Package 'spsann'

October 26, 2015

Type Package

Title Optimization of Sample Configurations using Spatial Simulated Annealing

Version 1.0-2

Date 2015-10-25

Description Methods to optimize sample configurations using spatial simulated annealing. Multiple objective functions are implemented for various purposes, such as variogram estimation, trend estimation, and spatial interpolation. A general purpose spatial simulated annealing function enables the user to define his/her own objective function.

License GPL (>= 2)

Imports pedometrics, Rcpp (>= 0.11.3), sp, SpatialTools

Suggests gstat

LinkingTo Rcpp

Encoding UTF-8

NeedsCompilation yes

Author Alessandro Samuel-Rosa [aut, cre],
Lucia Helena Cunha dos Anjos [ths],
Gustavo de Mattos Vasques [ths],
Gerard B M Heuvelink [ths],
Edzer Pebesma [ctb],
Jon Skoien [ctb],
Joshua French [ctb],
Pierre Roudier [ctb],
Dick Brus [ctb],
Murray Lark [ctb]

Maintainer Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

Repository CRAN

Date/Publication 2015-10-26 00:56:39

R topics documented:

`spsann-package` *spsann: Optimization of Sample Configurations using Spatial Simulated Annealing*

Description

Methods to optimize sample configurations using spatial simulated annealing. Multiple objective functions are implemented for various purposes, such as variogram estimation, trend estimation, and spatial interpolation. A general purpose spatial simulated annealing function enables the user to define his/her own objective function.

Details

spsann is the R package for the optimization of sample configurations using spatial simulated annealing. It includes multiple functions with different objective functions to optimize sample configurations for variogram estimation (number of points or point-pairs per lag distance class), trend estimation (association/correlation and marginal distribution of the covariates), and spatial interpolation (mean squared shortest distance). **spsann** also includes objective functions that can be used when the model of spatial variation is known (mean (or maximum) kriging variance). Some objective functions were combined to optimize sample configurations when we are ignorant about the model of spatial variation (*terra incognita*). A general purpose function enables to user to define his/her own objective function and plug it in the spatial simulated annealing algorithm.

Spatial simulated annealing is a well known method with widespread use to solve optimization problems in the environmental sciences. This is mainly due to its robustness against local optima. At each iteration, the algorithm evaluates if a worsening solution can be accepted. The chance of accepting worsening solutions reduces as the number of iterations increases.

Package:	spsann
Type:	Package
Version:	1.0-1.9000
Date:	2015-10-25

License: GPL (>= 2)

Support

spsann was initially developed as part of the PhD research project entitled 'Contribution to the Construction of Models for Predicting Soil Properties', developed by Alessandro Samuel-Rosa under the supervision of Lúcia Helena Cunha dos Anjos (Universidade Federal Rural do Rio de Janeiro, Brazil), Gustavo de Mattos Vasques (Embrapa Solos, Brazil), and Gerard B. M. Heuvelink (ISRIC - World Soil Information, the Netherlands). The project was/is supported from 2012 to 2016 by the CAPES Foundation, Ministry of Education of Brazil, and the CNPq Foundation, Ministry of Science and Technology of Brazil.

Contributors

Some of the solutions used to build **spsann** were found in the source code of other R-packages. The skeleton of the optimization functions was adopted from the **intamapInteractive** package, authored by Edzer Pebesma <edzer.pebesma@uni-muenster.de> and Jon Skoien <jon.skoien@gmail.com>.

A few small solutions were adopted from the **SpatialTools** package, authored by Joshua French <joshua.french@ucdenver.edu>, and **clhs** package, authored by Pierre Roudier <roudierp@landcareresearch.co.nz>.

Conceptual contributions were made by Gerard Heuvelink <gerard.heuvelink@wur.nl>, Dick Brus <dick.brus@wur.nl>, Murray Lark <mlark@bgs.ac.uk>, and Edzer Pebesma <edzer.pebesma@uni-muenster.de>.

Author(s)

Author and Maintainer: Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

`objSPSANN` *Auxiliary tools*

Description

Auxiliary tools used in the optimization of sample configurations using spatial simulated annealing.

Usage

```
objSPSANN(OSC, at = "end", n = 1)
```

Arguments

`OSC`	Optimized Sample Configuration.
`at`	Point of the optimization at which the energy state should be returned. Available options: `"start"`, for the start, and `"end"`, for the end of the optimization. Defaults to `at = "end"`.
`n`	Number of instances that should be returned. Defaults to `n = 1`.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

`optimACDC` *Optimization of sample configurations for spatial trend identification and estimation (III)*

Description

Optimize a sample configuration for spatial trend identification and estimation. An utility function U is defined so that the sample reproduces the bivariate association/correlation between the covariates, as well as their marginal distribution (**ACDC**). The utility function is obtained aggregating two single objective functions: **CORR** and **DIST**.

Usage

```
optimACDC(points, candi, iterations = 100, covars, strata.type = "area",
  use.coords = FALSE, x.max, x.min, y.max, y.min, acceptance = list(initial
  = 0.99, cooling = iterations/10), stopping = list(max.count =
  iterations/10), plotit = FALSE, track = FALSE, boundary,
  progress = TRUE, verbose = FALSE, greedy = FALSE, weights = list(CORR
  = 0.5, DIST = 0.5), nadir = list(sim = NULL, seeds = NULL, user = NULL, abs
  = NULL), utopia = list(user = NULL, abs = NULL))

objACDC(points, candi, covars, strata.type = "area", use.coords = FALSE,
  weights = list(CORR = 0.5, DIST = 0.5), nadir = list(sim = NULL, seeds =
  NULL, user = NULL, abs = NULL), utopia = list(user = NULL, abs = NULL))
```

Arguments

`points`	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
`candi`	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
`iterations`	Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.
`covars`	Data frame or matrix with the covariates in the columns.

`strata.type` Character value setting the type of stratification that should be used to create the marginal sampling strata (or factor levels) for the numeric covariates. Available options are `"area"`, for equal-area, and `"range"`, for equal-range. Defaults to `strata.type = "area"`.

`use.coords` Logical value. Should the geographic coordinates be used as covariates? Defaults to `use.coords = FALSE`.

`x.max,x.min,y.max,y.min`
Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.

`acceptance` List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.

`stopping` List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.

`plotit` Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.

`track` Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.

`boundary` SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from `candi`.

`progress` Logical for printing a progress bar. Defaults to `progress = TRUE`.

`verbose` Logical for printing messages about the progress of the optimization. Defaults to `verbose = FALSE`.

`greedy` Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to `greedy = FALSE`. (experimental)

`weights` List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.

`nadir` List with named sub-arguments. Three options are available: 1) `sim` – the number of simulations that should be used to estimate the nadir point, and `seeds` – vector defining the random seeds for each simulation; 2) `user` – a list of user-defined nadir values named after the respective objective function to which they apply; 3) `abs` – logical for calculating the nadir point internally (experimental).

`utopia` List with named sub-arguments. Two options are available: 1) `user` – a list of user-defined values named after the respective objective function to which they apply; 2) `abs` – logical for calculating the utopia point internally (experimental).

Value

`optimACDC` returns a matrix: the optimized sample configuration.

`objACDC` returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Association/Correlation between covariates

The *correlation* between two numeric covariates is measured using the Pearson's r, a descriptive statistic that ranges from -1 to $+1$. This statistic is also known as the linear correlation coefficient.

When the set of covariates includes factor covariates, all numeric covariates are transformed into factor covariates. The factor levels are defined using the marginal sampling strata created from one of the two methods available (equal-area or equal-range strata).

The *association* between two factor covariates is measured using the Cramér's v, a descriptive statistic that ranges from 0 to $+1$. The closer to $+1$ the Cramér's v is, the stronger the association between two factor covariates. The main weakness of using the Cramér's v is that, while the Pearson's r shows the degree and direction of the association between two covariates (negative or positive), the Cramér's v only measures the degree of association (weak or strong).

Marginal distribution of covariates

Reproducing the marginal distribution of the numeric covariates depends upon the definition of marginal sampling strata. These marginal sampling strata are also used to define the factor levels of all numeric covariates that are passed together with factor covariates.

Two types of marginal sampling strata can be used. *Equal-area* sampling strata are defined using the sample quantiles estimated with `quantile` using a discontinuous function (`type = 3`). This is to avoid creating breakpoints that do not occur in the population of existing covariate values.

The function `quantile` commonly produces repeated break points. A break point will always be repeated if that value has a relatively high frequency in the population of covariate values. The number of repeated break points increases with the number of marginal sampling strata. Only unique break points are used to create marginal sampling strata.

Equal-range sampling strata are defined by breaking the range of covariate values into pieces of equal size. This method usually creates break points that do not occur in the population of existing covariate values. Such break points are replaced by the nearest existing covariate value identified using Euclidean distances.

Both stratification methods can produce marginal sampling strata that cover a range of values that do not exist in the population of covariate values. Any empty marginal sampling strata is merged with the closest non-empty marginal sampling strata. These are identified using Euclidean distances.

The approaches used to define the marginal sampling strata result in each numeric covariate having a different number of marginal sampling strata, some of them with different area/size. Because the goal is to have a sample that reproduces the marginal distribution of the covariate, each marginal sampling strata will have a different number of sample points. The wanted distribution of the number of sample points per marginal strata is estimated empirically as the proportion of points in the population of existing covariate values that fall in each marginal sampling strata.

Note

This function was derived with modifications from the method known as the *conditioned Latin Hypercube sampling* originally proposed by Minasny and McBratney (2006), and implemented in the R-package **clhs** by Pierre Roudier.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Cramér, H. *Mathematical methods of statistics*. Princeton: Princeton University Press, p. 575, 1946.

Everitt, B. S. *The Cambridge dictionary of statistics*. Cambridge: Cambridge University Press, p. 432, 2006.

Hyndman, R. J.; Fan, Y. Sample quantiles in statistical packages. *The American Statistician*, v. 50, p. 361-365, 1996.

Minasny, B.; McBratney, A. B. A conditioned Latin hypercube method for sampling in the presence of ancillary information. *Computers & Geosciences*, v. 32, p. 1378-1388, 2006.

Minasny, B.; McBratney, A. B. Conditioned Latin Hypercube Sampling for calibrating soil sensor data to soil properties. Chapter 9. Viscarra Rossel, R. A.; McBratney, A. B.; Minasny, B. (Eds.) *Proximal Soil Sensing*. Amsterdam: Springer, p. 111-119, 2010.

Mulder, V. L.; de Bruin, S.; Schaepman, M. E. Representing major soil variability at regional scale by constrained Latin hypercube sampling of remote sensing data. *International Journal of Applied Earth Observation and Geoinformation*, v. 21, p. 301-310, 2013.

Roudier, P.; Beaudette, D.; Hewitt, A. A conditioned Latin hypercube sampling algorithm incorporating operational constraints. *5th Global Workshop on Digital Soil Mapping*. Sydney, p. 227-231, 2012.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

See Also

`clhs`, `cramer`

Examples

```
## Not run:
# This example takes more than 5 seconds to run!
require(sp)
```

```
data(meuse.grid)
candi <- meuse.grid[, 1:2]
nadir <- list(sim = 10, seeds = 1:10)
utopia <- list(user = list(DIST = 0, CORR = 0))
covars <- meuse.grid[, 5]
set.seed(2001)
res <- optimACDC(points = 100, candi = candi, covars = covars,
                 nadir = nadir, use.coords = TRUE, utopia = utopia)
objSPSANN(res) - # 0.5272031
  objACDC(points = res, candi = candi, covars = covars, use.coords = TRUE,
          nadir = nadir, utopia = utopia)
# MARGINAL DISTRIBUTION
par(mfrow = c(3, 3))
# Covariates
i <- sample(1:nrow(candi), 100)
hist(candi[, 1], breaks = 10)
hist(candi[, 2], breaks = 10)
hist(covars, breaks = 10)
# Optimized sample
hist(candi[res[, 1], 1], breaks = 10)
hist(candi[res[, 1], 2], breaks = 10)
hist(covars[res[, 1]], breaks = 10)
# Random sample
hist(candi[i, 1], breaks = 10)
hist(candi[i, 2], breaks = 10)
hist(covars[i], breaks = 10)

# LINEAR CORRELATION
# Covariates
cor(cbind(candi[, 1], candi[, 2], covars))
# Optimized sample
cor(cbind(candi[res[, 1], 1], candi[res[, 1], 2], covars[res[, 1]]))
# Random sample
cor(cbind(candi[i, 1], candi[i, 2], covars[i]))

## End(Not run)
```

---

optimCLHS *Optimization of sample configurations for spatial trend identification and estimation (IV)*

Description

Optimize a sample configuration for spatial trend identification and estimation using the method proposed by Minasny and McBratney (2006), known as the conditioned Latin hypercube sampling. An utility function U is defined so that the sample reproduces the marginal distribution and correlation matrix of the numeric covariates, and the class proportions of the factor covariates (**CLHS**). The utility function is obtained aggregating three objective functions: **O1**, **O2**, and **O3**.

Usage

```
optimCLHS(points, candi, iterations = 50000, covars, use.coords = FALSE,
  x.max, x.min, y.max, y.min, acceptance = list(initial = 0.8, cooling =
  iterations/10), stopping = list(max.count = iterations/10),
  plotit = FALSE, track = FALSE, boundary, progress = TRUE,
  verbose = FALSE, greedy = FALSE, weights = list(O1 = 1/3, O2 = 1/3, O3 =
  1/3))

objCLHS(points, candi, covars, use.coords = FALSE, weights = list(O1 = 1/3,
  O2 = 1/3, O3 = 1/3))
```

Arguments

`points`	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
`candi`	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
`iterations`	Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.
`covars`	Data frame or matrix with the covariates in the columns.
`use.coords`	Logical value. Should the geographic coordinates be used as covariates? Defaults to `use.coords = FALSE`.
`x.max,x.min,y.max,y.min`	Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.
`acceptance`	List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.
`stopping`	List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.
`plotit`	Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in

	the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.
`track`	Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.
`boundary`	SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from `candi`.
`progress`	Logical for printing a progress bar. Defaults to `progress = TRUE`.
`verbose`	Logical for printing messages about the progress of the optimization. Defaults to `verbose = FALSE`.
`greedy`	Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to `greedy = FALSE`. (experimental)
`weights`	List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.

Value

`optimCLHS` returns a matrix: the optimized sample configuration.

`objCLHS` returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are

placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Marginal sampling strata

Reproducing the marginal distribution of the numeric covariates depends upon the definition of marginal sampling strata. *Equal-area* marginal sampling strata are defined using the sample quantiles estimated with `quantile` using a continuous function (`type = 7`), that is, a function that interpolates between existing covariate values to estimate the sample quantiles – this is the procedure implemented in the method of Minasny and McBratney (2006), which creates breakpoints that do not occur in the population of existing covariate values. Depending on the level of discretization of the covariate values, that is, how many significant digits they have, this can create repeated breakpoints, resulting in empty marginal sampling strata. The number of empty marginal sampling strata will ultimately depend on the frequency distribution of the covariate, and on the number of sampling points.

Correlation between numeric covariates

The *correlation* between two numeric covariates is measured using the sample Pearson's *r*, a descriptive statistic that ranges from -1 to $+1$. This statistic is also known as the sample linear correlation coefficient.

Multi-objective optimization

A method of solving a multi-objective optimization problem (MOOP) is to aggregate the objective functions into a single *utility function U*. In the **spsann** package, as in the original CLHS, the aggregation is performed using the *weighted sum method*, which uses weights to incorporate the preferences of the user about the relative importance of each objective function. When the user has no preference, the objective functions receive equal weights.

The weighted sum method is affected by the relative magnitude of the different objective function values. The objective functions implemented in `optimCLHS` have different units and orders of magnitude. The consequence is that the objective function with the largest values, generally **O1**, may have a numerical dominance during the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear – the optimization will favour the objective function which is numerically dominant.

An efficient solution to avoid numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values, at least in the end of the optimization. However, as in the original CLHS, `optimCLHS` uses the naive aggregation method, which ignores that the three objective functions have different units and orders of magnitude. The same aggregation procedure is implemented in the **clhs** package.

Note

The (only) difference of the `optimCLHS` function to the original Fortran implementation of Minasny and McBratney (2006), and to the `clhs` function implemented in the **clhs** package by Pierre Roudier, is in the annealing schedule.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Minasny, B.; McBratney, A. B. A conditioned Latin hypercube method for sampling in the presence of ancillary information. *Computers & Geosciences*, v. 32, p. 1378-1388, 2006.

Minasny, B.; McBratney, A. B. Conditioned Latin Hypercube Sampling for calibrating soil sensor data to soil properties. Chapter 9. Viscarra Rossel, R. A.; McBratney, A. B.; Minasny, B. (Eds.) *Proximal Soil Sensing*. Amsterdam: Springer, p. 111-119, 2010.

Roudier, P.; Beaudette, D.; Hewitt, A. A conditioned Latin hypercube sampling algorithm incorporating operational constraints. *5th Global Workshop on Digital Soil Mapping*. Sydney, p. 227-231, 2012.

See Also

`clhs`, `optimACDC`

Examples

```
## Not run:
require(sp)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
covars <- meuse.grid[, 5]
weights <- list(O1 = 0.5, O3 = 0.5)
set.seed(2001)
res <- optimCLHS(points = 100, candi = candi, covars = covars,
                 use.coords = TRUE, weights = weights, iterations = 100)
objSPSANN(res) - # 106.0691
```

```
objCLHS(points = res, candi = candi, covars = covars, use.coords = TRUE,
        weights = weights)

# MARGINAL DISTRIBUTION
par(mfrow = c(3, 3))
# Covariates
i <- sample(1:nrow(candi), 100)
hist(candi[, 1], breaks = 10)
hist(candi[, 2], breaks = 10)
hist(covars, breaks = 10)
# Optimized sample
hist(candi[res[, 1], 1], breaks = 10)
hist(candi[res[, 1], 2], breaks = 10)
hist(covars[res[, 1]], breaks = 10)
# Random sample
hist(candi[i, 1], breaks = 10)
hist(candi[i, 2], breaks = 10)
hist(covars[i], breaks = 10)

# LINEAR CORRELATION
# Covariates
cor(cbind(candi[, 1], candi[, 2], covars))
# Optimized sample
cor(cbind(candi[res[, 1], 1], candi[res[, 1], 2], covars[res[, 1]]))
# Random sample
cor(cbind(candi[i, 1], candi[i, 2], covars[i]))

## End(Not run)
```

---

optimCORR *Optimization of sample configurations for spatial trend identification and estimation (I)*

Description

Optimize a sample configuration for spatial trend identification and estimation. A criterion is defined so that the sample reproduces the bivariate association/correlation between the covariates (**CORR**).

Usage

```
optimCORR(points, candi, iterations = 100, covars, strata.type = "area",
  use.coords = FALSE, x.max, x.min, y.max, y.min, acceptance = list(initial
  = 0.99, cooling = iterations/10), stopping = list(max.count =
  iterations/10), plotit = FALSE, track = FALSE, boundary,
  progress = TRUE, verbose = FALSE, greedy = FALSE, weights = NULL,
  nadir = NULL, utopia = NULL)

objCORR(points, candi, covars, strata.type = "area", use.coords = FALSE)
```

Arguments

points	Integer value, integer vector, data frame or matrix. If points is an integer value, it defines the number of points that should be randomly sampled from candi to form the starting system configuration. If points is a vector of integer values, it contains the row indexes of candi that correspond to the points that form the starting system configuration. If points is a data frame or matrix, it must have three columns in the following order: [, "id"] the row indexes of candi that correspond to each point, [, "x"] the projected x-coordinates, and [, "y"] the projected y-coordinates. Note that in the later case, points must be a subset of candi.
candi	Data frame or matrix with the candidate locations for the perturbed points. candi must have two columns in the following order: [, "x"] the projected x-coordinates, and [, "y"] the projected y-coordinates.
iterations	Integer. The maximum number of iterations that should be used for the optimization. Defaults to iterations = 100.
covars	Data frame or matrix with the covariates in the columns.
strata.type	Character value setting the type of stratification that should be used to create the marginal sampling strata (or factor levels) for the numeric covariates. Available options are "area", for equal-area, and "range", for equal-range. Defaults to strata.type = "area".
use.coords	Logical value. Should the geographic coordinates be used as covariates? Defaults to use.coords = FALSE.
x.max,x.min,y.max,y.min	Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from candi.
acceptance	List with two named sub-arguments: initial – numeric value between 0 and 1 defining the initial acceptance probability, and cooling – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to acceptance = list(initial = 0.99, cooling = iterations / 10).
stopping	List with one named sub-argument: max.count – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to stopping = list(max.count = iterations / 10).
plotit	Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to plotit = FALSE.
track	Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If track = FALSE (the default), only the starting and ending energy state values are returned with the result.
boundary	SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from candi.

progress	Logical for printing a progress bar. Defaults to progress = TRUE.
verbose	Logical for printing messages about the progress of the optimization. Defaults to verbose = FALSE.
greedy	Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to greedy = FALSE. (experimental)
weights	List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.
nadir	List with named sub-arguments. Three options are available: 1) sim – the number of simulations that should be used to estimate the nadir point, and seeds – vector defining the random seeds for each simulation; 2) user – a list of user-defined nadir values named after the respective objective function to which they apply; 3) abs – logical for calculating the nadir point internally (experimental).
utopia	List with named sub-arguments. Two options are available: 1) user – a list of user-defined values named after the respective objective function to which they apply; 2) abs – logical for calculating the utopia point internally (experimental).

Value

optimCORR returns a matrix: the optimized sample configuration.

objCORR returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in points. Because the more points there are in points, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in spsample. Because the candidate locations are

placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Association/Correlation between covariates

The *correlation* between two numeric covariates is measured using the Pearson's r, a descriptive statistic that ranges from -1 to $+1$. This statistic is also known as the linear correlation coefficient.

When the set of covariates includes factor covariates, all numeric covariates are transformed into factor covariates. The factor levels are defined using the marginal sampling strata created from one of the two methods available (equal-area or equal-range strata).

The *association* between two factor covariates is measured using the Cramér's v, a descriptive statistic that ranges from 0 to $+1$. The closer to $+1$ the Cramér's v is, the stronger the association between two factor covariates. The main weakness of using the Cramér's v is that, while the Pearson's r shows the degree and direction of the association between two covariates (negative or positive), the Cramér's v only measures the degree of association (weak or strong).

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Cramér, H. *Mathematical methods of statistics*. Princeton: Princeton University Press, p. 575, 1946.

Everitt, B. S. *The Cambridge dictionary of statistics*. Cambridge: Cambridge University Press, p. 432, 2006.

Hyndman, R. J.; Fan, Y. Sample quantiles in statistical packages. *The American Statistician*, v. 50, p. 361-365, 1996.

Minasny, B.; McBratney, A. B. A conditioned Latin hypercube method for sampling in the presence of ancillary information. *Computers & Geosciences*, v. 32, p. 1378-1388, 2006.

Minasny, B.; McBratney, A. B. Conditioned Latin Hypercube Sampling for calibrating soil sensor data to soil properties. Chapter 9. Viscarra Rossel, R. A.; McBratney, A. B.; Minasny, B. (Eds.) *Proximal Soil Sensing*. Amsterdam: Springer, p. 111-119, 2010.

Mulder, V. L.; de Bruin, S.; Schaepman, M. E. Representing major soil variability at regional scale by constrained Latin hypercube sampling of remote sensing data. *International Journal of Applied Earth Observation and Geoinformation*, v. 21, p. 301-310, 2013.

Roudier, P.; Beaudette, D.; Hewitt, A. A conditioned Latin hypercube sampling algorithm incorporating operational constraints. *5th Global Workshop on Digital Soil Mapping*. Sydney, p. 227-231, 2012.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

See Also

clhs, cramer

Examples

```
require(sp)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
covars <- meuse.grid[, 5]
set.seed(2001)
## Not run:
# This example takes more than 5 seconds to run!
res <- optimCORR(points = 100, candi = candi, covars = covars,
                 use.coords = TRUE)
objSPSANN(res) # 0.06386069
objCORR(points = res, candi = candi, covars = covars, use.coords = TRUE)

## End(Not run)
# Random sample
pts <- sample(1:nrow(candi), 5)
pts <- cbind(pts, candi[pts, ])
objCORR(points = pts, candi = candi, covars = covars, use.coords = TRUE)
```

optimDIST *Optimization of sample configurations for spatial trend identification and estimation (II)*

Description

Optimize a sample configuration for spatial trend identification and estimation. A criterion is defined so that the sample reproduces the marginal distribution of the covariates (**DIST**).

Usage

```
optimDIST(points, candi, iterations = 100, covars, strata.type = "area",
  use.coords = FALSE, x.max, x.min, y.max, y.min, acceptance = list(initial
  = 0.99, cooling = iterations/10), stopping = list(max.count =
  iterations/10), plotit = FALSE, track = FALSE, boundary,
  progress = TRUE, verbose = FALSE, greedy = FALSE, weights = NULL,
  nadir = NULL, utopia = NULL)

objDIST(points, candi, covars, strata.type = "area", use.coords = FALSE)
```

Arguments

points	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
candi	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
iterations	Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.
covars	Data frame or matrix with the covariates in the columns.
strata.type	Character value setting the type of stratification that should be used to create the marginal sampling strata (or factor levels) for the numeric covariates. Available options are `"area"`, for equal-area, and `"range"`, for equal-range. Defaults to `strata.type = "area"`.
use.coords	Logical value. Should the geographic coordinates be used as covariates? Defaults to `use.coords = FALSE`.

`x.max,x.min,y.max,y.min`
: Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.

`acceptance`
: List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.

`stopping`
: List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.

`plotit`
: Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.

`track`
: Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.

`boundary`
: SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from `candi`.

`progress`
: Logical for printing a progress bar. Defaults to `progress = TRUE`.

`verbose`
: Logical for printing messages about the progress of the optimization. Defaults to `verbose = FALSE`.

`greedy`
: Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to `greedy = FALSE`. (experimental)

`weights`
: List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.

`nadir`
: List with named sub-arguments. Three options are available: 1) `sim` – the number of simulations that should be used to estimate the nadir point, and `seeds` – vector defining the random seeds for each simulation; 2) `user` – a list of user-defined nadir values named after the respective objective function to which they apply; 3) `abs` – logical for calculating the nadir point internally (experimental).

`utopia`
: List with named sub-arguments. Two options are available: 1) `user` – a list of user-defined values named after the respective objective function to which they apply; 2) `abs` – logical for calculating the utopia point internally (experimental).

Value

`optimDIST` returns a matrix: the optimized sample configuration.

objDIST returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in points. Because the more points there are in points, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in spsample. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of mag-

nitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Marginal distribution of covariates

Reproducing the marginal distribution of the numeric covariates depends upon the definition of marginal sampling strata. These marginal sampling strata are also used to define the factor levels of all numeric covariates that are passed together with factor covariates.

Two types of marginal sampling strata can be used. *Equal-area* sampling strata are defined using the sample quantiles estimated with `quantile` using a discontinuous function (`type = 3`). This is to avoid creating breakpoints that do not occur in the population of existing covariate values.

The function quantile commonly produces repeated break points. A break point will always be repeated if that value has a relatively high frequency in the population of covariate values. The number of repeated break points increases with the number of marginal sampling strata. Only unique break points are used to create marginal sampling strata.

Equal-range sampling strata are defined by breaking the range of covariate values into pieces of equal size. This method usually creates break points that do not occur in the population of existing covariate values. Such break points are replaced by the nearest existing covariate value identified using Euclidean distances.

Both stratification methods can produce marginal sampling strata that cover a range of values that do not exist in the population of covariate values. Any empty marginal sampling strata is merged with the closest non-empty marginal sampling strata. These are identified using Euclidean distances.

The approaches used to define the marginal sampling strata result in each numeric covariate having a different number of marginal sampling strata, some of them with different area/size. Because the goal is to have a sample that reproduces the marginal distribution of the covariate, each marginal sampling strata will have a different number of sample points. The wanted distribution of the number of sample points per marginal strata is estimated empirically as the proportion of points in the population of existing covariate values that fall in each marginal sampling strata.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* intamap *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Cramér, H. *Mathematical methods of statistics*. Princeton: Princeton University Press, p. 575, 1946.

Everitt, B. S. *The Cambridge dictionary of statistics*. Cambridge: Cambridge University Press, p. 432, 2006.

Hyndman, R. J.; Fan, Y. Sample quantiles in statistical packages. *The American Statistician*, v. 50, p. 361-365, 1996.

Minasny, B.; McBratney, A. B. A conditioned Latin hypercube method for sampling in the presence of ancillary information. *Computers & Geosciences*, v. 32, p. 1378-1388, 2006.

Minasny, B.; McBratney, A. B. Conditioned Latin Hypercube Sampling for calibrating soil sensor data to soil properties. Chapter 9. Viscarra Rossel, R. A.; McBratney, A. B.; Minasny, B. (Eds.) *Proximal Soil Sensing*. Amsterdam: Springer, p. 111-119, 2010.

Mulder, V. L.; de Bruin, S.; Schaepman, M. E. Representing major soil variability at regional scale by constrained Latin hypercube sampling of remote sensing data. *International Journal of Applied Earth Observation and Geoinformation*, v. 21, p. 301-310, 2013.

Roudier, P.; Beaudette, D.; Hewitt, A. A conditioned Latin hypercube sampling algorithm incorporating operational constraints. *5th Global Workshop on Digital Soil Mapping*. Sydney, p. 227-231, 2012.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

See Also

clhs

Examples

```
require(sp)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
covars <- meuse.grid[, 5]
set.seed(2001)
## Not run:
# This example takes more than 5 seconds to run!
res <- optimDIST(points = 100, candi = candi, covars = covars,
                 use.coords = TRUE)
objSPSANN(res) # 1.6505
objDIST(points = res, candi = candi, covars = covars, use.coords = TRUE)

## End(Not run)
# Random sample
pts <- sample(1:nrow(candi), 5)
pts <- cbind(pts, candi[pts, ])
objDIST(points = pts, candi = candi, covars = covars, use.coords = TRUE)
```

---

optimMKV *Optimization of sample configurations for spatial interpolation*

Description

Optimize a sample configuration for spatial interpolation with a known linear model. A criterion is defined so that the sample configuration minimizes the mean/maximum kriging variance (**MKV**).

Usage

```
optimMKV(points, candi, iterations = 100, covars, eqn = z ~ 1, vgm,
  krige.stat = "mean", ..., x.max, x.min, y.max, y.min,
  acceptance = list(initial = 0.99, cooling = iterations/10),
  stopping = list(max.count = iterations/10), plotit = FALSE,
  track = FALSE, boundary, progress = TRUE, verbose = FALSE,
  greedy = FALSE, weights = NULL, nadir = NULL, utopia = NULL)

objMKV(points, candi, covars, eqn = z ~ 1, vgm, krige.stat = "mean", ...)
```

Arguments

points	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
candi	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
iterations	Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.
covars	Data frame or matrix with the covariates in the columns.
eqn	Formula string that defines the dependent variable `z` as a linear model of the independent variables contained in `covars`. Defaults to `eqn = z ~ 1`, that is, ordinary kriging. See the argument `formula` in the function `krige` for more information.
vgm	Object of class "variogramModel". See the argument `model` in the function `krige` for more information.
krige.stat	Character value defining the statistic that should be used to summarize the kriging variance. Available options are `"mean"` and `"max"` for the mean and maximum kriging variance, respectively. Defaults to `krige.stat = "mean"`.
...	further arguments passed to `krige`.
x.max,x.min,y.max,y.min	Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.
acceptance	List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.

`stopping` List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.

`plotit` Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.

`track` Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.

`boundary` SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from `candi`.

`progress` Logical for printing a progress bar. Defaults to `progress = TRUE`.

`verbose` Logical for printing messages about the progress of the optimization. Defaults to `verbose = FALSE`.

`greedy` Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to `greedy = FALSE`. (experimental)

`weights` List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.

`nadir` List with named sub-arguments. Three options are available: 1) `sim` – the number of simulations that should be used to estimate the nadir point, and `seeds` – vector defining the random seeds for each simulation; 2) `user` – a list of user-defined nadir values named after the respective objective function to which they apply; 3) `abs` – logical for calculating the nadir point internally (experimental).

`utopia` List with named sub-arguments. Two options are available: 1) `user` – a list of user-defined values named after the respective objective function to which they apply; 2) `abs` – logical for calculating the utopia point internally (experimental).

Value

`optimMKV` returns a matrix: the optimized sample configuration.

`objMKV` returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the

spsann package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Note

This function is based on the method originally proposed by Heuvelink, Brus and de Gruijter (2006) and implemented in the R-package **intamapInteractive** by Edzer Pebesma and Jon Skoien.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

Brus, D. J. & Heuvelink, G. B. M. Optimization of sample patterns for universal kriging of environmental variables. *Geoderma*. v. 138, p. 86-95, 2007.

Heuvelink, G. B. M.; Brus, D. J. & de Gruijter, J. J. Optimization of sample configurations for digital mapping of soil properties with universal kriging. In: Lagacherie, P.; McBratney, A. & Voltz, M. (Eds.) *Digital soil mapping - an introductory perspective*. Elsevier, v. 31, p. 137-151, 2006.

Examples

```
## Not run:
# This example takes more than 5 seconds to run!
require(sp)
require(gstat)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
covars <- as.data.frame(meuse.grid)
vgm <- vgm(psill = 10, model = "Exp", range = 500, nugget = 8)
set.seed(2001)
res <- optimMKV(points = 100, candi = candi, covars = covars, maxdist = 500,
                eqn = z ~ dist, vgm = vgm)
objSPSANN(res) # 11.9878
objMKV(points = res, candi = candi, covars = covars, eqn = z ~ dist,
       vgm = vgm, maxdist = 500)

## End(Not run)
```

`optimMSSD` *Optimization of sample configurations for spatial interpolation*

Description

Optimize a sample configuration for spatial interpolation. The criterion used is the mean squared shortest distance (**MSSD**).

Usage

```
optimMSSD(points, candi, iterations = 100, x.max, x.min, y.max, y.min,
  acceptance = list(initial = 0.99, cooling = iterations/10),
  stopping = list(max.count = iterations/10), plotit = FALSE,
  track = FALSE, boundary, progress = TRUE, verbose = FALSE,
```

```
  greedy = FALSE, weights = NULL, nadir = NULL, utopia = NULL)

objMSSD(points, candi)
```

Arguments

points	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
candi	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
iterations	Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.
x.max,x.min,y.max,y.min	Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.
acceptance	List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.
stopping	List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.
plotit	Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.
track	Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.
boundary	SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from `candi`.
progress	Logical for printing a progress bar. Defaults to `progress = TRUE`.
verbose	Logical for printing messages about the progress of the optimization. Defaults to `verbose = FALSE`.

`greedy` Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to `greedy = FALSE`. (experimental)

`weights` List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.

`nadir` List with named sub-arguments. Three options are available: 1) `sim` – the number of simulations that should be used to estimate the nadir point, and `seeds` – vector defining the random seeds for each simulation; 2) `user` – a list of user-defined nadir values named after the respective objective function to which they apply; 3) `abs` – logical for calculating the nadir point internally (experimental).

`utopia` List with named sub-arguments. Two options are available: 1) `user` – a list of user-defined values named after the respective objective function to which they apply; 2) `abs` – logical for calculating the utopia point internally (experimental).

Value

`optimMSSD` returns a matrix: the optimized sample configuration.

`objMSSD` returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the

point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Note

This function was derived with modifications from the method known as the *spatial coverage sampling* originally proposed by Brus, de Gruijter and van Groenigen (2006), and implemented in the R-package **spcosa** by Dennis Walvoort, Dick Brus and Jaap de Gruijter.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

Brus, D. J.; de Gruijter, J. J.; van Groenigen, J. W. Designing spatial coverage samples using the k-means clustering algorithm. In: P. Lagacherie, A. M.; Voltz, M. (Eds.) *Digital soil mapping - an introductory perspective*. Elsevier, v. 31, p. 183-192, 2006.

de Gruijter, J. J.; Brus, D.; Bierkens, M.; Knotters, M. *Sampling for natural resource monitoring*. Berlin: Springer, p. 332, 2006.

Walvoort, D. J. J.; Brus, D. J.; de Gruijter, J. J. An R package for spatial coverage sampling and random sampling from compact geographical strata by k-means. *Computers and Geosciences*. v. 36, p. 1261-1267, 2010.

See Also

distanceFromPoints, stratify.

Examples

```
require(sp)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
set.seed(2001)
## Not run:
# This example takes more than 5 seconds to run!
res <- optimMSSD(points = 100, candi = candi)
objSPSANN(res) # 11531.03
objMSSD(candi = candi, points = res)

## End(Not run)
# Random sample
pts <- sample(1:nrow(candi), 5)
pts <- cbind(pts, candi[pts, ])
objMSSD(candi = candi, points = pts)
```

---

optimPPL *Optimization of sample configurations for variogram identification and estimation*

Description

Optimize a sample configuration for variogram identification and estimation. A criterion is defined so that the optimized sample configuration has a given number of points or point-pairs contributing to each lag-distance class (**PPL**).

Usage

```
optimPPL(points, candi, iterations = 100, lags = 7,
  lags.type = "exponential", lags.base = 2, cutoff,
  criterion = "distribution", distri, pairs = FALSE, x.max, x.min, y.max,
  y.min, acceptance = list(initial = 0.99, cooling = iterations/10),
  stopping = list(max.count = iterations/10), plotit = FALSE,
  track = FALSE, boundary, progress = TRUE, verbose = FALSE,
  greedy = FALSE, weights = NULL, nadir = NULL, utopia = NULL)

objPPL(points, candi, lags = 7, lags.type = "exponential", lags.base = 2,
  cutoff, distri, criterion = "distribution", pairs = FALSE, x.max, x.min,
  y.max, y.min)

countPPL(points, candi, lags = 7, lags.type = "exponential",
  lags.base = 2, cutoff, pairs = FALSE, x.max, x.min, y.max, y.min)
```

Arguments

points	Integer value, integer vector, data frame or matrix. If points is an integer value, it defines the number of points that should be randomly sampled from candi to form the starting system configuration. If points is a vector of integer values, it contains the row indexes of candi that correspond to the points that form the starting system configuration. If points is a data frame or matrix, it must have three columns in the following order: [, "id"] the row indexes of candi that correspond to each point, [, "x"] the projected x-coordinates, and [, "y"] the projected y-coordinates. Note that in the later case, points must be a subset of candi.
candi	Data frame or matrix with the candidate locations for the perturbed points. candi must have two columns in the following order: [, "x"] the projected x-coordinates, and [, "y"] the projected y-coordinates.
iterations	Integer. The maximum number of iterations that should be used for the optimization. Defaults to iterations = 100.
lags	Integer value. The number of lag-distance classes. Alternatively, a vector of numeric values with the lower and upper limits of each lag-distance class. The lowest value must be larger than zero. Defaults to lags = 7.
lags.type	Character value. The type of lag-distance classes, with options "equidistant" and "exponential". Defaults to lags.type = "exponential".
lags.base	Numeric value. Base of the exponential expression used to create exponentially spaced lag-distance classes. Used only when lags.type = "exponential". Defaults to lags.base = 2.
cutoff	Numeric value. The maximum distance up to which lag-distance classes are created. Used only when lags is an integer value.
criterion	Character value. The feature used to describe the energy state of the system configuration, with options "minimum" and "distribution". Defaults to objective = "distribution".
distri	Numeric vector. The distribution of points or point-pairs per lag-distance class that should be attained at the end of the optimization. Used only when criterion = "distribution". Defaults to a uniform distribution.
pairs	Logical value. Should the sample configuration be optimized regarding the number of point-pairs per lag-distance class? Defaults to pairs = FALSE.
x.max,x.min,y.max,y.min	Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from candi.
acceptance	List with two named sub-arguments: initial – numeric value between 0 and 1 defining the initial acceptance probability, and cooling – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to acceptance = list(initial = 0.99, cooling = iterations / 10).
stopping	List with one named sub-argument: max.count – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to stopping = list(max.count = iterations / 10).

plotit	Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to plotit = FALSE.
track	Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If track = FALSE (the default), only the starting and ending energy state values are returned with the result.
boundary	SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from candi.
progress	Logical for printing a progress bar. Defaults to progress = TRUE.
verbose	Logical for printing messages about the progress of the optimization. Defaults to verbose = FALSE.
greedy	Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to greedy = FALSE. (experimental)
weights	List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.
nadir	List with named sub-arguments. Three options are available: 1) sim – the number of simulations that should be used to estimate the nadir point, and seeds – vector defining the random seeds for each simulation; 2) user – a list of user-defined nadir values named after the respective objective function to which they apply; 3) abs – logical for calculating the nadir point internally (experimental).
utopia	List with named sub-arguments. Two options are available: 1) user – a list of user-defined values named after the respective objective function to which they apply; 2) abs – logical for calculating the utopia point internally (experimental).

Value

optimPPL returns a matrix: the optimized sample configuration.

objPPL returns a numeric value: the energy state of the sample configuration - the objective function value.

countPPL returns a data.frame with three columns: a) the lower and b) upper limits of each lag-distance class, and c) the number of points or point-pairs per lag-distance class.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper

bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Lag-distance classes

Two types of lag-distance classes can be created by default. The first are evenly spaced lags (`lags.type = "equidistant"`). They are created by simply dividing the distance interval from 0.0001 to `cutoff` by the required number of lags. The minimum value of 0.0001 guarantees that a point does not form a pair with itself. The second type of lags is defined by exponential spacings (`lags.type = "exponential"`). The spacings are defined by the base b of the exponential expression b^n, where n is the required number of lags. The base is defined using the argument `lags.base`.

Using the default uniform distribution means that the number of point-pairs per lag-distance class (`pairs = TRUE`) is equal to $n \times (n-1)/(2 \times lag)$, where n is the total number of points and lag is the number of lags. If `pairs = FALSE`, then it means that the number of points per lag is equal to the total number of points. This is the same as expecting that each point contributes to every lag. Distributions other than the available options can be easily implemented changing the arguments `lags` and `distri`.

There are two optimizing criteria implemented. The first is called using `criterion = "distribution"` and is used to minimize the sum of the absolute differences between a pre-specified distribution and the observed distribution of points or point-pairs per lag-distance class. The second criterion is

called using `criterion = "minimum"`. It corresponds to maximizing the minimum number of points or point-pairs observed over all lag-distance classes.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

Bresler, E.; Green, R. E. *Soil parameters and sampling scheme for characterizing soil hydraulic properties of a watershed*. Honolulu: University of Hawaii at Manoa, p. 42, 1982.

Pettitt, A. N. & McBratney, A. B. Sampling designs for estimating spatial variance components. *Applied Statistics*. v. 42, p. 185, 1993.

Russo, D. Design of an optimal sampling network for estimating the variogram. *Soil Science Society of America Journal*. v. 48, p. 708-716, 1984.

Truong, P. N.; Heuvelink, G. B. M.; Gosling, J. P. Web-based tool for expert elicitation of the variogram. *Computers and Geosciences*. v. 51, p. 390-399, 2013.

Warrick, A. W.; Myers, D. E. Optimization of sampling locations for variogram calculations. *Water Resources Research*. v. 23, p. 496-500, 1987.

Examples

```
## Not run:
# This example takes more than 5 seconds to run!
require(sp)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
set.seed(2001)
res <- optimPPL(points = 100, candi = candi)
objSPSANN(res) # 160
objPPL(points = res, candi = candi)
countPPL(points = res, candi = candi)

## End(Not run)
```

optimSPAN *Optimization of sample configurations for variogram and spatial trend identification and estimation, and for spatial interpolation*

Description

Optimize a sample configuration for variogram and spatial trend identification and estimation, and for spatial interpolation. An utility function U is defined so that the sample points cover, extend over, spread over, **SPAN** the feature, variogram and geographic spaces. The utility function is obtained aggregating four single objective functions: **CORR**, **DIST**, **PPL**, and **MSSD**.

Usage

```
optimSPAN(points, candi, iterations = 100, covars, strata.type = "area",
  use.coords = FALSE, lags = 7, lags.type = "exponential",
  lags.base = 2, cutoff, criterion = "distribution", distri,
  pairs = FALSE, x.max, x.min, y.max, y.min, acceptance = list(initial =
  0.99, cooling = iterations/10), stopping = list(max.count = iterations/10),
  plotit = FALSE, track = FALSE, boundary, progress = TRUE,
  verbose = FALSE, greedy = FALSE, weights = list(CORR = 1/6, DIST = 1/6,
  PPL = 1/3, MSSD = 1/3), nadir = list(sim = NULL, seeds = NULL, user = NULL,
  abs = NULL), utopia = list(user = NULL, abs = NULL))

objSPAN(points, candi, covars, strata.type = "area", use.coords = FALSE,
  lags = 7, lags.type = "exponential", lags.base = 2, cutoff,
  criterion = "distribution", distri, pairs = FALSE, x.max, x.min, y.max,
  y.min, weights = list(CORR = 1/6, DIST = 1/6, PPL = 1/3, MSSD = 1/3),
  nadir = list(sim = NULL, seeds = NULL, user = NULL, abs = NULL),
  utopia = list(user = NULL, abs = NULL))
```

Arguments

points	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
candi	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
iterations	Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.

`covars` Data frame or matrix with the covariates in the columns.

`strata.type` Character value setting the type of stratification that should be used to create the marginal sampling strata (or factor levels) for the numeric covariates. Available options are `"area"`, for equal-area, and `"range"`, for equal-range. Defaults to `strata.type = "area"`.

`use.coords` Logical value. Should the geographic coordinates be used as covariates? Defaults to `use.coords = FALSE`.

`lags` Integer value. The number of lag-distance classes. Alternatively, a vector of numeric values with the lower and upper limits of each lag-distance class. The lowest value must be larger than zero. Defaults to `lags = 7`.

`lags.type` Character value. The type of lag-distance classes, with options `"equidistant"` and `"exponential"`. Defaults to `lags.type = "exponential"`.

`lags.base` Numeric value. Base of the exponential expression used to create exponentially spaced lag-distance classes. Used only when `lags.type = "exponential"`. Defaults to `lags.base = 2`.

`cutoff` Numeric value. The maximum distance up to which lag-distance classes are created. Used only when `lags` is an integer value.

`criterion` Character value. The feature used to describe the energy state of the system configuration, with options `"minimum"` and `"distribution"`. Defaults to `objective = "distribution"`.

`distri` Numeric vector. The distribution of points or point-pairs per lag-distance class that should be attained at the end of the optimization. Used only when `criterion = "distribution"`. Defaults to a uniform distribution.

`pairs` Logical value. Should the sample configuration be optimized regarding the number of point-pairs per lag-distance class? Defaults to `pairs = FALSE`.

`x.max,x.min,y.max,y.min`
Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.

`acceptance` List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.

`stopping` List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.

`plotit` Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.

`track` Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.

boundary	SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from candi.
progress	Logical for printing a progress bar. Defaults to progress = TRUE.
verbose	Logical for printing messages about the progress of the optimization. Defaults to verbose = FALSE.
greedy	Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to greedy = FALSE. (experimental)
weights	List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.
nadir	List with named sub-arguments. Three options are available: 1) sim – the number of simulations that should be used to estimate the nadir point, and seeds – vector defining the random seeds for each simulation; 2) user – a list of user-defined nadir values named after the respective objective function to which they apply; 3) abs – logical for calculating the nadir point internally (experimental).
utopia	List with named sub-arguments. Two options are available: 1) user – a list of user-defined values named after the respective objective function to which they apply; 2) abs – logical for calculating the utopia point internally (experimental).

Value

optimSPAN() returns a matrix: the optimized sample configuration.

objSPAN returns a numeric value: the energy state of the sample configuration - the objective function value.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in points. Because the more points there are in points, the more likely it is that the new location already is occupied by

another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger

than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Association/Correlation between covariates

The *correlation* between two numeric covariates is measured using the Pearson's r, a descriptive statistic that ranges from -1 to $+1$. This statistic is also known as the linear correlation coefficient.

When the set of covariates includes factor covariates, all numeric covariates are transformed into factor covariates. The factor levels are defined using the marginal sampling strata created from one of the two methods available (equal-area or equal-range strata).

The *association* between two factor covariates is measured using the Cramér's v, a descriptive statistic that ranges from 0 to $+1$. The closer to $+1$ the Cramér's v is, the stronger the association between two factor covariates. The main weakness of using the Cramér's v is that, while the Pearson's r shows the degree and direction of the association between two covariates (negative or positive), the Cramér's v only measures the degree of association (weak or strong).

Marginal distribution of covariates

Reproducing the marginal distribution of the numeric covariates depends upon the definition of marginal sampling strata. These marginal sampling strata are also used to define the factor levels of all numeric covariates that are passed together with factor covariates.

Two types of marginal sampling strata can be used. *Equal-area* sampling strata are defined using the sample quantiles estimated with `quantile` using a discontinuous function (`type = 3`). This is to avoid creating breakpoints that do not occur in the population of existing covariate values.

The function `quantile` commonly produces repeated break points. A break point will always be repeated if that value has a relatively high frequency in the population of covariate values. The number of repeated break points increases with the number of marginal sampling strata. Only unique break points are used to create marginal sampling strata.

Equal-range sampling strata are defined by breaking the range of covariate values into pieces of equal size. This method usually creates break points that do not occur in the population of existing covariate values. Such break points are replaced by the nearest existing covariate value identified using Euclidean distances.

Both stratification methods can produce marginal sampling strata that cover a range of values that do not exist in the population of covariate values. Any empty marginal sampling strata is merged with the closest non-empty marginal sampling strata. These are identified using Euclidean distances.

The approaches used to define the marginal sampling strata result in each numeric covariate having a different number of marginal sampling strata, some of them with different area/size. Because the goal is to have a sample that reproduces the marginal distribution of the covariate, each marginal sampling strata will have a different number of sample points. The wanted distribution of the number of sample points per marginal strata is estimated empirically as the proportion of points in the population of existing covariate values that fall in each marginal sampling strata.

Lag-distance classes

Two types of lag-distance classes can be created by default. The first are evenly spaced lags (`lags.type = "equidistant"`). They are created by simply dividing the distance interval from 0.0001 to `cutoff` by the required number of lags. The minimum value of 0.0001 guarantees that a point does not form a pair with itself. The second type of lags is defined by exponential spacings (`lags.type = "exponential"`). The spacings are defined by the base b of the exponential expression b^n, where n is the required number of lags. The base is defined using the argument `lags.base`.

Using the default uniform distribution means that the number of point-pairs per lag-distance class (`pairs = TRUE`) is equal to $n \times (n-1)/(2 \times lag)$, where n is the total number of points and lag is the number of lags. If `pairs = FALSE`, then it means that the number of points per lag is equal to the total number of points. This is the same as expecting that each point contributes to every lag. Distributions other than the available options can be easily implemented changing the arguments `lags` and `distri`.

There are two optimizing criteria implemented. The first is called using `criterion = "distribution"` and is used to minimize the sum of the absolute differences between a pre-specified distribution and the observed distribution of points or point-pairs per lag-distance class. The second criterion is called using `criterion = "minimum"`. It corresponds to maximizing the minimum number of points or point-pairs observed over all lag-distance classes.

Spatial coverage sampling

Spatial coverage sampling is based on the knowledge that the kriging variance depends upon the distance between the sample points. As such, the better the spread of the sample points in the spatial domain, the smaller the kriging variance. This is similar to using a regular grid of sample points. However, a regular grid usually is suboptimal for irregularly shaped areas.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Cramér, H. *Mathematical methods of statistics*. Princeton: Princeton University Press, p. 575, 1946.

Everitt, B. S. *The Cambridge dictionary of statistics*. Cambridge: Cambridge University Press, p. 432, 2006.

Hyndman, R. J.; Fan, Y. Sample quantiles in statistical packages. *The American Statistician*, v. 50, p. 361-365, 1996.

Minasny, B.; McBratney, A. B. A conditioned Latin hypercube method for sampling in the presence of ancillary information. *Computers & Geosciences*, v. 32, p. 1378-1388, 2006.

Minasny, B.; McBratney, A. B. Conditioned Latin Hypercube Sampling for calibrating soil sensor data to soil properties. Chapter 9. Viscarra Rossel, R. A.; McBratney, A. B.; Minasny, B. (Eds.) *Proximal Soil Sensing*. Amsterdam: Springer, p. 111-119, 2010.

Mulder, V. L.; de Bruin, S.; Schaepman, M. E. Representing major soil variability at regional scale by constrained Latin hypercube sampling of remote sensing data. *International Journal of Applied Earth Observation and Geoinformation*, v. 21, p. 301-310, 2013.

Roudier, P.; Beaudette, D.; Hewitt, A. A conditioned Latin hypercube sampling algorithm incorporating operational constraints. *5th Global Workshop on Digital Soil Mapping*. Sydney, p. 227-231, 2012.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

Bresler, E.; Green, R. E. *Soil parameters and sampling scheme for characterizing soil hydraulic properties of a watershed*. Honolulu: University of Hawaii at Manoa, p. 42, 1982.

Pettitt, A. N. & McBratney, A. B. Sampling designs for estimating spatial variance components. *Applied Statistics*. v. 42, p. 185, 1993.

Russo, D. Design of an optimal sampling network for estimating the variogram. *Soil Science Society of America Journal*. v. 48, p. 708-716, 1984.

Truong, P. N.; Heuvelink, G. B. M.; Gosling, J. P. Web-based tool for expert elicitation of the variogram. *Computers and Geosciences*. v. 51, p. 390-399, 2013.

Warrick, A. W.; Myers, D. E. Optimization of sampling locations for variogram calculations. *Water Resources Research*. v. 23, p. 496-500, 1987.

Examples

```
## Not run:
# This example takes more than 5 seconds to run!
require(sp)
data(meuse.grid)
candi <- meuse.grid[, 1:2]
nadir <- list(sim = 10, seeds = 1:10)
utopia <- list(user = list(DIST = 0, CORR = 0, PPL = 0, MSSD = 0))
covars <- meuse.grid[, 5]
set.seed(2001)
res <- optimSPAN(points = 100, candi = candi, covars = covars, nadir = nadir,
                 use.coords = TRUE, utopia = utopia)
objSPSANN(res) - # 0.776184
  objSPAN(points = res, candi = candi, covars = covars, nadir = nadir,
          use.coords = TRUE, utopia = utopia)

## End(Not run)
```

---

optimUSER *Optimization of sample configurations using a user-defined objective function*

Description

Optimize a sample configuration using a user-defined objective function.

Usage

```
optimUSER(points, candi, iterations = 100, fun, ..., x.max, x.min, y.max,
  y.min, acceptance = list(initial = 0.99, cooling = iterations/10),
  stopping = list(max.count = iterations/10), plotit = FALSE,
  track = FALSE, boundary, progress = TRUE, verbose = FALSE,
  greedy = FALSE, weights = NULL, nadir = NULL, utopia = NULL)
```

Arguments

points	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
candi	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.

`iterations` Integer. The maximum number of iterations that should be used for the optimization. Defaults to `iterations = 100`.

`fun` A function defining the objective function that should be used to evaluate the energy state of the system configuration at each iteration. See 'Details' for more information.

`...` Other arguments passed to the objective function. See 'Details' for more information.

`x.max,x.min,y.max,y.min`
Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.

`acceptance` List with two named sub-arguments: `initial` – numeric value between 0 and 1 defining the initial acceptance probability, and `cooling` – a numeric value defining the exponential factor by which the acceptance probability decreases at each iteration. Defaults to `acceptance = list(initial = 0.99, cooling = iterations / 10)`.

`stopping` List with one named sub-argument: `max.count` – integer value defining the maximum allowable number of iterations without improvement of the objective function value. Defaults to `stopping = list(max.count = iterations / 10)`.

`plotit` Logical for plotting the optimization results. This includes a) the progress of the objective function values and acceptance probabilities, and b) the original points, the perturbed points and the progress of the maximum perturbation in the x- and y-coordinates. The plots are updated at each 10 iterations. Defaults to `plotit = FALSE`.

`track` Logical value. Should the evolution of the energy state and acceptance probability be recorded and returned with the result? If `track = FALSE` (the default), only the starting and ending energy state values are returned with the result.

`boundary` SpatialPolygon. The boundary of the spatial domain. If missing, it is estimated from `candi`.

`progress` Logical for printing a progress bar. Defaults to `progress = TRUE`.

`verbose` Logical for printing messages about the progress of the optimization. Defaults to `verbose = FALSE`.

`greedy` Logical value. Should the optimization be done using a greedy algorithm, that is, accepting only better system configurations? Defaults to `greedy = FALSE`. (experimental)

`weights` List with named sub-arguments. The weights assigned to each one of the objective functions that form the multi-objective optimization problem (MOOP). They must be named after the respective objective function to which they apply. The weights must be equal to or larger than 0 and sum to 1. The default option gives equal weights to all objective functions.

`nadir` List with named sub-arguments. Three options are available: 1) `sim` – the number of simulations that should be used to estimate the nadir point, and `seeds` – vector defining the random seeds for each simulation; 2) `user` – a list of user-defined nadir values named after the respective objective function to which they apply; 3) `abs` – logical for calculating the nadir point internally (experimental).

`utopia` List with named sub-arguments. Two options are available: 1) `user` – a list of user-defined values named after the respective objective function to which they apply; 2) `abs` – logical for calculating the utopia point internally (experimental).

Details

The user-defined objective function `fun` must be an object of class `function` and include the argument `points`. The argument `points` is defined in `optimUSER` as a matrix with three columns: `[, 1]` the identification of each sample point given by the respective row indexes of `candi`, `[, 2]` the x-coordinates, and `[, 3]` the y-coordinates. The identification is useful to retrieve information from any data matrix used by the objective function defined by the user.

Value

`optimUSER` returns a matrix: the optimized sample configuration.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the co-

ordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Multi-objective optimization

A method of solving a multi-objective optimization problem is to aggregate the objective functions into a single *utility function*. In the **spsann** package, the aggregation is performed using the *weighted sum method*, which incorporates in the weights the preferences of the user regarding the relative importance of each objective function.

The weighted sum method is affected by the relative magnitude of the different function values. The objective functions implemented in the **spsann** package have different units and orders of magnitude. The consequence is that the objective function with the largest values will have a numerical dominance in the optimization. In other words, the weights will not express the true preferences of the user, and the meaning of the utility function becomes unclear.

A solution to avoid the numerical dominance is to transform the objective functions so that they are constrained to the same approximate range of values. Several function-transformation methods can be used and the **spsann** offers a few of them. The *upper-lower-bound approach* requires the user to inform the maximum (nadir point) and minimum (utopia point) absolute function values. The resulting function values will always range between 0 and 1.

Using the *upper-bound approach* requires the user to inform only the nadir point, while the utopia point is set to zero. The upper-bound approach for transformation aims at equalizing only the upper bounds of the objective functions. The resulting function values will always be smaller than or equal to 1.

Sometimes, the absolute maximum and minimum values of an objective function can be calculated exactly. This seems not to be the case of the objective functions implemented in the **spsann** package. If the user is uncomfortable with informing the nadir and utopia points, there is the option for using *numerical simulations*. It consists in computing the function value for many random sample configurations. The mean function value is used to set the nadir point, while the the utopia point is set to zero. This approach is similar to the upper-bound approach, but the function values will have the same orders of magnitude only at the starting point of the optimization. Function values larger than one are likely to occur during the optimization. We recommend the user to avoid this approach whenever possible because the effect of the starting point on the optimization as a whole usually is insignificant or arbitrary.

The *upper-lower-bound approach* with the *Pareto maximum and minimum values* is the most elegant solution to transform the objective functions. However, it is the most time consuming. It works as follows:

1. Optimize a sample configuration with respect to each objective function that composes the MOOP;
2. Compute the function value of every objective function for every optimized sample configuration;
3. Record the maximum and minimum absolute function values computed for each objective function–these are the Pareto maximum and minimum.

For example, consider that a MOOP is composed of two objective functions (A and B). The minimum absolute value for function A is obtained when the sample configuration is optimized with

respect to function A. This is the Pareto minimum of function A. Consequently, the maximum absolute value for function A is obtained when the sample configuration is optimized regarding function B. This is the Pareto maximum of function A. The same logic applies for function B.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

Arora, J. *Introduction to optimum design*. Waltham: Academic Press, p. 896, 2011.

Marler, R. T.; Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, v. 26, p. 369-395, 2004.

Marler, R. T.; Arora, J. S. Function-transformation methods for multi-objective optimization. *Engineering Optimization*, v. 37, p. 551-570, 2005.

Marler, R. T.; Arora, J. S. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, v. 41, p. 853-862, 2009.

Examples

```
## Not run:
# This example takes more than 5 seconds to run!
require(sp)
require(SpatialTools)
data(meuse.grid)
candi <- meuse.grid[, 1:2]

# Define the objective function - number of points per lag distance class
objUSER <-
  function (points, lags, n_lags, n_pts) {
    dm <- SpatialTools::dist1(points[, 2:3])
    ppl <- vector()
    for (i in 1:n_lags) {
      n <- which(dm > lags[i] & dm <= lags[i + 1], arr.ind = TRUE)
      ppl[i] <- length(unique(c(n)))
    }
    distri <- rep(n_pts, n_lags)
    res <- sum(distri - ppl)
  }
lags <- seq(1, 1000, length.out = 10)

# Run the optimization using the user-defined objective function
set.seed(2001)
timeUSER <- Sys.time()
```

```
resUSER <- optimUSER(points = 100, fun = objUSER, lags = lags, n_lags = 9,
                     n_pts = 100, candi = candi)
timeUSER <- Sys.time() - timeUSER

# Run the optimization using the respective function implemented in spsann
set.seed(2001)
timePPL <- Sys.time()
resPPL <- optimPPL(points = 100, candi = candi, lags = lags)
timePPL <- Sys.time() - timePPL

# Compare results
timeUSER
timePPL
lapply(list(resUSER, resPPL), countPPL, candi = candi, lags = lags)
objSPSANN(resUSER) # 58
objSPSANN(resPPL) # 58

## End(Not run)
```

`spJitterFinite` *Random perturbation of spatial points*

Description

Perturb the coordinates of spatial points ('jittering').

Usage

```
spJitterFinite(points, candi, x.max, x.min, y.max, y.min, which.point)
```

Arguments

`points`	Integer value, integer vector, data frame or matrix. If `points` is an integer value, it defines the number of points that should be randomly sampled from `candi` to form the starting system configuration. If `points` is a vector of integer values, it contains the row indexes of `candi` that correspond to the points that form the starting system configuration. If `points` is a data frame or matrix, it must have three columns in the following order: `[, "id"]` the row indexes of `candi` that correspond to each point, `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates. Note that in the later case, `points` must be a subset of `candi`.
`candi`	Data frame or matrix with the candidate locations for the perturbed points. `candi` must have two columns in the following order: `[, "x"]` the projected x-coordinates, and `[, "y"]` the projected y-coordinates.
`x.max,x.min,y.max,y.min`	Numeric value. The minimum and maximum quantity of random noise to be added to the projected x- and y-coordinates. The minimum quantity should be equal to, at least, the minimum distance between two neighbouring candidate

locations. The units are the same as of the projected x- and y-coordinates. If missing, they are estimated from `candi`.

`which.point` Integer values defining which point should be perturbed.

Value

A matrix with the jittered projected coordinates of the points.

Jittering methods

There are two ways of jittering the coordinates. They differ on how the set of candidate locations is defined. The first method uses an *infinite* set of candidate locations, that is, any point in the spatial domain can be selected as the new location of a perturbed point. All that this method needs is a polygon indicating the boundary of the spatial domain. This method is not implemented in the **spsann** package (yet) because it is computationally demanding: every time a point is jittered, it is necessary to check if it is inside the spatial domain.

The second method consists of using a *finite* set of candidate locations for the perturbed points. A finite set of candidate locations is created by discretizing the spatial domain, that is, creating a fine grid of points that serve as candidate locations for the perturbed points. This is the only method currently implemented in the **spsann** package because it is one of the least computationally demanding.

Using a finite set of candidate locations has one important inconvenience. When a point is selected to be jittered, it may be that the new location already is occupied by another point. If this happens, another location is iteratively sought for as many times as there are points in `points`. Because the more points there are in `points`, the more likely it is that the new location already is occupied by another point. If a solution is not found, the point selected to be jittered point is kept in its original location.

A more elegant method can be defined using a finite set of candidate locations coupled with a form of *two-stage random sampling* as implemented in `spsample`. Because the candidate locations are placed on a finite regular grid, they can be seen as being the centre nodes of a finite set of grid cells (or pixels of a raster image). In the first stage, one of the "grid cells" is selected with replacement, i.e. independently of already being occupied by another sample point. The new location for the point chosen to be jittered is selected within that "grid cell" by simple random sampling. This method guarantees that any location in the spatial domain can be a candidate location. It also discards the need to check if the new location already is occupied by another point. This method is not implemented (yet) in the **spsann** package.

Distance between two points

The distance between two points is computed as the Euclidean distance between them. This computation assumes that the optimization is operating in the two-dimensional Euclidean space, i.e. the coordinates of the sample points and candidate locations should not be provided as latitude/longitude. Package **spsann** has no mechanism to check if the coordinates are projected, and the user is responsible for making sure that this requirement is attained.

Author(s)

Alessandro Samuel-Rosa <alessandrosamuelrosa@gmail.com>

References

Edzer Pebesma, Jon Skoien with contributions from Olivier Baume, A. Chorti, D.T. Hristopulos, S.J. Melles and G. Spiliopoulos (2013). *intamapInteractive: procedures for automated interpolation - methods only to be used interactively, not included in* `intamap` *package.* R package version 1.1-10.

van Groenigen, J.-W. *Constrained optimization of spatial sampling: a geostatistical approach.* Wageningen: Wageningen University, p. 148, 1999.

See Also

`ssaOptim`, `zerodist`, `jitter`, `jitter2d`.

Examples

```
require(sp)
data(meuse.grid)
meuse.grid <- as.matrix(meuse.grid[, 1:2])
meuse.grid <- matrix(cbind(1:dim(meuse.grid)[1], meuse.grid), ncol = 3)
pts1 <- sample(c(1:dim(meuse.grid)[1]), 155)
pts2 <- meuse.grid[pts1, ]
pts3 <- spJitterFinite(points = pts2, candi = meuse.grid, x.min = 40,
                       x.max = 100, y.min = 40, y.max = 100, which.point = 10)
plot(meuse.grid[, 2:3], asp = 1, pch = 15, col = "gray")
points(pts2[, 2:3], col = "red", cex = 0.5)
points(pts3[, 2:3], pch = 19, col = "blue", cex = 0.5)

# Cluster of points
pts1 <- c(1:55)
pts2 <- meuse.grid[pts1, ]
pts3 <- spJitterFinite(points = pts2, candi = meuse.grid, x.min = 40,
                       x.max = 80, y.min = 40, y.max = 80, which.point = 1)
plot(meuse.grid[, 2:3], asp = 1, pch = 15, col = "gray")
points(pts2[, 2:3], col = "red", cex = 0.5)
points(pts3[, 2:3], pch = 19, col = "blue", cex = 0.5)
```

Index