# SPATIALSTATS
# Version 1.5 Supplement

November 2008

TIBCO Spotfire Inc.

## Proprietary Notice



The correct bibliographical reference for this document is as follows:

*SPATIALSTATS Version 1.5 Supplement,* Data Analysis Products Division, TIBCO Software Inc.

Printed in the United States.

## Copyright Notice

# CONTENTS

**Chapter 1 Fitting Variograms** 1

**Chapter 2 Block Kriging** 3

Block Kriging with the Coal Ash Data 4

**Chapter 3 Summarizing and Plotting Spatial Neighbor Objects** 5

**Chapter 4 Simulating Nonhomogeneous Poisson Patterns** 9

**Appendix A: Data and Function Reference** 13

# Welcome to SPATIALSTATS 1.5 for UNIX

Congratulations on acquiring Version 1.5 of the SPATIALSTATS package of Spotfire S+. This new version includes the following functionality:

- Variogram Fitting
- Block Kriging
- Summary and plot methods for spatial neighbor objects
- Simulation of Nonhomogenous Poisson Point Patterns
- New data set: `Glasgow.SMR`
- Numerous Bug fixes: `find.neighbor`, `quad.tree`, `sids` dataset names

Use this supplemental guide to:

- Get started using the new features in SPATIALSTATS,
- Access new help files for the new functions and for those that were modified.

# FITTING VARIOGRAMS 1

Use the function `variogram.fit` to fit a theoretical variogram model to an empirical variogram. The function uses the bounded non-linear optimizer in Spotfire S+, `nlminb`, to ensure that the estimated parameters are valid (typically non-negative). The objective function that is minimized is the following weighted least squares estimator proposed by Cressie (1985):

$$\sum_{j=1}^{K} |N(h_j)| \left[ \frac{\gamma(h_j)}{\gamma(h_j;\theta)} - 1 \right]^2$$

where $|N(h_j)|$ is the number of distinct pairs in lag j, K is the number of lags, $\gamma(h_j)$ is the value of the empirical variogram at lag j, and $\gamma(h_j;\theta)$ is the known theoretical variogram model with unknown parameters $\theta$.

The `variogram.fit` function requires an object of class `"variogram"` as its first argument. This is typically created by the function `variogram`. Other arguments, all optional, include: `param`, a named vector of parameters to optimize over; `fun`, the theoretical variogram function; and `lower` and `upper`, which specify the upper and lower limits of parameters.

The function assumes that the first argument to `fun` is distance and all the other arguments to `fun` are parameters to be estimated.

If the `param` argument is not provided then the function gets the parameter names from `fun`. If `fun` is one of the known variogram functions included with SPATIALSTATS (`exp.vgram`, `spher.vgram`, `gauss.vgram`, `power.vgram`, or `linear.vgram`) and initial parameter values are not supplied the function will set special starting values appropriate for those functions. If `fun` is some other variogram function and `param` is not provided then it is set to a vector of ones.

Here is how you would fit a spherical variogram model to the `iron.ore` data set:

1. First compute and plot the empirical variogram:

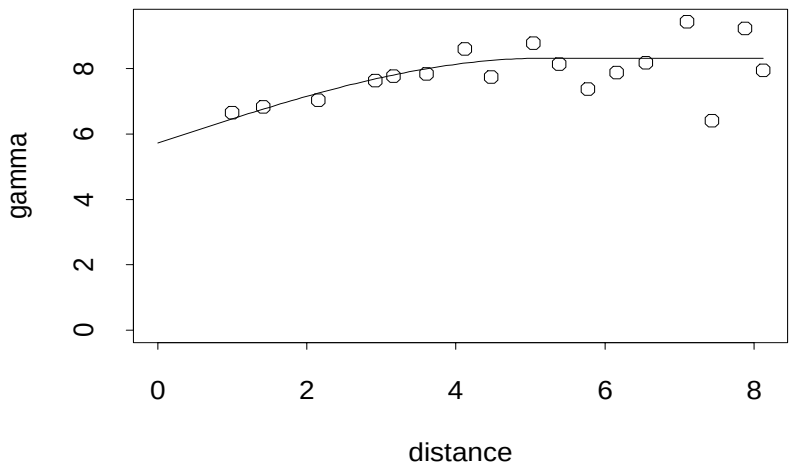
```
> vg.iron <- variogram(residuals ~ loc(easting, northing),
+ data=iron.ore)
> plot(vg.iron)
```

2. From the plot you can guess at initial values for the variogram model and use them when calling `variogram.fit`:

```
> vfit.iron <- variogram.fit(vg.iron, param=c(range=8.7,
+ sill=3.5, nugget=4.8), fun=spher.vgram)
```

3. Examine the resulting fit using the print and plot methods for objects created by `variogram.fit`:

```
> vfit.iron
Function: spher.vgram
Parameters:
    range     sill   nugget
5.184546 2.593336 5.722811
Objective function value: 26.69099
> plot(vfit.iron, add=T)
```



# BLOCK KRIGING

# 2

*Block kriging* is the general term used for the prediction of the average value of a random field over a segment, surface, or volume. The term *point kriging* refers to prediction of the field at a point. The block value over a block A at location $\boldsymbol{x}_0$ is:

$$z(\boldsymbol{x}_0) = \frac{1}{|A|}\int_A z(\boldsymbol{u})d\boldsymbol{u}$$

$$\cong \frac{1}{N}\sum_{i=1}^{N} z(u_i)$$

where | A | is the length, area or volume of the block A.

In SPATIALSTATS, block kriging is computed by the `predict` method for objects of call `"krige"`, that is, `predict.krige`.

Block kriging is restricted to prediction of the average value over a rectangular area. The integral over the rectangular block is approximated by the average of the point prediction. You control the number of points in the average by the `nxy` argument to `predict.krige`.

The `nxy` argument consists of two values: the number of points in the x direction, and the number of points in the y direction. Thus the total number of points in the block average is `nxy[1]*nxy[2]`.

The size of the block is specifed by the `blocksize` argument which is a vector of length two: the dimension of the block in the x direction and the dimension in the y direction.

The predictions are calculated with the supplied prediction locations in the center of the block.

## Block Kriging with the Coal Ash Data

Whether you are using point or block kriging, you create the `krige` object in the same way:

```
> kcoal <- krige(coal ~ loc(x, y) + x + x^2, data =
+   coal.ash, covfun = spher.cov, range = 4.31, sill = 0.14,
+   nugget = 0.89)
```

Create a small data frame of prediction locations and compute both block (over 2 x 2 blocks) and point kriging predictions at them:

```
> newdf <- data.frame(x = c(4.5,5.5,9.5,10.5), y =
+   c(7.5,13.5,9.5,18.5))
# Block Kriging:
> predict(kcoal, newdata = newdf, blocksize = c(2,2),
+    nxy = c(5,5))
     x    y       fit    se.fit
1  4.5  7.5 10.774804 0.2250666
2  5.5 13.5 10.061807 0.2239514
3  9.5  9.5  9.475169 0.2256258
4 10.5 18.5  8.838148 0.2247773
# Point Kriging:
> predict(kcoal, newdata = newdf)
     x    y       fit    se.fit
1  4.5  7.5 10.799256 0.9905583
2  5.5 13.5 10.068892 0.9903889
3  9.5  9.5  9.454931 0.9907360
4 10.5 18.5  8.811028 0.9905622
```

The predicted values (fit) are very similar between the two as we would expect. The standard errors are much smaller for the block kriging since the predictions are averages.

# SUMMARIZING AND PLOTTING SPATIAL NEIGHBOR OBJECTS

# 3

Functions for plotting (`plot.spatial.neighbor`) and summarizing (`summary.spatial.neighbor`) were added to SPATIALSTATS.

The help files for these functions contain details for the new functions. These functions are methods for the generic functions `plot` and `summary`. They can be called as `plot` and `summary` if the first argument given to them is an object of class `"spatial.neighbor"`.

We illustrate these new functions here using the new spatial neighbor data set `Glasgow.neighbor` and its associated data set, `Glasgow.SMR`.

The data set consists of standardized mortality rates (SMR) for a number of diseases in 87 community medicine areas in Glasgow, Scotland. The neighbor relationships in `Glasgow.neighbor` are based on contiguous community medicine areas. The `Glasgow.SMR` data frame also has columns labeled `Easting` and `Northing` containing the coordinates for the center of each community medicine area.

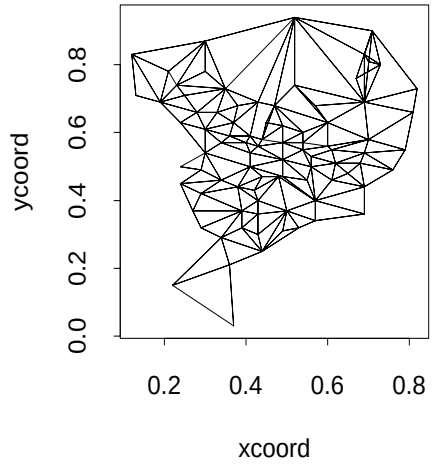
```
> summary(Glasgow.neighbor)
Matrix was NOT defined as symmetric
Number of Regions: 87
Average Number of Connections: 5.172414
Average Weight: 1
Least Number of Connections: 1 for Regions with Indices:
[1] 81
Maximum Number of Connections: 11 for Regions with Indices:
[1] 39
Missing Row Indices:
[1] "none"
Missing Column Indices:
[1] "none"
Indices of Regions with No Connections (islands):
[1] "none"
```

A spatial neighbor object contains indices only for points that are neighbors; it does not contain the actual spatial locations. To use `plot.spatial.neighbor` you need to supply the x and y coordinates along with an object of class "`spatial.neighbor`". For plotting the `Glasgow.neighbor` use the `Easting` and `Northing` values from `Glasgow.SMR`:

The plot draws line segments between locations that are defined as neighbors.

```
> plot(Glasgow.neighbor, xcoord=Glasgow.SMR$Easting,
+  ycoord=Glasgow.SMR$Northing)
```

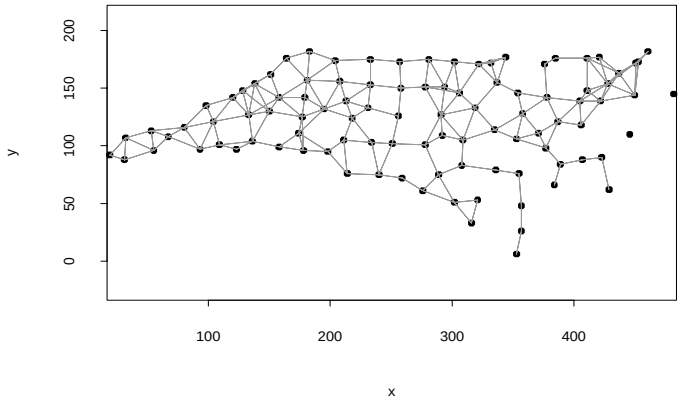
You can also use the `add` argument to `plot.spatial.neighbor` to add the neighbor line segments to an existing plot.




The example below uses the `scaled.plot` function to first draw the locations of the county seats from the North Carolina SIDS data using an equal scales plot and then adds the lines connecting the neighbors. There is also a `scaled` argument to `plot.spatial.neighbor` to request an equal scaled plot to be drawn directly.

```
> scaled.plot(sids$easting, sids$northing, pch=16)
```

```
> plot(sids.neighbor, xcoord=sids$easting,
+ ycoord=sids$northing, add=T, line.col=2)
```




Looking at the summary of the SPATIALSTATS object `sids.neighbor` you can see that the two counties with no neighbors shown in the plot are counties 28 and 48:

```
> summary(sids.neighbor)
Matrix was NOT defined as symmetric
Number of Regions: 100
Average Number of Connections: 4.020408
Average Weight: 0.1306507
Least Number of Connections: 1 for Regions with Indices:
[1] 10 16 67
Maximum Number of Connections: 8 for Regions with Indices:
[1] 21
Missing Row Indices:
[1] 28 48
Missing Column Indices:
[1] 28 48
Indices of Regions with No Connections (islands):
[1] 28 48
```

# SIMULATING NONHOMOGENEOUS POISSON PATTERNS

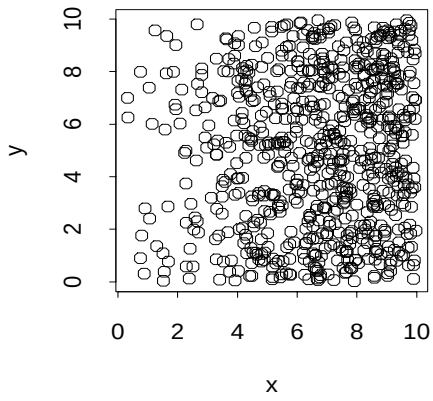
# 4

The `make.pattern` function can simulate nonhomogeneous Poisson patterns over rectangular regions. To do this you set the `process` argument to be `"poisson"` and then supply the intensity function for the Poisson process as the `lambda` argument.

This should be a function of two arguments that is non-negative over the rectangular region specified in `boundary`.

To simulate a nonhomogeneous Poisson with linear rate in the x direction over a 10 x 10 square you use:

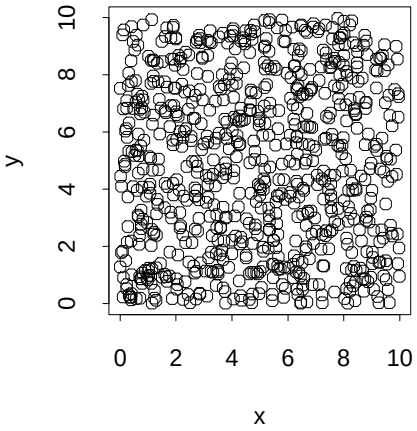
```
> linx <- function(x, y) 1.5*x
> xy <- make.pattern(proc="poisson",
+        boundary=bbox(x=c(0,10),
+        y=c(0,10)), lambda=linx)
> plot(xy)
```



The expected number of points for a Poisson process is the integral of the intensity over the region. For the above example the integral over the 10 x 10 square is 750.

Here is a homogeneous Poisson process with the same expected number of points:

```
> plot(make.pattern(proc="poisson",
+        boundary=bbox(x=c(0,10),
+        y=c(0,10)), lambda=7.5))
```




The nonhomogeneous Poisson process is simulated by a rejection sampling method (Diggle, 1983). The algorithm requires the maximum value of the intensity function over the region. This can be supplied as the argument `maxlambda` to `make.pattern`. If it is not supplied, a nonlinear optimization (using `nlminb`) is done to find this maximum.

The algorithm proceeds by generating a homogenous Poisson process with intensity `maxlambda` over the region. Then points are retained with probability `lambda(x, y)/maxlambda`.

**Reference:**

**Diggle, Peter J.** (1983). *Statistical Analysis of Spatial Point Patterns*. Academic Press, London.

# APPENDIX A: DATA AND FUNCTION REFERENCE

The functions and data sets described in this appendix are included with SPATIALSTATS. The information in this appendix is also found in the online help.

**anisotropy.plot** Explore Corrections For Geometric Anisotropy **anisotropy.plot**

DESCRIPTION

Computes corrections for geometric anisotropy for two dimensional spatial data and plots variograms based on the corrections.

USAGE

```
anisotropy.plot(formula=formula(data), data=sys.parent(),
                subset, na.action, lag=<<see below>>,
                nlag=20, tol.lag=lag/2, maxdist=<<see below>>,
                angle=c(0, 45, 90, 135),
                ratio=seq(1.25, 2, length = 4),
                minpairs=6, method="classical",
                smooth=T, plot.it=T, panel=panel.xyplot, ...)
```

REQUIRED ARGUMENTS

`formula` formula defining the response and the predictors. In general, its form is:

```
z ~ x + y
```

The `z` variable is a numeric response. Variables `x` and `y` are the locations. All variables in the formula must be vectors of equal length with no missing values (`NA`s). The formula may also contain expressions for the variables, for example, `sqrt(count)`, `log(age+1)` or `I(2*x)`. (The `I()` is required since the `*` operator has a special meaning on the right side of a formula.)

OPTIONAL ARGUMENTS

`data` an optional data frame in which to find the objects mentioned in `formula`.

`subset` expression saying which subset of the rows of the data should be used in the fit. This can be a logical vector (which is replicated to have length equal to the number of observations), or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included.

`na.action` a function to filter missing data. This is applied to the `model.frame` after any `subset` argument has been used. The default (with `na.fail`) is to create an error if any missing values are found. A possible alternative is `na.omit`, which deletes observations that contain one or more missing values.

`lag` a numeric value, the width of the lags. If missing, `lag` is set to `maxdist / nlag`.

`nlag` an integer, the maximum number of lags to calculate.

`tol.lag` a numeric value, the distance tolerance.

`maxdist` the maximum distance to include in the returned output. The default is half the maximum distance in the transformed data.

`angle` a vector of direction angles (in degrees, clockwise from North) to consider as directions of anisotropy.

`ratio` a vector of ratios of anisotropy. These should all be greater than 1.

`minpairs` the minimum number of pairs of points (minimum value for `np`) that must be used in calculating a variogram value. If `np` is less than `minpairs`, that value is dropped from the variogram.

`method` a character string to select the method for estimating the variogram. The possible values are `"classical"` for Matheron's (1963) estimate and `"robust"` for Cressie and Hawkins (1980) robust estimator. Only the first character of the string needs to be given.

`smooth` a logical flag, if `TRUE`, a loess smooth line is drawn for each variogram panel. If `panel` is supplied then this value is ignored.

`panel` a panel function to be used in plotting the variograms. If `plot.it=FALSE`, this value is ignored.

`plot.it` a logical flag, if `TRUE`, a plot of all the variogram is drawn.

`...` additional arguments to be passed down to the panel function for plotting.

VALUE

a data frame with columns:

`distance` the average distance for pairs in the lag.

`gamma` the variogram estimate.

`np` the number of pairs in each lag.

`angle` a factor denoting the angle for the geometric anisotropy.

`ratio` a factor with levels denoting the ratio for the geometric anisotropy.

SIDE EFFECTS

If `plot.it=TRUE` (the default) the variogram for each combination of `angle` and `ratio` is plotted. The plot is drawn using `xyplot`.

DETAILS

For each combination of `angle` and `ratio` the locations are corrected for geometric anisotropy. The correction consists of multipling each location pair (`x[i],y[i]`) by the symmetric 2 x 2 matrix `A` where `A[1,1]=cos(angle)^2+ratio*sin(angle)^2`, `A[1,2]=(1- ratio) * sin(angle) * cos(angle)` and `A[2,2]=sin(angle)^2+ratio*cos(angle)^2`. See Journel and Huijbregts (1978, pp 179-181). The variogram is then estimated using these corrected locations.

REFERENCES

Cressie, N. and Hawkins, D. M. (1980). Robust estimation of the variogram. *Mathematical Geology* **12**, 115-125.

Journel, A. G. and Huijbregts, Ch. J. (1978). *Mining Geostatistics.* Academic Press, New York.

Matheron, G. (1963). Principles of geostatistics. *Economic Geology* **58**, 1246-1266.

SEE ALSO

`loc`, `variogram`, `xyplot`.

EXAMPLES

```
anisotropy.plot(log(tcatch+1) ~ long + lat, data=scallops, lag=.075)
```

## `check.islands` — Detect Isolated Spatial Regions — `check.islands`

DESCRIPTION

Given an object of class `"spatial.neighbor"` detects spatial units that have no neighbors (islands).

USAGE

```
check.islands(x, remap=F)
```

REQUIRED ARGUMENTS

`x` an object of class `"spatial.neighbor"`.

OPTIONAL ARGUMENTS

`remap` logical flag: if there is an island, should we recode the indexing of the spatial contiguity matrix to eliminate the rows and columns with all zeroes? That is, should we renumber components `row.id` and `col.id` of the spatial neighbor object?

VALUE

if `remap=FALSE` the list of existing islands is returned. Otherwise, an object of class `"spatial.neighbor"` with remapped `row.id` and `col.id`.

SIDE EFFECTS

the attribute "nregion" of the output may differ from that of x whenremap=T.

SEE ALSO

spatial.neighbor, spatial.subset, spatial.weights

EXAMPLES

```
sids.nhbr2 <- check.islands(sids.neighbor,remap=T)
```

**find.neighbor** Find the Nearest Neighbors of a Point **find.neighbor**

DESCRIPTION

Find the k nearest neighbors of a vector x in a matrix of data contained in an object of class "quad.tree".

USAGE

```
find.neighbor(x, quadtree=quad.tree(x), k=1, metric="euclidean",
              max.dist=NULL, drop.self=F)
```

REQUIRED ARGUMENTS

x a vector (or matrix) containing the multidimensional point(s) at which the nearest neighbors are desired. The vector must have the same number of elements as the number of columns in the numeric matrix used to construct quadtree. If a matrix is used, the matrix must have the same number of columns as the numeric matrix used to construct quadtree, and nearest neighbors are found for each row in the matrix.

OPTIONAL ARGUMENTS

quadtree an object of class "quad.tree" containing the sorted matrix of data for which a nearest neighbor search is desired. Defaults to quad.tree(x) if x is a matrix but it is required when x is a vector.

k the number of nearest neighbors to be found. If the data x is the same data that was used to construct the "quad.tree" object, then k = 1 results in each element having itself as its own nearest neighbor.

metric a character string giving the metric to be used when finding "nearest" neighbors. Partial matching is allowed. Possible values are: "euclidean", "city block", and "maximum absolute value" for the $l_2$, $l_1$, and $l_\infty$ norm, respectively. For two vectors $x$ and $y$, these are defined as:

$$l_1 = \sum_i |x_i - y_i|,$$

$$l_2 = \sqrt{\sum_i (x_i - y_i)^{2)}},$$

$$l_\infty = \max_i |x_i - y_i|$$

max.dist if max.dist is given, argument k is ignored, and all of the neighbors within distance max.dist of each row in x are found.

drop.self a logical value, if TRUE then rows with distances equal to 0 and index1 == index2 (self neighbors) are dropped from the returned object. This definition retains coincident points as neighbors although their distance apart is zero. If quadtree is not supplied, k=1, and drop.self=T, a warning is printed (since this results in nothing being returned) and the value of k is set to 2.

VALUE

a matrix with three named columns:

`index1` if `x` is a matrix, the row in `x` for this nearest neighbor. If `x` is not a matrix, the value 1.
`index2` the row in the matrix from which the quad tree was formed for this nearest neighbor. If the quad tree was formed from a matrix `y`, then x[index1[i],] and y[index2[i],] are neighbors.
`distances` the corresponding nearest neighbor distances.

DETAILS

An efficient recursive algorithm is used to find all nearest neighbors. First the quad tree is traversed to find the leaf with medians nearest the point for which neighbors are desired. Then all observations in the leaf are searched to find nearest neighbors. Finally, if necessary, adjoining leaves are searched for nearest neighbors.

REFERENCES

Friedman, J., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software* **3**, 209-226.

SEE ALSO

`quad.tree`.

EXAMPLES

```
x <- cbind(sids$easting, sids$northing)
sids.nhbr <- find.neighbor(x, max.dist = 30)

# Find the nearest neighbors for the Lansing hickories
hickory <- lansing[lansing[,3] == "hickory", 1:2]
hickory.nhbr1 <- find.neighbor(hickory, k=2, drop.self=T)

# Now find the closest maple for each hickory
maple <- lansing[lansing[,3] == "maple", 1:2]
hmn <- find.neighbor(hickory, quad.tree(maple))
# and plot the tree locations with lines joing the neighbors
par(pty='s')
plot(maple[,1], maple[,2], pch=16)
points(hickory[,1], hickory[,2], pch=1, col=2)
segments(hickory[hmn[,1],1], hickory[hmn[,1],2],
     maple[hmn[,2],1], maple[hmn[,2],2])
```

## `Glasgow.neighbor` Neighbors for Glasgow Mortality Rate Data `Glasgow.neighbor`

SUMMARY

An object of class `"spatial.neighbor"` containing the neighbor specification among the 87 community medicine areas in Glasgow, Scotland. The standardized mortality rate (SMR) values for this data are contained in `Glasgow.SMR`.

DATA DESCRIPTION

Four hundred and fifty neighbor relationships are specified. The neighbor relationships are not symmetric. See `spatial.neighbor.object` for a description of the data within an object of class `"spatial.neighbor"`.

SOURCE

The data are presented and analyzed in Haining (1990).

REFERENCES

Haining, R. (1990). *Spatial Data Analysis in the Social and Environmental Sciences.* Cambridge University Press. Cambridge.

SEE ALSO

`Glasgow.SMR`.

---

**`Glasgow.SMR`** Standardized Mortality Rates for Glasgow **`Glasgow.SMR`**

---

SUMMARY

The `Glasgow.SMR` data frame contains standardized mortality rates for 87 community medicine areas in Glasgow, Scotland for 1980-1982.

DATA DESCRIPTION

This data frame contains the following columns:

| | |
|---|---|
| `AllDeaths` | the standardized mortality rate (SMR) for all deaths. |
| `Accidents` | the SMR for death by accidents. |
| `Cancer` | the SMR for deaths due to cancer. |
| `Respiratory` | the SMR for deaths due to respiratory disease accidents. |
| `Heart` | the SMR for deaths due to ischaemic heart disease. |
| `Cerebrovascular` | the SMR for deaths due to cerebrovascular disease. |
| `Population` | the population (in 1000's). |
| `Easting` | the x coordinate of the community medicine area (CMA) relative to an arbitrary origin, where the x-axis is parallel to the latitude. |
| `Northing` | the y coordinate of the CMA relative to an arbitrary origin, where the y-axis is parallel to the longitude. |

DETAILS

The standardized mortality rate for a community medicine area is the observed deaths due to that cause divided by the expected number of deaths given the age and sex combination in that area multiplied by 100.

SOURCE

The data are presented and analyzed in Haining (1990).

REFERENCES

Haining, R. (1990). *Spatial Data Analysis in the Social and Environmental Sciences.* Cambridge University Press. Cambridge.

SEE ALSO

`Glasgow.neighbor`.

**Kenv** Compute Simulations of `Khat` **Kenv**

DESCRIPTION

Computes `Khat` (`Lhat`) for simulations of point processes. Returns upper and lower bounds, as well as the average of all simulated values.

USAGE

```
Kenv(object, nsims=100, maxdist=<<see below>>, ndist=100,
     process="binomial", boundary=bbox(object), add=T, ...)
Lenv(object, nsims=100, maxdist=<<see below>>, ndist=100,
     process="binomial", boundary=bbox(object), add=T, ...)
```

REQUIRED ARGUMENTS

`object` an object of class `"spp"` representing a spatial point pattern, or a data frame or matrix with first two columns containing locations of a point pattern.

OPTIONAL ARGUMENTS

`nsims` integer. Number of desired simulations.

`maxdist` numeric value indicating the maximum distance at which `Khat` (or `Lhat`) should be estimated. Defaults to half the length of a diagonal of the sample's bounding box.

`ndist` desired number of default distances at which to compute `Khat` (or `Lhat`). Default is `100`.

`process` a character string with one of five possible processes for the spatial arrangement of the resulting pattern. This must be one of `"binomial"`, `"poisson"`, `"cluster"`, `"Strauss"`, or `"SSI"`. See the help file for `make.pattern` for information on parameters for each process.

`add` logical flag: should the envelope be added to an already existing plot of `Khat` (or `Lhat` for `Lenv`)? Defaults to `TRUE`.

`...` other parameters as needed by the requested process.

VALUE

invisibly returns a list with 4 numeric vectors each representing:

`dist` the distances at which all values were computed.

`lower` the minimum of all resulting `Khat` (or `Lhat` for `Lenv`) for the simulations.

`upper` the maximum of all resulting `Khat` (or `Lhat` for `Lenv`) for the simulations.

`average` the average of all resulting `Khat` (or `Lhat` for `Lenv`) for the simulations.

SIDE EFFECTS

if `add=TRUE` an envelope is added to an existing plot of `Khat`.

SEE ALSO

`Khat`, `Lhat`, `make.pattern`.

EXAMPLES

```
Khat(bramble)
Kenv(bramble,nsims=50)
Lhat(lansing)
Lenv(lansing,nsims=50)
```

**Khat** Ripley's K Function for a Spatial Point Pattern Object **Khat**

DESCRIPTION

Calculates `K(t)`, Ripley's K function for a spatial point pattern.

USAGE

```
Khat(object, maxdist=<<see below>>, ndist=100, boundary=bbox(object),
     plot.it=T)
```

REQUIRED ARGUMENTS

`object` an object of class `"spp"` representing a spatial point pattern, or a data frame or matrix with first two columns containing locations of a point pattern.

OPTIONAL ARGUMENTS

`maxdist` numeric value indicating the maximum distance at which `Khat` should be estimated. Defaults to half the length of a diagonal of the sample's bounding box.

`ndist` desired number of default distances at which to compute `Khat`. Default is `100`. The distances for which `Khat` will be estimated are calculated as `seq(0,maxdist,ndist)`, both `maxdist` and `ndist` will change if not reasonable for the given `object`.

`boundary` points defining the boundary polygon for the spatial point pattern. This version accepts only rectangles, for which `boundary` should be given as a list with named components `"x"` and `"y"` denoting the corners of the rectangular region. For example, for the unit square the boundary could be given as `bbox(x=c(0,1),y=c(0,1))`, the bounding box of two diagonally opposed points. Defaults to a rectangle covering the range of points.

`plot.it` logical flag: should the resulting `K`-estimates be plotted? Default is `TRUE`.

VALUE

a list containing components :

`values` a two column matrix. The first column, named `dist`, contains the distances at which `Khat` was computed, and the second column, named `Khat`, contains the values of `K(dist)`.

`ndist` number of distances returned. This could be smaller than its input value if the extent of the distances is too large.

`mindist` minimum distance between any pair of points.

`maxdev` maximum deviation from `K(t)=t`. See DETAILS.

SIDE EFFECTS

if `plot.it=TRUE`, a plot of the value of `K(t)` against distance will be produced on the current graphics device.

DETAILS

`Khat` computes Ripley's (1976) estimate of K(t) for a spatial point pattern:

$$K(t) = \lambda^{-1} E[\textit{number of events} \leq \textit{distance } t \textit{ of an arbitrary event}].$$

where $\lambda$ is the intensity of the spatial point pattern.

The theoretical K-function for a Poisson (completely spatially random) process is $K(t) = \pi t^2$, so $L(t) = \sqrt{K(t)/\pi}$ is equal to `t`, the distances. The default plots `K(t)` versus `t`. See function `Lhat` for estimation of `L(t)`.

REFERENCES

Ripley, Brian D. (1976). The second-order analysis of stationary point processes. *Journal of Applied*

*Probability*, **13**,255-266.

SEE ALSO

`Kenv`, `Lhat`.

EXAMPLES

```
lansing.spp <- as.spp(lansing)
lansing.khat <- Khat(lansing.spp)
Khat(wheat)
abline(0,1)
```

## `krige` Ordinary and Universal Kriging `krige`

DESCRIPTION

Performs ordinary or universal kriging for two dimensional spatial data. The function `predict.krige` can then be called to compute interpolation surfaces and prediction errors.

USAGE

```
krige(formula, data=sys.parent(), subset, na.action=na.fail,
      covfun, nc=10000, ...)
```

REQUIRED ARGUMENTS

`formula` a formula describing the kriging variable and the spatial location variables and optionally a polynomial trend surface. Its simplest form is:

```
z ~ loc(x,y)
```

where `z` is the kriging variable and `x` and `y` are the spatial locations, that is, `z[i]` is observed at the location (`x[i],y[i]`). The right hand side must contain a call to the function `loc`. A polynomial trend surface is of the form:

```
z ~ loc(x,y) + x + y + x^2 + y^2
```

The polynomial must be in the same variables as the first two arguments used in the `loc` function. A constant term is always fit. All terms on the right hand side must be entered with a `+` sign. The `loc` call can include arguments `angle` and `ratio` to correct for geometric anisotropy; see the `loc` help file. Note that an evaluated `loc` object cannot be used in `formula`.

`covfun` a function that returns the distanced based covariance between two points. The first argument to the function must be the distance. Additional parameters will be passed through the `...`.

OPTIONAL ARGUMENTS

`data` an optional data frame in which to find the objects mentioned in `formula`.

`subset` expression saying which subset of the rows of the data should be used in the fit. This can be a logical vector (which is replicated to have length equal to the number of observations), or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included.

`na.action` a function to filter missing data. This is applied to the data in `formula` after any `subset` argument has been used. The default (with `na.fail`) is to create an error if any missing values are found. A possible alternative is `na.omit`, which deletes observations that contain one or more missing values.

`nc` the number of points to use internally by the algorithm in approximating the distance-based covariance function. Note: this argument has nothing to do with the number of observed points used in computing the kriging. All observed points are used in computing kriging predictions.

`...` additional named arguments can be passed to `covfun`.

VALUE

an object of class `"krige"` with components:

`x` the first spatial location vector i.e. the first argument in `loc` function call in `formula`.

`y` the second spatial location vector i.e. the second argument in `loc` function call in `formula`.

`coefficients` the vector of coefficients for the trend surface. These are for the polynomial based on the scaled spatial location vectors (see the DETAILS section).

`residuals` the vector of residuals from the trend surface.

`call` an image of the call that produced the object.

Other components are included that are used by `predict.krige` for computing interpolations.

DETAILS

The kriging system is solved using generalized least squares (see Ripley, 1981). The polynomial terms are scaled to (-1, 1) internally to avoid numeric problems; the `coefficients` component returned is for these scaled terms.

This implementation of kriging does not handle multiple observations at a point.

Methods for objects of class `"krige"` include `predict` and `print`.

REFERENCES

Cressie, Noel A. C. (1993). *Statistics for Spatial Data*, Revised Edition. Wiley, New York.

Ripley, Brian D. (1981). *Spatial Statistics.* Wiley, New York

SEE ALSO

`exp.cov`, `loc`, `predict.krige`.

EXAMPLES

```
# krige the Coal Ash data with a quadratic trend in the x direction
#   using a spherical covariance function:
kcoal <- krige(coal ~ loc(x, y) + x + x^2, data = coal.ash,
        covfun = spher.cov, range = 4.31, sill = 0.14, nugget = 0.89)
# predictions over default 30 x 30 grid
pcoal <- predict(kcoal)
# plot prediction surface
wireframe(fit ~ x * y, data = pcoal,
        screen = list(z = 300, x = -60, y = 0), drape = T)
```

## Lhat — Ripley's K Function for a Spatial Point Pattern Object — Lhat

DESCRIPTION

Calculates `L(t)=sqrt(K(t)/pi)`, where `K(t)` is Ripley's K function for a spatial point pattern and `L(t)` is linear for a completely random point process.

USAGE

```
Lhat(object, maxdist=<<see below>>, ndist=100, boundary=bbox(object),
     plot.it=T)
```

REQUIRED ARGUMENTS

`object` an object of class `"spp"` representing a spatial point pattern, or a data frame or matrix with first two columns containing locations of a point pattern.

OPTIONAL ARGUMENTS

`maxdist` numeric value indicating the maximum distance at which `Lhat` should be estimated. Defaults to half the length of a diagonal of the sample's bounding box.

`ndist` desired number of default distances at which to compute `Lhat`. Default is `100`. The distances for which `Lhat` will be estimated are calculated as `seq(0,maxdist,ndist)`, both `maxdist` and `ndist` will change if not reasonable for the given `object`.

`boundary` points defining the boundary polygon for the spatial point pattern. This version accepts only rectangles, for which `boundary` should be given as a list with named components `"x"` and `"y"` denoting the corners of the rectangular region. For example, for the unit square the boundary could be given as `bbox(x=c(0,1),y=c(0,1))`, the bounding box of two diagonally opposed points. Defaults to a rectangle covering the range of points.

`plot.it` logical flag: should the resulting `K`-estimates be plotted? Default is `TRUE`.

VALUE

a list containing components :

`values` a two column matrix. The first column, called `dist`, contains the distances at which `Lhat` was computed, and the second column, called `Lhat`, contains the values of `L(dist)`.

`ndist` number of distances returned. This could be smaller than its input value if the extent of the distances is too large.

`mindist` minimum distance between any pair of points.

`maxdev` maximum deviation from `L(t)=t`. See DETAILS.

SIDE EFFECTS

if `plot.it=TRUE`, a plot of the value of `L(t)` against distance will be produced on the current graphics device.

DETAILS

`Khat` computes Ripley's (1976) estimate of K(t) for a spatial point pattern:

$$K(t) = \lambda^{-1} E[\textit{number of events} \leq \textit{distance } t \textit{ of an arbitrary event}].$$

where $\lambda$ is the intensity of the spatial point pattern.

The theoretical K-function for a Poisson (completely spatially random) process is $K(t) = \pi t^2$, so $L(t) = \sqrt{K(t)/\pi}$ is equal to `t`, the distances. The default plots `L(t)` versus `t` which should approximate a straight line for a homogeneous process with no spatial dependence. See function `Khat` for estimation of `K(t)`.

REFERENCES

Ripley, Brian D. (1976). The second-order analysis of stationary point processes. *Journal of Applied Probability* **13**,255-266.

SEE ALSO

`Lenv`, `Khat`.

EXAMPLES

```
lansing.spp <- as.spp(lansing)
lansing.khat <- Lhat(lansing.spp)

Lhat(wheat)
abline(0,1)
```

**make.pattern** Generate a Spatial Point Process **make.pattern**

DESCRIPTION

Generates points in two-dimensional space given their desired spatial distribution.

USAGE

```
make.pattern(n, process="binomial", object, boundary=bbox(x=c(0,1),
             y=c(0,1)), lambda, maxlambda, radius, cpar)
```

REQUIRED ARGUMENTS

`n` integer denoting the desired number of points in the resulting object.

OPTIONAL ARGUMENTS

`process` a character string with one of five possible processes for the spatial arrangement of the resulting pattern. This must be one of `"binomial"`, `"poisson"`, `"cluster"`, `"Strauss"`, or `"SSI"`. See the DETAILS section for each definition. Defaults to `"binomial"` for a completely spatially random process conditioned to `n` points within `boundary`. Partial matching is allowed.

`object` a spatial point pattern object. An object of class `"spp"`. When this is given, the resulting pattern has the same `n` and its `boundary` is that same as the bounding box of `object`.

`boundary` points defining the boundary polygon for the spatial point pattern. This version accepts only rectangles, for which `boundary` should be given as a list with named components `"x"` and `"y"` denoting the corners of the rectangular region. For example, for the unit square the boundary could be given as `bbox(x=c(0,1),y=c(0,1))`, the bounding box of two diagonally opposed points. Defaults to `bbox(object)` if `object` is given or to the unit square otherwise.

`lambda` the intensity when `process="poisson"`. If `lambda` is a numerical value then `make.pattern` simulates a two dimensional homogeneous Poisson process with that constant intensity. `lambda` can also be a function with two arguments that defines the intensity over the region. `n`, if given, will be ignored if this argument is provided.

`maxlambda` if `lambda` is a function then this should be the maximum value of the function over the region. If this is not supplied, a nonlinear optimization will be run (using `nlminb`) to find the maximum. Supplying this value will speed up the simulation and avoid any possible problems with the nonlinear optimization. `maxlambda` is used only if `lambda` is a function.

`radius` the inhibition distance. This is needed for process `"Strauss"`, `"SSI"` and `"cluster"`. Options `"Strauss"` and `"SSI"` will NOT generate points closer than `radius`. For this reason, this parameter needs to be reasonably small. The exception is when `process="cluster"` in which case it should contain the desired size of the clusters. See DETAILS section for more information.

`cpar` the inhibition parameter needed when `process="Strauss"`. This parameter is also required if `process="cluster"`. In that case, it represents the intensity of the "parent" Poisson process which will determine the random placement of clusters and their number. See the DETAILS section for more information.

VALUE

an object of class `"spp"` whose `n` points are distributed according to `process`. If `process="poisson"` results on a process with zero points, the returned value will be a classless matrix with zero rows and a warning will be issued.

DETAILS

The `"binomial"` process option generates a spatially random pattern of `n` points within the given `boundary`. This is in essence a homogeneous Poisson process conditional on the given number of points `n`.

The `"poisson"` process option generates a Poisson process with intensity `lambda`. This argument is required for this option. If `lambda` is a function the Poisson process is generated by a rejection sampling algorithm (Diggle, 1983): a homogeneous Poisson process with intensity `maxlambda` is generated over the region and then points are retained with probability `lambda(x, y)/maxlambda`.

The `"SSI"` process generates a random pattern where no two points are within the inhibition distance determined by its parameter `radius`. This process is equivalent to sequentially laying down discs of radius `radius` which will not overlap.

The `"Strauss"` process accepts each randomly generated point with probability `cpar^s` where `s` is the number of existing points within radius `radius` of the potential new point. The parameter `cpar` must be in [0,1] for this process, where `cpar=0` corresponds to complete inhibition at distances up to `radius`.

The user should exercise caution when determining the value of `radius`, for if it is too big in relation to the area defined by `boundary`, the algorithm will run out of possible area to place the subsequent disc and the generation of the desired process may be impossible or very slow.

The option `"cluster"` generates a Poisson cluster process. This is defined by generating a "parent" Poisson process with intensity `cpar` and a "daughter" process of clusters with radii determined by the value of `radius`.

WARNING

If `radius` is too large, it may be impossible or nearly impossible to generate the number of requested points. The call may "hang" in some extreme cases.

REFERENCES

Diggle, Peter J. (1983). *Statistical Analysis of Spatial Point Patterns.* Academic Press, London.

Ripley, Brian D. (1981). *Spatial Statistics.* John Wiley & Sons, New York.

Ripley, Brian D. (1976). The second-order analysis of stationary point processes. *Journal of Applied Probability* **13**,255-266.

SEE ALSO

`runif`, `rnorm`, `rpois`, `rbinom`.

EXAMPLES

```
# A completely random process in the unit square
rand <- make.pattern(100)

plot(make.pattern(100, process="Strauss", rad=0.1, c=0.5))

plot(make.pattern(500, proc="cluster", rad=20, c=10,
       boundary=list(x=c(0,200), y=c(0,200))))

# A nonhomogeneous Poisson pattern with a linear trend in x
#  over a 10 x 10 square
lxy <- function(x, y) 1.5*x
xy <- make.pattern(proc="poisson", boundary=bbox(x=c(0,10),
       y=c(0,10)), lambda=lxy)
plot(xy)
```

**`model.variogram`** Display a Variogram Object and Theoretical Model **`model.variogram`**

DESCRIPTION

Plots an empirical variogram object and displays the fit of a theoretical variogram model on that plot. Optionally allows interactive parameter updates to the theoretical model and displays the new fit.

USAGE

```
model.variogram(object, fun, ..., ask=T, objective.fun=<<see below>>,
                plot.it=T)
```

REQUIRED ARGUMENTS

`object` an object that inherits from class `"variogram"` (this includes classes `"covariogram"` and `"correlogram"`). The `azimuth` column should have only one level.

`fun` a theoretical variogram function (or covariogram or correlogram function, depending on the class of `"object"`). Its first argument should be distance. Its remaining arguments are considered parameters that can be changed to update the fit of `fun` to `object`.

OPTIONAL ARGUMENTS

`...` additional arguments to `fun` that do not have default values must be specified here by full name.

`ask` a logical value, if `TRUE`, a command line menu is displayed allowing the user to change the values of the parameters to `fun`. After changing a value the plot is updated. If `FALSE`, the data in `object` is plotted, the value of `fun` evaluated at `object$distance` is added to the graph, and the function returns.

`objective.fun` a function with three arguments, `y`, `yf`, and `n` that gives a measure of the fit of `yf` to `y` with weights `n`. It is used as a measure of fit of `fun` to the data in `object`. The default is the sum of squared residuals, `sum((y-yf)^2)`.

`plot.it` a logical value, if `TRUE`, a plot of the variogram and its fitted model is displayed.

VALUE

invisibly returns a named list of the final parameters used. This list has the last value of the objective function as an attribute.

DETAILS

This function can be used to fit a variogram or covariogram model "by eye". The value of `objective.fun` is displayed on the plot.

A weighted least squares objective function for variograms (Cressie, 1993, p. 97) is:

```
objective.fun <- function(y,yh,n) sum(n*(y/yh-1)^2)
```

REFERENCES

Cressie, Noel. (1993). *Statistics For Spatial Data*, Revised Edition. Wiley, New York.

SEE ALSO

`correlogram`, `plot.variogram`, `variogram`.

EXAMPLES

```
vg.iron <- variogram(residuals ~ loc(easting, northing), data=iron.ore)
model.variogram(vg.iron, spher.vgram, range=8.7, sill=3.5, nugget=4.8)
```

**plot.spatial.neighbor** Plot a spatial.neighbor Object **plot.spatial.neighbor**

DESCRIPTION

Plot an object of class "spatial.neighbor" with lines connecting points that are neighbors.

USAGE

```
plot.spatial.neighbor(x, xcoord, ycoord, line.col=1, line.type=1,
                      line.width=1, matrix.id=1, add=F, arrows=F,
                      size.arrow=0.1, scaled=T, ...)
```

REQUIRED ARGUMENTS

`x` an object of class "spatial.neighbor".

`xcoord` a numeric vector containing the x-coordinates of the data whose neighbor relations are defined in `x`.

`ycoord` a numeric vector containing the y-coordinates of the data whose neighbor relations are defined in `x`. Must be the same length as `xcoord`.

OPTIONAL ARGUMENTS

`line.col` a numeric value indicating the color to draw the lines connecting the points that are neighbors. See the `col` parameter in the `par` help file.

`line.type` a numeric value indicating the line type to use for the lines connecting the points that are neighbors. See the `lty` parameter in the `par` help file.

`line.width` a numeric value indicating the line width to use for the lines connecting the points that are neighbors. See the `lwd` parameter in the `par` help file.

`matrix.id` a positive integer indicating which spatial neighbor matrix is to be plotted. Only one spatial neighbor matrix can be plotted per call to the function but objects of class "spatial.neightbor" can contain more than one matrix.

`add` a logical value, if `TRUE` no initial plot is drawn, only the lines joining the neighbors are added to the current plot.

`arrows` a logical value, if `TRUE`, arrows are drawn from each point to its neighbor, if `FALSE`, segments are drawn from each point to its neighbor. Plotting with arrows can be useful when there are one way neighbor relations in `x` i.e. point B is a neighbor of point A but point A is not a neighbor of point B. If `x` is a symmetric spatial neighbor object, (`attr(x,symmetric)` is `TRUE`) then all neighbor relations are bi-directional and setting `arrows=TRUE` just results in a messy graph.

`size.arrows` the size of the arrowhead width in inches. See the `arrows` help file for details.

`scaled` a logical value, if `TRUE` then `scaled.plot` is used to set up the plot coordinates instead of `plot`. This produces an equally scaled plot which is often useful when `xcoord` and `ycoord` are geographic locations.

Graphical parameters may also be supplied as arguments to this function (see `par`).

SIDE EFFECTS

a plot is produced on the current graphics device or lines are added to the current plot (if `add=T`).

DETAILS

The coordinate system for the plot is drawn based on the values in `xcoord`, `ycoord`. The graphical parameters specified in `...` are used to draw this initial graph. If `scaled=TRUE` the `scale.ratio` parameter to `scaled.plot` can also be passed in the `...` arguments. The lines are added through a call to `segments` or `arrows`. The graphical parameters `line.col`, `line.type` and `line.width` are used in the call to `segments` or `arrows`.

This function is a method for the generic function `plot` for class `spatial.neighbor` . It can be invoked by calling `plot` for an object of the appropriate class, or directly by calling `plot.spa-`

`tial.neighbor` regardless of the class of the object.

BUGS

With S-PLUS 5.1, if you are calling this function as a plot method (i.e. `plot`) you must specify the `xcoord` and `ycoord` arguments by name (not by position) otherwise the wrong method will be called.

SEE ALSO

`spatial.neighbor`, `spatial.neighbor.object`, `plot`, `scaled.plot`, `par`, `segments`.

EXAMPLES

```
# Plot the sids.neighbor object using the easting and northing
#   values from sids as the coordinates
plot(sids.neighbor, xc=sids$easting, yc=sids$northing, scaled=T)

# Create a second order spatial neighbor object on a 10 x 10 grid
ng10 <- neighbor.grid(10, 10, neighbor.type="second.order")
sn10 <- spatial.neighbor(ng10)
# Generate a 10 x 10 set of coordinates
xy <- expand.grid(x=1:10, y=1:10)
# Plot the spatial neighbor object
plot(sn10, xc=xy$x, yc=xy$y)

# Create and plot spatial neighbor object for the bramble canes
#   nearest neighbors
nb <- find.neighbor(bramble, k=2, drop.self=T)
sn <- spatial.neighbor(nb)
plot(sn, xc=bramble$x, yc=bramble$y)
```

## `plot.vgram.fit` Plot Results from `variogram.fit` `plot.vgram.fit`

DESCRIPTION

Plot a `vgram.fit` object, ustually the result from a call to `variogram.fit`.

USAGE

```
plot.vgram.fit(x, line.col=1, line.type=1, line.width=1, add=T,
               npoints=100, ...)
```

REQUIRED ARGUMENTS

`x` an object of class `"vgram.fit"`.

OPTIONAL ARGUMENTS

`line.col` a numeric value indicating the color for the variogram fit line. See the `col` parameter in the `par` help file.

`line.type` a numeric value indicating the line type for the variogram fit line. See the `lty` parameter in the `par` help file.

`line.width` a numeric value indicating the line width for the variogram fit line. See the `lwd` parameter in the `par` help file.

`add` a logical value, if `TRUE` no initial plot is drawn, only the variogram fitted line is added to the current plot.

`npoints` a numeric value, the number of to evalute the variogram function at.

Graphical parameters may also be supplied as arguments to this function (see `par`).

SIDE EFFECTS

a plot is produced on the current graphics device or lines are added to the current plot (if `add=T`).

DETAILS

The function specified by `x$funName` must exist. It is evaluated at `npoints` between `0` and `x$distRange[2]`.

This function is a method for the generic function `plot` for class `vgram.fit` . It can be invoked by calling `plot` for an object of the appropriate class, or directly by calling `plot.vgram.fit` regardless of the class of the object.

SEE ALSO

`variogram.fit`, `variogram`.

EXAMPLES

```
vg.iron <- variogram(residuals ~ loc(easting, northing), data=iron.ore)
vfit.iron <- variogram.fit(vg.iron, param=c(range=8.7, sill=3.5,
          nugget=4.8), fun=spher.vgram)
plot(vg.iron)
plot(vfit.iron, add=T)
```

## `points.in.poly` Find Points Inside a Given Polygon `points.in.poly`

DESCRIPTION

Determine whether points are inside a polygon.

USAGE

```
points.in.poly(x, y, polygon)
```

REQUIRED ARGUMENTS

| | |
|---|---|
| `x` | the X-coordinates of the points |
| `y` | the Y-coordinates of the points. Must be the same length as `x`. |
| `polygon` | a list with named components "x" and "y". |

VALUE

a logical vector the same length as `x`. If `TRUE` then the corresponding point is inside the given polygon and so on.

BUG

if a ray from a point to an edge intersects a horizontal edge, i.e. is collinear with it, the C program will return `TRUE` even if such point is not in the polygon.

SEE ALSO

`poly.grid`, `poly.area`.

EXAMPLES

```
# 100 points on a unit square
x <- runif(100); y <- runif(100)
# A square polygon in the center:
pcenter <- list(x=c(.25,.25,.75,.75), y=c(.25,.75,.75,.25))
```

```
pin <- points.in.poly(x, y, pcenter)
# Plot the unit square and the center square:
plot(x, y, type='n'); polygon(pcenter, density=0, col=2)
# Plot only the points in the center square:
points(x[pin], y[pin], col=3)
```

**poly.grid** Generate a Grid Inside a Given Polygonal Boundary **poly.grid**

DESCRIPTION

Generates a grid of points and then clips them to lie within a given boundary.

USAGE

```
poly.grid(boundary, nx, ny, size)
```

REQUIRED ARGUMENTS

`boundary` a list with components named `"x"` and `"y"` or a matrix with 2 columns representing the vertices of a convex polygon. Endpoint need not be repeated.

`nx` integer representing the number of cells in the horizontal direction.

`ny` integer representing the number of cells in the vertical direction.

OPTIONAL ARGUMENTS

`size` numeric vector containing the size of each cell. If it has length one then the cells will be squared with the same side sizes. If it has length two then the cells will have width `size[1]` and height `size[2]`.

VALUE

a two-column matrix containing the coordinates of the resulting grid.

DETAILS

A rectangular `nx` by `ny` grid is overlaid on the polygon defined by `boundary` and then those points that fall outside are dropped. If `size` is given then the values `nx` and `ny` are redundant and if given will be ignored.

SEE ALSO

`points.in.poly`

EXAMPLES

```
plot(as.spp(bramble))
bramble.chull <- bramble[chull(bramble),]
polygon(bramble.chull, den=0)
points(poly.grid(bramble.chull, size=c(.1,.1)), col=2)
```

**predict.krige** Point and Block Kriging Prediction **predict.krige**

DESCRIPTION

Computes point or block kriging predictions and standard errors at locations in `newdata` using an object returned by `krige`.

USAGE

```
predict.krige(object, newdata, se.fit=T, grid=<<see below>>,
              blocksize=c(1, 1), nxy=c(1, 1))
```

REQUIRED ARGUMENTS

`object` an object of class `"krige"` as returned by the function `krige`.

OPTIONAL ARGUMENTS

`newdata` a data frame or list containing the spatial locations for the predictions. The names must match the names of the locations used in the call to `krige` (see `attr(object,"call")`).

`se.fit` a logical value, if `TRUE`, the standard errors of the predictions are returned. Currently the standard errors are always computed internally. This `se.fit` only determines if the returned data frame includes the `se` column.

`grid` a list containing two vectors, the names of the vectors must match the names of the locations used in the call to `krige`. The vectors are each of length 3 and specify the minimum, maximum and number of locations in that spatial coordinate, respectively. A grid is then computing using `expand.grid`. The default value is to use the range of the original location data for the minimum and maximum, and 30 points. This argument is ignored if `newdata` is supplied.

`blocksize` for block kriging, a numeric vector of length 2 specifying the size of the block in x (first value) and y (second value) direction. The locations specified by `newdata` or `grid` are at the center of the blocks.

`nxy` for block kriging, a numeric vector of length 2 specifying the number of discretization points inside the block. If both values are set to `1` (the default) then point kriging predictions are computed.

VALUE

a data frame where the first two columns are the locations of the prediction along with:

`fit` the predicted values.

`se.fit` the standard error of the prediction. Only included if `se.fit = TRUE`.

DETAILS

This function is a method for the generic function `predict` for class `krige` . It can be invoked by calling `predict` for an object of the appropriate class, or directly by calling `predict.krige` regardless of the class of the object.

REFERENCES

Ripley, Brian D. (1981). *Spatial Statistics.* Wiley, New York.

SEE ALSO

`krige`, `loc`.

EXAMPLES

```
# krige the Coal Ash data
kcoal <- krige(coal ~ loc(x, y) + x + x^2, data = coal.ash,
        covfun = spher.cov, range = 4.31, sill = 0.14, nugget = 0.89)
# predictions over default 30 x 30 grid
pcoal <- predict(kcoal)
```

```
# plot prediction surface
wireframe(fit ~ x * y, data = pcoal,
        screen = list(z = 300, x = -60, y = 0), drape = T)
# block kriging predictions with block of size 2 x 2 at 4 locations
predict(kcoal, data.frame(x=c(4,5,9,11), y=c(7,13,9,18)),
       blocksize=c(2,2), nxy=c(5,5))
```

## spatial.neighbor — Create a "spatial.neighbor" Object — spatial.neighbor

DESCRIPTION

Function used to create an object of class "spatial.neighbor" given its component parts.

USAGE

```
spatial.neighbor(row.id, col.id, weights=rep(1, length(row.id)),
                 neighbor.matrix, nregion=max(c(row.id,col.id)),
                 symmetric=F, matrix.id=<<see below>>)
```

REQUIRED ARGUMENTS

`row.id` an integer vector containing the row indices of the non-zero elements of the neighbor weight matrix. The `i`-th element of `row.id` and the `i`-th element of `col.id` specify two regions which are spatial neighbors. Two regions are spatial neighbors if observations from the two regions have a non-zero spatial weight and vice-versa. `row.id` can also be a two column matrix containing the row indices (the first column) and the column indices (the second column). This argument is ignored if `neighbor.matrix` is given.

`col.id` integer vector (of the same length as `row.id`) with the column indices of the non-zero elements of the neighbor weight matrix. This is ignored if `neighbor.matrix` is given or if `row.id` is a matrix.

It is important to note that even if a pair of regions `c(row.id[i],col.id[i])` are spatial neighbors, the permuted pair `c(col.id[i],row.id[i])` does not have to define spatial neighbors (corresponding contiguity matrix element can be zero). For example, consider two regions on a river, and suppose that a region corresponding to `row.id[i]` is downstream from the region in `col.id[i]` and neighbors. By this definition, "downstream of" the transpose pairing need not satisfy a neighbor relationship. See argument `symmetric` below.

`neighbor.matrix` a matrix of neighbor weights (where all weights are often `1`) from which the object of class "`spatial.neighbor`" is to be constructed. This must be a square matrix such that if element `[i,j]` is non-zero, then spatial regions `i` and `j` are considered neighbors, and its value is used as a weight in measures of correlation or in further model-fitting. This is also known as the contiguity matrix.

OPTIONAL ARGUMENTS

`weights` numeric vector of the same length as `row.id` and `col.id`. `weights[i]` gives a weight for the corresponding neighbor pair relationship, given in `c(row.id[i],col.id[i])`. If `weights` is not specified (and argument `neighbor.matrix` is not used), then the spatial weights are all set equal to `1`. Each spatial weight defines the strength of the association between two neighbors. This argument is ignored if `neighbor.matrix` is given as each of the matrix elements are then considered to be neighbor weights.

`nregion` integer stating the total number of regions or spatial units. If not given, this value is computed from the number of unique elements in `row.id` and `col.id` as the maximum of all the regions given therein `max(c(row.id,col.id))`.

`symmetric` logical flag: should the neighbor matrix be considered symmetric?. If `TRUE`, the spatial weights matrix is computed by assuming that if the i-th neighbor pair `c(row.id[i],col.id[i])` has neighbor weight given by `w=weights[i]` then so does the matrix element `c(col.id[i],row.id[i])`. Only half of the weights need be specified in this case. If `TRUE`, routine `spatial.condense` is called to re-

move redundant values. When `neighbor.matrix` is given, its symmetry is determined within the function, otherwise, it defaults to `FALSE`.

`matrix.id` integer vector of length equal to the total number of spatial neighbors. This can be used to differentiate various types of neighbors. For example, spatial regression models may differentiate between north-south neighbors as compared to east-west neighbors. The values of vector `matrix.id` should then indicate the neighbor types. If missing, a single neighbor type is assumed (with one neighbor matrix).

VALUE

an object of class `"spatial.neighbor"`. This object inherits from class `"data.frame"` and describes the relationship among spatial regions using a sparse representation of the Weight or Contiguity matrix (or matrices). It has columns `row.id`, `col.id`, `weights` and `matrix` (determined by `matrix.id`).

DETAILS

Objects of class `"spatial.neighbor"` are required by the spatial regression, spatial correlation, and other functions in SPATIALSTATS. Two methods for constructing a spatial neighbor object are available. A matrix of weights (where all weights are often `1`) can be given as input, and the `"spatial.neighbor"` object is constructed from its non-negative elements. In this case argument `neighbor.matrix` must be a square matrix such that if element `c(i,j)` of the matrix is non-zero, then spatial regions `i` and `j` are neighbors, with weight given by the value of the element (usually a `1`).

Another method for constructing an object of class `"spatial.neighbor"` is by directly specifying the row and column numbers (and the weight value) of the non-zero elements of the contiguity matrix which is usually a sparse matrix. A sparse representation is usually preferred in practice. In this case, `row.id[i]` gives the row of the `i`-th non-negative element of the neighbor matrix, and the corresponding element `col.id[i]` gives its column index. Thus, each pair `c(row.id[i],col.id[i])` represents a pair of neighboring spatial units. The strength of their association can then be given by `weights[i]`.

Notice that `row.id` and `col.id` contain INDICES of the contiguity matrix and NOT the region identifiers which could be character strings or some such. These are used to expand the full contiguity matrix, so we should have representation for all indices `1` through `nregion`, though it is possible to have islands in between. Use the function `check.islands` to check for these islands, and remap their indexing if that is desirable.

It is possible to specify two or more types of neighbor relationships. For example, the user may want to model a spatial relationship depending upon the angle of the line connecting neighbor centers i.e. considering directional relationships. For this example, let Type-1 neighbors be north-south neighbors, and let Type-2 neighbors be east-west neighbors; neighbors along a diagonal could be modeled with weights proportional to `.707` (the sine of 45 degrees), for instance.

Consider the elements of `row.id`, `col.id`, and `weights` corresponding to a distinct value, `k`, of the vector `matrix.id`. The spatial neighbor matrix can be expressed as a matrix `A[k]` such that `A[k][row.id,col.id]=weights`, and all other elements are zero. Consider a parameter vector `rho` of length `g`, many spatial covariance matrices used in spatial regression models can be expressed as a weighted linear combination of the contiguity matrices `A[k]`, `rho[k]*A[k]`, for values of `k` varying in `1:g`.

SEE ALSO

`check.islands`, `plot.spatial.neighbor`, `summary.spatial.neighbor`.

EXAMPLES

```
row.index <- c(1,1,2,2,3)
col.index <- c(2,3,1,3,4)
```

```
# Assume we have no information about the strength of the spatial
# association.  All weights are 1.
nghb <- spatial.neighbor(row.id=row.index, col.id=col.index)
summary(nghb)
# Another way to create the same spatial.neighbor object:
nmat <- matrix(c(0, 1, 1, 0,
                 1, 0, 1, 0,
                 0, 0, 0, 1,
                 0, 0, 0, 0), ncol=4, byrow=T)
nghb2 <- spatial.neighbor(neighbor.matrix=nmat)
```

## `spatial.neighbor.object` Class `"spatial.neighbor"` `spatial.neighbor.object`

DESCRIPTION

Class of objects used to define neighbor relationships for spatial data on a regular or irregular lattice.

GENERATION

This class of objects is constructed using the function `spatial.neighbor`. Alternatively, the functions `read.neighbor`, or `neighbor.grid` may be used. In general, the user must construct these objects whenever estimates of spatial correlation and spatial regression are desired.

An object of class `"spatial.neighbor"` contains all the information required to determine which spatial units on a region of interest are neighbors, as well as the strength of their relationship.

METHODS

The class `"spatial.neighbor"` has associated methods, `print.spatial.neighbor`, `plot.spatial.neighbor`, and `summary.spatial.neighbor`.

INHERITANCE

Class `"spatial.neighbor"` inherits from class `"data.frame"`.

STRUCTURE

The `"spatial.neighbor"` object is in essence a data frame with additional attributes. Each row of the data frame denotes a pair of neighboring spatial units. The data frame contains the following columns:

- `row.id` the row index in the neighbor matrix that corresponds to a region or spatial unit. This implies a numbering of regions from `1` to the total number of regions.
- `col.id` the column index in the neighbor matrix that corresponds to the neighbor of the region defined by the corresponding element of `row.id`.
- `weights` a numeric value giving the relative strength of the neighbor relationship. The larger the value, the stronger the relationship.
- `matrix` if multiple types of neighbor matrices are possible, this column contains the type of the neighbor this weight represents - it gives a numeric identifier for each spatial neighbor [contiguity] matrix.

SPECIAL ATTRIBUTES

- `nregion` the number of total regions in the study. The row and column identifiers given in `row.id` and `col.id` might not include ALL the spatial units in the area of interest. This happens when units are isolated, i.e. have no neighboring regions. In this case, `nregion` must be used to determine the total number of rows and columns in the contiguity matrix.

`symmetric` logical flag. It provides an indication of whether the contiguity matrix is symmetric (`TRUE`) or not (`FALSE`). If `TRUE`, only the weights for the upper (or lower) triangle of the contiguity matrix need to be specified in the object. Use the function `spatial.weights` to expand the full symmetric weights matrix.

DETAILS

An object of class `"spatial.neighbor"` is a sparse matrix representation of a square matrix (or a number of square matrices).

The function `plot.spatial.neighbor` will show a graphical view of the spatial.neighbor object and `summary.spatial.neighbor` will compute summary statistics on the object.

The functions `spatial.multiply`, and `spatial.cg.solve` can be used to form products of the form `rho[i]*N[i]*x` and `(rho[i]*N[i])^(-1)*x`, for neighbor weight matrices `N[i]`, vector of constants or parameters, `rho[i]`, and arbitrary vectors `x`, should that be needed to form a neighbor or contiguity matrix as a weighted linear combination of others.

SEE ALSO

`spatial.neighbor plot.spatial.neighbor`, `summary.spatial.neighbor`, `read.neighbor`, `neighbor.grid`, `spatial.multiply`, `spatial.cg.solve`, `spatial.weights`.

## `spatial.solve` — Solve `Sb=x` — `spatial.solve`

DESCRIPTION

Solves `Sb=x` for `b`, where `S` is a sparse matrix obtained from an object of class `"spatial.neighbor"`.

USAGE

```
spatial.solve(neighbor, x, transpose=F, rho=0, product=F,
              weights=NULL, region.id=NULL, absThreshold=0,
              relThreshold=0, diagPivoting=0, shareMemory=F)
```

REQUIRED ARGUMENTS

`neighbor` an object of class `"spatial.neighbor"` containing the sparse matrix representation of the spatial neighbor matrix (or matrices, see function `spatial.neighbor`).

`x` the right hand side for which a solution is desired. Alternatively, `x` can be a matrix. In this case, a solution is obtained for each column in `x`.

OPTIONAL ARGUMENTS

`transpose` with the default arguments, `S` is taken as `I` minus the sum over `i` of `rho[i] * A[i]`. Here `I` is an identity matrix, `rho[i]` is a scalar, and `A[i]` is the `i`-th weight matrix in `neighbor`. If `transpose` is `TRUE`, then the transpose of this matrix is used for `A`.

`rho` a scalar (or vector) of constants used in defining the matrix `S` (see argument `transpose`).

`product` let `B=I` minus the sum of `rho[i]*A[i]` as described in argument `transpose`. When `product` is `FALSE`, `S=B`. When `product` is `TRUE`, `S` is `t(B)%*%B`.

`weights` if provided, the inverse weights are included along the diagonal matrix `W` and incorporated into the model for `S` as follows: Let `R` be `I` minus the sum of `rho[i]*A[i]`. Then

| product | transpose | S |
|---|---|---|
| F | F | `R %*% W` |
| F | T | `t(R) %*% W` |
| T | F | `t(R) %*% W %*% R` |

```
                T   |   T   | R %*% W %*% t(R)
```

`region.id` a vector with length equal to the number of regions in the spatial lattice. If variables `row.id` and `col.id` of argument `neighbor` are not integer valued variables with sequential values from 1 to the number or regions in the lattice, then argument `region.id` must be specified and is used to obtain a sequential coding of the lattice regions.

`absThreshold` the pivot threshold (between zero and 1). Values near 1 result in complete pivoting, while values near zero result in a strict Markowitz solution. In general, you should choose a value as close to zero as roundoff error will permit. A value of 0.001 has been recommended by Kundert (1988) in some cases.

`relThreshold` the absolute magnitude an element must have to be considered as a pivot candidate, except as a last resort. This should be set to a small fraction of the smallest (absolute) diagonal element.

`diagPivoting` if `TRUE`, pivot selection should be confined to the diagonal if possible.

`shareMemory` if `TRUE`, the in-memory representation of the sparse matrix will be shared by other routines. If memory is shared, it needs to be released later. One way to release the memory is to call `.C("destroy_sparse_matrix")` after the in-memory representation of the matrix is no long needed. Most users should use the default value, `FALSE`.

VALUE

a matrix (or vector), `b`, solving the linear system `Sb=x`.

DETAILS

This routine uses the sparse matrix code of Kenneth Kundert and Alberto Sangiovanni-Vincentelli (1988). The University of California, Berkeley, holds the copyright for these routines.

REFERENCES

Kundert, Kenneth S. and Sangiovanni-Vincentelli, Alberto (1988). A Sparse Linear Equation Solver. Department of EE and CS, University of California, Berkeley.

SEE ALSO

`spatial.cg.solve`, `spatial.multiply`, `spatial.neighbor`, `spatial.neighbor.object`.

EXAMPLES

```
x <- 1:4
row.id <- c(1,1,2,2,3)
col.id <- c(1,3,1,3,4)
alpha <- 0.3
neighbor <- spatial.neighbor(row.id=row.id, col.id=col.id, symmetric=T)
a <- solve(diag(attr(neighbor, "nregion"))-alpha*
       spatial.weights(neighbor), x)
b <- spatial.solve(neighbor, x, rho=alpha)$result
print(max(abs(a-b)) < 1e-14)
```

## `summary.spatial.neighbor` Summary Method `summary.spatial.neighbor`

DESCRIPTION

Returns a summary list for objects of class `"spatial.neighbor"`.

USAGE

```
summary.spatial.neighbor(object)
```

REQUIRED ARGUMENTS

`object` an object that inherits from class `"spatial.neighbor"`.

VALUE

an object of class `"summary.spatial.neighbor"` which is a list of lists, one list for each unique value of `matrix` in `object`. The sublists each contain the components that summarize the particular spatial neighbor matrix:

`nregion` an integer indicating the number of regions `object` covers. This is the same as `attr(object,"nregion")`

`symmetric` a logical value, if `TRUE` the `object` is assumed to be symmetric. This is the same as `attr(object,"symmetric")`

`minConnected` a named vector of the least connected regions. The names are the row indices that have the smallest number of connections for the i-th matrix in `object`. The values (all the same) are the minimum number of neighbors.

`maxConnected` a named vector of the most connected regions. The names are the row indices that have the largest number of connections for the i-th matrix in `object`. The values (all the same) are the maximum number of neighbors.

`aveNumLinks` a single value giving the mean number of neighbors each region has.

`aveWeight` a single value giving the mean weight value for this matrix.

`rowMissing` a vector of indices that are not present in `object$row.id` for the i-th matrix. This will be printed as `"none"` by the print method if there are no missing row indices and it is not printed at all if `object` is a symmetric spatial neighbor matrix since all missing row indices will be islands (see below).

`colMissing` a vector of indices that are not present in `object$col.id` for the i-th matrix. This will be printed as `"none"` by the print method if there are no missing column indices and it is not printed at all if `object` is a symmetric spatial neighbor matrix since all missing column indices will be islands (see below).

`islands` the indices for regions that have no neighbors. These indices do not appear in either the `object$col.id` or `object$row.id` for the i-th matrix. This will be printed as `"none"` by the print method if there are no islands.

DETAILS

This function is a method for the generic function `summary` for class `spatial.neighbor` . It can be invoked by calling `summary` for an object of the appropriate class, or directly by calling `summary.spatial.neighbor` regardless of the class of the object.

SEE ALSO

`spatial.neighbor`, `check.islands`.

EXAMPLES

```
summary(sids.neighbor)

# Create two symmetric spatial neighbor matrices with one island
#   in the second matrix:
ri <- c(1,1,2,3,4,5,1,1,2,5,5)
ci <- c(2,3,3,4,5,6,2,3,3,3,6)
mat <- c(1,1,1,1,1,1,2,2,2,2,2)
sn <- spatial.neighbor(ri, ci, symm=T, matrix=mat)
summary(sn)
```

**`triangulate`** Delaunay's Triangulation **`triangulate`**

DESCRIPTION

Calculate Delaunay's triangulation for points with given coordinates `x` and `y`.

USAGE

```
triangulate(x, y, plot.it=T, shrink=0.1)
```

REQUIRED ARGUMENTS

`x` a list with components `"x"` and `"y"`, a 2-column matrix, or a vector containing the horizontal coordinates of the vertices that form the polygon of interest.

OPTIONAL ARGUMENTS

`y` if `x` is a vector of X-coordinates then `y` must contain the corresponding vertical or Y-coordinates.

`plot.it` logical flag: should the resulting triangulation be plotted? Default is `TRUE`.

`shrink` fraction by which the triangles will be shrunken for better discrimination of the individual triangles in the plot, no edges overlap if `shrink > 0`.

VALUE

invisibly returns a list with 2 components:

`ipt` a matrix with 3 rows, for each column the 3 row-values can be used to index `x` and `y` and extract corresponding triangle vertices. This provides an ordering of the triangles as well.

`ipl` another integer matrix with 3 rows. These are the point numbers of the end points of the border line segments and their corresponding triangle number.

SIDE EFFECTS

if `plot.it = TRUE` a colorful representation of the triangulation is produced.

DETAILS

A Delaunay triangulation of a point set is a triangulation whose vertices are the point set, with the property that no point in the point set falls in the interior of the circumcircle (circle that passes through all three vertices) of any triangle in the triangulation.

EXAMPLES

```
triangulate(scallops[,c("lat","long")])
```

**`variogram.cloud`** Calculate Variogram Cloud **`variogram.cloud`**

DESCRIPTION

Calculates all pairwise differences in a random field data set.

USAGE

```
variogram.cloud(formula, data=<<see below>>, subset=<<see below>>,
                na.action=<<see below>>, azimuth=0, tol.azimuth=90,
                maxdist=<<see below>>, bandwidth=1e+307,
                FUN=function(zi, zj) (zi - zj)^2/2))
```

REQUIRED ARGUMENTS

`formula` formula defining the response and the predictors. In general, its form is:

```
z ~ x + y
```

The `z` variable is a numeric response. Variables `x` and `y` are the locations. All variables in the formula must be vectors of equal length with no missing values (`NA`s). The formula may also contain expressions for the variables, e.g. `sqrt(count)` or `log(age+1)`. The right hand side may also be a call to the `loc` function e.g. `loc(x,y)`. The `loc` function can be used to correct for geometric anisotropy, see the `loc` help file.

OPTIONAL ARGUMENTS

`data` an optional data frame in which to find the objects mentioned in `formula`.

`subset` expression saying which subset of the rows of the data should be used in the fit. This can be a logical vector (which is replicated to have length equal to the number of observations), or a numeric vector indicating which observation numbers are to be included, or a character vector of the row names to be included.

`na.action` a function to filter missing data. This is applied to the `model.frame` after any `subset` argument has been used. The default (with `na.fail`) is to create an error if any missing values are found. A possible alternative is `na.omit`, which deletes observations that contain one or more missing values.

`azimuth` the clockwise direction angle in degrees from North-South. Only pairs of points in this direction plus or minus `tol.azimuth` will be included in the output.

`tol.azimuth` the tolerance angle, in degrees. `tol.azimuth` greater than or equal to 90 implies the of use all directions.

`maxdist` the maximum distance to consider. The default is half the maximum observed distance.

`bandwidth` the maximum perpendicular distance to consider.

`FUN` a function of two variables that is to be computed. The default function is the contribution to the classical empirical variogram for the pair `z[i]`, `z[j]`.

VALUE

an object of class `"vgram.cloud"` that inherits from `"data.frame"`. The columns are:

`distance` the distance between the two points.

`gamma` the value of `FUN` for the `z[iindex]`, `z[jindex]`.

`iindex` the index into the original data for the first value of the pair.

`jindex` the index into the original data for the second value of the pair.

The return object has an attribute `call` with an image of the call that produced the object.

DETAILS

Methods for class `"vgram.cloud"` include `boxplot`, `plot` and `identify`.

If all directions and distances are included the return object will have n*(n-1)/2 rows where n is the number of observations. This can get very large, even for relatively small n. The argument `maxdist` can be used to limit the size. Typically values beyond half the maximum distance in the data are not used in estimating the variogram function.

REFERENCES

Cressie, Noel. (1993). *Statistics For Spatial Data*, Revised Edition. Wiley, New York.

SEE ALSO

`boxplot.vgram.cloud`, `identify.vgram.cloud`, `plot.vgram.cloud`, `variogram`.

EXAMPLES

```
v1 <- variogram.cloud(coal ~ x + y, data=coal.ash)
plot(v1)
```

```
boxplot(v1)
```

**variogram.fit** Fit a Variogram Model **variogram.fit**

DESCRIPTION

Fits a theoretical variogram model to an empirical variogram object using a local minimizer for smooth non-linear functions subject to bounded parameters.

USAGE

```
variogram.fit(vobj, param, fun=spher.vgram, lower=rep(0, n.param),
              upper=Inf)
```

REQUIRED ARGUMENTS

`vobj` an object that inherits from class `"variogram"` representing an empirical variogram. Usually, the result of the `variogram` function.

OPTIONAL ARGUMENTS

`param` a named vector with initial values for the parameters to fit. Usually, these are the `"nugget"`, `"sill"`, and `"range"` or a subset of these. If missing, the function will try to determine the parameter names and initial values based on the arguments to the function specified in `fun`.

`fun` a theoretical variogram function The first argument should be distance. The remaining arguments are considered parameters that can be changed to update the fit of fun to object.

`lower` either a single numeric value or a vector of length equal to the number of parameters giving lower bounds for the parameter values. If it is a single value then all parameters have that as their lower bound. See the help page for `nlminb` for more information.

`upper` either a single numeric value or a vector of length equal to the number of parameters giving upper bounds for the parameter values. If it is a single value then all parameters have that as their upper bound. See the help page for `nlminb` for more information.

VALUE

an object of class `"vgram.fit"` with components:

`parameters` a named vector with the fitted values for the parameters.

`objective` the final value of the objective function.

`funName` the `fun` argument as a character string.

`distRange` a numeric vector containing the minimum and maximum distance values from `vobj`.

DETAILS

If fun is one of `exp.vgram`, `gauss.vgram`, `linear.vgram`, `power.vgram` or `spher.vgram` and `param` is not supplied the function sets special initial starting values for `param`. Otherwise, if `param` is not supplied it is set to a vector of ones.

The weighted least squares objective function used in the fitting process (Cressie, 1993, p. 97) is:

```
objective.fun <- function(y,yh,n) sum(n*(y/yh-1)^2)
```

The `nlminb` function is used for the optimization.

REFERENCES

Cressie, Noel. (1993). *Statistics For Spatial Data*, Revised Edition. Wiley, New York.

SEE ALSO

`variogram`, `plot.vgram.fit`, `model.variogram`, `nlminb`.

EXAMPLES

```
vg.iron <- variogram(residuals ~ loc(easting, northing), data=iron.ore)
vfit.iron <- variogram.fit(vg.iron, param=c(range=8.7, sill=3.5,
          nugget=4.8), fun=spher.vgram)
plot(vg.iron)
plot(vfit.iron, add=T)
```