Package ‘maptools’

April 17, 2022

Version 1.1-4

Date 2022-04-18

Title Tools for Handling Spatial Objects

Encoding UTF-8

Depends R (>= 2.10), sp (>= 1.0-11)

Imports foreign (>= 0.8), methods, grid, lattice, stats, utils, grDevices

Suggests rgeos (>= 0.1-8), spatstat.geom (>= 1.65-0), PBSmapping, maps, RColorBrewer, raster, polyclip, plotrix, spatstat.linnet (>= 1.65-3), spatstat.utils (>= 1.19.0), spatstat (>= 2.0-0)

Enhances gpclib

Description Please note that 'maptools' will be retired by the end of 2023, plan transition at your earliest convenience; some functionality will be moved to 'sp'. Set of tools for manipulating geographic data. It includes binary access to 'GSHHG' shoreline files. The package also provides interface wrappers for exchanging spatial objects with packages such as 'PBSmapping', 'spatstat.geom', 'maps', and others.

License GPL (>= 2)

URL http://maptools.r-forge.r-project.org/,
https://r-forge.r-project.org/projects/maptools/

RoxygenNote 6.1.1

NeedsCompilation yes

Author Roger Bivand [cre, aut] (<https://orcid.org/0000-0003-2392-6140>),
Nicholas Lewin-Koh [aut],
Edzer Pebesma [ctb],
Eric Archer [ctb],
Adrian Baddeley [ctb],
Nick Bearman [ctb],
Hans-Jörg Bibiko [ctb],
Steven Brey [ctb],
Jonathan Callahan [ctb],
German Carrillo [ctb],

Stéphane Dray [ctb],
David Forrest [ctb],
Michael Friendly [ctb],
Patrick Giraudoux [ctb],
Duncan Golicher [ctb],
Virgilio Gómez Rubio [ctb],
Patrick Hausmann [ctb],
Karl Ove Hufthammer [ctb],
Thomas Jagger [ctb],
Kent Johnson [ctb],
Matthew Lewis [ctb] (<https://orcid.org/0000-0003-2244-4078>),
Sebastian Luque [ctb],
Don MacQueen [ctb],
Andrew Niccolai [ctb],
Edzer Pebesma [ctb],
Oscar Perpiñán Lamigueiro [ctb],
Ethan Plunkett [ctb],
Ege Rubak [ctb] (<https://orcid.org/0000-0002-6675-533X>),
Tom Short [ctb],
Greg Snow [ctb],
Ben Stabler [ctb],
Murray Stokely [ctb],
Rolf Turner [ctb]

Maintainer Roger Bivand <Roger.Bivand@nhh.no>

Repository CRAN

Date/Publication 2022-04-17 19:12:32 UTC

R topics documented:

`as.im` *Coercion between sp objects and spatstat im objects*

Description

Functions to convert between **spatstat**s raster format `im` and **sp**s `SpatialGridDataFrame` as well as one-way conversion from **raster**s `rasterLayer` to `im`. S4-style `as()` coercion can be used between `im` and `SpatialGridDataFrame` objects.

Usage

```
as.im.SpatialGridDataFrame(from)
as.SpatialGridDataFrame.im(from)
as.im.RasterLayer(from, factor.col.name = NULL)
```

Arguments

`from`	object to coerce from
`factor.col.name`	column name of `levels(from)` to be treated as a factor; if NULL defaults to last column of `from`. Ignored if `from` is not a raster with factor values.

Details

A `SpatialGridDataFrame` object may contain several columns of data such that several values are associated with each grid cell. In contrast an `im` object can only contain a single variable value for each cell. In `as.im.SpatialGridDataFrame()` the first data column is used. To convert another column to `im` format simply extract this column first as shown in the example below.

Methods

coerce `signature(from = "SpatialGridDataFrame",to = "im")`

coerce `signature(from = "im",to = "SpatialGridDataFrame")`

Warning

In **spatstat** all spatial objects are assumed to be planar. This means that **spatstat** is not designed to work directly with geographic (longitude and latitude) coordinates. If a **sp** object is declared to have geographic (unprojected) coordinates **maptools** refuses to convert directly to **spatstat** format. Rather, these should be projected first using e.g. `spTransform`. If you know what you are doing, and really want to force coercion, you can overwrite the `proj4string` of the **sp** object with `NA`, `proj4string(x) <-CRS(NA)`, which will fool the system to think that the data is in local planar coordinates. This is probably not a good idea!

Author(s)

Edzer Pebesma <`edzer.pebesma@uni-muenster.de`>, Roger Bivand

See Also

Other converters between **sp** and **spatstat**: as.ppp.SpatialPoints, as.psp.SpatialLines, as.owin.SpatialPolygons, as.SpatialPolygons.tess.

Examples

```
run <- FALSE
if (require("spatstat.geom", quietly=TRUE)) run <- TRUE
if (run) {
## Extract an example SpatialGridDataFrame and plot it
data(meuse.grid) # A data.frame
gridded(meuse.grid) = ~x+y # Now a SpatialPixelsDataFrame
fullgrid(meuse.grid) <- TRUE # Finally a SpatialGridDataFrame
mg_dist <- meuse.grid["dist"] # A SpatialGridDataFrame with a single column
image(mg_dist, axes=TRUE)
}
if (run) {
## Convert to im format and plot it
mg_im <- as(mg_dist, "im")
plot(mg_im)
}
if (run) {
## Convert back to SpatialGridDataFrame and plot it again
mg2 <- as.SpatialGridDataFrame.im(mg_im)
image(mg2, axes=TRUE)
}
run <- run && require(raster, quietly=TRUE)
if (run) {
## Convert SpatialGridDataFrame -> RasterLayer -> im and plot it
r <- as(mg2, "RasterLayer")
r_im <- as.im.RasterLayer(r)
plot(r_im)
}
if (run) {
rr <- raster(nrow=2, ncol=3)
values(rr) <- 1:6
rr <- as.factor(rr)
rrr <- rr
f <- levels(rrr)[[1]]
f$name <- c("vector", "of", "values")
f$name2 <- letters[1:6]
levels(rrr) <- f
print(levels(rrr))
}
if (run) {
iii <- as.im.RasterLayer(rrr)
plot(iii)
}
if (run) {
iv <- as.im.RasterLayer(rrr, factor.col.name = "name")
plot(iv)
}
```

```
if (run) {
}
```

---

as.linnet.SpatialLines *Convert SpatialLines to Linear Network*

Description

Convert an object of class `SpatialLines` or `SpatialLinesDataFrame` (from package **sp**), representing a collection of polygonal lines, into an object of class `linnet` (from package **spatstat**), representing a network.

Usage

```
  as.linnet.SpatialLines(X, ..., fuse = TRUE)
  ## S4 method for signature 'SpatialLines,linnet'
coerce(from, to = "linnet", strict = TRUE)
  ## S4 method for signature 'SpatialLinesDataFrame,linnet'
coerce(from, to = "linnet",
    strict = TRUE)
```

Arguments

`X, from`	Object of class `SpatialLines` or `SpatialLinesDataFrame` to be converted.
`to`	output object of class “linnet”.
`strict`	logical flag. If TRUE, the returned object must be strictly from the target class.
`...`	Ignored.
`fuse`	Logical value indicating whether to join different curves which have a common vertex.

Details

This function converts an object of class `SpatialLines` or `SpatialLinesDataFrame` into an object of class `linnet`. It is not a method for the **spatstat** generic function `as.linnet`, but like other S4 coercion functions for **sp** classes to **spatstat** classes, it may be called directly as a function.

An object of class SpatialLines or `SpatialLinesDataFrame` (from package **sp**) represents a list of lists of the coordinates of lines, such as a list of all roads in a city. An object of class `linnet` in the **spatstat** package represents a linear network, such as a road network.

If `fuse=FALSE`, each “Line” object in `X` will be treated as if it were disconnected from the others. The result is a network that consists of many disconnected sub-networks, equivalent to the list of “Line” objects.

If `fuse=TRUE` (the default), the code will search for identical pairs of vertices occurring in different “Line” objects, and will treat them as identical vertices, effectively joining the two “Line” objects at this common vertex.

If X belongs to class SpatialLinesDataFrame, the associated columns of data in the auxiliary data frame slot(X,"data") will be copied to the output as the marks attached to the line segments of the network. See the Examples.

Value

An object of class linnet.

Author(s)

Adrian Baddeley.

See Also

as.linnet

Examples

```
run <- FALSE
if(require("spatstat.geom", quietly=TRUE) &&
  require("spatstat.linnet", quietly=TRUE)) run <- TRUE
if (run) {
   dname <- system.file("shapes", package="maptools")
   fname <- file.path(dname, "fylk-val.shp")
   fylk <- readShapeSpatial(fname, proj4string=CRS("+proj=utm +zone=33 +ellps=WGS84"))
   is.projected(fylk)
}
if (run) {
   L <- as(fylk, "linnet")
   print(max(vertexdegree(L)))
}
if (run) {
   L0 <- as.linnet.SpatialLines(fylk, fuse=FALSE)
   print(max(vertexdegree(L0)))
}
if (run) {
   ## extract data associated with each network segment
   head(marks(as.psp(L)))
}
if (run) {
   fname <- file.path(dname, "fylk-val-ll.shp")
   fylk <- readShapeSpatial(fname, proj4string=CRS("+proj=longlat +ellps=WGS84"))
   is.projected(fylk)
}
if (run) {
   try(L <- as(fylk, "linnet"))
  }
```

as.owin *Coercion between sp objects and spatstat owin objects*

Description

Functions to convert between **spatstat**s observation window (`owin`) format and various **sp** formats. S4-style `as()` coercion can be used as well.

Usage

```
as.owin.SpatialPolygons(W, ..., fatal)
as.owin.SpatialGridDataFrame(W, ..., fatal)
as.owin.SpatialPixelsDataFrame(W, ..., fatal)
as.SpatialPolygons.owin(x)
```

Arguments

`W`	`SpatialPolygons` object to coerce to `owin`
`x`	`owin` object to coerce to `SpatialPolygons` format
`...`	ignored
`fatal`	formal coercion argument; ignored

Details

An observation window in **spatstat** defines a planar region. It is typically used to represent a sampling region. It comes in three different formats: a simple rectangle, a polygon (vector format) or a binary mask (TRUE/FALSE grid; raster format). These can all be coerced to polygonal form internally in **spatstat** and then converted to `SpatialPolygons`, which is what `as.SpatialPolygons.owin()` does. For objects of class `SpatialPolygons` (and `SpatialPolygonsDataFrame`) the **sp** polygons are simply extracted and cast into **spatstat**s polygon format inside the `owin` object. For `SpatialPixelsDataFrame` and `SpatialGridDataFrame` the grid is extracted and cast into **spatstat**s mask format inside the `owin` object. In all cases any data apart from the spatial region itself are discarded.

Methods

coerce `signature(from = "SpatialPolygons", to = "owin")`

coerce `signature(from = "SpatialPixelsDataFrame", to = "owin")`

coerce `signature(from = "SpatialGridDataFrame", to = "owin")`

coerce `signature(from = "owin", to = "SpatialPolygons")`

Warning

In **spatstat** all spatial objects are assumed to be planar. This means that **spatstat** is not designed to work directly with geographic (longitude and latitude) coordinates. If a **sp** object is declared to have geographic (unprojected) coordinates **maptools** refuses to convert directly to **spatstat** format. Rather, these should be projected first using e.g. `spTransform`. If you know what you are doing, and really want to force coercion, you can overwrite the `proj4string` of the **sp** object with an empty string, `proj4string(x) <-""`, which will fool the system to think that the data is in local planar coordinates. This is probably not a good idea!

Note

When coercing a SpatialPolygons object to an owin object, full topology checking is enabled by default. To avoid checking, set `spatstat.options(checkpolygons=FALSE)` (from spatstat (1.14-6)). To perform the checking later, `owinpolycheck(W,verbose=TRUE)`.

Author(s)

Edzer Pebesma <`edzer.pebesma@uni-muenster.de`>, Roger Bivand

Examples

```
run <- FALSE
if (require("spatstat.geom", quietly=TRUE)) run <- TRUE
if (run) {
## SpatialPixelsDataFrame -> owin
data(meuse.grid) # A data.frame
gridded(meuse.grid) = ~x+y # Now a SpatialPixelsDataFrame
mg_owin <- as(meuse.grid, "owin")
mg_owin
}
if (run) {
## SpatialGridDataFrame -> owin
fullgrid(meuse.grid) <- TRUE # Now a SpatialGridDataFrame
mg_owin2 <- as(meuse.grid, "owin")
}
if (run) {
## SpatialPolygons region with a hole
ho_sp <- SpatialPolygons(list(Polygons(list(Polygon(cbind(c(0,1,1,0,0),
  c(0,0,1,1,0))), Polygon(cbind(c(0.6,0.4,0.4,0.6,0.6),
  c(0.2,0.2,0.4,0.4,0.2)), hole=TRUE)), ID="ho")))
plot(ho_sp, col="red", pbg="pink")
}
if (run) {
## SpatialPolygons -> owin
ho <- as(ho_sp, "owin")
plot(ho)
}
if (run) {
## Define owin directly and check they are identical
ho_orig <- owin(poly=list(list(x=c(0,1,1,0), y=c(0,0,1,1)),
  list(x=c(0.6,0.4,0.4,0.6), y=c(0.2,0.2,0.4,0.4))))
identical(ho, ho_orig)
}
if (run) {
## owin -> SpatialPolygons
ho_sp1 <- as(ho, "SpatialPolygons")
all.equal(ho_sp, ho_sp1, check.attributes=FALSE)
}
```

`as.ppp` *Coercion between sp objects and spatstat ppp objects*

Description

Functions to convert between **spatstat**s planar point pattern (`ppp`) format and **sp**s `SpatialPoints` and `SpatialPointsDataFrame` as well as one-way conversion from `SpatialGridDataFrame` to `ppp`. S4-style `as()` coercion can be used as well.

Usage

```
as.ppp.SpatialPoints(X)
as.ppp.SpatialPointsDataFrame(X)
as.SpatialPoints.ppp(from)
as.SpatialPointsDataFrame.ppp(from)
as.SpatialGridDataFrame.ppp(from)
```

Arguments

`from, X`	object to coerce from

Details

The main conversion is between **sp**s `SpatialPoints`/`SpatialPointsDataFrame` and **spatstat**s `ppp`. Conversion between `SpatialGridDataFrame` and `ppp` should rarely be used; using `as.owin.SpatialGridDataFrame` is more transparent.

Methods

coerce `signature(from = "SpatialPoints", to = "ppp")`

coerce `signature(from = "SpatialPointsDataFrame", to = "ppp")`

coerce `signature(from = "ppp", to = "SpatialGridDataFrame")`

coerce `signature(from = "ppp", to = "SpatialPointsDataFrame")`

coerce `signature(from = "ppp", to = "SpatialPoints")`

Warning

In **spatstat** all spatial objects are assumed to be planar. This means that **spatstat** is not designed to work directly with geographic (longitude and latitude) coordinates. If a **sp** object is declared to have geographic (unprojected) coordinates **maptools** refuses to convert directly to **spatstat** format. Rather, these should be projected first using e.g. `spTransform`. If you know what you are doing, and really want to force coercion, you can overwrite the `proj4string` of the **sp** object with `NA`, `proj4string(x) <-CRS(NA)`, which will fool the system to think that the data is in local planar coordinates. This is probably not a good idea!

Note

The `ppp` format requires an observation window which is the sampling region. The **sp** formats contain no such information and by default the bounding box of the points is simply used. This is almost never the correct thing to do! Rather, information about the sampling region should be converted into **spatstat**s `owin` format and assigned as the observation window. Usually conversion from `ppp` to **sp** format simply discards the `owin`. However, `as.SpatialGridDataFrame.ppp` actually first discards the points(!), second checks that the corresponding `owin` is in a grid format (matrix of TRUE/FALSE for inside/outside sampling region), and finally converts the TRUE/FALSE grid to a `SpatialGridDataFrame`.

Author(s)

Edzer Pebesma <`edzer.pebesma@uni-muenster.de`>, Roger Bivand

Examples

```
run <- FALSE
if (require("spatstat.geom", quietly=TRUE)) run <- TRUE
if (run) {
## Convert SpatialPointsDataFrame into a marked ppp
data(meuse)
coordinates(meuse) = ~x+y
meuse_ppp <- as(meuse, "ppp")
meuse_ppp # Window is the bounding rectangle
}
if (run) {
plot(meuse_ppp, which.marks = "zinc")
}
if (run) {
## Convert SpatialPoints into an unmarked ppp
meuse2 <- as(meuse, "SpatialPoints")
as(meuse2, "ppp")
}
if (run) {
## Get sampling region in grid format and assign it as observation window
data(meuse.grid)
gridded(meuse.grid) <- ~x+y
mg_owin <- as(meuse.grid, "owin")
Window(meuse_ppp) <- mg_owin
meuse_ppp # Window is now a binary image mask (TRUE/FALSE grid)
}
```

```
if (run) {
plot(meuse_ppp, which.marks = "zinc")
}
if (run) {
## Convert marked ppp back to SpatialPointsDataFrame
rev_ppp_SPDF <- as.SpatialPointsDataFrame.ppp(meuse_ppp)
summary(rev_ppp_SPDF)
}
if (run) {
## Convert marked ppp back to SpatialPoints (discarding marks)
rev_ppp_SP <- as.SpatialPoints.ppp(meuse_ppp)
summary(rev_ppp_SP)
}
if (run) {
## Convert marked ppp back to SpatialGridDataFrame (extracting the window grid)
rev_ppp_SGDF <- as.SpatialGridDataFrame.ppp(meuse_ppp)
summary(rev_ppp_SGDF)
}
```

`as.psp` *Coercion between sp objects and spatstat psp objects*

Description

Functions to convert between **spatstat**s planar segment pattern (psp) format and various **sp** line formats. S4-style `as()` coercion can be used as well.

Usage

```
as.psp.Line(from, ..., window=NULL, marks=NULL, fatal)
as.psp.Lines(from, ..., window=NULL, marks=NULL, fatal)
as.psp.SpatialLines(from, ..., window=NULL, marks=NULL, characterMarks
                  = FALSE, fatal)
as.psp.SpatialLinesDataFrame(from, ..., window=NULL, marks=NULL, fatal)
as.SpatialLines.psp(from)
```

Arguments

`from`	object to coerce from
`...`	ignored
`window`	window of class `owin` as defined in the spatstat package
`marks`	marks as defined in the spatstat package
`characterMarks`	default FALSE, if TRUE, do not convert NULL marks to factor from character
`fatal`	formal coercion argument; ignored

Methods

coerce `signature(from = "Line", to = "psp")`

coerce `signature(from = "Lines", to = "psp")`

coerce `signature(from = "SpatialLines", to = "psp")`

coerce `signature(from = "SpatialLinesDataFrame", to = "psp")`

coerce `signature(from = "psp", to = "SpatialLines")`

Warning

In **spatstat** all spatial objects are assumed to be planar. This means that **spatstat** is not designed to work directly with geographic (longitude and latitude) coordinates. If a **sp** object is declared to have geographic (unprojected) coordinates **maptools** refuses to convert directly to **spatstat** format. Rather, these should be projected first using e.g. `spTransform`. If you know what you are doing, and really want to force coercion, you can overwrite the `proj4string` of the **sp** object with `NA`, `proj4string(x) <-CRS(NA)`, which will fool the system to think that the data is in local planar coordinates. This is probably not a good idea!

Author(s)

Edzer Pebesma <`edzer.pebesma@uni-muenster.de`>, Roger Bivand

Examples

```
run <- FALSE
if (require("spatstat.geom", quietly=TRUE)) run <- TRUE
if (run) {
data(meuse.riv)
mr <- Line(meuse.riv)
mr_psp <- as(mr, "psp")
mr_psp
}
if (run) {
plot(mr_psp)
}
if (run) {
xx_back <- as(mr_psp, "SpatialLines")
plot(xx_back)
}
if (run) {
xx <- readShapeLines(system.file("shapes/fylk-val.shp", package="maptools")[1],
 proj4string=CRS("+proj=utm +zone=33 +ellps=WGS84"))
xx_psp <- as(xx["LENGTH"], "psp")
xx_psp
}
if (run) {
plot(xx_psp)
}
if (run) {
xx_back <- as(xx_psp, "SpatialLines")
plot(xx_back)
}
if (run) {
xx <- readShapeLines(system.file("shapes/fylk-val-ll.shp", package="maptools")[1],
 proj4string=CRS("+proj=longlat +ellps=WGS84"))
try(xx_psp <- as(xx["LENGTH"], "psp"))
}
```

as.SpatialPolygons.tess

Coercion of spatstat tess object to sp SpatialPolygons object

Description

This function coerces **spatstat**s tessellation objects of class `tess` to **sp**s `SpatialPolygons` class. S4-style as() coercion works as well.

Usage

```
as.SpatialPolygons.tess(x)
```

Arguments

`x`	**spatstat** object of class `tess` to coerce from

Methods

coerce `signature(from = "tess", to = "SpatialPolygons")`

Author(s)

Edzer Pebesma <`edzer.pebesma@uni-muenster.de`>, Roger Bivand

Examples

```
run <- FALSE
if (require("spatstat.geom", quietly=TRUE)) run <- TRUE
if (run) {
A <- tess(xgrid=0:4,ygrid=0:4)
A_sp <- as(A, "SpatialPolygons")
plot(A_sp)
text(coordinates(A_sp), labels=row.names(A_sp), cex=0.6)
}
```

CCmaps *Conditioned choropleth maps*

Description

Conditioned choropleth maps permit the conditioning of a map of a variable on the values of one or two other variables coded as factors or shingles. This function uses spplot after constructing multiple subsets of the variable of interest defined by the intervals given by the conditioning variables.

Usage

```
CCmaps(obj, zcol = NULL, cvar = NULL, cvar.names = NULL, ..., names.attr,
 scales = list(draw = FALSE), xlab = NULL, ylab = NULL,
 aspect = mapasp(obj, xlim, ylim), sp.layout = NULL, xlim = bbox(obj)[1, ],
 ylim = bbox(obj)[2, ])
```

Arguments

`obj`	object of class SpatialPolygonsDataFrame
`zcol`	single variable name as string
`cvar`	a list of one or two conditioning variables, which should be of class factor or shingle
`cvar.names`	names for conditioning variables, if not given, the names of the variables in the `cvar` list
`...`	other arguments passed to spplot and levelplot
`names.attr`	names to use in panel, if different from zcol names
`scales`	scales argument to be passed to Lattice plots; use `list(draw = TRUE)` to draw axes scales
`xlab`	label for x-axis
`ylab`	label for y-axis
`aspect`	aspect ratio for spatial axes; defaults to "iso" (one unit on the x-axis equals one unit on the y-axis) but may be set to more suitable values if the data are e.g. if coordinates are latitude/longitude
`sp.layout`	NULL or list; see spplot
`xlim`	numeric; x-axis limits
`ylim`	numeric; y-axis limits

Value

The function returns a SpatialPolygonsDataFrame object with the `zcol` variable and the partitions of the `cvars` list variables invisibly.

Author(s)

Roger Bivand

References

Carr D, Wallin J, Carr D (2000) Two new templates for epidemiology applications: linked micromap plots and conditioned choropleth maps. *Statistics in Medicine* 19(17-18): 2521-2538 Carr D, White D, MacEachren A (2005) Conditioned choropleth maps and hypothesis generation. *Annals of the Association of American Geographers* 95(1): 32-53 Friendly M (2007) A.-M. Guerry's Moral Statistics of France: challenges for multivariable spatial analysis. *Statistical Science* 22(3): 368-399

See Also

spplot

Examples

```
nc.sids <- readShapeSpatial(system.file("shapes/sids.shp",
 package="maptools")[1], IDvar="FIPSNO",
 proj4string=CRS("+proj=longlat +ellps=clrk66"))
nc.sids$ft.SID74 <- sqrt(1000)*(sqrt(nc.sids$SID74/nc.sids$BIR74) +
 sqrt((nc.sids$SID74+1)/nc.sids$BIR74))
nc.sids$ft.NWBIR74 <- sqrt(1000)*(sqrt(nc.sids$NWBIR74/nc.sids$BIR74) +
 sqrt((nc.sids$NWBIR74+1)/nc.sids$BIR74))
library(lattice)
sh_nw4 <- equal.count(nc.sids$ft.NWBIR74, number=4, overlap=1/5)
CCmaps(nc.sids, "ft.SID74", list("Nonwhite_births"=sh_nw4),
 col.regions=colorRampPalette(c("yellow1", "brown3"))(20),
 main="Transformed SIDS rates 1974-8")
```

checkPolygonsHoles *Check holes in Polygons objects*

Description

The function checks holes in Polygons objects. Use of the rgeos package functions is prefered, and if rgeos is available, they will be used automatically. In this case, member Polygon objects are checked against each other for containment, and the returned Polygons object has component hole slots set appropriately. In addition, the output Polygons object may be provided with a comment string, encoding the external and internal rings. For gpclib use, see details below.

Usage

```
checkPolygonsHoles(x, properly=TRUE, avoidGEOS=FALSE, useSTRtree=FALSE)
gpclibPermitStatus()
gpclibPermit()
rgeosStatus()
```

Arguments

`x`	An Polygons object as defined in package sp
`properly`	default TRUE, use `gContainsProperly` rather than `gContains`
`avoidGEOS`	default FALSE; if TRUE force use of **gpclib** even when **rgeos** is available
`useSTRtree`	default FALSE, if TRUE, use **rgeos** STRtree in checking holes, which is much faster, but uses a lot of memory and does not release it on completion (work in progress)

Details

If the gpclib package is used, an intersection between a gpc.poly object with one or more polygon contours and its bounding box is used to set the hole flag. The function will set single polygon contours to hole=FALSE, and if multiple polygon contours are holes, will set them TRUE. The `gpclibPermit` function is used to choose to permit the use of gpclib if installed, and `gpclibPermitStatus` reports its status. The licence for gpclib is not Free or Open Source and explicitly forbids commercial use. See `library(help=gpclib)`.

Value

An Polygons object re-created from the input object.

Author(s)

Roger Bivand

Examples

```
if (rgeosStatus()) {
nc1 <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 proj4string=CRS("+proj=longlat +ellps=clrk66"))
pl <- slot(nc1, "polygons")
sapply(slot(pl[[4]], "Polygons"), function(x) slot(x, "hole"))
pl[[4]] <- Polygons(list(slot(pl[[4]], "Polygons")[[1]],
 Polygon(slot(slot(pl[[4]], "Polygons")[[2]], "coords"), hole=TRUE),
 slot(pl[[4]], "Polygons")[[3]]), slot(pl[[4]], "ID"))
sapply(slot(pl[[4]], "Polygons"), function(x) slot(x, "hole"))
pl_new <- lapply(pl, checkPolygonsHoles)
sapply(slot(pl_new[[4]], "Polygons"), function(x) slot(x, "hole"))
srs <- slot(slot(pl[[1]], "Polygons")[[1]], "coords")
hle2 <- structure(c(-81.64093, -81.38380, -81.34165, -81.66833, -81.64093,
 36.57865, 36.57234, 36.47603, 36.47894, 36.57865), .Dim = as.integer(c(5, 2)))
hle3 <- structure(c(-81.47759, -81.39118, -81.38486, -81.46705, -81.47759,
 36.56289, 36.55659, 36.49907, 36.50380, 36.56289), .Dim = as.integer(c(5, 2)))
x <- Polygons(list(Polygon(srs), Polygon(hle2), Polygon(hle3)),
 ID=slot(pl[[1]], "ID"))
sapply(slot(x, "Polygons"), function(x) slot(x, "hole"))
res <- checkPolygonsHoles(x)
sapply(slot(res, "Polygons"), function(x) slot(x, "hole"))
## Not run:
opar <- par(mfrow=c(1,2))
SPx <- SpatialPolygons(list(x))
plot(SPx)
text(t(sapply(slot(x, "Polygons"), function(i) slot(i, "labpt"))),
 labels=sapply(slot(x, "Polygons"), function(i) slot(i, "hole")), cex=0.6)
title(xlab="Hole slot values before checking")
SPres <- SpatialPolygons(list(res))
plot(SPres)
text(t(sapply(slot(res, "Polygons"), function(i) slot(i, "labpt"))),
 labels=sapply(slot(res, "Polygons"), function(i) slot(i, "hole")), cex=0.6)
title(xlab="Hole slot values after checking")
par(opar)
p1 <- Polygon(cbind(x=c(0, 0, 10, 10, 0), y=c(0, 10, 10, 0, 0))) # I
p2 <- Polygon(cbind(x=c(3, 3, 7, 7, 3), y=c(3, 7, 7, 3, 3))) # H
p8 <- Polygon(cbind(x=c(1, 1, 2, 2, 1), y=c(1, 2, 2, 1, 1))) # H
p9 <- Polygon(cbind(x=c(1, 1, 2, 2, 1), y=c(5, 6, 6, 5, 5))) # H
p3 <- Polygon(cbind(x=c(20, 20, 30, 30, 20), y=c(20, 30, 30, 20, 20))) # I
p4 <- Polygon(cbind(x=c(21, 21, 29, 29, 21), y=c(21, 29, 29, 21, 21))) # H
p14 <- Polygon(cbind(x=c(21, 21, 29, 29, 21), y=c(21, 29, 29, 21, 21))) # H
p5 <- Polygon(cbind(x=c(22, 22, 28, 28, 22), y=c(22, 28, 28, 22, 22))) # I
p15 <- Polygon(cbind(x=c(22, 22, 28, 28, 22), y=c(22, 28, 28, 22, 22))) # I
p6 <- Polygon(cbind(x=c(23, 23, 27, 27, 23), y=c(23, 27, 27, 23, 23))) # H
p7 <- Polygon(cbind(x=c(13, 13, 17, 17, 13), y=c(13, 17, 17, 13, 13))) # I
p10 <- Polygon(cbind(x=c(24, 24, 26, 26, 24), y=c(24, 26, 26, 24, 24))) # I
p11 <- Polygon(cbind(x=c(24.25, 24.25, 25.75, 25.75, 24.25),
 y=c(24.25, 25.75, 25.75, 24.25, 24.25))) # H
p12 <- Polygon(cbind(x=c(24.5, 24.5, 25.5, 25.5, 24.5),
 y=c(24.5, 25.5, 25.5, 24.5, 24.5))) # I
p13 <- Polygon(cbind(x=c(24.75, 24.75, 25.25, 25.25, 24.75),
 y=c(24.75, 25.25, 25.25, 24.75, 24.75))) # H
lp <- list(p1, p2, p13, p7, p6, p5, p4, p3, p8, p11, p12, p9, p10, p14, p15)
#           1   2    3   4   5   6   7   8   9   10   11  12   13   14   15
#           0   1   11   0   6   0   8   0   1   13    0   1    0  (7)  (6)
#           I   H    H   I   H   I   H   I   H    H    I   H    I   ?    ?
pls <- Polygons(lp, ID="1")
comment(pls)
pls1 <- checkPolygonsHoles(pls)
comment(pls1)
opar <- par(mfrow=c(1,2))
plot(SpatialPolygons(list(pls)), col="magenta", pbg="cyan", usePolypath=FALSE)
title(xlab="Hole slot values before checking")
plot(SpatialPolygons(list(pls1)), col="magenta", pbg="cyan", usePolypath=FALSE)
title(xlab="Hole slot values after checking")
par(opar)

## End(Not run)
}
```

ContourLines2SLDF *Converter functions to build SpatialLinesDataFrame objects*

Description

These functions show how to build converters to SpatialLinesDataFrame objects: `ArcObj2SLDF` from the list returned by the `get.arcdata` function in the RArcInfo package; `ContourLines2SLDF` from the list returned by the `contourLines` function in the graphics package (here the data frame is just the contour levels, with one Lines object made up of at least one Line object per level); and `MapGen2SL` reads a file in "Mapgen" format into a `SpatialLines` object.

Usage

```
ArcObj2SLDF(arc, proj4string=CRS(as.character(NA)), IDs)
ContourLines2SLDF(cL, proj4string=CRS(as.character(NA)))
MapGen2SL(file, proj4string=CRS(as.character(NA)))
```

Arguments

`arc`	a list returned by the `get.arcdata` function in the RArcInfo package
`IDs`	vector of unique character identifiers; if not given, suitable defaults will be used, and the same values inserted as data slot row names
`cL`	a list returned by the `contourLines` function in the graphics package
`proj4string`	Object of class `"CRS"`; see CRS-class
`file`	filename of a file containing a Mapgen line data set

Value

A SpatialLinesDataFrame object

Note

Coastlines of varying resolution may be chosen online and downloaded in "Mapgen" text format from `https://www.ngdc.noaa.gov/mgg/shorelines/shorelines.html`, most conveniently using the interactive selection tool, but please note the 500,000 point limit on downloads, which is easy to exceed.

Author(s)

Roger Bivand; Edzer Pebesma

See Also

SpatialLines-class

Examples

```
#data(co37_d90_arc) # retrieved as:
# library(RArcInfo)
# fl <- "http://www.census.gov/geo/cob/bdy/co/co90e00/co37_d90_e00.zip"
# download.file(fl, "co37_d90_e00.zip")
# e00 <- zip.file.extract("co37_d90.e00", "co37_d90_e00.zip")
# e00toavc(e00, "ncar")
# arc <- get.arcdata(".", "ncar")
#res <- arcobj2SLDF(arc)
#plot(res)
#invisible(title(""))
res <- ContourLines2SLDF(contourLines(volcano))
plot(res, col=terrain.colors(nrow(as(res, "data.frame"))))
title("Volcano contours as SpatialLines")
```

dotsInPolys *Put dots in polygons*

Description

Make point coordinates for a dot density map

Usage

```
dotsInPolys(pl, x, f = "random", offset, compatible = FALSE)
```

Arguments

`pl`	an object of class SpatialPolygons or SpatialPolygonsDataFrame
`x`	integer vector of counts of same length as pl for dots
`f`	type of sampling used to place points in polygons, either "random" or "regular"
`offset`	for regular sampling only: the offset (position) of the regular grid; if not set, `c(0.5,0.5)`, that is the returned grid is not random
`compatible`	what to return, if TRUE a a list of matrices of point coordinates, one matrix for each member of pl, if false a SpatialPointsDataFrame with polygon ID values

Details

With f="random", the dots are placed in the polygon at random, f="regular" - in a grid pattern (number of dots not guaranteed to be the same as the count). When the polygon is made up of more than one part, the dots will be placed in proportion to the relative areas of the clockwise rings (anticlockwise are taken as holes). From maptools release 0.5-2, correction is made for holes in the placing of the dots, but depends on hole values being correctly set, which they often are not.

Value

If compatible=TRUE, the function returns a list of matrices of point coordinates, one matrix for each member of pl. If x[i] is zero, the list element is NULL, and can be tested when plotting - see the examples. If compatible=FALSE (default), it returns a SpatialPointsDataFrame with polygon ID values as the only column in the data slot.

Note

Waller and Gotway (2004) Applied Spatial Statistics for Public Health Data (Wiley, Hoboken, NJ) explicitly warn that care is needed in plotting and interpreting dot density maps (pp. 81-83)

Author(s)

Roger Bivand <Roger.Bivand@nhh.no>

See Also

spsample

Examples

```
nc_SP <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 proj4string=CRS("+proj=longlat  +ellps=clrk66"))
## Not run:
pls <- slot(nc_SP, "polygons")
pls_new <- lapply(pls, checkPolygonsHoles)
nc_SP <- SpatialPolygonsDataFrame(SpatialPolygons(pls_new,
 proj4string=CRS(proj4string(nc_SP))), data=as(nc_SP, "data.frame"))

## End(Not run)
try1 <- dotsInPolys(nc_SP, as.integer(nc_SP$SID74))
plot(nc_SP, axes=TRUE)
plot(try1, add=TRUE, pch=18, col="red")
try2 <- dotsInPolys(nc_SP, as.integer(nc_SP$SID74), f="regular")
plot(nc_SP, axes=TRUE)
plot(try2, add=TRUE, pch=18, col="red")
```

---

elide-methods *Methods for Function elide in Package 'maptools'*

Description

Methods for function `elide` to translate and disguise coordinate placing in the real world.

Usage

```
elide(obj, ...)
```

Arguments

`obj`	object to be elided
`...`	other arguments: **bb** if NULL, uses bounding box of object, otherwise the given bounding box **shift** values to shift the coordinates of the input object; this is made ineffective by the scale argument **reflect** reverse coordinate axes **scale** if NULL, coordinates not scaled; if TRUE, the longer dimension is scaled to lie within [0,1] and aspect maintained; if a scalar, the output range of [0,1] is multiplied by scale

flip translate coordinates on the main diagonal

rotate default 0, rotate angle degrees clockwise around center

center default NULL, if not NULL, the rotation center, numeric of length two

unitsq logical, default FALSE, if TRUE and scale TRUE, impose unit square bounding box (currently only points)

Value

The methods return objects of the input class object with elided coordinates; the coordinate reference system is not set. Note that if the input coordinates or centroids are in the data slot data.frame of the input object, they should be removed before the use of these methods, otherwise they will betray the input positions.

Methods

obj = "SpatialPoints" elides object

obj = "SpatialPointsDataFrame" elides object

obj = "SpatialLines" elides object

obj = "SpatialLinesDataFrame" elides object

obj = "SpatialPolygons" elides object

obj = "SpatialPolygonsDataFrame" elides object

Note

Rotation code kindly contributed by Don MacQueen

Examples

```
data(meuse)
coordinates(meuse) <- c("x", "y")
proj4string(meuse) <- CRS("+init=epsg:28992")
data(meuse.riv)
river_polygon <- Polygons(list(Polygon(meuse.riv)), ID="meuse")
rivers <- SpatialPolygons(list(river_polygon))
proj4string(rivers) <- CRS("+init=epsg:28992")
rivers1 <- elide(rivers, reflect=c(TRUE, TRUE), scale=TRUE)
meuse1 <- elide(meuse, bb=bbox(rivers), reflect=c(TRUE, TRUE), scale=TRUE)
opar <- par(mfrow=c(1,2))
plot(rivers, axes=TRUE)
plot(meuse, add=TRUE)
plot(rivers1, axes=TRUE)
plot(meuse1, add=TRUE)
par(opar)
meuse1 <- elide(meuse, shift=c(10000, -10000))
bbox(meuse)
bbox(meuse1)
rivers1 <- elide(rivers, shift=c(10000, -10000))
bbox(rivers)
bbox(rivers1)
```

```
meuse1 <- elide(meuse, rotate=-30, center=apply(bbox(meuse), 1, mean))
bbox(meuse)
bbox(meuse1)
plot(meuse1, axes=TRUE)
```

`gcDestination` *Find destination in geographical coordinates*

Description

Find the destination in geographical coordinates at distance dist and for the given bearing from the starting point given by lon and lat.

Usage

```
gcDestination(lon, lat, bearing, dist, dist.units = "km",
 model = NULL, Vincenty = FALSE)
```

Arguments

`lon`	longitude (Eastings) in decimal degrees (either scalar or vector)
`lat`	latitude (Northings) in decimal degrees (either scalar or vector)
`bearing`	bearing from 0 to 360 degrees (either scalar or vector)
`dist`	distance travelled (scalar)
`dist.units`	units of distance "km" (kilometers), "nm" (nautical miles), "mi" (statute miles)
`model`	choice of ellipsoid model ("WGS84", "GRS80", "Airy", "International", "Clarke", "GRS67"
`Vincenty`	logical flag, default FALSE

Details

The bearing argument may be a vector when lon and lat are scalar, representing a single point.

Value

A matrix of decimal degree coordinates with Eastings in the first column and Northings in the second column.

Author(s)

Eric Archer and Roger Bivand

References

http://www.movable-type.co.uk/scripts/latlong.html#ellipsoid,

the file earlier available at http:\/\/williams.best.vwh.net/avform.htm,

http://www.movable-type.co.uk/scripts/latlong-vincenty.html#direct,

Original reference https://www.ngs.noaa.gov/PUBS_LIB/inverse.pdf:

Vincenty, T. 1975. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. Survey Review 22(176):88-93

See Also

gzAzimuth

Examples

```
data(state)
res <- gcDestination(state.center$x, state.center$y, 45, 250, "km")
plot(state.center$x, state.center$y, asp=1, pch=16)
arrows(state.center$x, state.center$y, res[,1], res[,2], length=0.05)
llist <- vector(mode="list", length=length(state.center$x))
for (i in seq(along=llist)) llist[[i]] <- gcDestination(state.center$x[i],
  state.center$y[i], seq(0, 360, 5), 250, "km")
plot(state.center$x, state.center$y, asp=1, pch=3)
nll <- lapply(llist, lines)
```

---

getinfo.shape *Get shapefile header information*

Description

Get shapefile header information; the file should be given including its ".shp" extension, and the function will reconstruct the names of the database (dbf) file and the index (shx) file from these.

Usage

```
getinfo.shape(filen)
## S3 method for class 'shapehead'
print(x, ...)
```

Arguments

filen	name of file with *.shp extension
x	a shapehead list as returned by getinfo.shape
...	other arguments passed to print

Details

The function calls code from shapelib to read shapefiles, a file format used by ESRI GIS software among others

Value

The function returns a list of class shapehead.

Author(s)

Roger Bivand <`Roger.Bivand@nhh.no`>; shapelib by Frank Warmerdam

References

`http://shapelib.maptools.org/`

Examples

```
res <- getinfo.shape(system.file("shapes/fylk-val.shp", package="maptools")[1])
res
str(res)
```

`getKMLcoordinates` *Get a list of coordinates out of a KML file*

Description

This function parses a KML file to get the content of `<coordinates>` tags and returns a list of matrices representing the longitude-latitute or if `ignoreAltitude` is `FALSE` the longitude-latitute-altitude coordinates of a KML geometry.

Usage

```
getKMLcoordinates(kmlfile, ignoreAltitude=FALSE)
```

Arguments

`kmlfile`	connection object or a character string of the KML file
`ignoreAltitude`	if set to `TRUE` the altitude values of a KML points will be ignored

Value

`coords` is a list of matrices representing the longitude-latitute or if `ignoreAltitude` is `FALSE` the longitude-latitute-altitude coordinates

Author(s)

Hans-J. Bibiko

See Also

kmlPolygon, kmlLine

Examples

```
data(wrld_simpl)
## creates a KML file containing the polygons of South Africa (plus hole)
sw <- slot(wrld_simpl[wrld_simpl$NAME=="South Africa",], "polygons")[[1]]
tf <- tempfile()
kmlPolygon(sw, kmlfile=tf, name="South Africa", col="#df0000aa", lwd=5,
    border=4, kmlname="R Test",
   kmldescription="This is <b>only</b> a <a href='http://www.r-project.org'>R</a> test.")
zz <- getKMLcoordinates(tf, ignoreAltitude=TRUE)
str(zz)
zz <- getKMLcoordinates(system.file("shapes/Testing.kml", package="maptools"))
str(zz)
```

GE_SpatialGrid *Create SpatialGrid for PNG output to GE*

Description

The function sets up metadata in the form of a SpatialGrid object for defining the size and placing of a PNG image overlay in Google Earth. The internal function `Sobj_SpatialGrid` can also be called to build a grid for arbitrary Spatial* objects.

Usage

```
GE_SpatialGrid(obj, asp = NA, maxPixels = 600)
Sobj_SpatialGrid(obj, asp=1, maxDim=100, n=NULL)
```

Arguments

`obj`	a Spatial* object
`asp`	if NA, will be set to the latitude corrected value
`maxPixels`	the maximum dimension of the output PNG
`maxDim`	the maximum dimension of the output grid; ignored if `n` not NULL
`n`	if not NULL, the minimum number of cells in the returned grid

Details

The function is used together with `kmlOverlay` to wrap around the opening of a PNG graphics device, plotting code, and the closing of the device. The computed values take account of the adjustment of the actual data bounding box to an integer number of rows and columns in the image file.

The approach may be used as an alternative to writing PNG files from SpatialGrid and SpatialPixel objects in **rgdal** using `writeGDAL`, and to writing KML files using `writeOGR` for vector data objects.

The output PNG files are likely to be very much smaller than large vector data KML files, and hinder the retrieval of exact positional information.

Note that the geometries should be in geographical coordinates with datum WGS84 for export to KML.

Value

returns an S3 object of class `GE_SG` with components:

`height`	Integer raster height for png call
`width`	Integer raster width for png call
`SG`	a SpatialGrid object with the grid topology of the output PNG
`asp`	the aspect value used
`xlim`	xlim taken from SG
`ylim`	ylim taken from SG

Author(s)

Duncan Golicher, David Forrest and Roger Bivand

See Also

`kmlOverlay`

Examples

```
opt_exask <- options(example.ask=FALSE)
qk <- SpatialPointsDataFrame(quakes[, c(2:1)], quakes)
summary(Sobj_SpatialGrid(qk)$SG)
t2 <- Sobj_SpatialGrid(qk, n=10000)$SG
summary(t2)
prod(slot(slot(t2, "grid"), "cells.dim"))
proj4string(qk) <- CRS("+proj=longlat +ellps=WGS84")
tf <- tempfile()
SGqk <- GE_SpatialGrid(qk)
png(file=paste(tf, ".png", sep=""), width=SGqk$width, height=SGqk$height,
  bg="transparent")
par(mar=c(0,0,0,0), xaxs="i", yaxs="i")
plot(qk, xlim=SGqk$xlim, ylim=SGqk$ylim, setParUsrBB=TRUE)
dev.off()
kmlOverlay(SGqk, paste(tf, ".kml", sep=""), paste(tf, ".png", sep=""))
## Not run:
qk0 <- quakes
qk0$long <- ifelse(qk0$long <= 180, qk0$long, qk0$long-360)
qk0a <- SpatialPointsDataFrame(qk0[, c(2:1)], qk0)
proj4string(qk0a) <- CRS("+proj=longlat +ellps=WGS84")
# writeOGR(qk0a, paste(tf, "v.kml", sep=""), "Quakes", "KML")
# system(paste("googleearth ", tf, ".kml", sep=""))

## End(Not run)
options(example.ask=opt_exask)
```

gpcholes *Hisaji Ono's lake/hole problem*

Description

How to plot polygons with holes - holes are encoded by coordinates going anticlockwise, and overplotting is avoided by re-ordering the order in which polygons are plotted.

This example is retained for historical interest only, other solutions are present in the sp package.

Usage

```
data(gpcholes)
```

Details

"Date: Tue, 11 May 2004 12:54:20 +0900 From: Hisaji ONO To: r-help

I've tried to create a polygon with one hole by gpclib using following example script.

holepoly <- read.polyfile(system.file("poly-ex/hole-poly.txt", package="gpclib"), nohole = FALSE) area.poly(holepoly) plot(holepoly,poly.args=list(col="red",border="blue"))

And I noticed plot function couldn't draw polygons with holes correctly.

Does anyone know how to solve this situation?"

(h1pl has reversed the y component of polygon 1, to make its ring direction clockwise, h2pl reverses the order of the two polygons in holepoly1@pts)

Source

Data file included in "gpclib" package.

Examples

```
data(gpcholes)
opar <- par(mfrow=c(1,2))
plot(SpatialPolygons(list(h2pl)), col="red", pbg="white", border="blue")
plot(SpatialPolygons(list(h1pl)), col="red", pbg="white", border="blue")
par(opar)
```

`gzAzimuth` *Find azimuth for geographical coordinates*

Description

The function finds azimuth values for geographical coordinates given as decimal degrees from the `from` coordinates to the `to` coordinate. In function `trackAzimuth`, the azimuth values are found between successive rows of the input coordinate matrix.

Usage

```
gzAzimuth(from, to, type = "snyder_sphere")
trackAzimuth(track, type="snyder_sphere")
```

Arguments

`from`	a two column matrix of geographical coordinates given as decimal degrees (longitude first)
`track`	a two column matrix of geographical coordinates given as decimal degrees (longitude first)
`to`	a one row, two column matrix or two element vector of geographical coordinates given as decimal degrees (longitude first)
`type`	default is `"snyder_sphere"`, otherwise `"abdali"`; the results should be identical with slightly less trigonometry in `"abdali"`

Details

The azimuth is calculated on the sphere, using the formulae given by Snyder (1987, p. 30) and Abdali (1997, p. 17). The examples use data taken from Abdali (p. 17–18). There is a very interesting discussion of the centrality of azimuth-finding in the development of mathematics and mathematical geography in Abdali's paper. Among others, al-Khwarizmi was an important contributor. As Abdali puts it, "This is a veritable who's who of medieval science" (p. 3).

Value

values in decimal degrees - zero is North - of the azimuth from the `from` coordinates to the `to` coordinate.

Author(s)

Roger Bivand, with contributions by Sebastian Luque

References

Snyder JP (1987) Map projections - a working manual, USGS Professional Paper 1395; Abdali SK (1997) "The Correct Qibla", formerly at http://patriot.net/users/abdali/ftp/qibla.pdf

Examples

```
name <- c("Mecca", "Anchorage", "Washington")
long <- c(39.823333, -149.883333, -77.0166667)
lat <- c(21.423333, 61.2166667, 38.9)
x <- cbind(long, lat)
row.names(x) <- name
crib <- c(-9.098363, 56.575960)
r1 <- gzAzimuth(x[2:3,], x[1,])
r1
all.equal(r1, crib)
r2 <- gzAzimuth(x[2:3,], x[1,], type="abdali")
r2
all.equal(r2, crib)
trackAzimuth(x)
```

`kmlLine` *Create and write a KML file on the basis of a given Lines object*

Description

The function is used to create and write a KML file on the basis of a given Lines object (a list of Line objects) for the usage in Google Earth resp. Google Maps.

Usage

```
kmlLine(obj=NULL, kmlfile=NULL,
    name="R Line", description="", col=NULL, visibility=1, lwd=1,
    kmlname="", kmldescription="")
```

Arguments

`obj`	a `Lines` or `SpatialLinesDataFrame` object
`kmlfile`	if not `NULL` the name as character string of the kml file to be written
`name`	the name of the KML line
`description`	the description of the KML line (HTML tags allowed)
`col`	the stroke color (see also Color Specification) of the KML line
`visibility`	if set to `1` or `TRUE` specifies that the KML line should be visible after loading
`lwd`	the stroke width for the KML line
`kmlname`	the name of the KML layer
`kmldescription`	the description of the KML layer (HTML tags allowed)

Details

The function is used to convert a given `Lines` object (a list of Line objects) or the first `Lines` object listed in a passed `SpatialLinesDataFrame` object into KML line(s). If `kmlfile` is not `NULL` the result will be written into that file. If `kmlfile` is `NULL` the generated KML lines will be returned (see also value).

For a passed `Lines` object the function generates a <Style> tag whereby its id attribute is set to the passed object's ID.

Note that the geometries should be in geographical coordinates with datum WGS84.

The resulting KML line will be embedded in `<Placemark><MultiGeometry><LineString>`.

Value

x is a list with the elements `style` and `content` containing the generated lines of the KML file as character vectors if `kmlfile` is `NULL`.

y is a list with the elements `header` and `footer` representing the KML file' header resp. footer if `obj` is `NULL`.

Color Specification

The following color specifications are allowed: `'red'`, 2, or as hex code `'#RRGGBB'` resp. `'#RRGGBBAA'` for passing the alpha value.

Author(s)

Hans-J. Bibiko

See Also

`kmlOverlay`, `kmlPolygon`, `Line`

Examples

```
xx <- readShapeSpatial(system.file("shapes/fylk-val-ll.shp",
      package="maptools")[1], proj4string=CRS("+proj=longlat +ellps=WGS84"))
out <- sapply(slot(xx, "lines"), function(x) { kmlLine(x,
    name=slot(x, "ID"), col="blue", lwd=1.5,
    description=paste("river:", slot(x, "ID"))) })
tf <- tempfile()
kmlFile <- file(tf, "w")
tf
cat(kmlLine(kmlname="R Test", kmldescription="<i>Hello</i>")$header,
    file=kmlFile, sep="\n")
cat(unlist(out["style",]), file=kmlFile, sep="\n")
cat(unlist(out["content",]), file=kmlFile, sep="\n")
cat(kmlLine()$footer, file=kmlFile, sep="\n")
close(kmlFile)
```

kmlLines *Create and write a KML file on the basis of a given Lines object*

Description

The function is used to create and write a KML file on the basis of a given Lines object (a list of Line objects) for the usage in Google Earth and Google Maps.

Usage

```
kmlLines(obj=NULL, kmlfile=NULL,
        name="R Lines", description="", col=NULL, visibility=1, lwd=1,
        kmlname="", kmldescription="")
```

Arguments

`obj`	a `Lines` or `SpatialLinesDataFrame` object
`kmlfile`	if not `NULL` the name as character string of the kml file to be written
`name`	the name of the KML line
`description`	the description of the KML line (HTML tags allowed)
`col`	the stroke color (see also Color Specification) of the KML line
`visibility`	if set to `1` or `TRUE` specifies that the KML line should be visible after loading
`lwd`	the stroke width for the KML line
`kmlname`	the name of the KML layer
`kmldescription`	the description of the KML layer (HTML tags allowed)

Details

The function is used to convert a given `Lines` object (a list of Line objects) or the first `Lines` object listed in a passed `SpatialLinesDataFrame` object into KML line(s). If `kmlfile` is not `NULL` the result will be written into that file. If `kmlfile` is `NULL` the generated KML lines will be returned (see also value). Function no longer uses `append` greatly improving performance on large objects or lists.

For a passed `Lines` object the function generates a <Style> tag whereby its id attribute is set to the passed object's ID.

Note that the geometries should be in geographical coordinates with datum WGS84.

The resulting KML line will be embedded in `<Placemark><MultiGeometry><LineString>`.

Value

x is a list with the elements `style` and `content` containing the generated lines of the KML file as character vectors if `kmlfile` is `NULL`.

y is a list with the elements `header` and `footer` representing the KML file header and footer if `obj` is `NULL`.

Color Specification

The following color specifications are allowed: `'red'`, 2, or as hex code `'#RRGGBB'` resp. `'#RRGGBBAA'` for passing the alpha value.

Author(s)

Hans-J. Bibiko, Jon Callahan, Steven Brey

See Also

`kmlOverlay`, `kmlPolygon`, `Line`

Examples

```
# Maptools library required
library(maptools)
# load line object
rivers <- readShapeSpatial(system.file("shapes/fylk-val-ll.shp",
                   package="maptools")[1], proj4string=CRS("+proj=longlat +ellps=WGS84"))
# create kml file
td <- tempdir()
kmlfile <- paste(td, "rivers.kml", sep="/")
kmlLines(rivers, kmlfile = kmlfile, name = "R Lines",
         description = "Hello!", col = "blue", visibility = 1, lwd = 1,
         kmlname = "", kmldescription = "")
```

`kmlOverlay` *Create and write KML file for PNG image overlay*

Description

The function is used to create and write a KML file for a PNG image overlay for Google Earth.

Usage

```
kmlOverlay(obj, kmlfile = NULL, imagefile = NULL, name = "R image")
```

Arguments

`obj`	a `GE_SG` object from `GE_SpatialGrid`
`kmlfile`	if not NULL the name of the kml file to be written
`imagefile`	the name of the PNG file containing the image - this should be either relative (same directory as kml file) or abosolute (fully qualified)
`name`	the name used to describe the image overlay in GE

Details

The function is used together with `GE_SpatialGrid` to wrap around the opening of a PNG graphics device, plotting code, and the closing of the device. The computed values take account of the adjustment of the actual data bounding box to an integer number of rows and columns in the image file.

The approach may be used as an alternative to writing PNG files from SpatialGrid and SpatialPixel objects in **rgdal** using `writeGDAL`, and to writing KML files using `writeOGR` for vector data objects. The output PNG files are likely to be very much smaller than large vector data KML files, and hinder the retrieval of exact positional information.

Note that the geometries should be in geographical coordinates with datum WGS84.

Value

x is a character vector containing the generated lines of the kml file

Author(s)

Duncan Golicher, David Forrest and Roger Bivand

See Also

`GE_SpatialGrid`

Examples

```
opt_exask <- options(example.ask=FALSE)
qk <- SpatialPointsDataFrame(quakes[, c(2:1)], quakes)
proj4string(qk) <- CRS("+proj=longlat +ellps=WGS84")
tf <- tempfile()
SGqk <- GE_SpatialGrid(qk)
png(file=paste(tf, ".png", sep=""), width=SGqk$width, height=SGqk$height,
  bg="transparent")
par(mar=c(0,0,0,0), xaxs="i", yaxs="i")
plot(qk, xlim=SGqk$xlim, ylim=SGqk$ylim, setParUsrBB=TRUE)
dev.off()
kmlOverlay(SGqk, paste(tf, ".kml", sep=""), paste(tf, ".png", sep=""))
## Not run:
#library(rgdal)
#qk0 <- quakes
#qk0$long <- ifelse(qk0$long <= 180, qk0$long, qk0$long-360)
#qk0a <- SpatialPointsDataFrame(qk0[, c(2:1)], qk0)
#proj4string(qk0a) <- CRS("+proj=longlat +ellps=WGS84")
#writeOGR(qk0a, paste(tf, "v.kml", sep=""), "Quakes", "KML")
#system(paste("googleearth ", tf, ".kml", sep=""))

## End(Not run)
options(example.ask=opt_exask)
```

`kmlPoints` *Create and write a KML file on the basis of a given Points object*

Description

The function is used to create and write a KML file on the basis of a given SpatialPointsDataFrame object for the usage in Google Earth resp. Google Maps.

Usage

```
kmlPoints(obj=NULL, kmlfile=NULL, kmlname="", kmldescription="",
    name=NULL, description="",
   icon="http://www.gstatic.com/mapspro/images/stock/962-wht-diamond-blank.png")
```

Arguments

`obj`	a `SpatialPointsDataFrame` object
`kmlfile`	if not `NULL` the name as character string of the kml file to be written
`kmlname`	the name of the KML layer
`kmldescription`	the description of the KML layer (HTML tags allowed)
`name`	a character vector to be used as names for each KML Placemark
`description`	a character vector to be used as the description for each KML Placemark (HTML tags allowed)
`icon`	a character vector of icon URLs to be used in the style associated with each KML Placemark

Details

The function is used to convert a given `SpatialPointsDataFrame` object into a series of KML Placemarks, each with a single Point. If `kmlfile` is not `NULL` the result will be written into that file. If `kmlfile` is `NULL` the generated KML lines will be returned (see also value).

If `name=NULL`, the <name> tag for each Placemark will be `'site #'`. If a single value is used for `name` or `description`, that value will be replicated for each Placemark. If a single value is used for `icon`, only a single style will be created and that style will be referenced by each Placemark.

Note that the geometries should be in geographical coordinates with datum WGS84.

Value

x is a list with the elements `style` and `content` containing the generated lines of the KML file as character vectors if `kmlfile` is `NULL`.

y is a list with the elements `header` and `footer` representing the KML file' header resp. footer if `obj` is `NULL`.

KML icons

The default icon URL is http://www.gstatic.com/mapspro/images/stock/962-wht-diamond-blank.png. Additional icons are available at: http://sites.google.com/site/gmapsdevelopment.

Author(s)

Jonathan Callahan

See Also

kmlLine, kmlOverlay, kmlPolygon, Line

Examples

```
data(SplashDams)
num <- length(SplashDams)
td <- tempdir()
kmlfile <- paste(td, "OregonSplashDams.kml", sep="/")
kmlname <- "Oregon Splash Dams"
kmldescription <- paste("Data for Splash Dams in western Oregon.",
 "See http://www.fs.fed.us/pnw/lwm/aem/people/burnett.html#projects_activities",
 "for more information.")
icon <- "http://www.gstatic.com/mapspro/images/stock/962-wht-diamond-blank.png"
name <- paste("Dam on",SplashDams$streamName)
description <- paste("<b>owner:</b>", SplashDams$owner, "<br><b>dates:</b>", SplashDams$datesUsed)

kmlPoints(SplashDams, kmlfile=kmlfile, name=name, description=description,
          icon=icon, kmlname=kmlname, kmldescription=kmldescription)
```

---

kmlPolygon *Create and write a KML file on the basis of a given Polygons object*

Description

The function is used to create and write a KML file on the basis of a given Polygons object (a list of Polygon objects) for the usage in Google Earth resp. Google Maps.

Usage

```
kmlPolygon(obj=NULL, kmlfile=NULL,
    name="R Polygon", description="", col=NULL, visibility=1, lwd=1, border=1,
    kmlname="", kmldescription="")
```

Arguments

`obj`	a `Polygons` or `SpatialPolygonsDataFrame` object
`kmlfile`	if not `NULL` the name as character string of the kml file to be written
`name`	the name of the KML polygon
`description`	the description of the KML polygon (HTML tags allowed)
`col`	the fill color (see also Color Specification) of the KML polygon
`visibility`	if set to `1` or `TRUE` specifies that the KML polygon should be visible after loading
`lwd`	the stroke width for the KML polygon
`border`	the stroke color (see also Color Specification) for the KML polygon
`kmlname`	the name of the KML layer
`kmldescription`	the description of the KML layer (HTML tags allowed)

Details

The function is used to convert a given `Polygons` object (a list of Polygon objects) or the first `Polygons` object listed in a passed `SpatialPolygonsDataFrame` object into KML polygon. If `kmlfile` is not `NULL` the result will be written into that file. If `kmlfile` is `NULL` the generated KML lines will be returned (see also value).

The conversion can also handle polygons which are marked as holes inside of the Polygons object if these holes are listed right after that polygon in which these holes appear. That implies that a given plot order set in the Polygons object will **not** be considered.

For a passed `Polygons` object the function generates a <Style> tag whereby its id attribute is set to the passed object's ID.

Note that the geometries should be in geographical coordinates with datum WGS84.

The resulting KML polygon will be embedded in `<Placemark><MultiGeometry><Polygon>`.

Value

x is a list with the elements `style` and `content` containing the generated lines of the KML file as character vectors if `kmlfile` is `NULL`.

y is a list with the elements `header` and `footer` representing the KML file' header resp. footer if `obj` is `NULL` (see second example).

Color Specification

The following color specifications are allowed: `'red'`, `2`, or as hex code `'#RRGGBB'` resp. `'#RRGGBBAA'` for passing the alpha value.

Author(s)

Hans-J. Bibiko

See Also

`kmlOverlay`, `kmlLine`, `SpatialPolygons`

Examples

```
data(wrld_simpl)
## creates a KML file containing the polygons of South Africa (plus hole)
sw <- slot(wrld_simpl[wrld_simpl$NAME=="South Africa",], "polygons")[[1]]
tf <- tempfile()
kmlPolygon(sw, kmlfile=tf, name="South Africa", col="#df0000aa", lwd=5,
    border=4, kmlname="R Test",
   kmldescription="This is <b>only</b> a <a href='http://www.r-project.org'>R</a> test.")
tf

## creates a KML file containing the polygons of South Africa, Switzerland, and Canada
sw  <- wrld_simpl[wrld_simpl$NAME %in% c("South Africa", "Switzerland", "Canada"),]
out <- sapply(slot(sw, "polygons"), function(x) { kmlPolygon(x,
    name=as(sw, "data.frame")[slot(x, "ID"), "NAME"],
    col="red", lwd=1.5, border='black',
    description=paste("ISO3:", slot(x, "ID"))) })
tf <- tempfile()
kmlFile <- file(tf, "w")
tf
cat(kmlPolygon(kmlname="R Test", kmldescription="<i>Hello</i>")$header,
    file=kmlFile, sep="\n")
cat(unlist(out["style",]), file=kmlFile, sep="\n")
cat(unlist(out["content",]), file=kmlFile, sep="\n")
cat(kmlPolygon()$footer, file=kmlFile, sep="\n")
close(kmlFile)
```

---

kmlPolygons *Create and write a KML file on the basis of a given Polygons object or list of Polygons or SpatialPolygonsDataFrame*

Description

The function is used to create and write a KML file on the basis of a given Polygons object (a list of Polygon objects of SpatialPolygonsDataFrame class) for the usage in Google Earth and Google Maps.

Usage

```
kmlPolygons(obj=NULL, kmlfile=NULL,
    name="KML Polygons", description="", col=NULL, visibility=1, lwd=1,
    border="white", kmlname="", kmldescription="")
```

Arguments

`obj`	a `Polygons` or `SpatialPolygonsDataFrame` object or list of objects
`kmlfile`	if not `NULL` the name as character string of the kml file to be written to working directory as "NAME.kml"
`name`	the name of the KML polygon in Google Earth

`description`	the description of the KML polygon displayed in Google Earth or Maps (HTML tags allowed)
`col`	the fill color (see also Color Specification) of the KML polygon. If passing a list of `Polyons` or `SpatialPolygonsDataFrame` and `length(col)` is less than `length(object)` the first color in col will be applied to all objects in the list
`visibility`	if set to `1` or `TRUE` specifies that the KML polygon should be visible after loading
`lwd`	the stroke (polygon's border line) width for the KML polygon
`border`	the stroke color (see also Color Specification) for the KML polygon
`kmlname`	the name of the KML layer
`kmldescription`	the description of the KML layer (HTML tags allowed)

Details

The function is used to convert a given `Polygons` object (a list of Polygon objects) or the `Polygons` object listed in a passed `SpatialPolygonsDataFrame` object into KML polygon. If `kmlfile` is not `NULL` the result will be written into that file. If `kmlfile` is `NULL` the generated KML lines will be returned (see also value).

The conversion can also handle polygons which are marked as holes inside of the Polygons object if these holes are listed right after that polygon in which these holes appear. That implies that a given plot order set in the Polygons object will **not** be considered.

For a passed `Polygons` object the function generates a <Style> tag whereby its id attribute is set to the passed object's ID.

Note that the geometries should be in geographical coordinates with datum WGS84.

The resulting KML polygon will be embedded in `<Placemark><MultiGeometry><Polygon>`.

Value

x is a list with the elements `style` and `content` containing the generated lines of the KML file as character vectors if `kmlfile` is `NULL`.

y is a list with the elements `header` and `footer` representing the KML file' header resp. footer if `obj` is `NULL` (see second example).

Color Specification

The following color specifications are allowed: `'red'`, `2`, or as hex code `'#RRGGBB'` resp. `'#RRGGBBAA'` for passing the alpha value.

Author(s)

Hans-J. Bibiko, Jon Callihan, Steven Brey

See Also

`kmlPolygon`, `kmlLines`, `SpatialPolygons`, `kmlPoints`

Examples

```
data(wrld_simpl)
td <- tempdir()
kmlfile <- paste(td, "worldPolitical.kml", sep="/")
## creates a KML file containing the polygons of a political world map
kmlPolygons(wrld_simpl, kmlfile = kmlfile, name = "KML Polygons",
         description = "the world", col = "red",
         visibility = 1, lwd = 1, border = "white", kmlname = "R Test",
      kmldescription = "This is <b>only</b> a <a href='http://www.r-project.org'>R</a> test.")

data(wrld_simpl)
## create a KML file containing the polygons of Brazil, Uganda, and Canada
regions <- c("Brazil","Canada","Uganda")
wrld_simpl_subset <- wrld_simpl[wrld_simpl$NAME %in% regions,]
kmlfile <- paste(td, "worldPoliticalSubset.kml", sep="/")
kmlPolygons(wrld_simpl_subset, kmlfile = kmlfile,
 name = "KML Polygons subset", description = "three countries", col = "blue",
 visibility = 1, lwd = 1, border = "white", kmlname = "R Test 2",
 kmldescription = "This is <b>only</b> a <a href='http://www.r-project.org'>R</a> test.")
## combine to make a list of polygon objects to plot
polList <- c(regions,wrld_simpl)
kmlfile <- paste(td, "worldPoliticalandSubset.kml", sep="/")
kmlPolygons(wrld_simpl_subset, kmlfile = kmlfile,
 name = "KML Polygons subset", description = "three countries highlighted in world",
 col = sample(colours(), length(polList)), visibility = 1, lwd = 1, border = "white",
 kmlname = "R Test 2",
 kmldescription = "This is <b>only</b> a <a href='http://www.r-project.org'>R</a> test.")
```

`leglabs` *Make legend labels*

Description

`leglabs` makes character strings from the same break points. The `plot.polylist()` function may be used as a generic S3 method.

Usage

```
leglabs(vec, under="under", over="over", between="-", reverse=FALSE)
```

Arguments

`vec`	vector of break values
`under`	character value for under
`over`	character value for over
`between`	character value for between
`reverse`	flag to reverse order of values, you will also need to reorder colours, see example

Author(s)

Roger Bivand <Roger.Bivand@nhh.no>

See Also

findInterval

Examples

```
mappolys <- readShapeSpatial(system.file("shapes/columbus.shp", package="maptools")[1], ID="NEIGNO")
brks <- round(quantile(mappolys$CRIME, probs=seq(0,1,0.2)), digits=2)
colours <- c("salmon1", "salmon2", "red3", "brown", "black")
plot(mappolys, col=colours[findInterval(mappolys$CRIME, brks,
 all.inside=TRUE)])
legend(x=c(5.8, 7.1), y=c(13, 14.5), legend=leglabs(brks),
  fill=colours, bty="n")
title(main=paste("Columbus OH: residential burglaries and vehicle",
 "thefts per thousand households, 1980", sep="\n"))
#legend with reversed order
plot(mappolys, col=colours[findInterval(mappolys$CRIME, brks,
 all.inside=TRUE)])
legend(x=c(5.8, 7.1), y=c(13, 14.5), legend=leglabs(brks, reverse = TRUE),
  fill=rev(colours), bty="n")
title(main=paste("Columbus OH: residential burglaries and vehicle",
 "thefts per thousand households, 1980 (reversed legend)", sep="\n"))
```

---

lineLabel *Line label placement with spplot and lattice.*

Description

The `lineLabel` function produces and draws text grobs following the paths defined by a list of `Line` objects. The `sp.lineLabel` methods use this function to work easily with `spplot`.

Usage

```
lineLabel(line, label,
          spar=.6, position = c('above', 'below'),
          textloc = 'constantSlope',
          col = add.text$col,
          alpha = add.text$alpha,
          cex = add.text$cex,
          lineheight = add.text$lineheight,
          font = add.text$font,
          fontfamily = add.text$fontfamily,
          fontface = add.text$fontface,
          lty = add.line$lty,
          lwd = add.line$lwd,
          col.line = add.line$col,
          identifier = 'lineLabel',
          ...)

sp.lineLabel(object, labels, byid=TRUE,...)

label(object, text, ...)
```

Arguments

`line`	a `list` of `Lines`.
`object`	A `Lines` or `SpatialLines` object.
`label, labels, text`	a string or expression to be printed following the path of `line`. The `names` of `labels` should match the values of the `ID` slot of the lines to label. If `labels` is missing, the `ID` slot is used instead. The `label` method is a wrapper function to extract the `ID` slots and create a suitable `character` object with the correct `names` values.
`byid`	If TRUE (default) only the longest line of each unique `ID` value will be labelled.
`textloc`	a character or a numeric. It may be 'constantSlope', 'minSlope' or 'maxDepth', or the numeric index of the location. If it is a numeric, its length must coincide with the number of `Lines`.
`spar`	smoothing parameter. With values near zero, the label will closely follow the line. Default value is .6. See smooth.spline for details.
`position`	character string ('above' or 'below') to define where the text must be placed.
`col, alpha, cex, lineheight, font, fontfamily, fontface`	graphical arguments for the text. See gpar for details.
`lty, lwd, col.line`	graphical parameters for the line. See gpar for details.
`identifier`	A character string to identify the grob to be created.
`...`	other arguments

Details

Part of the label location code is adapted from `panel.levelplot`. `smooth.spline` is used to resample the segment of the line where the label is placed.

Author(s)

Oscar Perpiñán Lamigueiro.

See Also

`spplot` `sp.pointLabel` `pointLabel` `panel.levelplot` `smooth.spline`

Examples

```
data(meuse.grid)
coordinates(meuse.grid) = ~x+y
proj4string(meuse.grid) <- CRS("+init=epsg:28992")
gridded(meuse.grid) = TRUE

data(meuse)
coordinates(meuse) = ~x+y
data(meuse.riv)
meuse.sl <- SpatialLines(list(Lines(list(Line(meuse.riv)), "1")))

run <- FALSE
if (require("RColorBrewer", quietly=TRUE)) run <- TRUE
if (run) {
myCols <- adjustcolor(colorRampPalette(brewer.pal(n=9, 'Reds'))(100), .85)

labs <- label(meuse.sl, 'Meuse River')

## Maximum depth
sl1 <- list('sp.lineLabel', meuse.sl, label=labs,
            position='below', textloc='maxDepth',
            spar=.2,
            col='darkblue', cex=1,
            fontfamily='Palatino',
            fontface=2)

spplot(meuse.grid["dist"],
       col.regions=myCols,
       sp.layout = sl1)

## Constant slope
sl2 <- modifyList(sl1, list(textloc = 'constantSlope')) ## Default

spplot(meuse.grid["dist"],
       col.regions=myCols,
       sp.layout = sl2)

## Location defined by its numeric index
sl3 <- modifyList(sl1, list(textloc = 140, position='above'))

spplot(meuse.grid["dist"],
       col.regions=myCols,
       sp.layout = sl3)
}
```

map2SpatialPolygons *Convert map objects to sp classes*

Description

These functions may be used to convert map objects returned by the map function in the maps package to suitable objects defined in the sp package. In the examples below, arguments are shown for retrieving first polygons by name, then lines by window.

Usage

```
map2SpatialPolygons(map, IDs, proj4string = CRS(as.character(NA)), checkHoles=FALSE)
map2SpatialLines(map, IDs=NULL, proj4string = CRS(as.character(NA)))
pruneMap(map, xlim=NULL, ylim=NULL)
```

Arguments

`map`	a map object defined in the maps package and returned by the map function
`IDs`	Unique character ID values for each output Polygons object; the input IDs can be an integer or character vector with duplicates, where the duplicates will be combined as a single output Polygons object
`proj4string`	Object of class `"CRS"`; holding a valid proj4 string
`checkHoles`	default=FALSE, if TRUE call `checkPolygonsHoles`internally to check hole assignment, (by default no polygon objects are holes)
`xlim,ylim`	limits for pruning a map object - should only be used for lines, because polygons will not be closed

Details

Any zero area output geometries are dropped, and warnings are issued.

Value

`map2SpatialPolygons` returns a SpatialPolygons object and `map2SpatialLines` returns a SpatialLines object (objects defined in the sp package); `pruneMap` returns a modified map object defined in the maps package

Note

As the examples show, retrieval by name should be checked to see whether a window is not also needed: the "norway" polygons include "Norway:Bouvet Island", which is in the South Atlantic. Here, the IDs argument is set uniformly to "Norway" for all the component polygons, so that the output object contains a single Polygons object with multiple component Polygon objects. When retrieving by window, pruning may be needed on lines which are included because they begin within the window; interior=FALSE is used to remove country boundaries in this case.

Author(s)

Roger Bivand

See Also

`map`

Examples

```
run <- FALSE
if(require(maps)) run <- TRUE
if (run) {
nor_coast_poly <- map("world", "norway", fill=TRUE, col="transparent",
 plot=FALSE)
range(nor_coast_poly$x, na.rm=TRUE)
}
if (run) {
range(nor_coast_poly$y, na.rm=TRUE)
}
if (run) {
nor_coast_poly <- map("world", "norway", fill=TRUE, col="transparent",
 plot=FALSE, ylim=c(58,72))
nor_coast_poly$names
}
if (run) {
IDs <- sapply(strsplit(nor_coast_poly$names, ":"), function(x) x[1])
}
if (run) {
nor_coast_poly_sp <- map2SpatialPolygons(nor_coast_poly, IDs=IDs,
 proj4string=CRS("+proj=longlat +datum=WGS84"))
sapply(slot(nor_coast_poly_sp, "polygons"),
 function(x) length(slot(x, "Polygons")))
}
if (run) {
plot(nor_coast_poly_sp, col="grey", axes=TRUE)
}
if (run) {
nor_coast_lines <- map("world", interior=FALSE, plot=FALSE, xlim=c(4,32),
 ylim=c(58,72))
plot(nor_coast_lines, type="l")
}
if (run) {
nor_coast_lines <- pruneMap(nor_coast_lines, xlim=c(4,32), ylim=c(58,72))
lines(nor_coast_lines, col="red")
nor_coast_lines_sp <- map2SpatialLines(nor_coast_lines,
 proj4string=CRS("+proj=longlat +datum=WGS84"))
plot(nor_coast_poly_sp, col="grey", axes=TRUE)
}
if (run) {
plot(nor_coast_lines_sp, col="blue", add=TRUE)
}
if (run) {
worldmap <- map("world", fill=TRUE, plot=FALSE)
worldmapPolys <- map2SpatialPolygons(worldmap,
 IDs=sapply(strsplit(worldmap$names, ":"), "[", 1L),
 proj4string=CRS("+proj=longlat +datum=WGS84"))
if (rgeosStatus()) {
require(rgeos)
vals <- rgeos::gIsValid(worldmapPolys, byid=TRUE, reason=TRUE)
print(table(sapply(strsplit(vals, "\\["), "[", 1)))
}
}
```

nearestPointOnLine *Get the nearest point on a line to a given point*

Description

This function calculates the coordinates of the nearest point on a line to a given point. This function does not work with geographic coordinates.

Usage

```
nearestPointOnLine(coordsLine, coordsPoint)
```

Arguments

coordsLine	Matrix with coordinates of line vertices. Each row represents a vertex.
coordsPoint	A vector representing the X and Y coordinates of the point.

Value

Vector with the X and Y coordinates of the nearest point on a line to the given point.

Author(s)

German Carrillo

See Also

nearestPointOnSegment, snapPointsToLines

Examples

```
coordsLine = cbind(c(1,2,3),c(3,2,2))
coordsPoint = c(1.2,1.5)
nearestPointOnLine(coordsLine, coordsPoint)
```

`nearestPointOnSegment` *Get the nearest point on a segment to a given point*

Description

This function calculates the coordinates of and the distance to the nearest point on a segment to a given point. This function does not work with geographic coordinates.

Usage

```
nearestPointOnSegment(s, p)
```

Arguments

`s`	A matrix representing the coordinates of the segment. The matrix has 2x2 dimension where each row represents one of the end points.
`p`	A vector representing the X and Y coordinates of the point.

Value

A vector with three numeric values representing X and Y coordinates of the nearest point on a segment to a given point as well as the distance between both points.

Author(s)

German Carrillo

References

The function was ported to R based on this code: http://pastebin.com/n9rUuGRh

See Also

`nearestPointOnLine`, `snapPointsToLines`

Examples

```
segment = cbind(c(1,2),c(1,1.5))
point = c(1.2,1.5)
nearestPointOnSegment(segment, point)
```

nowrapRecenter *Break polygons at meridian for recentering*

Description

When recentering a world map, say to change an "Atlantic" view with longitude range -180 to 180, to a "Pacific" view, with longitude range 0 to 360, polygons crossed by the new offset, here 0/360, need to be clipped into left and right sub.polygons to avoid horizontal scratches across the map. The `nowrapSpatialPolygons` function performs this operation using polygon intersection, and `nowrapRecenter` recenters the output SpatialPolygons object.

Usage

```
nowrapRecenter(obj, offset = 0, eps = rep(.Machine$double.eps^(1/2.5), 2),
 avoidGEOS = FALSE)
nowrapSpatialPolygons(obj, offset = 0, eps=rep(.Machine$double.eps^(1/2.5), 2),
 avoidGEOS = FALSE)
```

Arguments

`obj`	A SpatialPolygons object
`offset`	offset from the Greenwich meridian
`eps`	vector of two (left and right) fuzz factors to retract the ring from the offset (2.5 root to accommodate **rgeos** precision rules)
`avoidGEOS`	default FALSE; use **polyclip** or **gpclib** code even if **rgeos** is available

Value

A SpatialPolygons object

Author(s)

Roger Bivand

See Also

`recenter-methods`, `nowrapSpatialLines`

Examples

```
run <- FALSE
if (require(maps)) run <- TRUE
## Not run:
if (run) {
world <- map("world", fill=TRUE, col="transparent", plot=FALSE)
worldSpP <- map2SpatialPolygons(world, world$names, CRS("+proj=longlat +ellps=WGS84"))
worldSpP <- worldSpP[-grep("Antarctica", row.names(worldSpP)),]
# incomplete polygons
worldSpP <- worldSpP[-grep("Ghana", row.names(worldSpP)),]
# self-intersection mouth of Volta
worldSpP <- worldSpP[-grep("UK:Great Britain", row.names(worldSpP)),]
# self-intersection Humber estuary
worldSpPr <- recenter(worldSpP)
plot(worldSpPr)
title("Pacific view without polygon splitting")
}
if (run) {
worldSpPnr <- nowrapRecenter(worldSpP)
plot(worldSpPnr)
title("Pacific view with polygon splitting")
}

## End(Not run)
if (!rgeosStatus()) run <- FALSE
if (run) {
crds <- matrix(c(-1, 1, 1, -1, 50, 50, 52, 52), ncol=2)
rcrds <- rbind(crds, crds[1,])
SR <- SpatialPolygons(list(Polygons(list(Polygon(rcrds)), ID="r1")),
 proj4string=CRS("+proj=longlat +ellps=WGS84"))
bbox(SR)
}
if (run) {
SRr <- recenter(SR)
bbox(SRr)
}
if (run) {
SRnr <- nowrapRecenter(SR)
bbox(SRnr)
}
```

nowrapSpatialLines *Split SpatialLines components at offset*

Description

When recentering a world map, most often from the "Atlantic" view with longitudes with range -180 to 180, to the "pacific" view with longitudes with range 0 to 360, lines crossing the offset (0 for this conversion) get stretched horizonally. This function breaks Line objects at the offset (usually Greenwich), inserting a very small gap, and reassembling the Line objects created as Lines. The **rgeos** package is required to use this function.

Usage

```
nowrapSpatialLines(obj, offset = 0, eps = rep(.Machine$double.eps^(1/2.5), 2))
```

Arguments

`obj`	A Spatial Lines object
`offset`	default 0, untried for other values
`eps`	vector of two fuzz values, both default 2.5 root of double.eps

Value

A Spatial Lines object

Author(s)

Roger Bivand

See Also

`recenter-methods`, `nowrapSpatialPolygons`

Examples

```
Sl <- SpatialLines(list(Lines(list(Line(cbind(sin(seq(-4,4,0.4)),
 seq(1,21,1)))), "1")), proj4string=CRS("+proj=longlat +ellps=WGS84"))
summary(Sl)
if (require(rgeos)) {
nwSL <- nowrapSpatialLines(Sl)
summary(nwSL)
if(require(maps)) {
worldmap <- map("world", plot=FALSE)
worldmapLines <- map2SpatialLines(worldmap, proj4string=CRS("+proj=longlat +datum=WGS84"))
bbox(worldmapLines)
t0 <- nowrapSpatialLines(worldmapLines, offset=180)
bbox(t0)
}
}
```

`pal2SpatialPolygons` *Making SpatialPolygons objects from RArcInfo input*

Description

This function is used in making SpatialPolygons objects from RArcInfo input.

Usage

```
pal2SpatialPolygons(arc, pal, IDs, dropPoly1=TRUE,
 proj4string=CRS(as.character(NA)))
```

Arguments

`IDs`	Unique character ID values for each output Polygons object; the input IDs can be an integer or character vector with duplicates, where the duplicates will be combined as a single output Polygons object
`proj4string`	Object of class `"CRS"`; holding a valid proj4 string
`arc`	Object returned by `get.arcdata`
`pal`	Object returned by `get.paldata`
`dropPoly1`	Should the first polygon in the AVC or e00 data be dropped; the first polygon is typically the compound boundary of the whole dataset, and can be detected by looking at the relative lengths of the list components in the second component of pal, which are the numbers of arcs making up the boundary of each polygon

Value

The functions return a SpatialPolygons object

Author(s)

Roger Bivand

Examples

```
nc1 <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1], ID="FIPS")
plot(nc1)
text(coordinates(nc1), labels=row.names(nc1), cex=0.6)
if(require(maps)){
ncmap <- map("county", "north carolina", fill=TRUE, col="transparent",
 plot=FALSE)
IDs <- sapply(strsplit(ncmap$names, "[,:]"), function(x) x[2])
nc2 <- map2SpatialPolygons(ncmap, IDs)
plot(nc2)
text(coordinates(nc2), labels=row.names(nc2), cex=0.6)
}
#if(require(RArcInfo)) {
#td <- tempdir()
#tmpcover <- paste(td, "nc", sep="/")
#if (!file.exists(tmpcover)) e00toavc(system.file("share/co37_d90.e00",
# package="maptools")[1], tmpcover)
#arc <- get.arcdata(td, "nc")
#pal <- get.paldata(td, "nc")
#pat <- get.tabledata(paste(td, "info", sep="/"), "NC.PAT")
#sapply(pal[[2]], function(x) length(x[[1]]))
#IDs <- paste(pat$ST[-1], pat$CO[-1], sep="")
#nc3 <- pal2SpatialPolygons(arc, pal, IDs=IDs)
#plot(nc3)
#text(coordinates(nc3), labels=row.names(nc3), cex=0.6)
#}
```

panel.pointLabel *Label placement with spplot and lattice.*

Description

Use optimization routines to find good locations for point labels without overlaps.

Usage

```
panel.pointLabel(x, y = NULL,
                              labels = seq(along = x),
                              method = c("SANN", "GA"),
                              allowSmallOverlap = FALSE,
                              col = add.text$col,
                              alpha = add.text$alpha,
                              cex = add.text$cex,
                              lineheight = add.text$lineheight,
                              font = add.text$font,
                              fontfamily = add.text$fontfamily,
                              fontface = add.text$fontface,
                              fill='transparent',
                              ...)

sp.pointLabel(object, labels, ...)
```

Arguments

`object`	A `SpatialPoints` object.
`x, y`	coordinates for the point labels. See `xy.coords` for details.
`labels`	a character vector or expression.
`method`	the optimization method, either SANN for simulated annealing (the default) or GA for a genetic algorithm.
`allowSmallOverlap`	logical; if TRUE, labels are allowed a small overlap. The overlap allowed is 2% of the diagonal distance of the plot area.
`col, alpha, cex, lineheight, font, fontfamily, fontface, fill`	Graphical arguments. See gpar for details
`...`	Additional arguments (currently not processed).

Author(s)

Tom Short wrote `pointLabel` for base graphics. Oscar Perpiñán Lamigueiro modified this function for `lattice` and `spplot`.

See Also

spplot

pointLabel

Examples

```
n <- 15
x <- rnorm(n)*10
y <- rnorm(n)*10
labels <- as.character(round(x, 5))


myTheme <- list(add.text=list(
                  cex=0.7,
                  col='midnightblue',
                  fontface=2,
                  fontfamily='mono'))
library(lattice)
xyplot(y~x,
       labels=labels,
       par.settings=myTheme,
       panel=function(x, y, labels, ...){
         panel.xyplot(x, y, ...)
         panel.pointLabel(x, y, labels=labels, ...)
       })



data(meuse.grid)
coordinates(meuse.grid) = ~x+y
proj4string(meuse.grid) <- CRS("+init=epsg:28992")
gridded(meuse.grid) = TRUE


pts <- spsample(meuse.grid, n=15, type="random")

Rauthors <- readLines(file.path(R.home("doc"), "AUTHORS"))[9:28]
someAuthors <- Rauthors[seq_along(pts)]

sl1 <- list('sp.points', pts, pch=19, cex=.8, col='midnightblue')
sl2 <- list('sp.pointLabel', pts, label=someAuthors,
            cex=0.7, col='midnightblue',
            fontfamily='Palatino')
run <- FALSE
if (require("RColorBrewer", quietly=TRUE)) run <- TRUE
if (run) {
myCols <- adjustcolor(colorRampPalette(brewer.pal(n=9, 'Reds'))(100), .85)
spplot(meuse.grid["dist"], col.regions=myCols, sp.layout=list(sl1, sl2))
}
```

pointLabel *Label placement for points to avoid overlaps*

Description

Use optimization routines to find good locations for point labels without overlaps.

Usage

```
pointLabel(x, y = NULL, labels = seq(along = x), cex = 1,
           method = c("SANN", "GA"),
           allowSmallOverlap = FALSE,
           trace = FALSE,
           doPlot = TRUE,
           ...)
```

Arguments

`x, y`	as with `plot.default`, these provide the x and y coordinates for the point labels. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details.
`labels`	as with `text`, a character vector or expression specifying the text to be written. An attempt is made to coerce other language objects (names and calls) to expressions, and vectors and other classed objects to character vectors by `as.character`.
`cex`	numeric character expansion factor as with `text`.
`method`	the optimization method, either "SANN" for simulated annealing (the default) or "GA" for a genetic algorithm.
`allowSmallOverlap`	logical; if `TRUE`, labels are allowed a small overlap. The overlap allowed is 2% of the diagonal distance of the plot area.
`trace`	logical; if `TRUE`, status updates are given as the optimization algorithms progress.
`doPlot`	logical; if `TRUE`, the labels are plotted on the existing graph with `text`.
`...`	arguments passed along to `text` to specify labeling parameters such as `col`.

Details

Eight positions are candidates for label placement, either horizontally, vertically, or diagonally offset from the points. The default position for labels is the top right diagonal relative to the point (considered the preferred label position).

With the default settings, simulating annealing solves faster than the genetic algorithm. It is an open question as to which settles into a global optimum the best (both algorithms have parameters that may be tweaked).

The label positioning problem is NP-hard (nondeterministic polynomial-time hard). Placement becomes difficult and slows considerably with large numbers of points. This function places all labels, whether overlaps occur or not. Some placement algorithms remove labels that overlap.

Note that only `cex` is used to calculate string width and height (using `strwidth` and `strheight`), so passing a different font may corrupt the label dimensions. You could get around this by adjusting the font parameters with `par` prior to running this function.

Value

An `xy` list giving the `x` and `y` positions of the label as would be placed by `text(xy,labels)`.

Author(s)

Tom Short, EPRI, `<tshort@epri.com>`

References

https://en.wikipedia.org/wiki/Automatic_label_placement

https://i11www.iti.uni-karlsruhe.de/map-labeling/bibliography/

http://www.eecs.harvard.edu/~shieber/Projects/Carto/carto.html

http://www.szoraster.com/Cartography/PracticalExperience.htm

The genetic algorithm code was adapted from the python code at

https://meta.wikimedia.org/wiki/Map_generator.

The simulated annealing code follows the algorithm and guidelines in:

Jon Christensen, Joe Marks, and Stuart Shieber. Placing text labels on maps and diagrams. In Paul Heckbert, editor, Graphics Gems IV, pages 497-504. Academic Press, Boston, MA, 1994. http://www.eecs.harvard.edu/~shieber/Biblio/Papers/jc.label.pdf

See Also

`text`, `thigmophobe.labels` in package **plotrix**

Examples

```
n <- 50
x <- rnorm(n)*10
y <- rnorm(n)*10
plot(x, y, col = "red", pch = 20)
pointLabel(x, y, as.character(round(x,5)), offset = 0, cex = .7)

plot(x, y, col = "red", pch = 20)
pointLabel(x, y, expression(over(alpha, beta[123])), offset = 0, cex = .8)
```

`ppp-class` *Virtual class "ppp"*

Description

Virtual S4 class definition for S3 classes in the spatstat package to allow S4-style coercion to these classes

Objects from the Class

A virtual Class: No objects may be created from it.

Author(s)

Edzer J. Pebesma

`readAsciiGrid` *read/write to/from (ESRI) asciigrid format*

Description

read/write to/from ESRI asciigrid format; a fuzz factor has been added to `writeAsciiGrid` to force cell resolution to equality if the difference is less than the square root of machine precision

Usage

```
readAsciiGrid(fname, as.image = FALSE, plot.image = FALSE,
 colname = basename(fname), proj4string = CRS(as.character(NA)),
 dec=options()$OutDec)
writeAsciiGrid(x, fname, attr = 1, na.value = -9999, dec=options()$OutDec, ...)
```

Arguments

`fname`	file name
`as.image`	logical; if TRUE, a list is returned, ready to be shown with the `image` command; if FALSE an object of class SpatialGridDataFrame-class is returned
`plot.image`	logical; if TRUE, an image of the map is plotted
`colname`	alternative name for data column if not file basename
`proj4string`	A CRS object setting the projection arguments of the Spatial Grid returned
`dec`	decimal point character. This should be a character string containing just one single-byte character — see note below.
`x`	object of class SpatialGridDataFrame

`attr`	attribute column; if missing, the first column is taken; a name or a column number may be given
`na.value`	numeric; value given to missing valued cells in the resulting map
`...`	arguments passed to write.table, which is used to write the numeric data

Value

`readAsciiGrid` returns the grid map read; either as an object of class SpatialGridDataFrame-class or, if as.image is TRUE, as list with components `x`, `y` and `z`.

Note

In ArcGIS 8, it was not in general necessary to set the `dec` argument; it is not necessary in a mixed environment with ArcView 3.2 (R writes and ArcView reads "."), but inter-operation with ArcGIS 9 requires care because the defaults used by ArcGIS seem to be misleading, and it may be necessary to override what appear to be platform defaults by setting the argument.

Author(s)

Edzer Pebesma, edzer.pebesma@uni-muenster.de

See Also

`image`, `image`

Examples

```
x <- readAsciiGrid(system.file("grids/test.ag", package="maptools")[1])
summary(x)
image(x)
xp <- as(x, "SpatialPixelsDataFrame")
abline(h=332000, lwd=3)
xpS <- xp[coordinates(xp)[,2] < 332000,]
summary(xpS)
xS <- as(xpS, "SpatialGridDataFrame")
summary(xS)
tmpfl <- paste(tempdir(), "testS.ag", sep="/")
writeAsciiGrid(xS, tmpfl)
axS <- readAsciiGrid(tmpfl)
opar <- par(mfrow=c(1,2))
image(xS, main="before export")
image(axS, main="after import")
par(opar)
unlink(tmpfl)
```

readGPS *GPSbabel read interface*

Description

The function reads a data frame from an attached GPS using the external program gpsbabel. The columns of the data frame need to be identified by hand because different GPS order NMEA data in different ways, and the columns should be converted to the correct classes by hand. Once the specifics of a particular GPS are identified, and ways of cleaning erroneous locations are found, the conversion of the output data frame into a usable one may be automated.

Usage

```
readGPS(i = "garmin", f = "usb:", type="w", invisible=TRUE, ...)
```

Arguments

`i`	INTYPE: a supported file type, default "garmin"
`f`	INFILE: the appropriate device interface, default "usb:", on Windows for serial interfaces commonly "com4:" or similar
`type`	"w" waypoints, or "t" track, or others provided in gpsbabel
`invisible`	Under Windows, do not open an extra window
`...`	arguments passed through to `read.table`

Details

The function just wraps: gpsbabel -i INTYPE -f INFILE -o tabsep -F - in system(), and reads the returned character vector of lines into a data frame. On some systems, INFILE may not be readable by ordinary users without extra configuration. The gpsbabel program must be present and on the user's PATH for the function to work. Typically, for a given GPS, the user will have to experiment first to find a set of data-cleaning tricks that work, but from then on they should be repeatable.

Value

A data frame of waypoint values

Author(s)

Patrick Giraudoux and Roger Bivand

References

https://www.gpsbabel.org

Examples

```
## Not run:
#b1 <- readGPS(f="usb:")
#str(b1)
#b2 <- b1[1:172,]
#wp0 <- b2[,c(2,3,4,8,9,19)]
#str(wp0)
#wp0$long <- wp0$V9
#wp0$lat <- as.numeric(as.character(wp0$V8))
#wp0$id <- as.character(wp0$V2)
#wp0$alt <- as.numeric(substring(as.character(wp0$V19), 1,
# (nchar(as.character(wp0$V19))-1)))
#wp0$time <- as.POSIXct(strptime(paste(as.character(wp0$V3),
# as.character(wp0$V4)), format="%d-%b-%y %H:%M:%S"))
#str(wp0)
#wp1 <- wp0[,-(1:6)]
#str(wp1)
#summary(wp1)

## End(Not run)
```

`readShapeLines` *Read arc shape files into SpatialLinesDataFrame objects*

Description

The use of this function is deprecated and it is not being maintained. Use `rgdal::readOGR()` or `sf::st_read()` instead - both of these read the coordinate reference system from the input file, while this deprecated function does not. For writing, use `rgdal::writeOGR()` or `sf::st_write()` instead.

The `readShapeLines` function reads data from an arc/line shapefile into a `SpatialLinesDataFrame` object; the shapefile may be of type polygon, but for just plotting for example coastlines, a SpatialLines object is sufficient. The `writeLinesShape` function writes data from a `SpatialLinesDataFrame` object to a shapefile. Note DBF file restrictions in `write.dbf`.

Usage

```
readShapeLines(fn, proj4string=CRS(as.character(NA)), verbose=FALSE,
 repair=FALSE, delete_null_obj=FALSE)
writeLinesShape(x, fn, factor2char = TRUE, max_nchar=254)
```

Arguments

`fn`	shapefile layer name, when writing omitting the extensions *.shp, *.shx and *.dbf, which are added in the function
`proj4string`	Object of class `CRS`; holding a valid proj4 string
`verbose`	default FALSE - report type of shapefile and number of shapes

`repair`	default FALSE: some shapefiles provided by Geolytics Inc. have values of object sizes stored in the *.shx index file that are eight bytes too large, leading the function to try to read past the end of file. If repair=TRUE, an attempt is made to repair the internal values, permitting such files to be read.
`delete_null_obj`	if TRUE, null geometries will be removed together with their data.frame rows
`x`	a `SpatialLinesDataFrame` object
`factor2char`	logical, default TRUE, convert factor columns to character
`max_nchar`	default 254, may be set to a higher limit and passed through to the DBF writer, please see Details in `write.dbf`

Details

The shpID values of the shapefile will be used as `Lines` ID values; when writing shapefiles, the object data slot row.names are added to the DBF file as column SL_ID.

Value

a SpatialLinesDataFrame object

Author(s)

Roger Bivand

See Also

`write.dbf`

Examples

```
xx <- readShapeLines(system.file("shapes/fylk-val.shp", package="maptools")[1],
 proj4string=CRS("+proj=utm +zone=33 +datum=WGS84"))
plot(xx, col="blue")
summary(xx)
xxx <- xx[xx$LENGTH > 30000,]
plot(xxx, col="red", add=TRUE)
tmpfl <- paste(tempdir(), "xxline", sep="/")
writeLinesShape(xxx, tmpfl)
getinfo.shape(paste(tmpfl, ".shp", sep=""))
axx <- readShapeLines(tmpfl, proj4string=CRS("+proj=utm +zone=33 +datum=WGS84"))
plot(xxx, col="black", lwd=4)
plot(axx, col="yellow", lwd=1, add=TRUE)
unlink(paste(tmpfl, ".*", sep=""))
xx <- readShapeLines(system.file("shapes/sids.shp", package="maptools")[1],
 proj4string=CRS("+proj=longlat +datum=NAD27"))
plot(xx, col="blue")
```

`readShapePoints` *Read points shape files into SpatialPointsDataFrame objects*

Description

The use of this function is deprecated and it is not being maintained. Use `rgdal::readOGR()` or `sf::st_read()` instead - both of these read the coordinate reference system from the input file, while this deprecated function does not.For writing, use `rgdal::writeOGR()` or `sf::st_write()` instead.

The `readShapePoints` reads data from a points shapefile into a `SpatialPointsDataFrame` object. The `writePointsShape` function writes data from a `SpatialPointsDataFrame` object to a shapefile. Both reading and writing can be carried out for 2D and 3D point coordinates. Note DBF file restrictions in `write.dbf`.

Usage

```
readShapePoints(fn, proj4string = CRS(as.character(NA)), verbose = FALSE,
 repair=FALSE)
writePointsShape(x, fn, factor2char = TRUE, max_nchar=254)
```

Arguments

`fn`	shapefile layer name, when writing omitting the extensions *.shp, *.shx and *.dbf, which are added in the function
`proj4string`	Object of class `CRS`; holding a valid proj4 string
`verbose`	default FALSE - report type of shapefile and number of shapes
`repair`	default FALSE: some shapefiles provided by Geolytics Inc. have values of object sizes stored in the *.shx index file that are eight bytes too large, leading the function to try to read past the end of file. If repair=TRUE, an attempt is made to repair the internal values, permitting such files to be read.
`x`	a `SpatialPointsDataFrame` object
`factor2char`	logical, default TRUE, convert factor columns to character
`max_nchar`	default 254, may be set to a higher limit and passed through to the DBF writer, please see Details in `write.dbf`

Value

a SpatialPointsDataFrame object

Author(s)

Roger Bivand

See Also

`write.dbf`

Examples

```
library(maptools)
xx <- readShapePoints(system.file("shapes/baltim.shp", package="maptools")[1])
plot(xx)
summary(xx)
xxx <- xx[xx$PRICE < 40,]
tmpfl <- paste(tempdir(), "xxpts", sep="/")
writePointsShape(xxx, tmpfl)
getinfo.shape(paste(tmpfl, ".shp", sep=""))
axx <- readShapePoints(tmpfl)
plot(axx, col="red", add=TRUE)
unlink(paste(tmpfl, ".*", sep=""))
xx <- readShapePoints(system.file("shapes/pointZ.shp", package="maptools")[1])
dimensions(xx)
plot(xx)
summary(xx)
```

---

readShapePoly — *Read polygon shape files into SpatialPolygonsDataFrame objects*

Description

The use of this function is deprecated and it is not being maintained. Use `rgdal::readOGR()` or `sf::st_read()` instead - both of these read the coordinate reference system from the input file, while this deprecated function does not.For writing, use `rgdal::writeOGR()` or `sf::st_write()` instead.

The `readShapePoly` reads data from a polygon shapefile into a `SpatialPolygonsDataFrame` object. The `writePolyShape` function writes data from a `SpatialPolygonsDataFrame` object to a shapefile. Note DBF file restrictions in `write.dbf`.

Usage

```
readShapePoly(fn, IDvar=NULL, proj4string=CRS(as.character(NA)),
 verbose=FALSE, repair=FALSE, force_ring=FALSE, delete_null_obj=FALSE,
 retrieve_ABS_null=FALSE)
writePolyShape(x, fn, factor2char = TRUE, max_nchar=254)
```

Arguments

`fn`	shapefile layer name, when writing omitting the extensions *.shp, *.shx and *.dbf, which are added in the function
`IDvar`	a character string: the name of a column in the shapefile DBF containing the ID values of the shapes - the values will be converted to a character vector
`proj4string`	Object of class `CRS`; holding a valid proj4 string
`verbose`	default FALSE - report type of shapefile and number of shapes

`repair`	default FALSE: some shapefiles provided by Geolytics Inc. have values of object sizes stored in the *.shx index file that are eight bytes too large, leading the function to try to read past the end of file. If repair=TRUE, an attempt is made to repair the internal values, permitting such files to be read.
`force_ring`	if TRUE, close unclosed input rings
`delete_null_obj`	if TRUE, null geometries will be removed together with their data.frame rows
`retrieve_ABS_null`	default FALSE, if TRUE and delete_null_obj also TRUE, the function will return a data frame containing the data from any null geometries inserted by ABS
`x`	a `SpatialPolygonsDataFrame` object
`factor2char`	logical, default TRUE, convert factor columns to character
`max_nchar`	default 254, may be set to a higher limit and passed through to the DBF writer, please see Details in `write.dbf`

Details

If no IDvar argument is given, the shpID values of the shapefile will be used as `Polygons` ID values; when writing shapefiles, the object data slot row.names are added to the DBF file as column SP_ID.

Value

a SpatialPolygonsDataFrame object

Author(s)

Roger Bivand

See Also

`write.dbf`

Examples

```
library(maptools)
xx <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
plot(xx, border="blue", axes=TRUE, las=1)
text(coordinates(xx), labels=row.names(xx), cex=0.6)
as(xx, "data.frame")[1:5, 1:6]
xxx <- xx[xx$SID74 < 2,]
plot(xxx, border="red", add=TRUE)
tmpfl <- paste(tempdir(), "xxpoly", sep="/")
writePolyShape(xxx, tmpfl)
getinfo.shape(paste(tmpfl, ".shp", sep=""))
axx <- readShapePoly(tmpfl, proj4string=CRS("+proj=longlat +ellps=clrk66"))
plot(xxx, border="black", lwd=4)
plot(axx, border="yellow", lwd=1, add=TRUE)
unlink(paste(tmpfl, ".*", sep=""))
```

readShapeSpatial *Read shape files into Spatial*DataFrame objects*

Description

The use of this function is deprecated and it is not being maintained. Use `rgdal::readOGR()` or `sf::st_read()` instead - both of these read the coordinate reference system from the input file, while this deprecated function does not. For writing, use `rgdal::writeOGR()` or `sf::st_write()` instead.

The `readShapeSpatial` reads data from a shapefile into a `Spatial*DataFrame` object. The `writeSpatialShape` function writes data from a `Spatial*DataFrame` object to a shapefile. Note DBF file restrictions in `write.dbf`.

Usage

```
readShapeSpatial(fn, proj4string=CRS(as.character(NA)),
verbose=FALSE, repair=FALSE, IDvar=NULL, force_ring=FALSE,
delete_null_obj=FALSE, retrieve_ABS_null=FALSE)
writeSpatialShape(x, fn, factor2char = TRUE, max_nchar=254)
```

Arguments

`fn`	shapefile layer name, when writing omitting the extensions *.shp, *.shx and *.dbf, which are added in the function
`proj4string`	Object of class `CRS`; holding a valid proj4 string
`verbose`	default FALSE - report type of shapefile and number of shapes
`repair`	default FALSE: some shapefiles provided by Geolytics Inc. have values of object sizes stored in the *.shx index file that are eight bytes too large, leading the function to try to read past the end of file. If repair=TRUE, an attempt is made to repair the internal values, permitting such files to be read.
`IDvar`	a character string: the name of a column in the shapefile DBF containing the ID values of the shapes - the values will be converted to a character vector (Polygons only)
`force_ring`	if TRUE, close unclosed input rings (Polygons only)
`delete_null_obj`	if TRUE, null geometries inserted by ABS will be removed together with their data.frame rows (Polygons and Lines)
`retrieve_ABS_null`	default FALSE, if TRUE and delete_null_obj also TRUE, the function will return a data frame containing the data from any null geometries inserted by ABS (Polygons only)
`x`	a vector data `Spatial*DataFrame` object
`factor2char`	logical, default TRUE, convert factor columns to character
`max_nchar`	default 254, may be set to a higher limit and passed through to the DBF writer, please see Details in `write.dbf`

Details

If no IDvar argument is given, the shpID values of the shapefile will be used as `Polygons` ID values; when writing shapefiles, the object data slot row.names are added to the DBF file as column SP_ID.

Value

a Spatial*DataFrame object of a class corresponding to the input shapefile

Author(s)

Roger Bivand

See Also

`write.dbf`

Examples

```
library(maptools)
xx <- readShapeSpatial(system.file("shapes/sids.shp", package="maptools")[1],
 IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
summary(xx)
xxx <- xx[xx$SID74 < 2,]
tmpfl <- paste(tempdir(), "xxpoly", sep="/")
writeSpatialShape(xxx, tmpfl)
getinfo.shape(paste(tmpfl, ".shp", sep=""))
unlink(paste(tmpfl, ".*", sep=""))
xx <- readShapeSpatial(system.file("shapes/fylk-val.shp",
 package="maptools")[1], proj4string=CRS("+proj=utm +zone=33 +datum=WGS84"))
summary(xx)
xxx <- xx[xx$LENGTH > 30000,]
plot(xxx, col="red", add=TRUE)
tmpfl <- paste(tempdir(), "xxline", sep="/")
writeSpatialShape(xxx, tmpfl)
getinfo.shape(paste(tmpfl, ".shp", sep=""))
unlink(paste(tmpfl, ".*", sep=""))
xx <- readShapeSpatial(system.file("shapes/baltim.shp", package="maptools")[1])
summary(xx)
xxx <- xx[xx$PRICE < 40,]
tmpfl <- paste(tempdir(), "xxpts", sep="/")
writeSpatialShape(xxx, tmpfl)
getinfo.shape(paste(tmpfl, ".shp", sep=""))
unlink(paste(tmpfl, ".*", sep=""))
```

readSplus *Read exported WinBUGS maps*

Description

The function permits an exported WinBUGS map to be read into an **sp** package class `SpatialPolygons` object.

Usage

```
readSplus(file, proj4string = CRS(as.character(NA)))
```

Arguments

`file`	name of file
`proj4string`	Object of class '"CRS"'; holding a valid proj4 string

Value

`readSplus` returns a SpatialPolygons object

Note

In the example, taken from the GeoBUGS manual, the smaller part of area1 has a counter-clockwise ring direction in the data, while other rings are clockwise. This implies that it is a hole, and does not get filled. Errant holes may be filled using `checkPolygonsHoles`. The region labels are stored in the `ID` slots of the `Polygons` objects.

Author(s)

Virgilio Gomez Rubio <Virgilio.Gomez@uclm.es>

References

http://www.mrc-bsu.cam.ac.uk/wp-content/uploads/geobugs12manual.pdf

See Also

`map2SpatialPolygons`

Examples

```
if (rgeosStatus()) {
geobugs <- readSplus(system.file("share/Splus.map", package="maptools"))
plot(geobugs, axes=TRUE, col=1:3)
row.names(geobugs)
pls <- slot(geobugs, "polygons")
sapply(pls, function(i) sapply(slot(i, "Polygons"), slot, "hole"))
pls1 <- lapply(pls, checkPolygonsHoles)
sapply(pls1, function(i) sapply(slot(i, "Polygons"), slot, "hole"))
plot(SpatialPolygons(pls1), axes=TRUE, col=1:3)
}
```

`Rgshhs` *Read GSHHS data into sp object*

Description

If the data are polygon data, the function will read GSHHS polygons into SpatialPolygons object for a chosen region, using binary shorelines from Global Self-consistant Hierarchical High-resolution (Shorelines) Geography, release 2.3.0 of February 1, 2014 (`http://www.soest.hawaii.edu/pwessel/gshhg/gshhg-bin-2.3.0.zip`).

The `getRgshhsMap` function calls `Rgshhs` internally to simplify the interface by returning only a SpatialPolygons object rather than a more complex list, and by calling `Rgshhs` twice either side of longitude 0 degrees for values of "xlim" straddling 0, then merging the polygons retrieved.

If the data are line data, the borders or river lines will be read into a SpatialLines object. The data are provided in integer form as millionths of decimal degrees. Reading of much earlier versions of the GSHHS binary files will fail with an error message. The netCDF GSHHS files distributed with GMT >= 4.2 cannot be read as they are in a very different format.

Usage

```
Rgshhs(fn, xlim = NULL, ylim = NULL, level = 4, minarea = 0, shift = FALSE,
verbose = TRUE, no.clip = FALSE, properly=FALSE, avoidGEOS=FALSE,
checkPolygons=FALSE)
getRgshhsMap(fn = system.file("share/gshhs_c.b", package= "maptools"),
 xlim, ylim, level = 1, shift = TRUE, verbose = TRUE, no.clip = FALSE,
 properly=FALSE, avoidGEOS=FALSE, checkPolygons=FALSE)
```

Arguments

`fn`	filename or full path to GSHHS 2.3.0 file to be read
`xlim`	longitude limits within 0-360 in most cases, negative longitudes are also found east of the Atlantic, but the Americas are recorded as positive values
`ylim`	latitude limits
`level`	maximum GSHHS level to include, defaults to 4 (everything), setting 1 will only retrieve land, no lakes
`minarea`	minimum area in square km to retrieve, default 0
`shift`	default FALSE, can be used to shift longitudes > 180 degrees to below zero, beware of artefacts involving unhandled polygon splitting at 180 degrees
`verbose`	default TRUE, print progress reports
`no.clip`	default FALSE, if TRUE, do not clip output polygons to bounding box

`properly`	default FALSE, if TRUE use `gContainsProperly` rather than `gContains`, here FALSE because clip rectangle touches clipped objects, so they are not properly contained
`avoidGEOS`	default FALSE; if TRUE force use of **gpclib** even when **rgeos** is available
`checkPolygons`	default FALSE, if TRUE, check using GEOS, which may re-order the member Polygon objects with respect to the returned polydata data frame rows

Details

The package is distributed with the coarse version of the shoreline data, and much more detailed versions may be downloaded from the referenced websites. The data is of high quality, matching the accuracy of SRTM shorelines for the full dataset (but not for inland waterbodies). In general, users will construct study region SpatialPolygons objects, which can then be exported (for example as a shapefile), or used in other R packages (such as PBSmapping). The largest land polygons take considerable time to clip to the study region, certainly many minutes for an extract from the full resolution data file including Eurasia (with Africa) or the Americas. For this reason, do not give up if nothing seems to be happening after the (verbose) message: "Rgshhs: clipping <m> of <n> polygons ..." appears. Clipping the largest polygons in full resolution also needs a good deal of memory.

Value

for polygon data, a list with the following components:

`polydata`	data from the headers of the selected GSHHS polygons
`belongs`	a matrix showing which polygon belongs to (is included in) which polygon, going from the highest level among the selected polygons down to 1 (land); levels are: 1 land, 2 lake, 3 island_in_lake, 4 pond_in_island_in_lake.
`new_belongs`	a ragged list of polygon inclusion used for making SP
`SP`	a SpatialPolygons object; this is the principal output object, and will become the only output object as the package matures

the `getRgshhsMap` returns only a SpatialPolygons object; for line data, a list with the following component:

`SP`	a SpatialLines object

Note

A number of steps are taken in this implementation that are unexpected, print messages, and so require explanation. Following the extraction of polygons intersecting the required region, a check is made to see if Antarctica is present. If it is, a new southern border is imposed at the southern ylim value or -90 if no ylim value is given. When clipping polygons seeming to intersect the required region boundary, it can happen that no polygon is left within the region (for example when the boundaries are overlaid, but also because the min/max polygon values in the header may not agree with the polygon itself (one case observed for a lake west of Groningen). The function then reports a null polygon. Another problem occurs when closed polygons are cut up during the finding of intersections between polygons and the required region boundary.

By default, if the rgeos package is available, it is used for topology operations. If it is not available, the gpclib package may be used. Please also note that gpclib has a restricted licence.

Author(s)

Roger Bivand

References

http://www.soest.hawaii.edu/pwessel/gshhg/, http://www.soest.hawaii.edu/pwessel/gshhg/gshhg-bin-2.3.0.zip; Wessel, P., and W. H. F. Smith, A Global Self-consistent, Hierarchical, High-resolution Shoreline Database, J. Geophys. Res., 101, 8741-8743, 1996.

Examples

```
if (rgeosStatus()) {
gshhs.c.b <- system.file("share/gshhs_c.b", package="maptools")
WEx <- c(-12, 3)
WEy <- c(48, 59)
WE <- getRgshhsMap(gshhs.c.b, xlim=WEx, ylim=WEy)
plot(WE, col="khaki", xlim=WEx, ylim=WEy, xaxs="i", yaxs="i", axes=TRUE)
NZx <- c(160,180)
NZy <- c(-50,-30)
NZ <- Rgshhs(gshhs.c.b, xlim=NZx, ylim=NZy)
plot(NZ$SP, col="khaki", pbg="azure2", xlim=NZx, ylim=NZy, xaxs="i", yaxs="i", axes=TRUE)
GLx <- c(265,285)
GLy <- c(40,50)
GL <- Rgshhs(gshhs.c.b, xlim=GLx, ylim=GLy)
plot(GL$SP, col="khaki", pbg="azure2", xlim=GLx, ylim=GLy, xaxs="i", yaxs="i", axes=TRUE)
BNLx <- c(2,8)
BNLy <- c(49,54)
wdb_lines <- system.file("share/wdb_borders_c.b", package="maptools")
BNLp <- Rgshhs(gshhs.c.b, xlim=BNLx, ylim=BNLy)
BNLl <- Rgshhs(wdb_lines, xlim=BNLx, ylim=BNLy)
plot(BNLp$SP, col="khaki", pbg="azure2", xlim=BNLx, ylim=BNLy, xaxs="i", yaxs="i", axes=TRUE)
lines(BNLl$SP)
xlims <- c(0,360)
ylims <- c(-90,90)
world <- Rgshhs(gshhs.c.b, xlim=xlims, ylim=ylims, level=1, checkPolygons=TRUE)
}
```

---

snapPointsToLines *Snap a set of points to a set of lines*

Description

This function snaps a set of points to a set of lines based on the minimum distance of each point to any of the lines. This function does not work with geographic coordinates.

Usage

```
snapPointsToLines(points, lines, maxDist=NA, withAttrs = TRUE, idField=NA)
```

Arguments

`points`	An object of the class SpatialPoints or SpatialPointsDataFrame.
`lines`	An object of the class SpatialLines or SpatialLinesDataFrame.
`maxDist`	Numeric value for establishing a maximum distance to avoid snapping points that are farther apart; its default value is NA.
`withAttrs`	Boolean value for preserving (TRUE) or getting rid (FALSE) of the original point attributes. Default: TRUE. This parameter is optional.
`idField`	A string specifying the field which contains each line's id. This id will be transferred to the snapped points data set to distinguish the line which each point was snapped to.

Value

SpatialPointsDataFrame object as defined by the R package 'sp'. This object contains the snapped points, therefore all of them lie on the lines.

Author(s)

German Carrillo and Ethan Plunkett

See Also

`nearestPointOnSegment`, `nearestPointOnLine`, `sp`

Examples

```
# From the sp vignette
l1 = cbind(c(1,2,3),c(3,2,2))
l1a = cbind(l1[,1]+.05,l1[,2]+.05)
l2 = cbind(c(1,2,3),c(1,1.5,1))
Sl1 = Line(l1)
Sl1a = Line(l1a)
Sl2 = Line(l2)
S1 = Lines(list(Sl1, Sl1a), ID="a")
S2 = Lines(list(Sl2), ID="b")
Sl = SpatialLines(list(S1,S2))
df = data.frame(z = c(1,2), row.names=sapply(slot(Sl, "lines"), function(x) slot(x, "ID")))
Sldf = SpatialLinesDataFrame(Sl, data = df)

xc = c(1.2,1.5,2.5)
yc = c(1.5,2.2,1.6)
Spoints = SpatialPoints(cbind(xc, yc))

if (rgeosStatus()) snapPointsToLines(Spoints, Sldf, maxDist=0.4)
```

sp2Mondrian *write map data for Mondrian*

Description

The function outputs a SpatialPolygonsDataFrame object to be used by Mondrian

Usage

```
sp2Mondrian(SP, file, new_format=TRUE)
```

Arguments

`SP`	a SpatialPolygonsDataFrame object
`file`	file where output is written
`new_format`	default TRUE, creates a text data file and a separate map file; the old format put both data sets in a single file - the map file is named by inserting "MAP_" into the file= argument after the rightmost directory separator (if any)

Note

At this release, the function writes out a text file with both data and polygon(s) identified as belonging to each row of data.

Author(s)

Patrick Hausmann and Roger Bivand

References

http://www.theusrus.de/Mondrian/

Examples

```
## Not run:
td <- tempdir()
xx <- readShapePoly(system.file("shapes/columbus.shp", package="maptools")[1])
sp2Mondrian(xx, file=file.path(td, "columbus1.txt"))
xx <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1])
sp2Mondrian(xx, file=file.path(td, "sids1.txt"))

## End(Not run)
```

sp2tmap *Convert SpatialPolygons object for Stata tmap command*

Description

The function converts a SpatialPolygons object for use with the Stata tmap command, by creating a data frame with the required columns.

Usage

```
sp2tmap(SP)
```

Arguments

`SP`	a SpatialPolygons object

Value

a data frame with three columns:

`_ID`	an integer vector of polygon identifiers in numeric order
`_X`	numeric x coordinate
`_Y`	numeric y coordinate

and an `ID_n` attribute with the named polygon identifiers

Author(s)

Roger Bivand

References

https://www.stata.com/search.cgi?query=tmap

See Also

write.dta

Examples

```
## Not run:
xx <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
plot(xx, border="blue", axes=TRUE, las=1)
tmapdf <- sp2tmap(as(xx, "SpatialPolygons"))
if (require(foreign)) {
td <- tempdir()
write.dta(tmapdf, file=file.path(td, "NCmap.dta"), version=7)
NCdf <- as(xx, "data.frame")
NCdf$ID_n <- attr(tmapdf, "ID_names")
write.dta(NCdf, file=file.path(td, "NC.dta"), version=7)
}

## End(Not run)
```

---

sp2WB *Export SpatialPolygons object as S-Plus map for WinBUGS*

Description

The function exports an sp SpatialPolygons object into a S-Plus map format to be import by WinBUGS.

Usage

```
sp2WB(map, filename, Xscale = 1, Yscale = Xscale, plotorder = FALSE)
```

Arguments

`map`	a SpatialPolygons object
`filename`	file where output is written
`Xscale, Yscale`	scales to be written in the output file
`plotorder`	default=FALSE, if TRUE, export polygons in plotting order

Author(s)

Virgilio GÃ³mez Rubio, partly derived from earlier code by Thomas Jagger

References

http://www.mrc-bsu.cam.ac.uk/wp-content/uploads/geobugs12manual.pdf

Examples

```
xx <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
plot(xx, border="blue", axes=TRUE, las=1)
tf <- tempfile()
sp2WB(as(xx, "SpatialPolygons"), filename=tf)
xxx <- readSplus(tf, proj4string=CRS("+proj=longlat +ellps=clrk66"))
all.equal(xxx, as(xx, "SpatialPolygons"), tolerance=.Machine$double.eps^(1/4),
 check.attributes=FALSE)
## Not run:
x <- readAsciiGrid(system.file("grids/test.ag", package="maptools")[1])
xp <- as(x, "SpatialPixelsDataFrame")
pp <- as(xp, "SpatialPolygons")
td <- tempdir()
```

```
sp2WB(pp, filename=file.path(td, "test.map"))

## End(Not run)
```

SpatialLines2PolySet *Convert sp line and polygon objects to PBSmapping PolySet objects*

Description

Functions SpatialLines2PolySet and SpatialPolygons2PolySet convert objects of sp classes to PolySet class objects as defined in the PBSmapping package, and PolySet2SpatialLines and PolySet2SpatialPolygons convert in the opposite direction.

Usage

```
SpatialLines2PolySet(SL)
SpatialPolygons2PolySet(SpP)
PolySet2SpatialLines(PS)
PolySet2SpatialPolygons(PS, close_polys=TRUE)
```

Arguments

SL	a SpatialLines object as defined in the sp package
SpP	a SpatialPolygons object as defined in the sp package
PS	a PolySet object
close_polys	should polygons be closed if open

Value

PolySet objects as defined in the PBSmapping package

Author(s)

Roger Bivand and Andrew Niccolai

See Also

PolySet, MapGen2SL

Examples

```
if(require(PBSmapping) && require(maps)) {
nor_coast_lines <- map("world", interior=FALSE, plot=FALSE, xlim=c(4,32),
 ylim=c(58,72))
nor_coast_lines <- pruneMap(nor_coast_lines, xlim=c(4,32), ylim=c(58,72))
nor_coast_lines_sp <- map2SpatialLines(nor_coast_lines,
 proj4string=CRS("+proj=longlat +datum=WGS84 +ellps=WGS84"))
nor_coast_lines_PS <- SpatialLines2PolySet(nor_coast_lines_sp)
summary(nor_coast_lines_PS)
plotLines(nor_coast_lines_PS)
o3 <- PolySet2SpatialLines(nor_coast_lines_PS)
plot(o3, axes=TRUE)
nor_coast_poly <- map("world", "norway", fill=TRUE, col="transparent",
 plot=FALSE, ylim=c(58,72))
IDs <- sapply(strsplit(nor_coast_poly$names, ":"), function(x) x[1])
nor_coast_poly_sp <- map2SpatialPolygons(nor_coast_poly, IDs=IDs,
 proj4string=CRS("+proj=longlat +datum=WGS84 +ellps=WGS84"))
nor_coast_poly_PS <- SpatialPolygons2PolySet(nor_coast_poly_sp)
summary(nor_coast_poly_PS)
plotPolys(nor_coast_poly_PS)
o1 <- PolySet2SpatialPolygons(nor_coast_poly_PS)
plot(o1, axes=TRUE)
}
```

SpatialLinesMidPoints *Line midpoints*

Description

The function onverts SpatialLinesDataFrame to SpatialPointsDataFrame with points at the midpoints of the line segments.

Usage

```
SpatialLinesMidPoints(sldf)
```

Arguments

sldf	A SpatialLines or SpatialLinesDataFrame object

Details

The function builds a SpatialPointsDataFrame from the midpoints of Line objects belonging to Lines objects in an object inheriting from a Spatial Lines object. The output data slot contains an index variable showing which Lines object the midpoints belong to.

Value

A SpatialPointsDataFrame object created from the input object.

Author(s)

Jonathan Callahan, modified by Roger Bivand

Examples

```
xx <- readShapeLines(system.file("shapes/fylk-val.shp", package="maptools")[1],
 proj4string=CRS("+proj=utm +zone=33 +datum=WGS84"))
plot(xx, col="blue")
spdf <- SpatialLinesMidPoints(xx)
plot(spdf, col="orange", add=TRUE)
```

`spCbind-methods` *cbind for spatial objects*

Description

`spCbind` provides cbind-like methods for Spatial*DataFrame objects in addition to the `$`, `[<-` and `[[<-` methods already available.

Methods

obj = "SpatialPointsDataFrame", x = "data.frame" cbind a data frame to the data slot of a SpatialPointsDataFrame object

obj = "SpatialPointsDataFrame", x = "vector" cbind a vector to the data slot of a SpatialPointsDataFrame object

obj = "SpatialLinesDataFrame", x = "data.frame" cbind a data frame to the data slot of a SpatialLinesDataFrame object; the data frame argument must have row names set to the Lines ID values, and should be re-ordered first by matching against a shared key column

obj = "SpatialLinesDataFrame", x = "vector" cbind a vector to the data slot of a SpatialLinesDataFrame object

obj = "SpatialPolygonsDataFrame", x = "data.frame" cbind a data frame to the data slot of a SpatialPolygonsDataFrame object; the data frame argument must have row names set to the Polygons ID values, and should be re-ordered first by matching against a shared key column

obj = "SpatialPolygonsDataFrame", x = "vector" cbind a vector to the data slot of a SpatialPolygonsDataFrame object

Author(s)

Roger Bivand

See Also

`spChFIDs-methods`, `spRbind-methods`

Examples

```
xx <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
  IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
library(foreign)
xtra <- read.dbf(system.file("share/nc_xtra.dbf", package="maptools")[1])
o <- match(xx$CNTY_ID, xtra$CNTY_ID)
xtra1 <- xtra[o,]
row.names(xtra1) <- xx$FIPSNO
xx1 <- spCbind(xx, xtra1)
names(xx1)
identical(xx1$CNTY_ID, xx1$CNTY_ID.1)
```

`SplashDams` *Data for Splash Dams in western Oregon*

Description

Data for Splash Dams in western Oregon

Usage

```
data(SplashDams)
```

Format

The format is: Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots ..@ data :'data.frame': 232 obs. of 6 variables: .. ..$ streamName : Factor w/ 104 levels "Abiqua Creek",..: 12 12 60 60 60 49 49 9 9 18 ... .$ locationCode: Factor w/ 3 levels "h","l","m": 1 1 1 1 1 1 1 1 1 1 ... .$ height : int [1:232] 4 4 NA NA NA NA 10 NA NA NA ... .$ lastDate : int [1:232] 1956 1956 1957 1936 1936 1929 1909 1919 1919 1919 ... .$ owner : Factor w/ 106 levels "A. Stefani","A.H. Blakesley",..: 42 42 42 84 84 24 24 25 25 25 ... .$ datesUsed : Factor w/ 118 levels "?-1870s-1899-?",..: 92 92 93 91 91 72 61 94 94 94 ... ..@ coords.nrs : num(0) ..@ coords : num [1:232, 1:3] -124 -124 -124 -124 -124 ... .- attr(*, "dimnames")=List of 2 .. .. ..$: NULL .. .. ..$: chr [1:3] "coords.x1" "coords.x2" "coords.x3" ..@ bbox : num [1:3, 1:2] -124.2 42.9 0 -122.4 46.2 ... .- attr(*, "dimnames")=List of 2 .. .. ..$: chr [1:3] "coords.x1" "coords.x2" "coords.x3" .. .. ..$: chr [1:2] "min" "max" ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots .. .. ..@ projargs: chr " +proj=longlat +ellps=WGS84"

Source

R. R. Miller (2010) Is the Past Present? Historical Splash-dam Mapping and Stream Disturbance Detection in the Oregon Coastal Province. MSc. thesis, Oregon State University; packaged by Jonathan Callahan

References

https://www.fs.fed.us/pnw/lwm/aem/docs/burnett/miller_rebecca_r2010rev.pdf

Examples

```
data(SplashDams)
plot(SplashDams, axes=TRUE)
```

---

spRbind-methods *rbind for spatial objects*

Description

`spRbind` provides rbind-like methods for Spatial*DataFrame objects

Methods

obj = "SpatialPoints", x = "SpatialPoints" rbind two SpatialPoints objects

obj = "SpatialPointsDataFrame", x = "SpatialPointsDataFrame" rbind two SpatialPointsDataFrame objects

obj = "SpatialLines", x = "SpatialLines" rbind two SpatialLines objects

obj = "SpatialLinesDataFrame", x = "SpatialLinesDataFrame" rbind two SpatialLinesDataFrame objects

obj = "SpatialPolygons", x = "SpatialPolygons" rbind two SpatialPolygons objects

obj = "SpatialPolygonsDataFrame", x = "SpatialPolygonsDataFrame" rbind two SpatialPolygonsDataFrame objects

Note

In addition to the `spRbind`-methods, there are also `rbind`-methods for Spatial* objects. The differences are:

1. `spRbind`-methods can bind 2 objects, whereas `rbind`-methods can bind multiple object
2. some `rbind` can accept objects with duplicated IDs, for all `spRbind`-methods these have to be modified explicitly, e.g. by calling `spChFIDs-methods`

Author(s)

Roger Bivand

See Also

`spChFIDs-methods`, `spCbind-methods`

Examples

```
xx <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
  IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
summary(xx)
xx$FIPSNO
xx1 <- xx[xx$CNTY_ID < 1982,]
xx2 <- xx[xx$CNTY_ID >= 1982,]
xx3 <- spRbind(xx2, xx1)
summary(xx3)
xx3$FIPSNO
```

`state.vbm` *US State Visibility Based Map*

Description

A SpatialPolygonsDataFrame object (for use with the maptools package) to plot a Visibility Based Map.

Usage

```
data(state.vbm)
```

Details

A SpatialPolygonsDataFrame object (for use with the maptools package) to plot a map of the US states where the sizes of the states have been adjusted to be more equal.

This map can be useful for plotting state data using colors patterns without the larger states dominating and the smallest states being lost.

The original map is copyrighted by Mark Monmonier. Official publications based on this map should acknowledge him. Comercial publications of maps based on this probably need permission from him to use.

Author(s)

Greg Snow <`greg.snow@imail.org`> (of this compilation)

Source

The data was converted from the maps library for S-PLUS. S-PLUS uses the map with permission from the author. This version of the data has not received permission from the author (no attempt made, not that it was refused), most of my uses I feel fall under fair use and do not violate copyright, but you will need to decide for yourself and your applications.

References

`http://www.markmonmonier.com/index.htm`, `http://euclid.psych.yorku.ca/SCS/Gallery/bright-ideas.html`

Examples

```
data(state.vbm)
plot(state.vbm)

tmp <- state.x77[,'HS Grad']
tmp2 <- cut(tmp, seq(min(tmp),max(tmp), length.out=11),
  include.lowest=TRUE)
plot(state.vbm,col=cm.colors(10)[tmp2])
```

---

sun-methods *Methods for sun ephemerides calculations*

Description

Functions for calculating sunrise, sunset, and times of dawn and dusk, with flexibility for the various formal definitions. They use algorithms provided by the National Oceanic & Atmospheric Administration (NOAA).

Usage

```
## S4 method for signature 'SpatialPoints,POSIXct'
crepuscule(crds, dateTime, solarDep, direction=c("dawn", "dusk"),
           POSIXct.out=FALSE)
## S4 method for signature 'matrix,POSIXct'
crepuscule(crds, dateTime,
           proj4string=CRS("+proj=longlat +datum=WGS84"), solarDep,
           direction=c("dawn", "dusk"), POSIXct.out=FALSE)
## S4 method for signature 'SpatialPoints,POSIXct'
sunriset(crds, dateTime, direction=c("sunrise", "sunset"),
         POSIXct.out=FALSE)
## S4 method for signature 'matrix,POSIXct'
sunriset(crds, dateTime,
         proj4string=CRS("+proj=longlat +datum=WGS84"),
         direction=c("sunrise", "sunset"), POSIXct.out=FALSE)
## S4 method for signature 'SpatialPoints,POSIXct'
solarnoon(crds, dateTime, POSIXct.out=FALSE)
## S4 method for signature 'matrix,POSIXct'
solarnoon(crds, dateTime,
          proj4string=CRS("+proj=longlat +datum=WGS84"),
          POSIXct.out=FALSE)
## S4 method for signature 'SpatialPoints,POSIXct'
solarpos(crds, dateTime, ...)
## S4 method for signature 'matrix,POSIXct'
solarpos(crds, dateTime,
         proj4string=CRS("+proj=longlat +datum=WGS84"), ...)
```

Arguments

`crds`	a `SpatialPoints` or `matrix` object, containing `x` and `y` coordinates (in that order).
`dateTime`	a POSIXct object with the date and time associated to calculate ephemerides for points given in crds.
`solarDep`	numeric vector with the angle of the sun below the horizon in degrees.
`direction`	one of "dawn", "dusk", "sunrise", or "sunset", indicating which ephemerides should be calculated.
`POSIXct.out`	logical indicating whether POSIXct output should be included.
`proj4string`	string with valid projection string describing the projection of data in `crds`.
`...`	other arguments passed through.

Details

NOAA used the reference below to develop their Sunrise/Sunset

`https://gml.noaa.gov/grad/solcalc/sunrise.html` and Solar Position

`https://gml.noaa.gov/grad/solcalc/azel.html` Calculators. The algorithms include corrections for atmospheric refraction effects.

Input can consist of one location and at least one POSIXct times, or one POSIXct time and at least one location. *solarDep* is recycled as needed.

Do not use the daylight savings time zone string for supplying *dateTime*, as many OS will not be able to properly set it to standard time when needed.

Value

`crepuscule`, `sunriset`, and `solarnoon` return a numeric vector with the time of day at which the event occurs, expressed as a fraction, if POSIXct.out is FALSE; otherwise they return a data frame with both the fraction and the corresponding POSIXct date and time.

`solarpos` returns a matrix with the solar azimuth (in degrees from North), and elevation.

Warning

Compared to NOAA's original Javascript code, the sunrise and sunset estimates from this translation may differ by +/- 1 minute, based on tests using selected locations spanning the globe. This translation does not include calculation of prior or next sunrises/sunsets for locations above the Arctic Circle or below the Antarctic Circle.

Note

NOAA notes that "for latitudes greater than 72 degrees N and S, calculations are accurate to within 10 minutes. For latitudes less than +/- 72 degrees accuracy is approximately one minute."

Author(s)

Sebastian P. Luque <spluque@gmail.com>, translated from Greg Pelletier's <gpel461@ecy.wa.gov> VBA code (available from https://ecology.wa.gov/Research-Data/Data-resources/Models-spreadsheets/Modeling-the-environment/Models-tools-for-TMDLs), who in turn translated it from original Javascript code by NOAA (see Details). Roger Bivand <roger.bivand@nhh.no> adapted the code to work with **sp** classes.

References

Meeus, J. (1991) Astronomical Algorithms. Willmann-Bell, Inc.

Examples

```
## Location of Helsinki, Finland, in decimal degrees,
## as listed in NOAA's website
hels <- matrix(c(24.97, 60.17), nrow=1)
Hels <- SpatialPoints(hels, proj4string=CRS("+proj=longlat +datum=WGS84"))
d041224 <- as.POSIXct("2004-12-24", tz="EET")
## Astronomical dawn
crepuscule(hels, d041224, solarDep=18, direction="dawn", POSIXct.out=TRUE)
crepuscule(Hels, d041224, solarDep=18, direction="dawn", POSIXct.out=TRUE)
## Nautical dawn
crepuscule(hels, d041224, solarDep=12, direction="dawn", POSIXct.out=TRUE)
crepuscule(Hels, d041224, solarDep=12, direction="dawn", POSIXct.out=TRUE)
## Civil dawn
crepuscule(hels, d041224, solarDep=6, direction="dawn", POSIXct.out=TRUE)
crepuscule(Hels, d041224, solarDep=6, direction="dawn", POSIXct.out=TRUE)
solarnoon(hels, d041224, POSIXct.out=TRUE)
solarnoon(Hels, d041224, POSIXct.out=TRUE)
solarpos(hels, as.POSIXct(Sys.time(), tz="EET"))
solarpos(Hels, as.POSIXct(Sys.time(), tz="EET"))
sunriset(hels, d041224, direction="sunrise", POSIXct.out=TRUE)
sunriset(Hels, d041224, direction="sunrise", POSIXct.out=TRUE)
## Using a sequence of dates
Hels_seq <- seq(from=d041224, length.out=365, by="days")
up <- sunriset(Hels, Hels_seq, direction="sunrise", POSIXct.out=TRUE)
down <- sunriset(Hels, Hels_seq, direction="sunset", POSIXct.out=TRUE)
day_length <- down$time - up$time
plot(Hels_seq, day_length, type="l")

## Using a grid of spatial points for the same point in time
## Not run:
grd <- GridTopology(c(-179,-89), c(1,1), c(359,179))
SP <- SpatialPoints(coordinates(grd),
                    proj4string=CRS("+proj=longlat +datum=WGS84"))
wint <- as.POSIXct("2004-12-21", tz="GMT")
win <- crepuscule(SP, wint, solarDep=6, direction="dawn")
SPDF <- SpatialGridDataFrame(grd,
 proj4string=CRS("+proj=longlat +datum=WGS84"),
 data=data.frame(winter=win))
image(SPDF, axes=TRUE, col=cm.colors(40))
```

```
## End(Not run)
```

`symbolsInPolys` *Place grids of points over polygons*

Description

Place grids of points over polygons with chosen density and/or symbols (suggested by Michael Wolf).

Usage

```
symbolsInPolys(pl, dens, symb = "+", compatible = FALSE)
```

Arguments

`pl`	an object of class SpatialPolygons or SpatialPolygonsDataFrame
`dens`	number of symbol plotting points per unit area; either a single numerical value for all polygons, or a numeric vector the same length as pl with values for each polygon
`symb`	plotting symbol; either a single value for all polygons, or a vector the same length as pl with values for each polygon
`compatible`	what to return, if TRUE a a list of matrices of point coordinates, one matrix for each member of pl, with a `symb` attribute, if false a SpatialPointsDataFrame with a `symb` column

Details

The dots are placed in a grid pattern with the number of points per polygon being polygon area times density (number of dots not guaranteed to be the same as the count). When the polygon is made up of more than one part, the dots will be placed in proportion to the relative areas of the clockwise rings (anticlockwise are taken as holes). From maptools release 0.5-2, correction is made for holes in the placing of the dots, but depends on hole values being correctly set, which they often are not.

Value

The function returns a list of matrices of point coordinates, one matrix for each member of pl; each matrix has a `symb` attribute that can be used for setting the `pch` argument for plotting. If the count of points for the given density and polygon area is zero, the list element is NULL, and can be tested when plotting - see the examples.

Note

Extension to plot pixmaps at the plotting points using addlogo() from the pixmap package is left as an exercise for the user.

Author(s)

Roger Bivand <Roger.Bivand@nhh.no>

See Also

spsample

Examples

```
nc_SP <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 proj4string=CRS("+proj=longlat +ellps=clrk66"))
## Not run:
pls <- slot(nc_SP, "polygons")
pls_new <- lapply(pls, checkPolygonsHoles)
nc_SP <- SpatialPolygonsDataFrame(SpatialPolygons(pls_new,
 proj4string=CRS(proj4string(nc_SP))), data=as(nc_SP, "data.frame"))

## End(Not run)
symbs <- c("-", "+", "x")
np <- sapply(slot(nc_SP, "polygons"), function(x) length(slot(x, "Polygons")))
try1 <- symbolsInPolys(nc_SP, 100, symb=symbs[np])
plot(nc_SP, axes=TRUE)
plot(try1, add=TRUE, pch=as.character(try1$symb))
```

---

thinnedSpatialPoly *Douglas-Peuker line generalization for Spatial Polygons*

Description

The function applies the implementation of the Douglas-Peuker algorithm for line generalization or simplification (originally from shapefiles) to objects inheriting from Spatial Polygons. It does not preserve topology, so is suitable for visualisation, but not for the subsequent analysis of the polygon boundaries, as artefacts may be created, and boundaries of neighbouring entities may be generalized differently. If the rgeos package is available, thinnedSpatialPolyGEOS will be used with partial topology preservation instead of the R implementation here by passing arguments through.

Usage

```
thinnedSpatialPoly(SP, tolerance, minarea=0, topologyPreserve = FALSE,
                   avoidGEOS = FALSE)
```

Arguments

SP	an object inheriting from the SpatialPolygons class
tolerance	the tolerance value in the metric of the input object
minarea	the smallest area of Polygon objects to be retained, ignored if **rgeos** used

`topologyPreserve`	choose between two **rgeos** options: logical determining if the algorithm should attempt to preserve the topology (nodes not complete edges) of the original geometry
`avoidGEOS`	use R DP code even if **rgeos** is available

Value

An object of the same class as the input object

Note

Warnings reporting: Non-finite label point detected and replaced, reflect the changes in the geometries of the polygons induced by line generalization.

Author(s)

Ben Stabler, Michael Friendly, Roger Bivand

References

Douglas, D. and Peucker, T. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer* 10(2). 112-122.

Examples

```
xx <- readShapeSpatial(system.file("shapes/sids.shp", package="maptools")[1],
      IDvar="FIPSNO", proj4string=CRS("+proj=longlat +ellps=clrk66"))
object.size(as(xx, "SpatialPolygons"))
xxx <- thinnedSpatialPoly(xx, tolerance=0.05, minarea=0.001)
object.size(as(xxx, "SpatialPolygons"))
par(mfrow=c(2,1))
plot(xx)
plot(xxx)
par(mfrow=c(1,1))
```

`unionSpatialPolygons` *Aggregate Polygons in a SpatialPolygons object*

Description

The function aggregates Polygons in a SpatialPolygons object, according to the IDs vector specifying which input Polygons belong to which output Polygons; internal boundaries are dissolved using the rgeos package `gUnaryUnion` function. If the rgeos package is not available, and if the gpclib package is available and the user confirms that its restrictive license conditions are met, its `union` function will be used.

Usage

```
unionSpatialPolygons(SpP, IDs, threshold=NULL, avoidGEOS=FALSE, avoidUnaryUnion=FALSE)
```

Arguments

`SpP`	A SpatialPolygons object as defined in package sp
`IDs`	A vector defining the output Polygons objects, equal in length to the length of the polygons slot of SpRs; it may be character, integer, or factor (try table(factor(IDs)) for a sanity check). It may contain NA values for input objects not included in the union
`threshold`	if not NULL, an area measure below which slivers will be discarded (some polygons have non-identical boundaries, for instance along rivers, generating slivers on union which are artefacts, not real sub-polygons)
`avoidGEOS`	default FALSE; if TRUE force use of gpclib even when GEOS is available
`avoidUnaryUnion`	avoid gUnaryUnion if it is available; not relevant before GEOS 3.3.0

Value

Returns an aggregated SpatialPolygons object named with the aggregated IDs values in their sorting order; see the ID values of the output object to view the order.

Warning

When using GEOS Unary Union, it has been found that some polygons are not dissolved correctly when the absolute values of the coordinates are very small. No work-around is available at present.

Author(s)

Roger Bivand

Examples

```
if (rgeosStatus()) {
nc1 <- readShapePoly(system.file("shapes/sids.shp", package="maptools")[1],
 proj4string=CRS("+proj=longlat +datum=NAD27"))
lps <- coordinates(nc1)
ID <- cut(lps[,1], quantile(lps[,1]), include.lowest=TRUE)
reg4 <- unionSpatialPolygons(nc1, ID)
row.names(reg4)
}
```

`wrld_simpl` *Simplified world country polygons*

Description

The object loaded is a `SpatialPolygonsDataFrame` object containing a slightly modified version of Bjoern Sandvik's improved version of world_borders.zip - TM_WORLD_BORDERS_SIMPL-0.2.zip dataset from the Mapping Hacks geodata site. The country Polygons objects and the data slot data frame row numbers have been set to the ISO 3166 three letter codes.

Usage

```
data(wrld_simpl)
```

Format

The format is: Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots; the data clot contains a data.frame with 246 obs. of 11 variables:

FIPS factor of FIPS country codes

ISO2 factor of ISO 2 character country codes

ISO3 factor of ISO 3 character country codes

UN integer vector of UN country codes

NAME Factor of country names

AREA integer vector of area values

POP2005 integer vector of population in 2005

REGION integer vector of region values

SUBREGION integer vector of subregion values

LON numeric vector of longitude label points

LAT numeric vector of latitude label points

The object is in geographical coordinates using the WGS84 datum.

Source

Originally "http://mappinghacks.com/data/TM_WORLD_BORDERS_SIMPL-0.2.zip", now available from `https://github.com/nasa/World-Wind-Java/tree/master/WorldWind/testData/shapefiles`

Examples

```
data(wrld_simpl)
plot(wrld_simpl)
```

Index