:function(opt,value)
- $ get:function(opt)

Details

opts_FRK is a list containing two functions, set and get, which can be used to set options and retrieve options, respectively. Currently FRK uses three options:

- "progress": a flag indicating whether progress bars should be displayed or not
- "verbose": a flag indicating whether certain progress messages should be shown or not
- "parallel": an integer indicating the number of cores to use. A number 0 or 1 indicates no parallelism

Examples

```
opts_FRK$set("progress",1L)
opts_FRK$get("parallel")
```

---

plane *plane*

Description

Initialisation of a 2D plane.

Usage

```
plane(measure = Euclid_dist(dim = 2L))
```

Arguments

measure	an object of class measure

Details

A 2D plane is initialised using a `measure` object. By default, the measure object (`measure`) is the Euclidean distance in 2 dimensions, Euclid_dist.

Examples

```
P <- plane()
print(type(P))
print(sp::dimensions(P))
```

---

plotting-themes *Plotting themes*

Description

Formats a ggplot object for neat plotting.

Usage

```
LinePlotTheme()

EmptyTheme()
```

Details

`LinePlotTheme()` creates `ggplot` object with a white background, a relatively large font, and grid lines. `EmptyTheme()` on the other hand creates a `ggplot` object with no axes or legends.

Value

Object of class ggplot

Examples

```
## Not run:
X <- data.frame(x=runif(100),y = runif(100), z = runif(100))
LinePlotTheme() + geom_point(data=X,aes(x,y,colour=z))
EmptyTheme() + geom_point(data=X,aes(x,y,colour=z))
## End(Not run)
```

---

real_line *real line*

Description

Initialisation of the real-line (1D) manifold.

Usage

```
real_line(measure = Euclid_dist(dim = 1L))
```

Arguments

measure	an object of class measure

Details

A real line is initialised using a measure object. By default, the measure object (measure) describes the distance between two points as the absolute difference between the two coordinates.

Examples

```
R <- real_line()
print(type(R))
print(sp::dimensions(R))
```

remove_basis *Removes basis functions*

Description

Takes a an object of class Basis and returns an object of class Basis with selected basis functions removed.

Usage

```
remove_basis(Basis, rmidx)

## S4 method for signature 'Basis'
remove_basis(Basis, rmidx)
```

Arguments

Basis	object of class Basis
rmidx	indices of basis functions to remove

See Also

auto_basis for automatically constructing basis functions and show_basis for visualising basis functions.

Examples

```
library(sp)
df <- data.frame(x = rnorm(10),
                 y = rnorm(10))
coordinates(df) <- ~x+y
G <- auto_basis(plane(),df,nres=1)
data.frame(G) # Print info on basis
G <- remove_basis(G,1:(nbasis(G)-1))
data.frame(G)
```

show_basis *Show basis functions*

Description

Generic plotting function for visualising the basis functions.

Usage

```
show_basis(basis, ...)

## S4 method for signature 'Basis'
show_basis(basis, g = ggplot() + theme_bw() + xlab("") +
  ylab(""))

## S4 method for signature 'TensorP_Basis'
show_basis(basis, g = ggplot())
```

Arguments

basis	object of class Basis
...	not in use
g	object of class gg (a ggplot object) over which to overlay the basis functions (optional)

Details

The function show_basis adapts its behaviour to the manifold being used. With real_line, the 1D basis functions are plotted with colour distinguishing between the different resolutions. With plane, only local basis functions are supported (at present). Each basis function is shown as a circle with diameter equal to the scale parameter of the function. Linetype distinguishes the resolution. With sphere, the centres of the basis functions are shown as circles, with larger sizes corresponding to coarser resolutions. Space-time basis functions of subclass TensorP_Basis are visualised by showing the spatial basis functions and the temporal basis functions in two separate plots.

See Also

auto_basis for automatically constructing basis functions.

Examples

```
library(ggplot2)
library(sp)
data(meuse)
coordinates(meuse) = ~x+y # change into an sp object
G <- auto_basis(manifold = plane(),data=meuse,nres = 2,regular=2,prune=0.1,type = "bisquare")
## Not run: show_basis(G,ggplot()) + geom_point(data=data.frame(meuse),aes(x,y))
```

SpatialPolygonsDataFrame_to_df

SpatialPolygonsDataFrame to df

Description

Convert SpatialPolygonsDataFrame or SpatialPixelsDataFrame object to data frame.

Usage

```
SpatialPolygonsDataFrame_to_df(sp_polys, vars = names(sp_polys))
```

Arguments

`sp_polys`	object of class `SpatialPolygonsDataFrame` or `SpatialPixelsDataFrame`
`vars`	variables to put into data frame (by default all of them)

Details

This function is mainly used for plotting `SpatialPolygonsDataFrame` objects with `ggplot` rather than `spplot`. The coordinates of each polygon are extracted and concatenated into one long data frame. The attributes of each polygon are then attached to this data frame as variables that vary by polygon `id` (the rownames of the object).

Examples

```
library(sp)
library(ggplot2)
opts_FRK$set("parallel",0L)
df <- data.frame(id = c(rep(1,4),rep(2,4)),
                 x = c(0,1,0,0,2,3,2,2),
                 y=c(0,0,1,0,0,1,1,0))
pols <- df_to_SpatialPolygons(df,"id",c("x","y"),CRS())
polsdf <- SpatialPolygonsDataFrame(pols,data.frame(p = c(1,2),row.names=row.names(pols)))
df2 <- SpatialPolygonsDataFrame_to_df(polsdf)
## Not run: ggplot(df2,aes(x=x,y=y,group=id)) + geom_polygon()
```

`sphere` *sphere*

Description

Initialisation of the 2-sphere, S2.

Usage

```
sphere(radius = 6371)
```

Arguments

`radius`	radius of sphere

Details

The 2D surface of a sphere is initialised using a `radius` parameter. The default value of the radius `R` is `R`=6371 km, Earth's radius, while the measure used to compute distances on the sphere is the great-circle distance on a sphere of radius `R`.

Examples

```
S <- sphere()
print(sp::dimensions(S))
```

`SRE-class` *Spatial Random Effects class*

Description

This is the central class definition of the `FRK` package, containing the model and all other information required for estimation and prediction.

Details

The spatial random effects (SRE) model is the model employed in Fixed Rank Kriging, and the `SRE` object contains all information required for estimation and prediction from spatial data. Object slots contain both other objects (for example, an object of class `Basis`) and matrices derived from these objects (for example, the matrix S) in order to facilitate computations.

Slots

`f` formula used to define the SRE object. All covariates employed need to be specified in the object `BAUs`

`data` the original data from which the model's parameters are estimated

`basis` object of class `Basis` used to construct the matrix S

`BAUs` object of class `SpatialPolygonsDataFrame`, `SpatialPixelsDataFrame` of `STFDF` that contains the Basic Areal Units (BAUs) that are used to both (i) project the data onto a common discretisation if they are point-referenced and (ii) provide a BAU-to-data relationship if the data has a spatial footprint

`S` matrix constructed by evaluating the basis functions at all the data locations (of class `Matrix`)

`S0` matrix constructed by evaluating the basis functions at all BAUs (of class `Matrix`)

`D_basis` list of distance-matrices of class `Matrix`, one for each basis-function resolution

`Ve` measurement-error variance-covariance matrix (typically diagonal and of class `Matrix`)

`Vfs` fine-scale variance-covariance matrix at the data locations (typically diagonal and of class `Matrix`) up to a constant of proportionality estimated using the EM algorithm

`Vfs_BAUs` fine-scale variance-covariance matrix at the BAU centroids (typically diagonal and of class `Matrix`) up to a constant of proportionality estimated using the EM algorithm

`Qfs_BAUs` fine-scale precision matrix at the BAU centroids (typically diagonal and of class `Matrix`) up to a constant of proportionality estimated using the EM algorithm

`Z` vector of observations (of class `Matrix`)

`Cmat` incidence matrix mapping the observations to the BAUs

`X` matrix of covariates

K_type type of prior covariance matrix of random effects. Can be "block-exponential" (correlation between effects decays as a function of distance between the basis-function centroids), or "unstructured" (all elements in K are unknown and need to be estimated)

mu_eta updated expectation of random effects (estimated)

S_eta updated covariance matrix of random effects (estimated)

Q_eta updated precision matrix of random effects (estimated)

Khat prior covariance matrix of random effects (estimated)

Khat_inv prior precision matrix of random effects (estimated)

alphahat fixed-effect regression coefficients (estimated)

sigma2fshat fine-scale variation scaling (estimated)

fs_model type of fine-scale variation (independent or CAR-based). Currently only "ind" is permitted

info_fit information on fitting (convergence etc.)

See Also

SRE for details on how to construct and fit SRE models.

STplane *plane in space-time*

Description

Initialisation of a 2D plane with a temporal dimension.

Usage

```
STplane(measure = Euclid_dist(dim = 3L))
```

Arguments

measure an object of class measure

Details

A 2D plane with a time component added is initialised using a measure object. By default, the measure object (measure) is the Euclidean distance in 3 dimensions, Euclid_dist.

Examples

```
P <- STplane()
print(type(P))
print(sp::dimensions(P))
```

`STsphere` *Space-time sphere*

Description

Initialisation of a 2-sphere (S2) with a temporal dimension

Usage

```
STsphere(radius = 6371)
```

Arguments

`radius`	radius of sphere

Details

As with the spatial-only sphere, the sphere surface is initialised using a `radius` parameter. The default value of the radius `R` is `R`=6371, which is the Earth's radius in km, while the measure used to compute distances on the sphere is the great-circle distance on a sphere of radius `R`. By default Euclidean geometry is used to factor in the time component, so that dist((s1,t1),(s2,t2)) = sqrt(gc_dist(s1,s2)^2 + (t1 - t2)^2). Frequently this distance can be used since separate correlation length scales for space and time are estimated in the EM algorithm (that effectively scale space and time separately).

Examples

```
S <- STsphere()
print(sp::dimensions(S))
```

`TensorP` *Tensor product of basis functions*

Description

Constructs a new set of basis functions by finding the tensor product of two sets of basis functions.

Usage

```
TensorP(Basis1, Basis2)

## S4 method for signature 'Basis,Basis'
TensorP(Basis1, Basis2)
```

Arguments

Basis1	first set of basis functions
Basis2	second set of basis functions

See Also

auto_basis for automatically constructing basis functions and show_basis for visualising basis functions.

Examples

```
library(spacetime)
library(sp)
library(dplyr)
sim_data <- data.frame(lon = runif(20,-180,180),
                       lat = runif(20,-90,90),
                       t = 1:20,
                       z = rnorm(20),
                       std = 0.1)
time <- as.POSIXct("2003-05-01",tz="") + 3600*24*(sim_data$t-1)
space <- sim_data[,c("lon","lat")]
coordinates(space) = ~lon+lat # change into an sp object
proj4string(space)=CRS("+proj=longlat +ellps=sphere")
STobj <- STIDF(space,time,data=sim_data)
G_spatial <- auto_basis(manifold = sphere(),
                        data=as(STobj,"Spatial"),
                        nres = 1,
                        type = "bisquare",
                        subsamp = 20000)
G_temporal <- local_basis(manifold=real_line(),loc = matrix(c(1,3)),scale = rep(1,2))
G <- TensorP(G_spatial,G_temporal)
# show_basis(G_spatial)
# show_basis(G_temporal)
```

---

type *Type of manifold*

Description

Retrieve slot type from object

Usage

```
type(.Object)

## S4 method for signature 'manifold'
type(.Object)
```

Arguments

`.Object` object of class `Basis` or `manifold`

See Also

`real_line`, `plane`, `sphere`, `STplane` and `STsphere` for constructing manifolds.

Examples

```
S <- sphere()
print(type(S))
```

`worldmap` *World map*

Description

This world map was extracted from the package `maps` v.3.0.1 by running `ggplot2::map_data("world")`. To reduce the data size, only every third point of this data frame is contained in `worldmap`.

Usage

```
worldmap
```

Format

A data frame with 33971 rows and 6 variables:

long longitude coordinate

lat latitude coordinate

group polygon (region) number

order order of point in polygon boundary

region region name

subregion subregion name

References

Original S code by Becker, R.A. and Wilks, R.A. This R version is by Brownrigg, R. Enhancements have been made by Minka, T.P. and Deckmyn, A. (2015) maps: Draw Geographical Maps, R package version 3.0.1.

Index