Package 'sensors4plumes'

April 3, 2017

Type Package

Title Test and Optimise Sampling Designs Based on Plume Simulations

Version 0.9

Date 2017-03-28

Author Kristina B. Helle

Maintainer Kristina B. Helle <kristina.helle@uni-muenster.de>

Description Test sampling designs by several flexible cost functions, usually based on the simulations, and optimise sampling designs using different optimisation algorithms; load plume simulations (on lattice or points) even if they do not fit into memory.

License GPL-3

Imports lattice, emdist, rgdal, FNN, graphics, conf.design, automap, genalg

Depends R (>= 3.0.0), methods, gstat, sp, raster

Collate s4p.R points2polygrid.R SpatialPolygridDataFrame-class.R SpatialPolygridDataFrame-methods.R SpatialIndexDataFrame-class.R SpatialIndexDataFrame-methods.R SpatialDataFrame-class.R subsetSDF.R areaSDF.R Simulations-class.R Simulations-methods.R plotSimulations.R cbindSimulations.R loadSimulations.R loadSimulations_raster.R loadSimulations_scan.R extractSpatialDataFrame.R SDF2simulations.R changeSimulationsPath.R subsetSimulations.R summaryPlumes.R summaryLocations.R replaceDefault.R simulationsApply.R fitMedianVariogram.R interpolate.R interpolationErrorFunctions.R interpolationError.R spatialSpreadFunctions.R spatialSpread.R measurementsResultMap.R optimiseSD_greedy.R optimiseSD_ssa.R optimiseSD_genetic.R completeSearch.R optimiseSD_global.R coordCentral.R optimiseSD_manual.R optimiseSD.R plotSD.R optimisationCurve.R postprocessing.R spplotLog.R

NeedsCompilation no

Repository CRAN

Date/Publication 2017-04-03 15:31:15 UTC

R topics documented:

`sensors4plumes-package`

Test and optimise sampling designs based on plume simulations

---

Description

The motivation of this package comes from radiological emergency preparedness, the functionality was needed for the planning of a network of sensors for the atmospheric gamma dose rate. These sensors should be placed in optimal locations to provide sufficient information in 'all' possible accident scenarios in the area of interest. For this purpose, a model for the potential radioactive pollution is needed; as there is a wide variety of possible accidents, this model has to be stochastic. We know that pollution spreads by the physical laws of dispersion and transport, uncertainty comes from parameters like the wind field, the amount and kind of emitted pollutant and maybe the location of the source. It is difficult to find a parametric model for the distribution of possible pollutions, but we may describe it by a representative set of sample scenarios that can be simulated numerically, varying the uncertain parameters. Each simulation represents a possible scenario for which we can check how well a proposed set of sensors can extract the required information: given the simulated values at the sensor locations we can mimic the process of information retrieval to decide if the result (trigger alarm, delineate evacuation area, etc.) fits the required result given the scenario. Quantifying this fitness and averaging it over the simulations, provides a global estimate about the fitness of a sensor set. This can guide algorithms to find optimal sampling designs.

The functionality of `sensors4plumes` is suited for the planning of monitoring sensor networks in all cases where sensors have to cope with a variety of possible scenarios that can be described by a set of simulations rather than by a parametric field. The package provides functions to load such simulations automatically. To handle data that does not fit into memory, it resorts to the raster-package. It provides several basic functions to quantify the fitness of sensor sets that may be modified by users. These or user-defined fitness functions can arbitrarily be combined with various optimisation algorithms.

Details

Package:	sensors4plumes
Type:	Package
Version:	0.9
Date:	2017-03-28
License:	

Author(s)

Kristina B. Helle

Maintainer: Kristina B. Helle <kristina.helle@uni-muenster.de>

areaSDF *Areas of elements of SpatialDataFrame objects*

Description

Returns the areas of all elements in a SpatialDataFrame object, e.g. of all grid cells in a SpatialPixelsDataFrame or of all Polygons in a SpatialPolygonsDataFrame. If the elements are Points, 0s are returned with a warning.

Usage

```
areaSDF(x)
```

Arguments

x	SpatialDataFrame-class

Value

Vector of the areas of elements of the SpatialDataFrame (same length as the SpatialDataFrame):
SpatialIndexDataFrame, SpatialPointsDataFrame: 0s, with warning.
SpatialPixelsDataFrame, SpatialGridDataFrame: cell size, repeated for each cell
SpatialPolygridDataFrame: sum of cell sizes of all cells related to the respective index value
SpatialPolygonsDataFrame: sum of area of all polygons related to the respective Polygons object
SpatialLinesDataFrame: sum of length of lines related to the respective Lines object

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SIndexDF)
areaSDF(SIndexDF)

data(SPolygridDF)
areaSDF(SPolygridDF)

data(SLinesDF)
areaSDF(SLinesDF)
```

`cbind.Simulations` *Combine plumes of Simulations objects with coinciding parameters*

Description

cbind-like method for `Simulations`: if `locations` are the same and the names of `plumes` and the layer names of the `values` coincide, the plumes of two or more `Simulations` can be combined by combining the values of the `plumes` slot and of the `values` slot.

Usage

```
cbind.Simulations(..., nameSave = NA, overwrite = FALSE)
```

Arguments

`...`	`Simulations`
`nameSave`	name to save resulting raster files in case they are too big to keep in memory and then contain the final data, files are created by appending names to it, starting with "_"
`overwrite`	if files at `nameSave` may be overwritten

Value

A `Simulations` object; if it is too big to keep in memory, it is saved in a file in `nameSave`.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SimulationsSmall)
## Not run:
## may create file
SimulationsSmall2 = cbind.Simulations(SimulationsSmall, SimulationsSmall)

## End(Not run)
```

changeSimulationsPath *Reset file paths in Simulations objects*

Description

Reset file paths in the values of Simulations objects and also in raster objects.

Usage

```
changeSimulationsPath(simulations, path)
```

Arguments

simulations	raster or Simulations
path	character with new path(s) for each of the layers

Value

The function returns simulations with replaced paths to the files. If simulations is in memory, it has no effect.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# if data are taken from files, the paths need to be updated
## Not run:
library(sensors4plumesData)
data(radioactivePlumes_area)
radioactivePlumes_area = changeSimulationsPath(radioactivePlumes_area,
c(paste0(path.package("sensors4plumesData"),
         "/extdata/radioactivePlumes_area_finaldose.grd"),
  paste0(path.package("sensors4plumesData"),
         "/extdata/radioactivePlumes_area_maxdose.grd"),
  paste0(path.package("sensors4plumesData"),
         "/extdata/radioactivePlumes_area_time.grd"))
)

## End(Not run)
```

copySimulations *Copy Simulations (including raster files)*

Description

Copy or move the raster files associated with a Simulations object to another directory and update the paths in the Simulations. The resulting Simulations may be saved.

Usage

```
copySimulations(simulations, newPath, newFile,
  overwrite = FALSE, deleteOld = FALSE)
```

Arguments

simulations	Simulations
newPath	path to the directory where to save the files; if it is missing, the raster files are not copied
newFile	filename where to save the resulting simulations; if missing, the simulations are not saved and just returned; the file is saved in the directory newPath if given and else in the current directory
overwrite	logical if files may be overwritten when copying the raster files – has no influence on saving of the simulations themselves as .Rdata file
deleteOld	logical if the old raster files are to be deleted (move files to new directory instead of copy)

Value

The simulations with paths in the values updated to the new raster files. In addition the raster files are copied or moved to newPath and the simulations may be saved in a .Rdata file. If the values of the simulations are in memory, there are no raster files to be copied. In this case the function will only save the simulations if a newFile is given.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## Not run:
library(sensors4plumesData)
data(radioactivePlumes_area)
radioactivePlumes_area2 =
  copySimulations(radioactivePlumes_area,
  newPath = paste0(path.package("sensors4plumesData"),
  newFile = "radioactivePlumes_area2")

## End(Not run)
```

`extractSpatialDataFrame` *Extract some values of Simulations to a SpatialDataFrame with the same spatial properties.*

Description

Turn some of the `values` of a `Simulations` object into a `SpatialDataFrame`, keeping the spatial reference.

Usage

```
extractSpatialDataFrame(obj, kinds = 1:nlayers(obj@values),
  plumes = 1:ncol(obj@values), chunksize = 1e+7)
```

Arguments

`obj`	`Simulations` object
`kinds`	integer: values of which layers (kind) of the `values` to extract
`plumes`	integer: values of which plume to extract
`chunksize`	limit of cells to extract, if selected kinds and plumes exceed this limit, only the first are extracted (with warning)

Details

This function can be used to extract the maps in `Simulations` objects: the values are assigned to the spatial properties, for plotting, combining it with other spatial data etc. As the size of all `values` may exceed the memory, it may extract only the first of the kinds and plumes.

Value

A `SpatialDataFrame` with the same spatial reference as the `locations` of the `obj` with the values of the chosen `plume` and `layer` as attribute.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SimulationsSmall)
SimulationsSmall_134_21 = extractSpatialDataFrame(
  SimulationsSmall,
  plumes = c(1,3,4), kinds = c(2,1))
```

`interpolate` *Interpolate many maps at once – even if they do not fit into memory*

Description

`interpolate` is a wrapper for `idw0` and `krige0` to interpolate several maps at once if locations of input values and desired output agree, even if output and maybe even input does not fit into memory. Works by writing output (and maybe input and intermediate results) to raster files.

`fitMedianVariogram` is a wrapper of `autofitVariogram`: fits variograms to a sample of plumes of a `simulations` object and generates a variogram with the median parameters.

Usage

```
interpolate(simulations, locations, kinds = 1, fun_interpolation,
  tmpfile = "tmp_interpolate", overwrite = FALSE, chunksize = 1e+7)
fitMedianVariogram(simulations, plumes, locations, kinds = 1)
idw0z(formula = z ~ 1, data, newdata, y, idp = 2)
```

Arguments

`simulations`	`Simulations`
`locations`	indices of `locations` to be used as input; multiple and invalid values are ignored
`plumes`	indices of `plumes` to fit variograms to
`kinds`	layer of the `values` of `simulations` to be used; interpolation can only be applied to one layer: if `values` is a vector, only the first entry is used in both functions.
`fun_interpolation`	interpolation `function`, must have the parameters `y`, `data`, `newdata` (form as for `krige0`). All other parameters need default values, e.g. the `model` of `krige0` that can be set by `replaceDefault` with `type = "interpolation_fun.interpolate"`.
`tmpfile`	filename for the raster file in case the result does not fit into memory; if `FALSE` the function stops with a warning and does not create a file
`overwrite`	boolean, if the file at `tmpfile` may be overwritten
`chunksize`	maximal number of cells to be processed at once – forwarded to `blockSize` inside
`formula`	formula that defines the dependent variable as linear model of independent variables, forwarded to `idw0` – see also there, default is no independent variables
`data`	data frame with dependent variable and coordinates, forwarded to `idw0` – see also there
`newdata`	data frame or Spatial object with prediction locations, forwarded to `idw0` – see also there
`y`	matrix, forwarded to `idw0` – see also there
`idp`	, forwarded to `idw0` – see also there

Value

interpolate returns a RasterLayer-class with the interpolations; it has the same size as the values of the input simulations and belongs to the same locations and plumes. Also projection is the same, so they can be combined by stack. If it does not fit into memory it is saved at tmpfile with the extension "_interpolated.grd". The function may produce intermediate files at tmpfile (with name extensions) that are deleted in the end.

fitMedianVariogram returns a variogram model (vgm).

idw0z is idw0 with one extra default parameter: formula = z ~ 1. This way it can directly be used as fun_interpolation in interpolate.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## Not run:
## takes some time
# get data
data(radioactivePlumes)

# generate median variogram from plumes
\dontrun{
## takes some seconds
medianVariogram = fitMedianVariogram(simulations = radioactivePlumes,
                              plumes = 1:nPlumes(radioactivePlumes),
                              kinds = 1)
}
## the result is in:
data(medianVariogram)

# prepare interpolation function
krige0var = replaceDefault(krige0, newDefaults = list(
  formula = z ~ 1, model = medianVariogram, beta = NA, ... = NA),
  type = "fun_interpolation.interpolate")[[1]]

# sample locations: proposed sensors
sampleLocations = sample.int(nLocations(radioactivePlumes), 50)

# interpolate
interpolated = interpolate(
  simulations = radioactivePlumes,
  kinds = 1,
  locations = sampleLocations,
  fun_interpolation = krige0var)

# combine plot original and interpolated
originalAndInterpolated = radioactivePlumes
originalAndInterpolated@values = stack(
  originalAndInterpolated@values[[1]], interpolated)
```

```
OriginalAndInterpolated = extractSpatialDataFrame(
  originalAndInterpolated, plumes = 1:4)

samplePoints =
  as(OriginalAndInterpolated, "SpatialPointsDataFrame")[sampleLocations,]
spplotLog(OriginalAndInterpolated,
  sp.layout = list("sp.points",
    samplePoints, col = 3))
spplot(OriginalAndInterpolated,
  sp.layout = list("sp.points",
    samplePoints, col = 3))


## End(Not run)
```

`interpolationError` *Interpolate many maps at once, compare them to the original and determine a global error*

Description

This function calls `interpolate`, then it compares the result plume-and-location-wise to the original and summarises the resulting error values.

Usage

```
interpolationError(simulations, locations, kinds,
  fun_interpolation = NA, fun_error = NA,
  fun_Rpl = NA, fun_Rpl_cellStats = "mean", fun_l = NA,
  tmpfile = "tmp_interpolationError", overwrite = FALSE, chunksize = 1e+7)
```

Arguments

`simulations`	`Simulations`
`locations`	indices of `locations` to be used as input; multiple and invalid values are ignored
`kinds`	layer of the `values` of `simulations` to be used; interpolation can only be applied to one layer: if `kinds` is a vector, only the first entry is used
`fun_interpolation`	interpolation `function`, must have the parameters `y`, `data`, `newdata` (form as for `krige0`). All other parameters need default values, e.g. the `model` of `krige0` that can be set by `replaceDefault` with `type = "interpolation_fun.interpolate"`.
`fun_error`	function to compare original and interpolated map location-and-plume-wise; must have a parameter `x`, then `x[1]` is the original and `x[2]` the interpolated value; it is forwarded to `simulationsApply` as `fun_pl`, therefore it has to fulfil all requirements for such functions; `interpolationErrorFunctions` provides some common examples

fun_Rpl	function to summarise the location-and-plume-wise errors, is forwarded to simulationsApply as fun_Rpl; if input does not fit into memory, it cannot be applied (causes warning)
fun_Rpl_cellStats	alternative function to summarise the location-and-plume-wise errors, is forwarded to simulationsApply as fun_Rpl_cellStats
fun_l	function to compare original and interpolated location-wise, i.e. generate one global map that takes into account all plumes; must have parameter x, then x[,1] refers to the original and x[,2] to the interpolated values; it is forwarded to simulationsApply as fun_pl, therefore it has to fulfil all requirements for such functions; interpolationErrorFunctions provides some common examples
tmpfile	filename for the raster file in case the result does not fit into memory; if FALSE the function stops with a warning and does not create a file
overwrite	boolean, if the file at tmpfile may be overwritten
chunksize	maximal number of cells to be processed at once – forwarded to blockSize inside

Value

List of values and rasters (of same dimension as the values of the simulations):

"cost": result of fun_Rpl if available (if not, warning), else result of fun_Rpl_cellStats (to guarantee that there is always a value)

"cost_cellStats": result of fun_Rpl_cellStats (if this is not in "cost")

"error_locationsplumes": raster, result of fun_error

"interpolated": result of the interpolation with fun_interpolation

"costLocations": result of fun_l

Author(s)

kristina.helle@uni-muenster.de

Examples

```
data(radioactivePlumes)
## preparation
idw0z = replaceDefault(idw0, newDefaults = list(
  formula = z ~ 1))[[1]]
sampleLocations100 = sample.int(nLocations(radioactivePlumes), 100)
fun_Rpl_mean = function(x, nout = 1){
  mean(x[,1], na.rm = TRUE)
}
## compute interpolation error
## Not run:
## takes some seconds
  interpolationError_delineation <- interpolationError(
    simulations = radioactivePlumes,
```

```
    locations = sampleLocations100,
    kinds = 2,
    fun_interpolation = idw0z,
    fun_error = delineationError,
    fun_Rpl = fun_Rpl_mean,
    fun_Rpl_cellStats = "mean",
    fun_l = delineationErrorMap
  )
# cost
interpolationError_delineation[["cost_cellStats"]]
## plot error map
interpolationErrorMaps = radioactivePlumes
interpolationErrorMaps@values =
  stack(radioactivePlumes@values[[2]],
        interpolationError_delineation[["interpolated"]],
        interpolationError_delineation[["error_locationsplumes"]][[1]])
interpolationErrorMapsSDF = extractSpatialDataFrame(interpolationErrorMaps, plumes = 1:5)
interpolationErrorMapsSDF@data$costMap = interpolationError_delineation[["costLocations"]]
# original, interpolated, error (1: overestimation, 5: underestimation)
spplotLog(interpolationErrorMapsSDF, zcol = 1:15)
# error summary - mean error of all plumes
spplot(interpolationErrorMapsSDF, zcol = "costMap")


## End(Not run)
```

`interpolationErrorFunctions` *Compare original and interpolated maps point-wise*

Description

`absError` and `delineationError` compare an interpolated to an original map (the values in each point respectively). They are to be used as `fun_error` in `interpolationError`. `absError` is the absolute difference between the values. `delineationError` compares areas above a given threshold, and indicates false positive and false negative classification.

`absErrorMap` and `delineationErrorMap` are to be used as `fun_l` in `interpolationError`, they compute the average over all plumes of the respective error functions to generate a common map.

Usage

```
absError(x, nout = 1)
delineationError(x, nout = 1, threshold = 1e-7, weightFalseNeg = 5)
absErrorMap(x, nout = 1)
delineationErrorMap(x, nout = 1, threshold = 1e-7, weightFalseNeg = 5)
```

Arguments

x	vector of length 2 with x[1] the value and x[2] the interpolated value
nout	length of output – needed as the function is to be called via simulationsApply as fun_pl or fun_l
threshold	threshold to classify all locations
weightFalseNeg	weight for false negative classification, false positive classification is weighted 1

Value

absError returns a single numeric value, the absolute difference.

delineationError first computes if the original is above the threshold and if the interpolated is above the threshold; then it determines the result: 0 if classifications agree, 1 for false positive (i.e. original is below threshold, interpolated is above), and weightFalseNeg for false negative.

absErrorMap and delineationErrorMap return single values.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## see interpolationError
```

---

loadSimulations *Load values from raster or text files into Simulations objects*

Description

Plume simulations for one scenario are a set of maps of the same area, each showing one plume. This function generates a Simulations object from all data of a scenario which can then be used to compute cost and to optimise sampling designs. Data may extend the size of the memory; it can be provided either as raster maps or as values in text files (in this case spatial reference can be added later). The resulting values are saved to disk as a raster object.

Usage

```
loadSimulations(basicPath = ".", readBy = NA,
  multilayer = "kinds", region, bBox,
  nameSave = NA, overwrite = FALSE, ...)
```

Arguments

basicPath	path to directory with files; it must not contain any but the files with simulations to be used
multilayer	way to interpret files with several layers/columns; "kinds" (default): layers/columns are interpreted as different kinds of values (number of layers/columns must be the same in all files) or "plumes": layers/columns are interpreted as values of different plumes, all being of the same kind
readBy	method to read the files; for some suffixes the reading method is determined automatically (all files must have the same suffix): "scan" for ".txt" or ".csv", and "raster" for ".tif" or ".grd"; else it has to be specified here, (default is NA)
region	logical indicating in which locations to keep the values (for raster files, order is row-wise), length must equal number of locations (if combined with bBox length must fit locations inside bBox)
bBox	bounding box of locations to keep, given as vector c(xmin, xmax, ymin, ymax); works only for raster data, for text files it is ignored
nameSave	name to save resulting raster files that are created automatically and then contain the final data, files are created by appending names to it, starting with "_"
overwrite	logical if files at nameSave may be overwritten
...	parameters to be forwarded to read.table and scan like skip, sep, dec

Details

Plume simulations have to be a set of maps all defined by values at the same set of locations; there may be several kinds of maps for each plume. Such data can be loaded from files automatically if these have the following properties:
Files can either all be loaded by brick (.grd/.gri, .tif) or by scan (.txt, .csv). If they have one of the listed suffixes, the method is derived automatically; for other suffixes it must be given by the user in readBy.
Each file must contain the values of a plume in all locations.
Files may contain several kinds of values; in this case each layer (of .tif or .grd files) or each column (in .txt or .csv files) belongs to different kind of map, the structure (number and meaning of columns/layers and rows) must be common to all files.
If there is only one kind of map, files may contain columns/layers with the values of several plumes.
File names must follow some rules: They must not contain any "." except the one to start the suffix. They must not contain "_", except for the case described below; they are interpreted as the names of plumes. If there are several kinds of maps, either all kinds for a plume are in the same file or there is a separate file for each kind and each plume. In the latter case file names must be of the form plumeName_kindName.suffix. For each plume there must be a file for each kind.

Value

The output of this function depends on the file type: raster files result in a Simulations object, with spatial properties kept in the locations (projection etc. of GeoTiff); text files do not contain spatial information, therefore a list of plumes and values is returned that can be used to generate a Simulations object together with appropriate locations.

The function may generate (and overwrite or delete) files at nameSave.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## Not run:
# examples for possible input is given in package sensors4plumesData in the
# subfolders of inst/extdata/fileFormats

# from raster files
Simulations1 = loadSimulations(
  basicPath = paste0(path.package("sensors4plumesData"),
                     "/extdata/fileFormats/raster"),
  nameSave = "t1_",
  overwrite = TRUE
)

# from text files
Simulations2 = loadSimulations(
  basicPath = paste0(path.package("sensors4plumesData"),
                     "/extdata/fileFormats/text_multicolumn"),
  readBy = "scan",
  nameSave = "t2_",
  overwrite = TRUE,
  skip = 1
)

# from multilayer raster files, selecting locations in a bounding box
region = as.logical(replicate(20, sample.int(2,1)) - 1)
bBox = c(0.2, 0.7, 0.4, 0.8)

Simulations3 = loadSimulations(
  basicPath = paste0(path.package("sensors4plumesData"),
                     "/extdata/fileFormats/raster_multilayer"),
  nameSave = "t3_",
  region = region,
  bBox = bBox,
  overwrite = TRUE
)

## End(Not run)
```

---

measurementsResult *General cost function by plume-wise summary of values at locations*

Description

It evaluates values at locations plume-wise and summarises the result by calls to simulationsApply.

Usage

```
measurementsResult(simulations, locations, kinds,
  fun_p = NA, fun_Rp = NA, fun_pl = NA, fun_Rpl = NA)
```

Arguments

simulations	Simulations object
locations	indices of locations, i.e. rows of simulations@values to be taken into account; if nothing indicated, all rows are used (invalid)
kinds	index or name of the layer of simulations@values to be used
fun_p	function for plume-wise summary; forwarded to simulationsApply – see there for details
fun_Rp	function to summarise the result of fun_p; forwarded to simulationsApply – see there for details
fun_pl	function for value-wise summary; forwarded to simulationsApply – see there for details
fun_Rpl	function to summarise the result of fun_pl; forwarded to simulationsApply – see there for details

Details

It is a general cost function, after specifying some parameters via replaceDefault with type = "costFun" it can be used as costFun in optimiseSD. Examples are cost functions related to plume detection as given in measurementsResultFunctions.

Value

"cost": result of fun_Rp (must be a single value in order to qualify as costFun in optimiseSD)
"costPlumes": result of fun_p, a matrix where each row represents a plume.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
demo(radioactivePlumes_addProperties)

# sensor locations
sampleLocations1 = sample.int(nLocations(radioactivePlumes), 10)

# modify 'measurementsResult' to cost function 'singleDetection'
singleDetection = replaceDefault(measurementsResult, newDefaults = list(
  kinds = "detectable",
  fun_p = function(x, nout = 1){
    y = 1 - max(x)
    if (length(x) == 0){
      y = 1
```

```
    }
    return(y)
    },
  fun_Rp = function(x, weight = 1, nout = 1){
    mean(x * weight$totalDose)/mean(weight$totalDose)
  }),
  type = "costFun.optimiseSD")[[1]]

# compute cost for sensors at 'sampleLocations1'
  singleDetection1 = singleDetection(
    simulations = radioactivePlumes,
    locations = sampleLocations1)

# results
# global cost: fraction of non detected plumes, weighted by their total dose:
singleDetection1[["cost"]]
singleDetection1[["costPlumes"]] # for each plume if it is detected (0) or not (1)
```

`medianVariogram` *Variogram of radioactivePlumes*

Description

A {`variogramModel`} generated from `kind = "finaldose"` of `radioactivePlumes` with the code given in the examples of `fitMedianVariogram`.

Usage

```
data("medianVariogram")
```

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(medianVariogram)
```

`mesurementsResultFunctions`
Cost functions by plume-wise summary of values at locations

Description

`singleDetection` is average total dose of non-detected plumes.

`multipleDetection` is the average fraction of possible additional detections: given a plume can be detected in 10 locations and the given sensors detect it in 3, then the cost would be 7/10; this is averaged over all plumes.

`earlyDetection` is the average time between the plume entering the area and its detection.

Usage

```
singleDetection(simulations, locations, plot = FALSE)
multipleDetection(simulations, locations, plot = FALSE)
earlyDetection(simulations, locations, plot = FALSE)
```

Arguments

simulations	Simulations object, must contain the required data, see details.
locations	indices of locations, sensor positions
plot	if a map is generated (takes time), only implemented yet for singleDetection, else it is automatically set to FALSE

Value

"cost": global cost "costPlumes": plume-wise intermediate result, not to be interpreted as plume-specific cost

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# load data
demo(radioactivePlumes_addProperties)

# define 'min' function without warning for empty sets
min_ = function(x, na.rm = TRUE){
  if (length(x) == 0){
    out = Inf
  } else {
    out = min(x, na.rm = na.rm)
  }
  return(out)
}

# preprocess data to provide required input
### earliest possible detection of plume inside the area
radioactivePlumes@plumes$earliestDetection =
  summaryPlumes(radioactivePlumes, fun = min_, kinds = "time", na.rm = TRUE)[[2]]

# sample locations (sensors)
sampleLocations1 = sample.int(nLocations(radioactivePlumes), 10)

# compute cost
singleDetection1 = singleDetection(
  simulations = radioactivePlumes,
  locations = sampleLocations1)

multipleDetection1 = multipleDetection(
  simulations = radioactivePlumes,
```

```
  locations = sampleLocations1)

earlyDetection1 = earlyDetection(
  simulations = radioactivePlumes,
  locations = sampleLocations1)
```

---

optimalSD *Optimised sampling designs*

Description

For each of the optimisation algorithms a resulting sampling design is provided. These are taken from the examples of the respective cost functions.

Usage

```
data(SDgenetic)
data(SDglobal)
data(SDgreedy)
data(SDmanual)
data(SDssa)
```

Format

SDgenetic: list
SDglobal: list
SDgreedy: list
SDmanual: list
SDssa: list

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SDgenetic)
```

`optimisationCurve` *Plot Optimisation Curve*

Description

Generates algorithm-specific plots of optimisation or optimal sampling design(s).

Usage

```
optimisationCurve(optSD, type, nameSave, ...)
```

Arguments

`optSD`	result of `optimiseSD`
`type`	character, indicating type of `optimisationFun` used, of `"ssa"`, `"genetic"`, `"greedy"`, `"global"`, `"manual"`
`nameSave`	path of file where to save, without suffix, generates a .png file
`...`	parameters to be forwarded to png

Details

The type of plot depends on the algorithm:

`"ssa"`: curve of the cost in each iteration: proposed design (red dot), accepted design (blue line), best design until now (green line)

`"genetic"`: two plots in one panel. Optimisation curve: mean (blue line) and best (green circles) cost in each iteration. Population: cost versus number of sensors in the final population. It may be useful to choose larger `width` as plots are placed beside each other.

`"greedy"`: cost (blue) and number of sensors (red) in each iteration, combined in one plot with adjusted scales. Lowest cost is marked by solid dots.

`"global"`: a barplot of all sampling designs, showing how many plumes are detected by 1st, 2nd, etc. sensor.

`"manual"`: similar to the plots of `"greedy"`.

Value

Generates a plot or a file with it, no value returned.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SDgreedy)
curve_greedy1 = optimisationCurve(
  optSD = SDgreedy,
  type = "greedy")

data(SDgenetic)
curve_genetic1 = optimisationCurve(
  optSD = SDgenetic,
  type = "genetic")

data(SDglobal)
curve_global1 = optimisationCurve(
  optSD = SDglobal,
  type = "global")

data(SDmanual)
curve_manual1 = optimisationCurve(
  optSD = SDmanual,
  type = "manual")


data(SDssa)
curve_ssa1 = optimisationCurve(
  optSD = SDssa,
  type = "ssa")


## Not run:
# generates a file
curve_global1 = optimisationCurve(
  optSD = SDglobal,
  type = "global",
  nameSave = "optSD_global",
  width = 600, height = 300)

## End(Not run)
```

`optimiseSD` *(Spatial) optimisation of sampling designs*

Description

This function optimises a sampling design to achieve minimal cost for a given cost function by applying a given optimisation function. It may take into account fix locations or constrain search to a subset of locations.

Usage

```
optimiseSD(simulations, costFun,
  locationsAll = 1:nLocations(simulations), locationsFix = integer(0),
  locationsInitial = integer(0),
  aimCost = NA, aimNumber = NA,
  optimisationFun,
  nameSave = NA, plot = FALSE, verbatim = FALSE, ...)
```

Arguments

simulations	Simulations object
costFun	cost function, must have parameters simulations and locations (may be prepared by replaceDefault with type = "costFun.optimiseSD"); it must return cost as a single value or a list where this is the first entry
locationsAll	indices of the locations that are considered possible sensor locations, by default these are all locations
locationsFix	indices of locations with fix sensors – to be considered when computing cost, by default this is empty
locationsInitial	indices of locations where sensors are initially; by default this is empty; if not indicated aimNumber random locations may be used, depending on the
aimCost	limit value of the costFun: the result has to fall below
aimNumber	if no locationsInitial given, it starts from this number of random locations
optimisationFun	function that executes an optimisation algorithm; must have parameters simulations, costFun, locationsAll, locationsFix, locationsInitial, aimCost, aimNumber, nameSave (use type = "optimisationFun.optimiseSD" in replaceDefault). Output must be a list with SD (vector, matrix, or list of one or several SDs) and cost (belonging to the SD); it may return other values in report
nameSave	character path and name (without suffix!) where to save intermediate results (sampling design of each iteration)
plot	if iteration is plotted (currently disabled)
verbatim	print intermediate results and keep sampling designs and cost of all iterations
...	further parameters, currently unused

Details

For examples see optimiseSD_genetic, optimiseSD_global, optimiseSD_greedy, optimiseSD_manual,optimiseSD_ss

Value

A list

SD	list, each entry is a matrix of the best (lowest cost) sampling designs found of a size - sampling designs in rows; including locationsFix

evaluation	data.frame, each row belongs to the SD of one size, giving the number of sensors and the cost
aimSD	list indicating which of the SD fulfil the given aimCost and aimNumber, can be empty if no aims given or aims not reached
report	algorithm-specific, e.g. all tested SDs etc., see there.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

---

optimiseSD_genetic *Optimisation of sensor locations by a binary genetic algorithm*

Description

optimiseSD_genetic optimises sensors for a given Simulations object and costFunction. It starts from a population of sensor sets that can be provided and else are random. By testing them and recombining good locations, a new improved population is generated. This process continues until the aim or a stopping criterion is reached. It applies the binary genetic algorithm rbga.bin. numberPenalty is the default evaluation function, it returns the penalty if aimNumber is exceeded, else the cost computed by costFun.

Usage

```
optimiseSD_genetic(simulations, costFun,
  locationsAll = 1:nLocations(simulations), locationsFix = integer(0),
  locationsInitial = integer(0),
  aimCost = NA, aimNumber = NA,
  nameSave = NA, plot = FALSE, verbatim = FALSE,
  evalFunc = numberPenalty,
  popSize = 200, iters = 100,
  mutationChance = NA, elitism = NA)
numberPenalty(chromosome, simulations, costFun,
  locationsAll, locationsFix,
  aimNumberNF, penalty = 2)
```

Arguments

Some arguments are the same for all optimisation algorithms, they are marked by a *, for detail see optimiseSD. Others are directly forwarded to rbga.bin, they are marked by **

	*
simulations	*
locationsAll	*
locationsFix	*

locationsInitial	*; here it may also be a matrix, each row indicates a set of sensors
aimCost	*
aimNumber	*; if locationsInitial given it must be NULL and is ignored, else it defines the number of sensors in the sets of the initial population
aimNumberNF	aimNumber minus length of locationsFix - to be used in evaluation
nameSave	*
plot	not yet implemented
verbatim	if more output is to be written to the console
evalFunc	**; this function has to take a chromosome and return its cost (including fix sensors); it can be modified by replaceDefault with type = "evalFunc.rbga.bin"
popSize	**
iters	**
mutationChance	**
elitism	**
chromosome	a vector of 0 and 1 of the same length as locationsAll indicating sensors at the positions of 1.
penalty	numeric to be returned if number of 1s in chromosome exceeds aimNumber.

Details

In general the function is used within the wrapper optimiseSD. The **-parameters are specific to optimiseSD_greedy, they may be changed beforehand via replaceDefault with type = "optimisationFun.optimiseSD" all other parameters are forwarded from optimiseSD. If aimCost is given, the algorithm stops when the aim is reached by the best sampling design. In this case, nothing is returned, but the populations are saved (at nameSave and if nothing indicated at "opt_genetic.Rdata"), see examples how to extract the sampling designs in this case.

Value

A list, the first two entries are common to all optimisation algorithms, they are marked with *, see optimiseSD for details.

SD	* best sampling designs
evaluation	* cost and size of SD
report	the generated "rbga" object as returned by rbga.bin

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# load data and compute required properties
demo(radioactivePlumes_addProperties)

# initial, fix, possible sensor locations
I = nLocations(radioactivePlumes)
set.seed(1000)
locInit_l = t(replicate(10, sample.int(I, 5)))
locKeep_l = sample(setdiff(1:I, locInit_l), 2)
locAll_l = c(sample(setdiff(1:I, c(locInit_l, locKeep_l)),
                          round(I - (5 + 2)) * 0.5), locInit_l)

# the function is to be used inside of optimiseSD
# change algorithm specific parameters 'evalFunc', 'popSize', 'iters'
optimiseSD_genetic_1 = replaceDefault(
  optimiseSD_genetic, newDefaults = list(
    evalFunc = numberPenalty,
    popSize = 20,
    iters = 10
  ),
  type = "optimisationFun.optimiseSD")[[1]]

# run optimisation
## Not run:
## takes some time
OptSD_gen1 = optimiseSD(
  simulations = radioactivePlumes,
  costFun = singleDetection,
  optimisationFun = optimiseSD_genetic_1,
  locationsAll = locAll_l,
  locationsFix = locKeep_l,
  locationsInitial = locInit_l[1,])

## End(Not run)
## this result is also in data(SDgenetic)

## Not run:
# result is not returned but saved to file
## case with 'aimCost'
optimiseSD_genetic_2 = replaceDefault(
    optimiseSD_genetic, newDefaults = list(
      evalFunc = numberPenalty,
      popSize = 20,
      iters = 20
    ),
    type = "optimisationFun.optimiseSD")[[1]]

set.seed(07021916)
OptSD_gen2 = optimiseSD(
  simulations = radioactivePlumes,
  costFun = singleDetection,
  optimisationFun = optimiseSD_genetic_2,
```

```
  locationsAll = locAll_l,
  locationsFix = locKeep_l,
  locationsInitial = locInit_l[1,],
  aimCost = 0.05,
  nameSave = "optSD_genetic_2" # result is saved to file here
)
# OptSD_gen2 not found
## load generated populations and extract sampling designs and cost
load("optSD_genetic_2.Rdata")
finalIteration = sum(!is.na(populations[1,1,]))
costs = matrix(0, nrow = finalIteration, ncol = 20)
SDs = list()
for (j in 1:finalIteration){
  SDs[[j]] = list()
  for (i in 1:20){
    costs[j,i] = numberPenalty(populations[i,,j],
      simulations = radioactivePlumes,
      costFun = singleDetection,
      locationsAll = locAll_l,
      locationsFix = locKeep_l,
      aimNumber = length(locInit_l[1,]))
    SDs[[j]][[i]] = c(locAll_l[populations[i,,j] == 1], locKeep_l)
    if (sum(populations[i,,j]) > 5){
      costs[j,i] = 2
    }
    print(paste(j,i))
  }
}
apply(FUN = min, X = costs, MARGIN = 1) # best cost in each iteration
# best sampling design:
SDs[[finalIteration]][[which(costs[finalIteration,] == min(costs[finalIteration,]))]]

## End(Not run)
```

`optimiseSD_global` *Derive one or all globally optimal sampling designs for plume detection*

Description

This optimisation works only for a special cost function: if the values are logical and indicating for each plume if it can be detected at any sensor location and cost is the fraction of plumes not detected by any sensor. In this case it can derive one or all global optima. It may take very long (hours to weeks) for cases of realistic size (hundreds of locations, more than 2 sensors). The algorithm is described in detail in the reference.

Usage

```
optimiseSD_global(simulations, costFun = NA,
  locationsAll = 1:nLocations(simulations), locationsFix = integer(0),
```

```
  locationsInitial = integer(0),
  aimCost = NA, aimNumber = NA,
  nameSave = NA, plot = FALSE, verbatim = FALSE,
  detectable = 1, maxIterations = NA,
  findAllOptima = FALSE, findSensorNumber = FALSE
  )
```

Arguments

	Some arguments are the same for all optimisation algorithms, they are marked by a *, for detail see optimiseSD
	*
simulations	* ignored, as optimisation can only optimise plume detection
locationsAll	*
locationsFix	*
locationsInitial	* ignored, only if aimNumber missing it is defined by length of locationsInitial
aimCost	*, ignored
aimNumber	*
nameSave	*
plot	not implemented yet
verbatim	if more output is to be written to the console
detectable	which layer of the values of simulations to use, character or integer
maxIterations	stop when this number of sensor sets was tested (if findAllOptima = TRUE this is applied to each search i.e. to the one to find the maximal number of detectable plumes and to the one to find all optima); if maxIterations is below the number of sensor sets to actually be tested, the result may not be the global optimum.
findAllOptima	logical, if all optima to be found - requires to internally run search twice.
findSensorNumber	logical, only relevant if findAllOptima = FALSE and all plumes are detected; in this case findSensorNumber = TRUE calls further searches to determine the minimal number of sensors required to detect all plumes. If findAllOptima = TRUE these searches are run in any case.

Details

In general the function is used within the wrapper optimiseSD. The non-*-parameters are specific to optimiseSD_global, they may be changed beforehand via replaceDefault with type = "optimisationFun.optimiseSD" all other parameters are forwarded from optimiseSD.

If aimCost is given, the algorithm stops when the aim is reached by the best sampling design. In this case, nothing is returned, but the populations are saved (at nameSave and if nothing indicated at "opt_genetic.Rdata"), see examples how to extract the sampling designs in this case.

Value

A list, the first two entries are common to all optimisation algorithms, they are marked with *, see optimiseSD for details.

SD	* best sampling designs
evaluation	* cost and size of SD
report	a list with more details about the optimisation detectable the matrix of 0 and 1 used in the optimisation after selecting only locations in locationsAll, deleting locations in locationsFix and containing only plumes that cannot be detected at locationsFix but at locationsAll first results of the first run to find the detection limit, lowerLimit is this limit plus 1 all results of the run to find all optimal sampling designs

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

References

K.B. Helle, E. Pebesma (2015). Optimising sampling designs for the maximum coverage problem of plume detection. Spatial Statistics (13), 21-44.

Examples

```
# prepare data
# test case from reference
detectionMatrix = matrix(c(0,1,0,0,0,1,
                           1,0,1,0,0,0,
                           0,1,1,1,0,0,
                           1,0,0,0,0,0,
                           0,0,0,1,1,0,
                           1,1,0,0,0,0,
                           0,0,1,0,0,1),
                         nrow = 7, ncol = 6, byrow = TRUE)
data(SimulationsSmall)
completeExample = Simulations(
  locations = SimulationsSmall@locations[1:7,],
  plumes = SimulationsSmall@plumes[1:6,],
  values = raster(x = detectionMatrix,
             xmn = -90, xmx = 90, ymn = -90, ymx = 90,
             crs = "+init=epsg:4326"))

# the function is to be used inside of optimiseSD
# change algorithm specific parameters 'detectable', 'findAllOptima'
optSDglobal = replaceDefault(
  fun = optimiseSD_global,
  newDefaults = list(
    detectable = 1,
    findAllOptima = TRUE
```

```
    ),
  type = "optimisationFun.optimiseSD"
)

# run optimisation
## Not run:
# takes some (little) time
optSD_global = optimiseSD(
  simulations = completeExample,
  aimNumber = 3,
  costFun = NA,
  optimisationFun = optSDglobal[[1]],
  nameSave = NA
)

## End(Not run)
## this result is also in data(SDglobal)

data(SDglobal)
# visualise result
## which plumes are detected?
detAtSensors = matrix(completeExample@values[SDglobal$SD[[1]][1,],],
  byrow = TRUE, nrow = 3)
undetedtedPlumes = ! apply(FUN = any, X = detAtSensors, MARGIN = 2)
## cost is fraction of undetected plumes
sum(undetedtedPlumes)/nPlumes(completeExample) == SDglobal$cost

## map of undetected plumes
detectablePlumesLoc = matrix(getValues(completeExample@values),
  byrow = TRUE, ncol = nPlumes(completeExample))
completeExample@locations@data$detectable = apply(FUN = sum, X = detectablePlumesLoc,
MARGIN = 1)

SDopt = as(subsetSDF(completeExample@locations,
                  locations =  SDglobal$SD[[1]][1,]),
         "SpatialPointsDataFrame")

spplot(completeExample@locations, zcol = "detectable",
  sp.layout = list(list("sp.points",
  SpatialPoints(coords = coordinates(SDopt), proj4string = CRS(proj4string(SDopt))),
                        col = 3, cex = 2, lwd = 1.5)))
```

---

optimiseSD_greedy *Greedy optimisation algorithm*

Description

Runs greedy optimisation for a given Simulations object and cost function. The aim may be given as a number of sensors or as cost value, then the algorithm determines if sensors need to be added or deleted. Greedy optimisation adds (or respectively deletes) sensors one by one, always checking

all possibilities and adding/deleting the sensor that yields minimal cost, given the already defined sensors; it stops when the aim is reached. In addition this algorithm can continue search when the greedy optimum is found by adding and deleting sensors in turns, this can help to get rid of redundant sensors and thus may improve the result. The algorithm may take into account fix sensors and start from a given initial sensor set.

Usage

```
optimiseSD_greedy(simulations, costFun,
  locationsAll = 1:nLocations(simulations), locationsFix = integer(0),
  locationsInitial = integer(0),
  aimCost = NA, aimNumber = NA,
  nameSave = NA, plot = FALSE, verbatim = FALSE,
  maxIterations = 100, swap = FALSE)
```

Arguments

	Some arguments are the same for all optimisation algorithms, they are marked by a *, for detail see optimiseSD
simulations	*
costFun	*
locationsAll	*
locationsFix	*
locationsInitial	*
aimCost	*, ignored
aimNumber	*
nameSave	*
plot	not implemented yet
verbatim	logical, not implemented yet
maxIterations	maximal number of iterations, then it stops
swap	logical if algorithm continues, when greedy optimum is found, by adding and deleting sensors in turns

Details

In general the function is used within the wrapper optimiseSD. The parameters maxIterations and swap are specific to optimiseSD_greedy, they may be changed beforehand via replaceDefault with type = "optimisationFun.optimiseSD"; all other parameters are forwarded from optimiseSD.

If aimCost and aimNumber are given, aimNumber is ignored with a warning.

Value

A list, the first two entries are common to all optimisation algorithms, they are marked with *, see optimiseSD for details.

SD	* best sampling designs
evaluation	* cost and size of SD
report	a list of SDsa list of the sampling designs of all iterations evalSDs a data.frame: cost and number of sensors for the sampling designs SDs.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# optimisation function: is to be used inside of optimiseSD
# change parameters 'swap' and 'maxIterations'
optGr_20_true = replaceDefault(
  optimiseSD_greedy,
  newDefaults = list(
    maxIterations = 20,
    swap = TRUE)
)
# cost function
meanFun = function(x){mean(x, na.rm = TRUE)}
minDist = replaceDefault(
  spatialSpread, newDefaults = list(
    fun = minimalDistance,
    fun_R = meanFun
  ), type = "costFun.optimiseSD"
)[["fun"]]

# define possible, fix, and initial sensors
data(SimulationsSmall)
I = nLocations(SimulationsSmall)
set.seed(9345872)
locInit1 = sample.int(I, 2)
locKeep1 = sample(setdiff(1:I, locInit1), 2)
locAll1 = c(sample(setdiff(1:I, c(locInit1, locKeep1)), 4), locInit1)

# run optimisation
## Not run:
## takes some time
optSD_greedy = optimiseSD(
  simulations = SimulationsSmall,
  costFun = minDist,
  optimisationFun = optGr_20_true[[1]],
  locationsAll = locAll1,
  locationsFix = locKeep1,
```

```
  locationsInitial = locInit1,
  aimNumber = 7
)


## End(Not run)
## this result is also in data(SDgreedy)
```

---

optimiseSD_manual *Plot cost map for interactive adding and deleting of sensors*

Description

Given a costMap function and a set of sensors it computes the cost map and plots it. Then the user can interactively add or delete sensors and plot the resulting cost and cost map..

Usage

```
optimiseSD_manual(simulations, costFun,
  locationsAll = 1:nLocations(simulations), locationsFix = integer(0),
  locationsInitial = integer(0),
  aimCost = NA, aimNumber = NA,
  nameSave = NA, plot = TRUE, verbatim = FALSE,
  costMap = NA, maxIterations = 10,
  valuesPlot = integer(0), colors = grey.colors)
```

Arguments

Some arguments are the same for all optimisation algorithms, they are marked by a *, for detail see optimiseSD. Others are directly forwarded to rbga.bin, they are marked by **

	*
simulations	* ignored - cost is derived from costMap
costFun	
locationsAll	*
locationsFix	*
locationsInitial	*
aimCost	* ignored
aimNumber	* ignored
nameSave	* ignored
plot	* ignored
verbatim	ignored

costMap	A function to return a list of "cost": cost in general, a single value, "costLocations": a cost value for each location(vector of length equal to nLocations(simulations)). Needs to have the same parameters (simulations, locations) as costFun and may also be prepared by replaceDefault with type = "costFun.optimiseSD".
maxIterations	maximal number of iterations to add or delete sensors
valuesPlot	names of values in the data associated with the locations of simulations to be plotted in addition to the cost map
colors	color ramp for plotting - a function like grey.colors)

Value

A list, the first two entries are common to all optimisation algorithms, they are marked with *, see optimiseSD for details.

SD	* best sampling designs
evaluation	* cost and size of SD
report	a list of cost: cost in each iteration, based on fix and current sensors identify: identity of points chosen in each iteration; these identities are 'raw' -for "SpatialIndexDataFrame" and "SpatialPolygridDataFrame" they may differ from the indices of the actual locations as data is transformed into SpatialPixels before plotting, they may also include invalid and multiple choices. locationsCurrent: indices of sampling design in each iteration; locationsFix are not included

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# prepare data and functions
data(radioactivePlumes)

meanFun = function(x){mean(x, na.rm = TRUE)}
spatialSpreadMinDist = replaceDefault(
  spatialSpread,
  newDefaults = list(
    weightByArea = TRUE,
    fun = minimalDistance,
    fun_R = meanFun),
  type = "costFun.optimiseSD"
  )[[1]]

radioactivePlumes@locations@data$p1 = getValues(
  subset(radioactivePlumes, plumes = 1, kinds = 1)@values)
```

```
optimSD_man_minDist = replaceDefault(
  optimiseSD_manual,
  newDefaults = list(
    costMap = spatialSpreadMinDist
    )
)[["fun"]]

## Not run:
## interactive optimisation
# inside optimiseSD
optSD_manual1 = optimiseSD(simulations = radioactivePlumes,
                           costFun = spatialSpreadMinDist,
                           optimisationFun = optimSD_man_minDist,
                           locationsFix = seq(1, 2001, 300),
                           locationsInitial = seq(1, 2001, 70),
                           locationsAll = setdiff(1:2001, seq(1,2001, 30)))

# directly using optimiseSD_manual
optSD_manual2 = optimiseSD_manual(simulations = radioactivePlumes,
                                  costFun = spatialSpreadMinDist,
                                  costMap = spatialSpreadMinDist,
                                  locationsFix = seq(1, 2001, 300),
                                  locationsInitial = seq(1, 2001, 70),
                                  locationsAll = setdiff(1:2001, seq(1,2001, 30)))

## End(Not run)
## the result of such a manual optimisation is in data(SDmanual)
```

`optimiseSD_ssa` *Spatial Simulated Annealing optimisation algorithm*

Description

Runs Spatial Simulated Annealing (SSA) optimisation for a given Simulations object and cost function. The aim is a cost value. The algorithm moves the initial sensors to reach the aim. In each iteration SSA moves one or more sensors randomly within their neighbourhood; the size of shifts decreases during the run. If a movement decreases cost, it is accepted. Else it may be accepted, too: the decision is random, with higher probability for acceptance in the beginning of the run and if the move increases cost only little. Acceptance of worse results helps to escape from local optima

Usage

```
optimiseSD_ssa(simulations, costFun,
  locationsAll = 1:nLocations(simulations), locationsFix = integer(0),
  locationsInitial = integer(0),
  aimCost = NA, aimNumber = NA,
  nameSave = NA, plot = FALSE, verbatim = FALSE,
  maxShiftNumber = length(locationsInitial), startMoveProb = 0.5,
  maxShiftFactor = 0.2, minShiftFactor = 0.01,
```

```
  maxIterations = 5000, maxStableIterations = 500,
  maxIterationsJumpBack = maxStableIterations - 1,
  acceptanceMethod = "vanGroenigen",
  start_acc_vG = 0.5, end_acc_vG = 0.0001,
  cooling_vG = 0.999, startAcceptance_vG = 0.5,
  start_acc_iI = 0.2
  )
```

Arguments

	Some arguments are the same for all optimisation algorithms, they are marked by a *, for detail see optimiseSD. Others are directly forwarded to rbga.bin, they are marked by **
	*
constFactors	*
locationsAll	*
locationsFix	*
locationsInitial	*
aimCost	*
aimNumber	*; if no locationsInitial given, it starts from this number of random locations
nameSave	*
plot	not yet implemented
verbatim	if more output is to be written to the console
maxShiftNumber	how many of the sensors can maximally be shifted in each iteration
startMoveProb	initial probability to move sensors -in any case at least one sensor is moved
maxShiftFactor	initial maximal size of shifts (as fraction of extent)
minShiftFactor	final maximal size of shifts (as fraction of extent)
maxIterations	maximal number of iterations (iteration means tried sensor changes, not all are accepted)
maxStableIterations	it stops after this number of iterations in a row without improvement
maxIterationsJumpBack	if for this number of iterations the best result is not improved, the sampling design jumps back to the last best design; by default this does not occur
acceptanceMethod	formula to compute probability to accept worse result: for "vanGroenigen" acceptance decreases relative to worsening, for "intamapInteractive" it does not; for both probability of acceptance decreases during the run
start_acc_vG	acceptance worse results in first iteration (for acceptanceMethod = "vanGroenigen")
end_acc_vG	acceptance of worse results in maxIterations-th iteration (for acceptanceMethod = "vanGroenigen")

`cooling_vG` cooling parameter
(for `acceptanceMethod = "vanGroenigen"`), see details

`startAcceptance_vG` second cooling parameter
(for `acceptanceMethod = "van Groenigen"`), see details

`start_acc_iI` calibration factor of acceptance for
`acceptanceMethod = "intamapInteractive"` (equals `start_p` in `ssaOptim` in package `intamapInteractive`), see details

Details

In general the function is used within the wrapper `optimiseSD`. The following parameters are specific to `optimiseSD_ssa`: maxShiftNumber, startMoveProb, maxShiftFactor, minShiftFactor, maxIterations, maxStableIterations, maxIterationsJumpBack, acceptanceMethod, start_acc_vG, end_acc_vG, cooling_vG, startAcceptance_vG, start_acc_iI. They may be changed beforehand via `replaceDefault` with `type = "optimisationFun.optimiseSD"`; all other parameters are forwarded from `optimiseSD`.

In one iteration more than one sensor may be moved: first `maxShiftNumber` sensors are selected; for each of them, probability to move it is $startMoveProb * (1 - k/maxIterations)$ where k indicates the iteration; only the first is moved in any case. The decision if the movement is accepted refers to the common movement of all finally selected sensors. Moving many sensors allows for big changes, it can be useful if initial sensors are supposed to be far from the optimum.

`maxShiftFactor` and `minShiftFactor` are given as fractions of the extent. If the underlying spatial object of the `simulations` is gridded, a shift size below cellsize would not allow any shifts. Therefore in this case it is set to at least cell size (independently in both coordinates).

Acceptance of worse results for `acceptanceMethod = "vanGroenigen"` is $exp((costOld - costCurrent)/(startAcceptan$ $cooling^k)$. This means it is higher for small cost difference and decreases with iteration `k`. You may set the parameters `cooling_vG` and `startAcceptance_vG` directly - to achieve this, set either `start_acc_vG` or `end_acc_vG` to `NULL`. However, `start_acc_vG` and `end_acc_vG` provide a more intuitive way to define acceptance: the algorithm computes the median cost difference when moving one sensor randomly or within the close neighbourhood (to next point) and adjusts cooling and startAcceptance to achieve the indicated acceptance rates. For `acceptanceMethod = "intamapInteractive"` acceptance of worse results is $start_acc_iI * exp(-10 * k/maxIterations)$. It is the same no matter how much worse the new design is, it just decreases with iteration `k`.

Value

A list A list, the first two entries are common to all optimisation algorithms, they are marked with *, see `optimiseSD` for details.

`SD` * best sampling designs

`evaluation` * cost and size of `SD`

`report` a list of
`SD_final`: final sampling design (not necessarily best, including `locationsFix`)
`SD_best`: best sampling design ever found (including `locationsFix`), same as `SD[[1]]`
`cost_final`: cost of `SD_final`
`cost_best`: cost of `SD_best`

iterations: data.frame with columns cost and accepted, rows refer to iterations.

If verbatim = TRUE more information is returned in report:
The iterations has additional columns: costTest (cost of the design tested in this iteration, no matter if it was accepted; cost refers to the currently accepted design); costBest (best cost until then); chi (random value that was used for the acceptance decision).

In addition it returns a matrix each for SDs, SDs_test, SDs_test with the current, tested and best known sampling designs of all iterations.

If acceptanceMethod = "vanGroenigen" it returns the cooling parameters cooling and startAcceptance as they may be computed inside the algorithm.

It gives the iterations with jumpBack i.e. when the current sampling design was replaced by the known best.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# the function is to be used inside of optimiseSD
# change parameters
optimSD_ssa1 = replaceDefault(
  optimiseSD_ssa,  newDefaults = list(
    start_acc_vG = 0.1,
    aimCost = 0,
    verbatim = TRUE,
    maxIterations = 3000,
    maxStableIterations = 500,
    maxIterationsJumpBack = 200
  ),
  type = "optimisationFun.optimiseSD")[[1]]

# load data
demo(radioactivePlumes_addProperties)
# define possible, fix, and initial sensors
I = nLocations(radioactivePlumes)
set.seed(22347287)
locDel3 = sample.int(I, 5)
locKeep3 = sample(setdiff(1:I, locDel3), 10)
locAll3 = c(sample(setdiff(1:I,
  c(locDel3, locKeep3)), 10), locDel3)


costInitial1 = multipleDetection(simulations = radioactivePlumes,
  locations = c(locKeep3, locDel3))

# run optimisation
## Not run:
```

```
## takes some time
SDssa = optimiseSD(
  simulations = radioactivePlumes,
  costFun = multipleDetection,
  locationsAll = setdiff(1:nLocations(radioactivePlumes), c(locKeep3, locAll3)),
  locationsFix = locKeep3,
  locationsInitial = locDel3,
  aimCost = 0.05 * costInitial1[[1]],
  aimNumber = length(locDel3) + length(locKeep3),
  optimisationFun = optimSD_ssa1
)

## End(Not run)
## this result is also in data(SDssa)

# visualise
data(SDssa)
## cost curve
optimisationCurve(optSD = SDssa, type = "ssa")
## designs
singleDet = replaceDefault(singleDetection,
  newDefaults = list(plot = TRUE), type = "costFun.optimiseSD")[[1]]
plotSD(radioactivePlumes,
  SD = SDssa[[1]],
  locationsFix = locKeep3,
  locationsInitial = locDel3,
  locationsAll = setdiff(1:nLocations(radioactivePlumes), c(locKeep3, locAll3)),
  costMap = singleDet
)
```

`plot.Simulations` *Plot overview of* `simulations`.

Description

Generate a common plot for the values associated with `locations` and with `plumes` and the `values`.

Usage

```
## S3 method for class 'Simulations'
plot(x, ..., zcol = 1, main = "",
                          col = terrain.colors(100), maxpixels = 1e+7)
```

Arguments

`x`	`Simulations` object
`...`	parameters to be forwarded
`zcol`	integer, layer of `values` to be plotted
`main`	character, header: should have short lines to fit into the upper right corner.

col	color scheme
maxpixels	if values (of one layer) fit into memory

Details

The plots show all factorial and numeric parameters, character parameters (e.g. names of plumes) are ignored. The colours cover min to max of each parameter. If values (of one layer) fit into memory, they are plotted by image. If they are bigger, we use plot and only plot a sample of size maxpixels (with warning).

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(radioactivePlumes)
plot(radioactivePlumes, zcol = 3, col = bpy.colors(),
     main = "radioactive \n plumes \n area")
```

---

plotSD *Plot sampling designs and related cost maps*

Description

Given a sampling design and a function to compute cost maps it generates the related map plot, based on spplot. If no cost map function is available, it may plot it with data from related simulations.

Usage

```
plotSD(simulations, SD,
  locationsFix = integer(0),
  locationsInitial = integer(0),
  locationsAll = integer(0),
  costMap,
  zcol = 1, allIn1Plot = 0,
  pch = c(1, 20, 4), col = c("white", "white", "white", "white"),
  pointsKey = TRUE, mainCost = TRUE,
  pch.SDs, col.SDs, cex.SDs, ...)
```

Arguments

simulations	Simulations object
SD	indices of locations of sensors, can be a matrix (rows refer to SDs) or a list
locationsFix	indices of fix sensors, taken into account to compute cost map and plotted

locationsInitial	indices of initial sensors, for plotting
locationsAll	indices of all possible sensors, for plotting
costMap	function to compute the values for the background map; needs the parameters simulations and locations; output must be a list with "costLocations" of length like the locations of simulations
zcol	index or name: if no costMap given, this layer of the locations is used as background map
allIn1Plot	if several SDs are given, they are by default (allIn1Plot = 0) plotted each on a map; if allIn1Plot is an integer, all SDs are combined in one plot with the cost map of this SD as background
pch	point style of SD, locationsFix, locationsInitial; for locationsAll the pch = "." cannot be changed
col	colors for SD, locationsFix, locationsInitial, locationsAll
pointsKey	logical, if key for points is to be printed
mainCost	logical, if cost of SD is to be used as main (this is only done if costMap returns a list item "cost")
pch.SDs	point types for the different SDs, must have length that fits number of SDs (default: use pch[1] for all SDs but vary size)
col.SDs	point colours for different SDs
cex.SDs	point size for different SDs
...	further parameters to be forwarded to spplot

Value

The function generates plots (main and point key missing) and returns a list of trellis objects (including main and keys).

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
demo(radioactivePlumes_addProperties)
data(SDssa)
SDs = list(SDssa[["SD"]][[1]][1,], SDssa$report[["SD_final"]])
singleDet = replaceDefault(singleDetection,
  newDefaults = list(plot = TRUE), type = "costFun.optimiseSD")[[1]]
# separate maps
plotSD1a = plotSD(
  simulations = radioactivePlumes,
  SD = SDs,
  costMap = singleDet)
for (i in seq(along = SDs)){
  plot(plotSD1a[[i]])
}
```

```
# combined map (with customised parameters for fix and initial sensors)
I = nLocations(radioactivePlumes)
set.seed(22347287) # reconstruct initial and fix sensors of SDssa
locDel3 = sample.int(I, 5)
locKeep3 = sample(setdiff(1:I, locDel3), 10)
locAll3 = c(sample(setdiff(1:I,
  c(locDel3, locKeep3)), 10), locDel3)

plotSD2 = plotSD(
  simulations = radioactivePlumes,
  SD = SDs,
  locationsAll = setdiff(1:nLocations(radioactivePlumes), c(locKeep3, locAll3)),
  locationsFix = locKeep3,
  locationsInitial = locDel3,
  costMap = singleDet,
  allIn1Plot = 2,
  col = c("green", "blue", "red", "white"),
  pch = c(1,2,4),
  col.regions = grey.colors,
  colorkey = FALSE)


# combined map (with customised parameters for different SDs)
plotSD3 = plotSD(
  simulations = radioactivePlumes,
  SD = SDs,
  locationsFix = locKeep3,
  locationsAll = setdiff(1:nLocations(radioactivePlumes), c(locKeep3, locAll3)),
  allIn1Plot = 2,
  col = c(1, "white", "white", "white"),
  pch = c(1,20,0),
  col.regions = grey.colors,
  pointsKey = FALSE,
  pch.SDs = 2:5,
  cex.SDs = c(3.5,3,2.5,2),
  col.SDs = 4:8
  )
```

`points2polygrid` *Turn points (and data) into a SpatialPolygridDataFrame*

Description

`points2polygrid` determines a fine grid, such that each point is represented by a set of grid cells (nearest neighbourhood); to each grid cell it assigns the values of the nearest point. It may also assign cells of a given grid to the nearest neighbour points.

Usage

```
points2polygrid(points, grid, index, data,
tolerance = signif(mean(diff(t(bbox(points))))/1000, digits = 1))
```

Arguments

points	coordinates of the original points: data.frame, matrix with coordinates in columns 1 and 2, or SpatialPoints
grid	target grid: GridTopology or SpatialGrid
index	target index indicating for each grid cell the associated data: integer, length must fit number of cells in grid (order: top left to right, then to bottom)
data	data associated with points, to be assigned to grid cells: data.frame or vector of same length as points
tolerance	lower limit of the resolution of the generated grid, default generates about 1000 x 1000 grid cells; if points lie on a regular grid, the resulting resolution is automatically adapted

Details

It is sufficient to provide points, they can be used to generate a grid and associate the cells to the closest points. The generated grid covers all points; resolution is chosen such that there are cell boundaries between the x-coordinates and y-coordinates of all points respectively. If points lie on a regular grid, resolution is chosen to fit this grid. A grid can also be provided which saves computational effort.

If no index is given, grid cells are assigned to the nearest point (based on get.knnx). If index, grid and points are given, points are ignored. If points are at the same location or so close that they are merged, the associated data of only one point is used.

Value

A SpatialPolygridDataFrame. If no data is given, the returned data consists of one column 'Index' with the indices of the associated points. proj4string of points is kept if available, else the one of grid is taken.

Note

This function is used to coerce *SpatialPointsDataFrame* to *SpatialPolygridDataFrame*.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# prepare
data(SPolygridDF)
regularPoints = coordinates(SPolygridDF)
irregularPoints = regularPoints + runif(12, 0.1)
```

```
# grid for regular points
newSPolygrid_reg = points2polygrid(points = regularPoints,
                                   data = data.frame(a = 12:1 * 5))
spplot(newSPolygrid_reg,
       sp.layout = list(list("sp.points", regularPoints, col = 3)))

# generate grid for irregular points with given resolution
newSPolygrid_irreg = points2polygrid(points = irregularPoints, tolerance = 0.5)
spplot(newSPolygrid_irreg, sp.layout = list("sp.points", irregularPoints, col = 3))

# with given grid
newSPolygrid_grid = points2polygrid(points = irregularPoints,
  grid = GridTopology(c(2,2), c(1,1), c(10,10)))
spplot(newSPolygrid_grid, sp.layout = list("sp.points", irregularPoints, col = 3))
```

`polygrid2grid` *Coerce SpatialPolygridDataFrame into SpatialGridDataFrame and geoTiff file*

Description

Coerces SpatialPolygridDataFrame into SpatialGridDataFrame by copying the data to all associated grid cells. It may write the result to a geoTiff file.

Usage

```
polygrid2grid(obj, zcol = NA, returnSGDF = TRUE, geoTiffPath)
```

Arguments

obj	SpatialPolygridDataFrame-class
zcol	names or numbers of the data columns to be used
returnSGDF	if the generated SpatialGridDataFrame is to be returned
geoTiffPath	filename where to save the result as geoTiff (without suffix, '.tif' is added automatically); if missing, no geoTiff is generated

Details

The main reason to turn a SpatialPolygridDataFrame into a SpatialGridDataFrame is for plotting. It can make sense to generate a (multilayer) geoTiff instead of returning the transformed data into the workspace.

Value

A SpatialGridDataFrame with the same values assigned as in the SpatialPolygridDataFrame by copying the data to all associated grid cells. If returnSGDF = FALSE it returns TRUE.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SPolygridDF)

# return SpatialGridDataFrame
SGridDF1 = polygrid2grid(SPolygridDF, zcol = "b")

# generate geoTiff
polygrid2grid(SPolygridDF, returnSGDF = FALSE,
              geoTiffPath = "SPolygridDF1")

# plot SpatialGridDataFrame
spplot(SGridDF1)

# view geoTiff with functions from 'raster' and delete it
SGridDF2 = brick("SPolygridDF1.tif")
plot(SGridDF2)
rm(SGridDF2)
file.remove("SPolygridDF1.tif")
```

---

radioactivePlumes *Simulations of radioactive plumes*

Description

The simulations show maps of pollution spreading from a single source in the centre. The simulations were generated by the atmospheric dispersion model RIMPUFF. The maps show radioactivity after a release of 6 nuclids, of about 1e+13 Bq/s each, during 12 hours. The nuclids represent typical half-lives and deposition characteristics to simplify release patterns that are used in radiological emergency simulations. The weather data was taken from a flat, coastal area in marine middle Europe. It covers one year, a new simulation starting about every week (6 days + 6 hours) and running for one day. The size of the map is 41 km x 41 km, the resolution is telescopic: 0.5 km up to 5 km from the source, 1 km else. The simulations were generated within the EU FP7 project DETECT, for details see the reference. Bigger, more realistic datasets can be found in the package `simulations4plumesData` on GitHub whereof this is a subset.

Usage

```
data(radioactivePlumes)
```

Format

`radioactivePlumes`: Simulations

Details

The three kinds of the `values` are summaries over plumes in time. For each location they give:
`finaldose`: total dose after a week, in Sv;
`maxdose`: maximal hourly average dose within the week, in Sv/h;
`time`: time until the hourly average dose first reaches 1e-7 Sv/h, in seconds.
`maxdose` and `time` may be used to determine where and when a plume can be detected at a detection level of 1e-7 Sv/h. `finaldose` is of interest for total impact and may for example been combined with population data.
In `plumes`:
`date` indicates when it was started;
The exact source location is secret according to the rules of the DETECT project from which the data was taken. Thus the location parameters have been anonymised, the projection is arbitrary.

The values actually needed in many examples can be computed from the given data. They are added by running `demo(radioactivePlumes_addProperties)`:
In `values` the layer `detectable` indicates weather `maxdose` exceeds the threshold of 1e-7 Sv/h here.
In `locations` the `area` is given in km^2.
In `plumes` the `totalDose` is the area weighted sum of `finaldose` over the whole area in Sv and `nDetectable` is the number of locations where `detectable` is TRUE.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

References

K.B. Helle, T.O. M\"uller, P. Astrup, J.E. Dyve (2014). Automatic Optimisation of Gamma Dose Rate Sensor Networks: The DETECT Optimisation Tool. Computers \& Geosciences (66), 158-167.

Examples

```
data(radioactivePlumes)
plot(radioactivePlumes, zcol = 2)
# plot first plume, all kinds of values
spplotLog(extractSpatialDataFrame(radioactivePlumes, plumes = 1))
demo(radioactivePlumes_addProperties)
plot(radioactivePlumes, zcol = 2)
spplot(radioactivePlumes@locations)
plot(radioactivePlumes@plumes$totalDose)
```

`replaceDefault` *Replace default values in functions and check parameter list*

Description

replaceDefault takes a function and a list of values to replace the default values. In addition it checks that the resulting function is fit for use in other functions, e.g. as cost function in optimiseSD.

Usage

```
replaceDefault(fun, newDefaults = list(), type)
```

Arguments

fun	function
newDefaults	list of new default values: elements must be named, names have to be those of parameters of fun (not necessarily all of them); then default values of fun are replaced; elements without or with other names cause a warning.
type	character string indicating the purpose of the resulting function, e.g. to be used as summaryFun in summaryPlumes; these function calls require certain parameters: it is checked if the fun has them and if all additional parameters have default values (after replacing defaults), and generates warnings, if not. For possible values, see details.

Details

Dependent on the type, a list of required parameters is generated and compared to the parameters of fun: all required parameters must exist and all other parameters need to have default values (in the resulting function, when defaults of newDefaults have been inserted)

"summaryFun.summaryPlumes" (summaryFun in summaryPlumes): "x", "weight"
"fun.simulationsApply" (fun, fun_l, fun_p, fun_pl in simulationsApply): "x", "nout"
"funR.simulationsApply" (fun_Rp or fun_Rl in simulationsApply): "x", "nout", "weight"
"funRR.simulationsApply" (fun_Rpl in simulationsApply): "x", "nout", "weight_l", "weight_p"
"fun_interpolation.interpolate" (fun_interpolation in interpolate): "data", "newdata", "y"

"fun.spatialSpread" (fun in spatialSpread): "allLocations", "locations"
"fun_R.spatialSpread" (fun_R in spatialSpread): "x"
"costFun" (costFun in optimiseSD): "simulations", "locations"
"optimisationFun.optimiseSD" (optimisationFun in optimiseSD): "simulations", "costFun", "locationsAll", "locationsFix", "locationsInitial", "aimcost", "aimNumber", "nameSave"
"evalFunc.rbga.bin"(evalFunc in rbga.bin): "chromosome"

For "fun.simulationsApply" and "funR.simulationsApply" the parameter "nout" must exist and have default value.

replaceDefault checks, if fun is a function. It does not check if the new defaults have the same class as the old ones.

Value

A list

fun	function fun with replaced default values
accept	logical if the function passed all tests (i.e. no warnings) – mainly needed when replaceDefault is called inside of other functions (e.g. simulationsApply)

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# examples with warnings
fun1 = "a"
fun_1 = replaceDefault(fun1)
fun2 = function(x){sum(x)}
fun_2 = replaceDefault(fun2, type = "fun.simulationsApply")
fun3 = function(simulations, locations){1}
# change default of 'simulations'
fun_3 = replaceDefault(fun3, type = "costFun",
  newDefaults = list(nout = 1, simulations = 10))
# example without warnings
# (new default values have different class, not tested)
fun6 = function(x = 17, weight = "a", extraWeight = matrix(1:12, nrow = 3)){}
fun_6 = replaceDefault(fun6,
  newDefaults = list(weight = 3, extraWeight = "c"),
  type =  "summaryFun.summaryPlumes")
```

SDF2simulations *Turn a SpatialDataFrame into Simulations.*

Description

Turn a SpatialDataFrame into Simulations: turn some of the data into the values, keep the others as the data of the locations.

Usage

```
SDF2simulations(x, indices = matrix(1:ncol(x@data), nrow = 1))
```

Arguments

x	SpatialDataFrame
indices	matrix of integers indicating the columns of the data to be transformed to values; each row stands for a kind of values

Details

This function can be used to extract the maps in Simulations objects: the values are assigned to the spatial properties, for plotting, combining it with other spatial data etc. As the size of all values may exceed the memory, it may extract only the first of the kinds and plumes.

Value

A SpatialDataFrame with the same spatial reference as the locations of the obj with the values of the chosen plume and layer as attribute.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SimulationsSmall)
SimulationsSmall_134_21 = extractSpatialDataFrame(
  SimulationsSmall,
  plumes = c(1,3,4), kinds = c(2,1))
```

SDLonLat *Transform locations of simulations into longitude latitude coordinates*

Description

Given a Simulations object and indices of locations it returns SpatialPoints in longitude and latitude.

Usage

```
SDLonLat(simulations, SD)
```

Arguments

simulations	a Simulations object
SD	vector of locations indices or matrix (with SDs in rows) or list

Value

List of SpatialPoints with WGS84 longlat projection.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(radioactivePlumes)
data(SDgenetic)
LL1 = SDLonLat(simulations = radioactivePlumes, SD = SDgenetic[["SD"]])
plot(LL1[[1]])
```

similaritySD *Determine (spatial) similarity between sampling designs*

Description

Takes sampling designs and a reference sampling design and computes similarity between them by various algorithms: number of coinciding sensors, sensors within a k-neighbourhood (based on get.knnx), Earth Mover's Distance (based on emd2d).

Usage

```
similaritySD(simulations, SD, referenceSD, type = "equal", k = 9, ...)
```

Arguments

simulations	Simulations object, needed to define spatial properties; not needed if type = "equal"
SD	vector, matrix (sampling designs in rows) or list of sampling designs; indices of the locations of simulations
referenceSD	sampling design (indices of the locations of simulations); all SD are compared to this sampling design
type	character, indicating comparison method, one of "equal", "Kneighbours", "EarthMoversDistance"
k	neighbourhood size, to be used for type = "Kneighbours", forwarded to get.knnx
...	parameters to be used for type = "EarthMoversDistance", forwarded to emd2d

Details

Originally Earth Mover's Distance compares matrices of two distributions. We can use it for cases where all locations lie on a regular grid. Then a sampling design is represented by a matrix of all grid cells: each sensor is represented by 1, all other cells are 0 (for locations on a SpatialPolygrid, a sensor may belong to several cells, then the weight 1 is distributed evenly to all these cells).

Value

A vector of length referring to the number of sampling designs in SD with the similarity between these sampling designs and referenceSD respectively.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## ---------------- prepare data --------------------------
# simulations with SpatialPolygridDataFrame
data(radioactivePlumes)

# simulations with SpatialPixelsDataFrame
data(meuse.grid)
coordinates(meuse.grid) = ~ x + y
gridded(meuse.grid) = TRUE
meuseGSim = subset(radioactivePlumes,
                   locations = 1:length(meuse.grid), plumes = 1:10)
meuseGSim@locations = meuse.grid

# simulations with rectangular SpatialPixelsDataFrame
data(SPixelsDF)
rectSim = subset(radioactivePlumes,
                 locations = 1:length(SPixelsDF), plumes = 1:10)
rectSim@locations = SPixelsDF # not nice, no checking if sizes fit

# simulations with SpatialPointsDataFrame
data(meuse)
coordinates(meuse) = ~ x + y
meuseSim = subset(radioactivePlumes,
                  locations = length(meuse), plumes = 1:10)
meuseSim@locations = meuse # not nice, no checking if sizes fit

# for meuseSim
sds0 = c(10,  25,  42,  84,  90,  92,  94,  97, 120, 153)#sample.int(155, 10)
sds1 = matrix(c(97,   79,   40,   68,  131,
                57,   18,   38,  118,   14,
                23,   71,   22,   94,   27,
                125,  108,    4,   80,  129,
                130,   96,  101,  137,   19,
                77,  138,   32,   95,   88,
                140,   73,   43,  153,   12,
                8,  141,   92,   35,  102),
              byrow = TRUE, nrow = 8)#matrix(sample.int(155, 40), nrow = 8)
sds2 = list(
  c(28, 59, 64, 78, 81),
  c(5,  13,  21,  31,  45,  91,  92, 122, 130, 141),
  c(1,   2,  19,  36,  50,  51,  58,  59,  90, 103, 105, 107, 112, 123, 132),
  c(34,  48,  49,  50,  71,  76,  77,  86,  92,  97, 100, 103, 104, 106, 108,
  110, 113, 127, 134, 142)
)

# ------------- "equal" -----------------------------
  sim_1_1 = similaritySD(# no simulations needed
    SD = sds1, referenceSD = sds0, type = "equal")

#------------- "Kneighbours" -------------------
  sim_2_1_1 = similaritySD(simulations = meuseGSim,
                           SD = sds2, referenceSD = sds0,
```

```
                              type = "Kneighbours", k = 4)
  sim_2_1_2 = similaritySD(simulations = meuseGSim,
                           SD = sds2, referenceSD = sds0,
                           type = "Kneighbours", k = 9)

# plot points for visual comparison
plot(coordinates(meuseSim@locations), pch = ".")
   points(coordinates(meuseSim@locations)[sds0,], cex = 0.5)
   for (i in seq(along = sds2)){
     points(coordinates(meuseSim@locations)[sds2[[i]],],
            col = i + 1)
   }
#---------- "EarthMoversDistance"----------
  # SpatialPixelsDataFrame
  sim_3_1_3 = similaritySD(simulations = rectSim,
                                SD = c(2,9), referenceSD = c(3,8),
                                type = "EarthMoversDistance")

  sim_3_1_4 = similaritySD(simulations = rectSim,
                                SD = c(3,8), referenceSD = c(4,5),
                                type = "EarthMoversDistance")
  sim_3_1_5 = similaritySD(simulations = rectSim,
                                SD = c(3,8), referenceSD = c(4,5),
                                type = "EarthMoversDistance", dist = "manhattan")

  ## SpatialPolygridDataFrame
  sim_3_1_5 = similaritySD(simulations = radioactivePlumes,
                                SD = c(220, 503), referenceSD = c(224, 544),
                                type = "EarthMoversDistance")
```

`Simulations-class` *Class "Simulations"*

Description

Class for maps of plume simulations with spatial reference and plume properties.

Objects from the Class

Objects hold plume simulations and additional data: plume simulations are maps, all defined by values at the same set of locations; there may be several maps for each plume (e.g. the concentration of different pollutants in the same accident scenario, values at different times, etc.). As all maps have the same spatial properties, it is sufficient to hold the `locations` once as a `SpatialDataFrame`. The simulations themselves can be represented by a matrix where rows refer to location and columns refer to different plumes; in case of several maps per plume this can be extended to an array. These `values` arrays are actually implemented as `raster` objects; thus they can hold more data than it fits into memory. To keep information that refers to the single plumes like the kind of scenario they belong to or their probability to occur, a `data.frame` of `plumes` is part of each `Simulations` object.

Objects may be created by calls to the function `Simulations(locations, plumes, values)`.

Slots

`locations`: SpatialDataFrame; locations and location-related data, e.g. population or cost to put a sensor here (n locations)

`plumes`: `data.frame`; data related to plumes but not to locations, e.g. average impact of the plume (aggregated over space) or likelihood of this plume to occur — rows refer to plumes, columns to different kind of information (N plumes)

`values`: raster; maps of plume values; each column represents the map of one plume: raster layers do not represent maps, but contain the values of the maps of all plumes (n x N values) — spatial properties of the `values` are meanigless and must have the standardised form —; there can be different layers to contain different kind of data, e.g. cumulative concentrations of a pollutant and time when pollution started.

Methods

nLocations number of `locations` (equals `nrows(values)`)

nPlumes number of `plumes` (equals `ncols(values)`)

nKinds number of kind of `values` (equals `nlayers(values)`)

cbind cbind-like method to combine the plumes of different `Simulations` with same locations and value types; for details see `cbind`

extractSpatialDataFrame extract one `plume` from one layer of the `values` and assign it to the `locations` and return the resulting `SpatialDataFrame`; ; for details see `extractSpatialDataFrame`

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# generate Simulations object: small, artificial example
data(SPixelsDF)
plumes = data.frame(source = c("A", "A", "B", "B", "B"),
                    date = c("2000-01-01", "2000-04-01",
                             "2000-07-01", "2000-01-01", "2000-01-03"),
                    totalCost = runif(5, min = 5, max = 15))

values1 = replicate(n = nrow(plumes), expr = rlnorm(length(SPixelsDF), sdlog = 2))
values2 = replicate(n = nrow(plumes), expr = rnorm(length(SPixelsDF), m = 10, sd = 3))
values = stack(raster(x = values1, xmn = -90, xmx = 90, ymn = -90, ymx = 90,
                      crs = "+init=epsg:4326" ),
               raster(x = values2, xmn = -90, xmx = 90, ymn = -90, ymx = 90,
                      crs = "+init=epsg:4326" ))

Simulations1 = Simulations(locations = SPixelsDF, plumes, values)

# nLocations, nPlumes, nKinds
nLocations(Simulations1)
nPlumes(Simulations1)
nKinds(Simulations1)
```

`simulationsApply` *Apply functions value-wise, location-wise, or plume-wise to Simulations*

Description

Apply functions to the `@values` of Simulations objects or to Raster* objects. Functions can be applied to any margin: value-wise, plume-wise, or location-wise. The results can be summarised by a second function. Processing of the data may be done in blocks, this allows to process data that does not fit into memory. However, this only works if input and output of a single function fit into memory.

Usage

```
simulationsApply(simulations,
  locations = 1:nLocations(simulations),
  plumes = 1:nPlumes(simulations),
  kinds = 1:nKinds(simulations),
  fun = NA, fun_p = NA, fun_l = NA, fun_pl = NA,
  fun_Rp = NA, fun_Rl = NA, fun_Rpl = NA, fun_Rpl_cellStats = NA,
  nameSave = "simulationsApply", overwrite = FALSE,
  chunksize = 1e+7, keepSubset = FALSE, ...)
```

Arguments

`simulations`	Simulations object or Raster* object
`locations`	indices of locations, i.e. rows of `simulations@values` to be taken into account
`plumes`	indices of plumes, i.e. columns of `simulations@values` to be taken into account
`kinds`	index or name of the layer of `simulations@values` to be used
`fun`	function to be applied to all values of the subset at once, see details
`fun_p`	function to be applied plume-wise, see details
`fun_l`	function to be applied location-wise, see details
`fun_pl`	function to be applied value-wise, see details
`fun_Rp`	function to be applied to the result of `fun_p`, see details
`fun_Rl`	function to be applied to the result of `fun_l`, see details
`fun_Rpl`	function to be applied to the result of `fun_pl`, see details
`fun_Rpl_cellStats`	character string of function to be applied to the result of `fun_pl` via `cellStats`, must be one of `c("sum", "mean", "min", "max", "sd", "skew", "rms")`
`nameSave`	filename for the raster file in case the result does not fit into memory; if `FALSE` the function stops with a warning and does not create a file
`overwrite`	boolean, if the file at `nameSave` may be overwritten

chunksize	maximal number of cells to be processed at once – forwarded to blockSize inside
keepSubset	boolean, if the subset of the values of simulations generated by locations and plumes is to be kept
...	further parameters to be passed on to called functions

Details

The defaults for locations, plumes and kinds only work for simulations of class Simulations. If it is of class raster, the user has to provide these parameters. If plumes or locations contain an index several times, these plumes / locations are considered that many times, also order is kept. Invalid indices cause stop with warning.

The functions must have a certain form, else the function stops with a warning: they need the parameters x for the input values and nout, the length of the output (to generate the arrays to hold the results). Several functions to be applied can be specified. However, fun_Rp, fun_Rl, and fun_Rpl make only sense if the functions that generate their input are specified. fun_Rp and fun_Rl need the additional parameter weight, inside of simulationsApply this parameter is set to the data frames associated with the plumes or locations, respectively: you may refer to names of these data frames inside the functions (default is overwritten); if simulations is a raster object, no values are inserted as weight – the parameter must be given, but unless it has default values you cannot refer to it in the function. fun_Rpl needs the parameters weight_l and weight_p which are replaced by the data frames associated with locations and plumes. All functions are checked by replaceDefault automatically with type = "fun.simulationsApply"; fun_Rp, fun_Rl with type = "funR.simulationsApply" and fun_Rpl with type = "funRR.simulationsApply".

The output of fun, fun_p, or fun_l is supposed to be small enough to keep it in memory; if not, it stops with a warning. The results of fun_pl may be too big to keep in memory. If nameSave is a filename, these data are saved as raster files. The subset of values (defined by plumes and locations) is only kept if it fits into memory.

Value

List of values and arrays (dimensions: references (locations, plumes), output parameters of the function): "result_global": result of fun "result_plumes": result of fun_p "result_locations": result of fun_l "result_global_plumes": result of fun_Rp "result_global_locations": result of fun_Rl "result_locationsplumes": result of fun_pl; RasterBrick "result_global_locationsplumes": result of fun_Rpl "result_global_locationsplumes": result of fun_Rpl_cellStats "subset": subset of data, according to plumes and locations If some function cannot be applied (wrong parameters, input not in memory...), the result of this function is not returned with a warning.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## Not run:
## may create files
demo(radioactivePlumes_addProperties)
```

```
# number of not detected plumes for given set of 10 sensors,
# weighted by total dose of plumes
sensors = sample.int(nLocations(radioactivePlumes), 10)

nondetection = function(x, threshold = 1e-7, nout = 1){
  all(x[,2] < threshold)
}

sumWeighted = function(x, weight, nout = 1){
  sum(x * weight$totalDose)
}

weightedSumUndetectedAtSensors = simulationsApply(
  simulations = radioactivePlumes,
  locations = sensors,
  fun_p = nondetection,
  fun_Rp = sumWeighted
)

# map of average time until detection of plumes
# if a plume never reaches a location it is counted as being there after a week
meanDetectionTime = function(x, nout = 1){
  y = x
  y[is.na(x)] = 7 * 86400
  z = mean(y, na.rm = TRUE)
}

mapMeanDetectionTime = simulationsApply(
  simulations = radioactivePlumes,
  kinds = 3,
  fun_l = meanDetectionTime
)
radioactivePlumes@locations@data$meanDetectionTime =
  mapMeanDetectionTime[["result_locations"]]
spplot(radioactivePlumes@locations, zcol = "meanDetectionTime")

# general ratio and difference of 'maxdose' and 'finaldose'
ratioMaxFinal =  function(x, nout = 2){
  ratio = x[2]/x[1]
  diff = x[2] - x[1]
  ratio[!is.finite(ratio)] = NA
  out = c(ratio, diff)
}
valuesRatio = simulationsApply(
  simulations = radioactivePlumes,
  fun_pl = ratioMaxFinal,
  fun_Rpl_cellStats = "mean",
  nameSave = "ratio"
)
hist(valuesRatio[["result_locationsplumes"]], 1,
  xlim = c(0,1), breaks = c(seq(0, 1, 0.01), 10000))
hist(valuesRatio[["result_locationsplumes"]], 2,
```

```
  xlim = c(-0.001, 0.001), breaks = c(-1000, seq(-0.001, 0.001, 0.0001), 1000))


## End(Not run)
```

SimulationsSmall *Artificial, small test data.*

Description

From example in Simulations.Rd (with random values).

Usage

```
data(SimulationsSmall)
```

Format

SimulationsSmall: Simulations

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SimulationsSmall)
```

SpatialDataFrame-class
Class "SpatialDataFrame"

Description

Class for a data frame with associated spatial attributes (points, lines, polygons, (irregular) grid).

Usage

```
is.SpatialDataFrame(SpatialDataFrame)
```

Arguments

`SpatialDataFrame`
: an object, if its class belongs to one of the subclasses of `SpatialDataFrame-class` the function returns `TRUE` else `FALSE`

Objects from the Class

This class is a wrapper for some classes from package `sp`:

SpatialPointsDataFrame-class,

SpatialPixelsDataFrame-class,

SpatialPolygonsDataFrame-class,

SpatialLinesDataFrame-class

Besides it contains two classes specific to `sensors4plumes`:

SpatialIndexDataFrame-class: Objects of this class have no spatial reference, they are just `data.frames` with an extra index.

SpatialPolygridDataFrame: These objects are similar to those from SpatialGridDataFrame-class, but several grid cells may refer to the same attributes in the data frame.

Methods

The following methods exist for all objects of class *SpatialDataFrame*. They may be defined individually for the respective subclasses, refer to the subclasses for details.

is.SpatialDataFrame `TRUE` for objects of class `SpatialDataFrame`, else `FALSE`

length `signature(x = "SpatialDataFrame")`: length (number of spatial elements)

coordinates `signature(x = "SpatialDataFrame")`: retrieves coordinates (of centroids)

proj4string `signature(x = "SpatialDataFrame")`: retrieves the projection as CRS-class

bbox `signature(x = "SpatialDataFrame")`: retrieves the bounding box

spplot `signature(x = "SpatialDataFrame", zcol = "integer", ...)`: plots maps of the attributes `zcol` ("integer" or names of `x@data`); wrapper of `spplot`

subsetSDF `signature(x = "SpatialDataFrame", locations = "integer", data = "integer")`: selects spatial sub-objects (`locations`) and/or attributes (`data`: "integer" or "character" names of `x@data`), returns an object of the same class. This is a general method for all `SpatialDataFrame` objects. There are special `subset` or `"["` functions for the subclasses that may have more possibilities to choose (e.g. subset grid by a bounding box), see the subclasses for details.

areaSDF `signature(x = "SpatialDataFrame")`: computes the area/length of the elements (grid cells, polygons...), for point data it returns 0s.

Refer to the subclasses for further methods.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# generate a SpatialPixelsDataFrame
data(meuse.grid)
gridded(meuse.grid) = ~ x + y

# extract some locations and attibutes
meuse.subgrid = subsetSDF(meuse.grid,
        locations = c(1:45),
        data = c("dist", "soil", "ffreq"))

# plot
spplot(meuse.subgrid, zcol = 1:2)
```

SpatialIndexDataFrame-class

Class "SpatialIndexDataFrame"

Description

This class is an extension of the data.frame class. In addition it has an index vector of arbitrary length, consisting of integers that assign the rows of the data to the positions in the index. The class is needed to complete the SpatialDataFrame-class in cases where no spatial information is available.

Objects from the Class

Objects can be created by calls of the form `as(x,"SpatialIndexDataFrame")`, from `x` of class `data.fame`, or from scratch with `SpatialIndexDataFrame(index, data.frame)`.

Slots

`data`: Object of class `data.frame`.

`index`: Object of class `integer`; it must consist of the values `1:nrow(data)` and may contain `NA`. The length is arbitrary.

`bbox`: Object of class `matrix`; no values (only needed to fit the structure of `SpatialDataFrame`).

`proj4string`: Object of class `CRS`; no values (only needed to fit the structure of `SpatialDataFrame`).

Methods

coordinates Returns `NULL`, only needed to fit the structure of the `SpatialDataFrame` class.

proj4string Returns `NA`, only needed to fit the structure ofthe `SpatialDataFrame` class.

bbox Returns matrix of 0s, only needed to fit the structure ofthe `SpatialDataFrame` class.

spplot Turns index into a column (top to bottom) of grid cells with the respective values and plots it by `spplot`. Layout parameters like colours may be forwarded to spplot.

cbind Method to combine values of `SpatialIndexDataFrame` objects with identical `index`.

rbind Method to combine `SpatialIndexDataFrame` objects with `data` of identical type and name.

subsetSDF See subsetSDF.SpatialIndexDataFrame. There is no `"["` function for this class.

areaSDF See areaSDF, returns vector of 0s, needed to fit the structure of the `SpatialIndexDataFrame` class.

length `signature(x = "SpatialIndexDataFrame")`: number of elements, i.e. of groups of cells with respectively share the attributes.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

See Also

`SpatialDataFrame-class`, which is a wrapper.

Examples

```
# generate SpatialIndexDataFrame from scratch
index = c(2,2,1,1,1,1,3,2,3,3,3,1)
SIndexDF1 = SpatialIndexDataFrame(index = as.integer(index),
                          data = data.frame(a = c(1,2,4),
                                            b = c(0.1, 0.2, 0.3),
                                            c = c("A", "B", "A")))
# generate from data.frame
data(USArrests)
SIndexDF2 = as(USArrests,"SpatialIndexDataFrame")


# spplot
spplot(SIndexDF1, zcol = c("b", "c"), col.regions = grey.colors(10))

# cbind
SIndexDF3 = cbind(SIndexDF1, SIndexDF1)
```

SpatialPolygridDataFrame-class

Class "SpatialPolygridDataFrame"

Description

Class for a data frame with associated spatial grid, several grid cells may share one attribute.

Objects from the Class

Objects can be created by calls to the function `SpatialPolygridDataFrame`. They may also be created from `SpatialGridDataFrame` objects by calls of the form `y = as(x, "SpatialPolygridDataFrame")`, in this case all grid cells have individual attributes. Transformation in the opposite direction can be done by `x = as(y, "SpatialGridDataFrame")`; it copies the attributes to all assigned grid cells. SpatialPolygridDataFrame objects can also be created

from SpatialPointsDataFrame objects by calls of the form as(x, "SpatialPolygridDataFrame"); it generates a rectangular grid that covers all points and then assigns cells to the nearest neighbouring point – it may fail if points are too irregular to allow for a common grid.

Slots

data: object of class data.frame; contains the data, one row per index

index: object of class integer, assigns data to grid cells: order indicates reference to grid cells: upper left to right, then to bottom – length must agree with number of grid cells; value indicates the row of the data with the related attributes; may contain NA, and values outside of 1:nrow(data) are ignored.

grid: object of class GridTopology

bbox: bounding box of the grid

proj4string object of class CRS; projection of the grid, may be missing

Methods

coordinates signature(obj = "SpatialPolygridDataFrame"): Retrieves the mean coordinates for each index, i.e. the centroid of all grid cells belonging to it.

proj4string signature(obj = "SpatialPolygridDataFrame"): Returns the projection character string, the value of the slot proj4string.

bbox signature(obj = "SpatialPolygridDataFrame"): Returns the bounding box matrix, the value of the slot bbox.

spplot signature(obj = "SpatialPolygridDataFrame", geoTiffPath, zcol = names(obj@data), plot = TRUE, r
Method to plot maps, by coercing the SpatialPolygridDataFrame into a SpatialGridDataFrame – see above – and using spplot to plot it. Therefore it may take any layout parameters to be forwarded to spplot. To keep the resulting SpatialGridDataFrame it may be returned by returnSGDF = TRUE. It can also be saved to a (multilayer) tif file by providing a file path in geoTiffPath. To speed up the function, it may be run without graphical output by plot = FALSE, e.g. to directly create a tif file.

cbind signature(...): cbind-like method; if grids and indices agree, the data are combined by cbind.

subsetSDF Subsetting, see subsetSDF,SpatialPolygridDataFrame-method; there is no "[" function.

areaSDF Method to compute the areas of all polygrids, see areaSDF,SpatialPolygridDataFrame-method.

length signature(x = "SpatialPolygridDataFrame"): number of elements, i.e. of groups of cells with respectively share the attributes.

Note

The code for coercing is based on polygrid2grid and points2polygrid. Using these functions directly allows better control of the parameters, like the resolution of the grid.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

See Also

SpatialDataFrame-class, which is a wrapper.

Examples

```
# prepare
index1 = as.integer(
  c( 6, 6, 7, 7, 8, 8,
     6, 6, 7, 7, 8, 8,
     5, 5, 1, 2, 9, 9,
     5, 5, 4, 3, 9, 9,
     12,12,11,11,10,10,
     12,12,11,11,10,10))
dataFrame1 = data.frame(a = 1:12 * 10, b = 2^(12:1))
grid1 = GridTopology(c(1,1), c(2,2), c(6,6))
spatialGrid1 = SpatialGrid(grid1, CRS("+proj=longlat +datum=WGS84"))

#- - create object by function SpatialPolygridDataFrame - -#
SPolygridDF1 = SpatialPolygridDataFrame(
  grid = grid1,
  data = dataFrame1,
  index = index1,
  proj4string = CRS("+proj=longlat +datum=WGS84"))

#- - - - coerce from SpatialGridDataFrame - - - - - - - -#
spatialGridDataFrame1 = SpatialGridDataFrame(
    grid = spatialGrid1, data = dataFrame1[index1,])
# coerce, each cell keeps individual values
s = as(spatialGridDataFrame1, "SpatialPolygridDataFrame")

# coerce back (name of coordinates may change)
#spatialGridDataFrame2 = as(SPolygridDF2, "SpatialGridDataFrame")
spatialGridDataFrame2 = as(s, "SpatialGridDataFrame")

#- - - - coerce from SpatialPointsDataFrame - - - - - - - #
# from irregular points, grid is created, cells are assigned values of nearest point
spatialPoints1 = SpatialPoints(coordinates(SPolygridDF1) + runif(24))
spatialPointsDataFrame1 = SpatialPointsDataFrame(
    coords = spatialPoints1, data = dataFrame1)
## Not run:
## takes some seconds
SPolygridDF3 = as(spatialPointsDataFrame1, "SpatialPolygridDataFrame")


# from regular points: one cell per point
spatialPointsDataFrame2 = as(spatialGridDataFrame1, "SpatialPointsDataFrame")
SPolygridDF4 = as(spatialPointsDataFrame2, "SpatialPolygridDataFrame")

# - - - - - - - - - - - - - - coordinates - - - - - - - - - - - - - #
# compare irregular input points and centroids of output
plot(spatialPoints1)
points(coordinates(SPolygridDF3))
```

```
# - - - - - - - - - - - - - - proj4string - - - - - - - - - - - - -  #
# only to retrieve projection attributes
#proj4string(SPolygridDF2)
proj4string(s)
proj4string(SPolygridDF3)
# - - - - - - - - - - - - - - proj4string - - - - - - - - - - - - -  #
bbox(SPolygridDF3)
# - - - - - - - - - - - - - spplot - - - - - - - - - - - - - - - - - #

## takes some time
# plot map with original points
spplot(SPolygridDF3, zcol = "a",
  sp.layout = list(list("sp.points", spatialPoints1, col = 8)))
# plot map with centres of polygrid cells
spplot(SPolygridDF4, zcol = "a", col.regions = rainbow(16),
  sp.layout = list("sp.points", spatialPointsDataFrame2@coords, col = 8))


# - - - - - - - - - - - - - - cbind - - - - - - - - - - - - - - - - - #
#SPolygridDF5 = cbind(SPolygridDF2, SPolygridDF4)
SPolygridDF5 = cbind(s, SPolygridDF4)

## End(Not run)
```

`spatialSpread` *Cost function based on sensor locations only*

Description

This is a wrapper function to compute cost of a set of sensors based on their coordinates only. It takes as parameter `fun` an actual function to compute such cost for each location of the provided `simulations` and can summarise it with a function `fun_R`

Usage

```
spatialSpread(simulations, locations, weightByArea = TRUE, fun, fun_R)
```

Arguments

`simulations`	`Simulations` object or `SpatialDataFrame` to compute local cost in each `location` (`values` are ignored)
`locations`	indices of `locations` of sensors
`weightByArea`	if cost in the locations has to be multiplied by the area associated to them before summary; for `"SpatialPointsDataFrame"` the factor is 0.
`fun`	`function` to compute location-wise cost, for examples see `spatialSpreadFunctions`; needs the parameters `allLocations`, `locations`; can be the result of `replaceDefault` with `type = "fun.spatialSpread"`; result must have length number of locations

fun_R	function to summarise the result of fun; must have parameter x, generate by replaceDefault with type = "funR.spatialSpread"

Details

The function itself is only a wrapper to turn simulations and locations into the correct form and apply the functions.

Value

List

cost	result of fun_R, missing if no fun_R given
costLocations	result of fun, length equals number of locations in simulations

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# prepare data and functions
data(radioactivePlumes)
data(medianVariogram)

krigingVarianceMedian =
  replaceDefault(krigingVariance, newDefaults = list(model = medianVariogram))[["fun"]]

meanFun = function(x){mean(x, na.rm = TRUE)}

locationsSensors = sample.int(nLocations(radioactivePlumes), 50)

spatialSpread_minDist = spatialSpread(
    simulations = radioactivePlumes,
    locations = locationsSensors,
    weightByArea = TRUE,
    fun = minimalDistance,
    fun_R = meanFun
)
spatialSpread_krigingVar = spatialSpread(
    simulations = radioactivePlumes,
    locations = locationsSensors,
    weightByArea = TRUE,
    fun = krigingVarianceMedian,
    fun_R = meanFun
  )

# plot maps
## Not run:
## takes some seconds
y = radioactivePlumes@locations
y@data$minDist = spatialSpread_minDist[["costLocations"]]
```

```
y@data$krigVar = spatialSpread_krigingVar[["costLocations"]]
yPoints = as(y, "SpatialPointsDataFrame")

# distance to next sensor
spplot(y, zcol = "minDist",
       sp.layout = list("sp.points", yPoints[locationsSensors,],
                        col = 3))
# kriging variance
spplot(y, zcol = "krigVar",
       sp.layout = list("sp.points", yPoints[locationsSensors,],
                        col = 3))

## End(Not run)
```

`spatialSpreadFunctions` *Cost functions dependent only on sensor locations*

Description

`Kriging variance` computes the ordinary kriging variance at a given set of points for the case of certain sensor locations and variogram model.

`minimalDistance` computes for each given point the distance to the next sensor.

Both functions can be used as input to `spatialSpread` that runs them and turns the result into global cost.

Usage

```
krigingVariance(allLocations, locations, model)
minimalDistance(allLocations, locations, algorithm = "kd_tree")
```

Arguments

`allLocations`	`SpatialDataFrame` (not "SpatialIndexDataFrame"), for each of the locations a cost is computed
`locations`	indices of sensors in `allLocations`
`model`	`model` to be forwarded to `krige`
`algorithm`	`character` to be forwarded as `algorithm` to `get.knnx`

Value

A vector of length `allLocations` with cost for these locations.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
# see spatialSpread
```

---

spplotLog *Plot methods for spatial data with logarithmic scale*

Description

This function extends spplot; it enabels logarithmic scaling.

Usage

```
spplotLog(x, zcol, base, nTicks = 10, replace0 = FALSE, minNonzero = 0, ...)
```

Arguments

x	SpatialDataFrame-class object to be plotted
zcol	character; name(s) of attribute(s) to be plotted
base	integer; base for the logarithmic transformation, to be used in the ticks; if none is indicated it is selected automatically to fit the range of the data
nTicks	integer; number of ticks - approximate -
replace0	logical; if zeros are to be replaced by a value that can be displayed on logscale. The tick of the key at this value will show "0". If replace0 = FALSE 0 is plotted as missing value.
minNonzero	numeric >= 0; all values below minNonzero are replaced by 0. Can be used to avoid that very small values dominate the plot.
...	parameters to be forwarded to spplot for style, colour etc.)

Details

The key refers to the original values although in the background log transformed data are used (except from SpatialPointsDataFrame where just the cuts are changed).

Value

A plot.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SPixelsDF)
data(SPointsDF)
# add arbitrary data to show the effect of different values (< 0, == 0, small, big)
SPixelsDF@data$a = c(-5, 0, 0.003, 10, 57, 320, 444, 1000, 10000)
# replace 0 and small values
spplotLog(SPixelsDF, replace0 = TRUE, minNonzero = 0.05)
# choose base an number of ticks manually; forward parameters to spplot
spplotLog(SPixelsDF, base = 5, nTicks = 20,
          col.regions = heat.colors,
          sp.layout = list("sp.points", SPointsDF[1,]))
```

---

subset.Simulations *Subsetting Simulations*

Description

Return subsets of Simulations, subsetting by plumes, locations, the kinds of values or by the columns of data associated with plumes or locations – possibly combined.

Usage

```
## S3 method for class 'Simulations'
subset(x, ..., locations, plumes, kinds, dataLocations, dataPlumes,
  nameSave = NA, overwrite = FALSE, valuesOnly = FALSE)
```

Arguments

x	Simulations
...	parameters to be forwarded
locations	indices of locations to keep
plumes	indices of plumes to keep
kinds	indices of kinds of values to keep
dataLocations	indices of the data associated with the locations to keep
dataPlumes	indices of the data associated with the plumes to keep
nameSave	name for saving the new raster files for the values if they cannot be kept in memory (without suffix)
overwrite	logical if files may be overwritten by the raster files
valuesOnly	logical if only the subset of values is returned (if FALSE the result is the full new Simulations)

Details

Multiple or invalid values in `locations` and `plumes` are ignored, only the order is taken into account. (Because in `SpatialIndexDataFrame` and `SpatialPolygridDataFrame` no locations can be selected multiply.)

However, if `valuesOnly = TRUE`, multiplity is taken into account and invalid indices result in `NA` values. In this case only the `values` are returned.

Value

A `Simulations` object, subsetted by the given indices: subsetting `locations` or `plumes` results in subsetting of the `values`. If the resulting raster is too big to keep in memory, it is saved.

If `valuesOnly = TRUE`, only the new `values` are returned, i.e. a `raster` object.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
## Not run:
## may create files
data(SimulationsSmall)

SimulationsSmall_a = subset(SimulationsSmall, locations = c(2,4,6,8))
SimulationsSmall_b = subset(SimulationsSmall,
                            locations = 1:5, plumes = c(2,4), kinds = 1,
                            dataPlumes = 1:2)

# effect of 'valuesOnly'
data(radioactivePlumes)
locations1 = sample(nLocations(radioactivePlumes), 15)
plumes1 = sample(nPlumes(radioactivePlumes), 15)
subset_fm_RNA = subset(radioactivePlumes,
                       plumes = c(plumes1, NA, 10000, NA, 0, plumes1),
                       locations = c(locations1, NA, 10000, NA, 0,locations1))

subset_mf_RNA_v0 = subset(radioactivePlumes,
                          plumes = c(plumes1, NA, 0, 10000, NA, plumes1),
                          locations = c(locations1, NA, NA, 10000, 0, 0, locations1),
                          valuesOnly = TRUE)
image(subset_fm_RNA@values)
image(subset_mf_RNA_v0)   # with repetitions

## End(Not run)
```

subsetSDF *Subsetting objects of class SpatialDataFrame*

Description

Subsetting of objects of class SpatialDataFrame: selects according to the spatial objects or to the attribute(s), and returns an object of the same class.

Usage

```
subsetSDF(x, locations, data = names(x@data), ...)
```

Arguments

x	object of class SpatialDataFrame
locations	integer vector: keep object at these spatial locations
data	integer vector or names of columns of the x@data to be kept
...	further arguments to be passed to other methods

Value

An object of the same SpatialDataFrame class.

Note

This is a wrapper function for the subsetting methods in all subclasses of SpatialDataFrame, with the parameters that are common to all classes; for the general subsetting methods refer to subset.Spatial for the classes imported from package sp and to subsetSDF.SpatialIndexDataFrame and subsetSDF.SpatialPolygridDataFrame.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SIndexDF)
data(SPointsDF)
data(SPixelsDF)
data(SPolygridDF)
data(SPolygonsDF)
data(SLinesDF)

sub_Index = subsetSDF(SIndexDF, locations = c(1,3), data = "c")
sub_Points = subsetSDF(SPointsDF, locations = c(1,3), data = "z")
sub_Pixels = subsetSDF(SPixelsDF, locations = c(1,3), data = "z")
sub_Polygrid = subsetSDF(SPolygridDF, locations = c(1,3), data = "b")
sub_Polygons = subsetSDF(SPolygonsDF, locations = c(1,3), data = "a")
```

```
sub_Lines = subsetSDF(SLinesDF, locations = 1:2, data = "a")
```

`subsetSDF.SpatialIndexDataFrame` *Subsetting objects of class SpatialIndexDataFrame*

Description

Subsetting of objects of class `SpatialIndexDataFrame`: selects index and/or attribute(s), and returns an object of class `SpatialIndexDataFrame`.

Usage

```
subsetSDF.SpatialIndexDataFrame(x, locations, data = names(x@data), ..., grid)
```

Arguments

`x`	object of class `SpatialIndexDataFrame`
`locations`	integer vector: keep data where `x@index` has this value (multiplicity ignored, order taken into account)
`data`	integer vector or names of columns of the `x@data` to be kept (order and multiplicity ignored)
`...`	further arguments to be passed to other methods
`grid`	integer vector: indices of `x@index`, keep the corresponding data

Value

A `SpatialIndexDataFrame`; the subsetting can be done cumulatively, e.g. if `grid` and `locations` is given, only the index entries that fullfill both criteria are kept.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SIndexDF)
# subset
SIndexDF1 = subsetSDF(SIndexDF, grid = which(SIndexDF@index == 1))
SIndexDF2 = subsetSDF(SIndexDF, locations = 1) # identical to x_1
SIndexDF3 = subsetSDF(SIndexDF, grid = c(2:4, 6:8, 10:12, 14:16, 18:20), data = "c")
```

`subsetSDF.SpatialPolygridDataFrame` *Subsetting objects of class SpatialPolygridDataFrame*

Description

Subsetting of the grid by either indices or coordinates, or by a logical mask; subsetting the index or the data.

Usage

```
subsetSDF.SpatialPolygridDataFrame(x, locations, data = names(x@data), ...,
  coord_x, coord_y, grid_i, grid_j, grid_ij)
```

Arguments

`x`	object of class SpatialPolygridDataFrame whereof we want a subset
`locations`	integer vector: the locations of x with these indices to be kept (refers to `x@index`); order is taken into account, multiplicity is ignored
`data`	integer vector or names of columns of the data of x to be kept (refers to `x@data`)
`...`	further arguments to be passed to other methods
`grid_i`	integer vector: the columns of the grid of x to be kept, from top (North) to bottom (South)
`grid_j`	integer vector: the rows of the grid of x to be kept, from left (West) to right (East)
`coord_x`	two numerical values: min and max coordinates in x-direction (columns, from West), keep the part in between, including the grid cells with these x-coordinates
`coord_y`	two numerical values: min and max coordinates in y-direction (rows, from South), keep the part in between, including the grid cells with these y-coordinates
`grid_ij`	matrix of logical, same size as x (ncol(grid_ij) must be `x@grid@grid.cells.dim[1]`, `nrow(grid_ij)` must be `x@grid@grid.cells.dim[2]`); keep values in grid cells where `grid_ij` is `TRUE` For `grid_i`, `grid_j` order is ignored, original order of grid is kept.

Value

All subsetting parameters can be combined, the result is the intersection of all parameters, e.g. if `index_i` and `coord_y` is given, only the colums that have an index of `index_i` and coordinates within the limits of `coord_y` are returned. If invalid values are chosen (outside of the grid), a warning is thrown and the intersection of the parameters with `x` is used. – Only for `data`, invalid values result in an error.

The result is a SpatialPolygridDataFrame, cropped to the extent of the remaining grid cells with values. Nonselected parts within this grid are replaced by NA in the index.

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SPolygridDF)
# subset only attributes
SPolygridDF1 = subsetSDF(SPolygridDF, data = 2)
# subset by locations
SPolygridDF2 = subsetSDF(SPolygridDF, locations = c(1,3,5,7))
# subset by rows and columns of the grid
SPolygridDF3 = subsetSDF(SPolygridDF, grid_i = 3:4, grid_j = 3:9)
# subset by individual grid cells
Grid_ij = matrix(nrow = 6, ncol = 6, byrow = TRUE, data = c(
                         TRUE, FALSE, FALSE, FALSE, FALSE, TRUE,
                        FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
                        FALSE, FALSE,  TRUE,  TRUE, FALSE, FALSE,
                        FALSE, FALSE,  TRUE,  TRUE, FALSE, FALSE,
                         TRUE, FALSE, FALSE, FALSE, FALSE, TRUE,
                        FALSE, FALSE, FALSE, FALSE, FALSE, FALSE))
SPolygridDF4 = subsetSDF(SPolygridDF, grid_ij = Grid_ij)
# subset by bounding box of coordinates
SPolygridDF5 = subsetSDF(SPolygridDF, coord_x = c(3,11), coord_y = c(3,11))

# combined subset (overlap of all subsetting parameters), result has 1 cell
SPolygridDF6 = subsetSDF(SPolygridDF,
data = "a", locations = 4:1, coord_x = c(0,6), grid_i = 1:3)
```

---

summaryLocations *Summarise values of Simulations by locations*

Description

This function is similar to apply for the values of Simulations location-wise. It takes a function and returns a vector with a value for each location: the result of the function applied to all values belonging to this location. This can be useful to generate maps.

Usage

```
summaryLocations(simulations,
  locations = 1:nLocations(simulations),
  plumes = 1:nPlumes(simulations),
  kinds = 1,
  fun, summaryFun = weightedMean,
  weight = 1, na.rm = FALSE, ...)
```

Arguments

simulations	Simulations
locations	indices of locations to be taken into account (to apply the function to a subset of simulations); for invalid indices NA values are used
plumes	indices of plumes to be taken into account; for invalid indices NA values are used
kinds	index of kind of values to be taken into account; only one kind can be used, all but the first given index are ignored
fun	function to be applied; it must have a single parameter (vector) and return a single value; in addition it needs a parameter na.rm or ...; possible choices are sum, prod, max, min
summaryFun	function to summarise the plume-wise results (as first parameter) to a global value; by default it is a weighted mean; in addition it can have the parameters weight, or use ...
weight	to be used by summaryFun: either a numeric value of length 1 or same length as plumes or a character indicating a column of the data associated with the plumes of simulations
na.rm	logical how to treat missing values by fun (has no influence on summaryFun)
...	further arguments to be forwarded to fun or summaryFun

Details

summaryLocations is similar to summaryPlumes but is not restricted to associative functions. For a more general function see simulationsApply. To use a different summaryFun you may use replaceDefault with kind = "summaryFun.summaryPlumes" to check the function beforehand or to generate it by resetting the default parameters of an existing function.

Value

A list of

summary	the result of summaryFun
summaryLocations	the location-wise results of fun, one value for each selected location

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(radioactivePlumes)
plumeSample =  c(1:5, 16:20, 30:34, 45:49)# sample of plumes: jan, apr, jul, oct

# plume detection
# (number of plumes in plumeSample that can be detected at a level of more than 1e-7)
detection1em7 = function(x, ...){
  xExceed = x > 1e-7
```

```
  out = sum(xExceed)
  return(out)
}
plumeNr_radioactivePlumes =
  summaryLocations(radioactivePlumes,
                   plumes = plumeSample, fun = detection1em7, kinds = 2)
## plot map
plumeNrMap = radioactivePlumes@locations
plumeNrMap@data$plumeNr = plumeNr_radioactivePlumes[["summaryLocations"]]
spplot(plumeNrMap, zcol = "plumeNr")
```

---

summaryPlumes *Summarise values of Simulations by plumes*

Description

This function is similar to apply for the values of Simulations plume-wise. It takes a function and returns a vector with a value for each plume: the result of the function applied to all values belonging to this plume. As values may be too big to keep in memory, the function is applied chunk-wise; therefore it only works for associative functions. A (slower) method without this restriction is It is combined with a global summary function and thus provides a general basic form of a cost function.

Usage

```
summaryPlumes(simulations,
  locations = 1:nLocations(simulations),
  plumes = 1:nPlumes(simulations),
  kinds = 1,
  fun, summaryFun = weightedMean,
  weight = 1, na.rm = FALSE, ...)
weightedMean(x, weight, na.rm = FALSE)
```

Arguments

simulations	Simulations
locations	indices of locations to be taken into account (to apply the function to a subset of simulations); for invalid indices NA values are used; multiple are ignored
plumes	indices of plumes to be taken into account; for invalid indices NA values are used
kinds	index of kind of values to be taken into account; only one kind can be used, all but the first given index are ignored
fun	function to be applied; it must have a single parameter (vector), be associative, i.e. fun(c(a,b)) = fun(c(fun(a),b)), work also on vectors of length 0 and return a single value; in addition it needs a parameter na.rm or ...; possible choices are sum, prod, max, min
summaryFun	function to summarise the plume-wise results (as first parameter) to a global value; by default it is a weighted mean; in addition it can have the parameters weight, or use ...

weight	to be used by summaryFun: either a numeric value of length 1 or same length as plumes or a character indicating a column of the data associated with the plumes of simulations
na.rm	logical how to treat missing values by fun (has no influence on summaryFun)
...	further arguments to be forwarded to fun or summaryFun
x	vector

Details

To use a different summaryFun you may use replaceDefault with kind = "summaryFun.summaryPlumes" to check the function beforehand or to generate it by resetting the default parameters of an existing function.

Value

A list of

summary	the result of summaryFun
summaryPlumes	the plume-wise results of fun, one value for each selected plume

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(radioactivePlumes)
# sample of locations, e.g. proposed sensor set
locSample = sample.int(nLocations(radioactivePlumes), 10)

# plume detection
# (number of plumes not detected at any of locSample at a level of more than 1e-7)
plumeMin_radioactivePlumes =
  summaryPlumes(radioactivePlumes, locations = locSample, kinds = 2,
  fun = "min",
  summaryFun = function(x, weight = 1, na.rm = TRUE){sum(x < 1e-7)})
```

---

testDataArtificial *Small artificial test data*

Description

Small artificial test data sets for demonstration, taken from the examples of the respective classes.

Usage

```
data(SIndexDF)
data(SPointsDF)
data(SPixelsDF)
data(SPolygridDF)
data(SPolygonsDF)
data(SLinesDF)
```

Format

SIndexDF: SpatialIndexDataFrame
SPointsDF: SpatialPointsDataFrame
SPixelsDF: SpatialPixelsDataFrame
SPolygridDF: SpatialPolygridDataFrame
SPolygonsDF: SpatialPolygonsDataFrame
SLinesDF: SpatialLinesDataFrame

Author(s)

Kristina B. Helle, <kristina.helle@uni-muenster.de>

Examples

```
data(SPolygridDF)

spplot(SPolygridDF, zcol = "b",
       sp.layout = list("sp.points", SpatialPoints(coordinates(SPolygridDF))))

data(SIndexDF)
spplot(SIndexDF)

data(SLinesDF)
areaSDF(SLinesDF)
```

Index