Package: tigers (via r-universe)

October 17, 2024

Version 0.1-3.3

Date 2024-09-17

Title Integration of Geography, Environment, and Remote Sensing

Imports stats

ZipData no

Description Handling and manipulation polygons, coordinates, and other geographical objects. The tools include: polygon areas, barycentric and trilinear coordinates (Hormann and Floater, 2006, <doi:10.1145/1183287.1183295>), convex hull for polygons (Graham and Yao, 1983, <doi:10.1016/0196-6774(83)90013-5>), polygon triangulation (Toussaint, 1991, <doi:10.1007/BF01905693>), great circle and geodesic distances, Hausdorff distance, and reduced major axis.

License GPL-3

URL `https://github.com/emmanuelparadis/tigers`

BugReports `https://github.com/emmanuelparadis/tigers/issues`

Repository https://emmanuelparadis.r-universe.dev

RemoteUrl https://github.com/emmanuelparadis/tigers

RemoteRef HEAD

RemoteSha 563948b60ac118d102a3fa54f5d8420dde35fe83

Contents

`tigers-package` *Integration of Geography, Environment, and Remote Sensing*

Description

tigers provides functions for manipulating polygons, coordinates, . . .

All the tools programmed in **tigers** are “class-free”: they work on numeric vectors or matrices (even data frames) that store coordinates. So the functions in the present package can easily be interfaced with other packages such as **terra**, **sf**, or **sp**.

The majority of the computations done by **tigers** are performed by efficient C code which could be interfaced with other languages (e.g., Python).

The complete list of functions can be displayed with `library(help = tigers)`.

Author(s)

Emmanuel Paradis

Maintainer: Emmanuel Paradis <Emmanuel.Paradis@ird.fr>

`Anduki` *Anduki Forest Reserve*

Description

A set of coordinates delimiting the Anduki Forest Reserve in Brunei.

Usage

```
data(Anduki)
```

Format

A two-column matrix giving the coordinates in degrees (longitude and latitude, respectively).

Source

WDPA: World Database of Protected Areas. April 2024.

`area` *Area of Polygon*

Description

This function computes the area of a polygon with Euclidean coordinates (e.g., UTM).

Usage

```
area(x, y = NULL)
```

Arguments

`x, y`	the coordinates of the points given in the usual way in R.

Details

The unit of the area are the squared unit of input coordinates by default.

Value

a single numeric value giving the area of the polygon.

Author(s)

Emmanuel Paradis

See Also

geod

Examples

```
XY <- rbind(c(0, 0),
            c(1, 0),
            c(.25, .25),
            c(.5, .5),
            c(1.2, .8),
            c(1, .78),
            c(0, 1))
area(XY)
```

`axisMap` *Draw axes on Maps*

Description

Draw checkerboard-style axes on a map with appropriate scales and annotations.

Usage

```
axisMap(latitude = FALSE, width = 0.05, len = 1,
        cols = c("black", "white"), ...)
```

Arguments

`latitude`	by default axes are drawn for the longitudes (above and below the map). Use `latitude` = `TRUE` to draw axes for the latitudes (left and right sides of the map).
`width`	the width of the bands.
`len`	the length (increment) of the bands.
`cols`	the alternate colours of the bands.
`...`	further arguments passed to `axis` (see examples).

Details

The axes are drawn as bands of width given by the argument of the same name and alternate colors according to `len`.

Author(s)

Emmanuel Paradis

Examples

```
n <- 100
plot(runif(n, -180, 180), runif(n, -90, 90), pch = 3)
axisMap(len = 10)
## 'las' is passed with '...'
axisMap(TRUE, len = 10, las = 0)
```

---

barycoords *Computes Barycentric Coordinates*

Description

The barycentric coordinates of a point inside a polygon are weighted coordinates of the vertices of this polygon. The algorithm implemented in this function works for any concave or convex polygon (Hormann and Floater, 2006).

Usage

```
barycoords(XY, point)
```

Arguments

`XY`	A two-column matrix giving the coordinates of a polygon.
`point`	a vector with two values giving the coordinates of a point.

Details

If the polygon is a triangle, the `trilinear2Cartesian` can be used instead.

The polygon must be open (see `is.open`), and can be either in clockwise or in counterclockwise order (see `is.clockwise`).

For the moment, the function is not vectorized with respect to `point`, so it must be called for each point separately (see examples). This is likely to change in the future.

Value

a numeric vector giving the barycentric coordinates of the point (second argument). The length of the returned vector is equal to the number of vertices in the polygon (first argument).

Author(s)

Emmanuel Paradis

References

Hormann, K. and Floater, M. S. (2006) Mean value coordinates for arbitrary planar polygons. *ACM Transactions on Graphics* **25**, 1424–1441. <doi:10.1145/1183287.1183295>

See Also

trilinear2Cartesian

Examples

```
## a square:
xy <- cbind(c(0, 1, 1, 0), c(0, 0, 1, 1))

## a small function to get the coordinates directly:
f <- function(Pxy) barycoords(xy, Pxy)
## the CMYK scale:
F <- col2rgb(c("cyan", "magenta", "yellow", "black"))

n <- 1e5L
## random points in the square
Pxys <- matrix(runif(2 * n), n, 2)
system.time(res <- t(apply(Pxys, 1, f))) # < 1 sec
colnames(res) <- as.character(1:4)

## all rows should (approximately) sum to one:
all.equal(rowSums(res), rep(1, n), tol = 1e-15)

## transform the barycentric coordinates into colours:
COLS <- t(F %*% t(res)) / 255
rgbCOLS <- apply(COLS, 1, function(x) do.call(rgb, as.list(x)))
## add transparency:
rgbCOLS <- paste0(rgbCOLS, "33")
## plot the results:
plot(0:1, 0:1, "n", asp = 1, ann = FALSE, axes = FALSE)
points(Pxys, pch = ".", col = rgbCOLS, cex = 20)
## the visual effect is nicer with n <- 1e6L above and cex = 7
## in the last command


## the example below follows the same logic than the previous one

## an 8-vertex polygon:
xy <- cbind(c(0, 0.5, 1, 3, 1, 0.5, 0, -2),
            c(0, -2, 0, 0.5, 1, 3, 1, 0.5))

## random points in the square and in the 4 triangles:
Pxys <- rbind(matrix(runif(2 * n), n, 2),
              rpit(n, xy[1:3, ]),
       rpit(n, xy[3:5, ]),
       rpit(n, xy[5:7, ]),
       rpit(n, xy[c(7:8, 1), ]))

system.time(res <- t(apply(Pxys, 1, f))) # < 5 sec

colnames(res) <- as.character(1:8)
F <- col2rgb(c("black", "red", "orange", "green",
               "yellow", "blue", "purple", "white"))
```

```
## F <- col2rgb(rainbow(8)) # alternative
COLS <- t(F %*% t(res)) / 255.001
rgbCOLS <- apply(COLS, 1, function(x) do.call(rgb, as.list(x)))
rgbCOLS <- paste0(rgbCOLS, "33") # add transparency
plot(xy, , "n", asp = 1, ann = FALSE, axes = FALSE)
points(Pxys, pch = ".", col = rgbCOLS, cex = 5)
```

`buffer` *Buffer Around Polygons*

Description

Define a buffer zone around a polygon so that the outer points of the buffer are at a given distance to the nearest point of the input polygon.

Usage

```
buffer(x, r, smoothing = 1, unit = "degree", quiet = FALSE)
```

Arguments

`x`	a two-column numeric matrix giving the coordinates of the vertices of the polygon.
`r`	a single numeric value giving the distance of the buffer zone.
`smoothing`	a numeric value coding for the smooting of the arcs of the buffer. Larger values result in less smoothing; set `smoothing = Inf` for no smoothing at all (see examples).
`unit`	a character string, one of `"degree"`, `"km"`, or `"m"`. This argument defines a coefficient multiplied by the previous one to give better results.
`quiet`	a logical value specifying whether to print the progress of the computations.

Details

This version assumes that the coordinates in `x` are Euclidean (see examples for conversion from longitude-latitude to UTM).

The code may not work very well on polygons with complicated shapes and when the buffer size (argument `r`) is too large. This is still in progress.

Value

a two-column numeric matrix.

Author(s)

Emmanuel Paradis

Examples

```
data(Anduki)
r0 <- 0.005
z <- buffer(Anduki, r0)
plot(z, , "n", asp = 1, xlab = "Longitude", ylab = "Latitude")
R <- seq(r0, r0/5, length.out = 5)
COLS <- R * 1000 + 1
for (i in seq_along(R))
    polygon(buffer(Anduki, R[i], quiet = TRUE), col = COLS[i])
polygon(Anduki, col = "white")
legend("topleft", , c(paste("r =", R), "Anduki Forest Reserve"),
       pt.bg = c(COLS, "white"), pt.cex = 2.5, pch = 22, bty = "n")
title("Buffers around the Anduki Forest Reserve")

## a more realistic application with the same data:
x <- lonlat2UTM(Anduki) # convert to UTM coordinates
z <- buffer(x, 100)
plot(z, , "n", asp = 1, xlab = "Easting", ylab = "Northing")
title("Buffer zone of 100 m (UTM coordinates)")
polygon(z, col = "lightgrey", lwd = 1/3)
polygon(x, col = "white", lwd = 1/3)


## a concave hexagon
x <- rbind(c(0, 0), c(0, 1), c(0.5, 0.75),
           c(1, 1), c(1, 0), c(.5, .5))
r0 <- 0.1
z <- buffer(x, r0)
plot(z, , "n", asp = 1)
R <- seq(r0, r0/10, length.out = 10)
COLS <- R * 100 + 1
for (i in seq_along(R))
    polygon(buffer(x, R[i], quiet = TRUE), col = COLS[i])
polygon(x, col = "white")

## Not run:
## more fancy:
plot(z, , "n", asp = 1)
R <- seq(r0, r0/10, length.out = 1000)
COLS <- rainbow(1000)
for (i in seq_along(R))
    polygon(buffer(x, R[i], quiet = TRUE), col = COLS[i], border = NA)
polygon(x, col = "white", border = NA)

## End(Not run)

## Effect of smoothing
layout(matrix(1:4, 2, 2, TRUE))
for (sm in c(1, 30, 50, Inf)) {
    z <- buffer(x, 0.5, sm, quiet = TRUE)
    plot(z, , "n", asp = 1, ann = FALSE, axes = FALSE, )
    polygon(z, col = "lightblue")
    title(paste("smoothing =", sm))
    polygon(x, col = "white")
}
layout(1)
```

---

chullPolygon *Convex Hull of Polygon*

Description

Finds the convex hull of a polygon.

Note that the function chull (see link below) finds the convex hull of a set of points and is about twice slower than the present one when applied to a polygon.

Usage

```
chullPolygon(x, y = NULL)
```

Arguments

x, y	the coordinates of the points given in the usual way in R.

Details

This internal implementation requires the polygon to be open and in clockwise order (a crash will happen otherwise). Clockwise order is checked and possibly handled before calling the C code.

Value

a vector of integers which give the indices of the vertices of the input polygon defining the convex hull.

Author(s)

Emmanuel Paradis

References

Graham, R. L. and Yao, F. F. (1983) Finding the convex hull of a simple polygon. *Journal of Algorithms*, **4**, 324–331. <doi:10.1016/0196-6774(83)90013-5>

See Also

chull

Examples

```
XY <- rbind(c(0, 0),
            c(1, 0),
            c(.25, .25),
            c(.5, .5),
            c(1.2, .8),
            c(1, .78),
            c(0, 1))
(i <- chullPolygon(XY))
plot(XY, type = "n", asp = 1)
polygon(XY, lwd = 5, border = "lightgrey")
text(XY, labels = 1:nrow(XY), cex = 2/1.5)
polygon(XY[i, ], border = "blue", lty = 2, lwd = 3)
```

`convexPolygonOverlap` *Overlap of Two Convex Polygons*

Description

Find the intersection of two convex polygons.

Usage

```
convexPolygonOverlap(A, B)
```

Arguments

`A, B`	two two-column matrices giving the coordinates of two polygons.

Details

The intersection of two overlapping convex polygons is a single convex polygon.

The two input polygons must be in clockwise order.

Value

a two-column numeric matrix giving the coordinates of the overlap between the two input polygons.

Author(s)

Emmanuel Paradis

See Also

`is.clockwise`, `polygonOverlap`

Examples

```
X <- matrix(rnorm(3800), ncol = 2)
A <- X[chull(X), ]
Y <- matrix(rnorm(3800), ncol = 2)
B <- Y[chull(Y), ]

plot(rbind(A, B), type = "n", asp = 1)
polygon(A)
COLS <- c("blue", "red")
text(A, labels = 1:nrow(A), font = 2, cex = 1.5, col = COLS[1])
polygon(B)
text(B, labels = 1:nrow(B), font = 2, cex = 1.5, col = COLS[2])
legend("topleft", , c("A", "B"), text.font = 2, text.col = COLS)
O <- convexPolygonOverlap(A, B)
polygon(O, border = NA, col = rgb(1, 1, 0, 0.5))
```

`distance_to_line` *Distance to Line*

Description

These functions calculate the shortest distances from a set of points to a line (in Euclidean coordinates) or an arc (in angular coordinates).

`dtl` and `dta` are aliases to `distance_to_line` and `distance_to_arc`, respectively.

Usage

```
distance_to_line(x, y = NULL, x0, y0, x1, y1,
                 alpha = NULL, beta = NULL)
dtl(x, y = NULL, x0, y0, x1, y1, alpha = NULL, beta = NULL)
distance_to_arc(x, y = NULL, x0, y0, x1, y1, prec = 0.001)
dta(x, y = NULL, x0, y0, x1, y1, prec = 0.001)
```

Arguments

`x, y`	the coordinates of the points given in the usual way in R.
`x0, y0, x1, y1`	the coordinates of two points defining the line similar to `segments`. These are ignored if `alpha` and `beta` are given.
`alpha, beta`	alternatively to the previous arguments, the parameters of the line (`beta` is the slope).
`prec`	the precision of the estimated distances (see details).

Details

`distance_to_line` uses Euclidean geometry (see references). The coordinates can be in any units.

`distance_to_arc` uses distances along arcs on the (Earth) sphere. The coordinates must be in decimal degrees. The calculations are done by iterations using intervals of decreasing lengths along the arc. The iterations are stopped when the required precision is reached (see argument `prec`).

Value

a numeric vector giving the distances; `distance_to_line` returns them in the same unit than the input data; `distance_to_arc` returns them in kilometres (km).

Author(s)

Emmanuel Paradis

References

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

See Also

`great_circle_line`, `geoTrans`, `geod`

Examples

```
## distance from the topleft corner of the unity square to the diagonal:
(d <- dtl(matrix(c(1, 0), , 2), NULL, 0, 0, 1, 1))
all.equal(d, sqrt(2)/2)

## see also ?great_circle_line
x <- y <- 0:10/10
dta(x, y, 0, 0, 1, 1)
```

`fast2waytable` *Fast Two-Way Contingency Tables*

Description

Contingency tables of two vectors in a much faster way than the default `table`.

Usage

```
fast2waytable(x, y, levels = NULL)
```

Arguments

`x, y`	two vectors of the same length with integers (or values that can be coerced to).
`levels`	the unique values found in both vectors.

Details

If `levels` is known (and given), the running times are considerably shorter (up to several ten times; see examples).

`NA`'s are not handled in this version.

Value

a square matrix with the counts.

Author(s)

Emmanuel Paradis

See Also

`table`, `tabulate`

Examples

```
z <- 11:20 * 10L
n <- 13e6
x <- sample(z, n, TRUE)
y <- sample(z, n, TRUE)
system.time(res1 <- fast2waytable(x, y)) # ~ 0.4 sec
system.time(res2 <- fast2waytable(x, y, z)) # ~ 0.02 sec
system.time(res3 <- table(x, y)) # ~ 1.8 sec
all(res1 == res2)
all(res1 == res3)
```

`geod` *Geodesic Distances*

Description

This function calculates geodesic (or great-circle) distances between pairs of points with their longitudes and latitudes given in (decimal) degrees.

Usage

```
geod(lon, lat = NULL, R = 6371)
```

Arguments

`lon`	either a vector of numeric values with the longitudes in degrees, or, if `lat = NULL`, a matrix giving the longitudes (first column) and the latitudes (second column).
`lat`	a vector with the latitudes.
`R`	the mean radius of the Earth (see details).

Details

The default value of `R` is the mean radius of the Earth which is slightly smaller than the radius at the equator (6378.1 km).

Value

a numeric symmetric matrix with the distances between pairs of points in kilometres.

Author(s)

Emmanuel Paradis

References

https://en.wikipedia.org/wiki/Great-circle_distance

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

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

See Also

geoTrans, as.dist

Examples

```
## the distance between 0N 0E and 0N 180E...
geod(c(0, 180), c(0, 0)) # ~ 20015.09 km
## ... the same using the radius of the Earth at the equator:
geod(c(0, 180), c(0, 0), 6378.1) # ~ 20037.39 km
## The same comparison for two points 5 degrees apart:
geod(c(0, 5), c(0, 0)) # ~ 555.9746 km
geod(c(0, 5), c(0, 0), 6378.1) # ~ 556.5942 km
```

---

geoTrans *Manipulate Geographical Coordinates*

Description

geoTrans transforms geographical coordinates in degrees, minutes and seconds input as characters (or a factor) into numerical values in degrees. geoTrans2 does the reverse operation.

Usage

```
geoTrans(x, degsym = NULL, minsym = "'", secsym = "\"")
geoTrans2(lon, lat = NULL, degsym = NULL, minsym = "'",
          secsym = "\"", dropzero = FALSE, digits = 3,
          latex = FALSE)
```

Arguments

`x`	a vector of character strings storing geographical coordinates; this can be a factor with the levels correctly set.
`degsym`, `minsym`, `secsym`	a single character giving the symbol used for degrees, minutes and seconds, respectively.
`lon`	either a vector of numeric values with the longitudes in degrees, or, if `lat = NULL`, a matrix (or a data frame) giving the longitudes in the first column and the latitudes in the second column.
`lat`	a vector with the latitudes.
`dropzero`	a logical value: if `TRUE`, the number of arc-seconds is dropped if it is zero; similarly for the number of arc-minutes if the number of arc-seconds is also zero.
`digits`	an integer used for rounding the number of arc-seconds.
`latex`	a logical value: if `TRUE`, the returned character is formatted with LaTeX code.

Details

`geoTrans` should be robust to any pattern of spacing around the values and the symbols (see examples). If the letter S, W, or O is found is the coordinate, the returned value is negative. Note that longitude and latitude should not be mixed in the same character strings.

`geoTrans2` can be used with `cat` (see examples).

The default for `degsym` (`NULL`) is because the degree symbol (°) is coded differently in different character encodings. By default, the function will use the appropriate character depending on the system and encoding used.

Value

`geoTrans` returns a numeric vector with the coordinates in degrees (eventually as decimal values). `geoTrans2` returns a character vector.

Author(s)

Emmanuel Paradis

See Also

`geod`

Examples

```
coord <- c("N 43°27'30\"", "N43°27'30\"", "43°27'30\"N",
           "43° 27' 30\" N", "43 ° 27 ' 30 \" N",
           "43°27'30\"", "43°27.5'")
cat(coord, sep = "\n")
geoTrans(coord)
geoTrans("43 D 27.5'", degsym = "D")
geoTrans("43° 27' 30\" S")

XL <- c(100.6417, 102.9500)
YL <- c(11.55833, 14.51667)
cat(geoTrans2(XL, YL, dropzero = TRUE), sep = "\n")
cat(geoTrans2(XL, YL, latex = TRUE), sep = "\\\n")
```

---

great_circle_line *Great Circle Line*

Description

This function calculates the coordinates of the line on the surface of a sphere between two points. All coordinates are in decimal degrees.

`gcl` is simply an alias.

Usage

```
great_circle_line(x0, y0, x1, y1, linear = FALSE, npoints = 100)
gcl(x0, y0, x1, y1, linear = FALSE, npoints = 100)
```

Arguments

`x0, y0, x1, y1`	the coordinates of the two points similar to `segments`.
`linear`	a logical value.
`npoints`	an integer giving the number of points where the coordinates are calculated (should be at least two).

Details

The interval between `x0` and `x1` is split into regular segments, then the latitudes are computed, by default, using a great circle formula (Chamberlain and Duquette, 2007).

If `linear = TRUE`, the coordinates are treated as linear (i.e., Euclidean).

Value

a numeric matrix with two columns and colnames 'x' and 'y'.

Author(s)

Emmanuel Paradis

References

Chamberlain, R. G. and Duquette, W. H. (2007) Some algorithms for polygons on a sphere. JPL Open Repository. <doi:2014/41271>

See Also

geod

Examples

```
X1 <- 3; Y1 <- 49 # Paris
X2 <- 101; Y2 <- 13 # Bangkok
## if (require(maps))  map() else
plot(c(-180, 180), c(-90, 90), "n")
text(X1, Y1, "Paris")
text(X2, Y2, "Bangkok")
lines(gcl(X1, Y1, X2, Y2), col = "blue", lwd = 2)
lines(gcl(X1, Y1, X2, Y2, linear = TRUE), col = "red", lwd = 2)

## assess the error implied by using linear interpolation for the
## diagonal of a 1 degree by 1 degree square near the equator:
xya <- gcl(0, 0, 1, 1)
xyb <- gcl(0, 0, 1, 1, TRUE)
## the error in degrees:
error <- xya[, "y"] - xyb[, "y"]
plot(xya[, "x"], error * 3600, "o",
     xlab = "Longitude (degrees)", ylab = "Error (arc-seconds)")

## max (vertical) distance between these 2 curves:
geod(c(0.5, 0.5), c(0.5, 0.5 + max(error))) # ~6.5 m
## NOTE: the actual shortest (orthogonal) distance
## between these two curves is ~4.6 m
## (assuming the vertical distance helps to define a rectangular
## triangle, we have: 0.5 * sqrt(6.5^2 * 2)) ~ 4.6)

## NOTE2: dividing the coordinates by 10 results in dividing
## these deviations by 1000
```

HausdorffDistance *Hausdorff Distance*

Description

Computes the Hausdorff distance between two polygons. The distances can be directed (i.e., asymmetric) or not.

Usage

```
HausdorffDistance(A, B, directed = FALSE)
```

Arguments

A, B	two two-column matrices giving the coordinates of two polygons.
directed	a logical value. By default, the undirected distance is returned.

Details

If `directed = TRUE`, the order of the two polygons is important.

Value

a single numeric value.

Author(s)

Emmanuel Paradis

Examples

```
A <- cbind(c(0, 1, 1, 0), c(0, 0, 1, 1))
B <- A
B[, 1] <- B[, 1] + 2
B[c(1, 4), 1] <- 1.15

plot(rbind(A, B), type = "n", asp = 1)
COLS <- c("blue", "red")
polygon(A, border = COLS[1], lwd = 3)
polygon(B, border = COLS[2], lwd = 3)
text(mean(A[, 1]), mean(A[, 2]), "A", font = 2, col = COLS[1])
text(mean(B[, 1]), mean(B[, 2]), "B", font = 2, col = COLS[2])

(H <- HausdorffDistance(A, B))
(HAB <- HausdorffDistance(A, B, TRUE))
(HBA <- HausdorffDistance(B, A, TRUE))
arrows(0, 0.75, 1.15, 0.75, length = 0.1, code = 3)
text(0.5, 0.85, paste("H(A->B)", "=", HAB))
arrows(1, 0.15, 3, 0.15, length = 0.1, code = 3)
text(2, 0.25, paste("H(B->A)", "=", HBA))
text(1.5, -0.5, paste("H = max(H(A->B), H(B->A))", "=", H))
```

`haveOverlap` *Compare Two Polygons*

Description

These functions compare two polygons.

Usage

```
haveOverlap(A, B)
samePolygons(A, B, digits = 10)
```

Arguments

`A, B`	Two two-column matrices giving the coordinates of two polygons.
`digits`	the number of digits considered when comparing the coordinates.

Value

a single logical value

Author(s)

Emmanuel Paradis

See Also

`redundantVertices`

`is.insidePolygon` *Test If a Point Is Inside a Polygon*

Description

This function tests if a point is inside a polygon.

Usage

```
is.insidePolygon(XY, points)
```

Arguments

`XY`	A two-column matrix giving the coordinates of a polygon.
`points`	a vector with two values giving the coordinates of a point, or a matrix with two columns.

Details

The algorithm is based on "ray-tracing": a segment is traced between `points` and an arbitrary point far from the polygon. If this segment intersects an odd number of edges of the polygon, then `points` is inside the polygon.

The polygon must be open and can be in either clockwise or counterclockwise order. If the polygon is closed, it is modified internally without warning (the original polygon is not modified).

Value

a logical vector indicating whether each point is inside the polygon defined by `XY`.

Author(s)

Emmanuel Paradis

See Also

`is.open`

Examples

```
XY <- rbind(c(0, 0), c(0, 1), c(1, 1), c(1, 0))
stopifnot(is.insidePolygon(XY, c(0.5, 0.5)))
stopifnot(!is.insidePolygon(XY, c(1.5, 1.5)))
```

`lonlat2ECEF` *Conversions of Coordinates*

Description

Conversion between lon-lat and ECEF (Earth-centered, Earth-fixed) coordinates.

Usage

```
lonlat2ECEF(lon, lat = NULL, alt = 0, as.matrix = TRUE)
ECEF2lonlat(x, y = NULL, z = NULL)
```

Arguments

`lon, lat`	coordinates given as two vectors or as a matrix (or a data frame).
`alt`	a vector of altitudes.
`as.matrix`	a logical value specifying whether to return the results in a matrix. The default is to return them in a list.
`x, y, z`	three numeric vectors of the same length, or only `x` as a three-column matrix.

Details

This function uses an ellopsoid model of Earth shape with the following parameters: equatorial radius = 6,378,137 m, polar radius = 6,356,752 m.

Value

`lonlat2ECEF` returns a matrix with three columns or a list with three vectors; `ECEF2lonlat` returns a matrix with three columns.

Author(s)

Emmanuel Paradis

References

Blewitt, G. (2024) An improved equation of latitude and a global system of graticule distance coordinates. *Journal of Geodesy*, **98**, 6.

Examples

```
## From Blewitt's Table 1:
DF <- expand.grid(alt = c(1, 4, 10, 40) *1e3,
                  lat = seq(15, 75, 15),
                  lon = 0)
DF <- as.matrix(DF[3:1])

x <- lonlat2ECEF(DF[, -3], alt = DF[, 3], as.matrix = TRUE)

DFbis <- ECEF2lonlat(x)
xbis <- lonlat2ECEF(DFbis[, -3], alt = DFbis[, 3], as.matrix = TRUE)

summary(x - xbis)
summary(DF - DFbis)
```

`lonlat2UTM` *Conversions of Coordinates*

Description

Functions to convert coordinates between angular (longitude, latitude) and UTM systems.

Usage

```
lonlat2UTM(lon, lat = NULL, details = FALSE)
UTM2lonlat(x, y = NULL, zone = NULL, hemisphere = "N")
```

Arguments

`lon, lat`	coordinates given as two vectors or as a matrix (or a data frame).
`details`	a logical value indicating whether to return information on the UTM zones (see below).
`x, y`	either `x` is a data frame with four columns (as output from `lonlat2UTM`), or a matrix or a data frame giving the UTM coordinates in the standard way.
`zone, hemisphere`	an integer and a character string specifying the UTM square; can be single values (in which case they are recycled for all coordinates).

Details

`lonlat2UTM` works for all UTM zones. If the coordinates cover several UTM zones and/or hemispheres, the option `details` is switched to `TRUE`.

The formulas are in Karney (2011) originally from Kr\"uger (1912). The error is less than one micrometre. Karney (2011) presents algorithms that require arbitrary precision real numbers for errors of a few nanometres.

Value

a matrix or a data frame.

Author(s)

Emmanuel Paradis

References

Karney, C. F. F. (2011) Transverse Mercator with an accuracy of a few nanometers. *Journal of Geodesy*, **85**, 475–485.

Kr\"uger, J. H. L. (1912) Konforme Abbildung des Erdellipsoids in der Ebene. New Series 52. Royal Prussian Geodetic Institute, Potsdam.

`polygon2mask` *Convert Polygon to a Raster Mask*

Description

Takes a polygon and returns a matrix with a mask that can be input into a raster.

Usage

```
polygon2mask(XY, extent = NULL, k = 360,
             value = 1L, backgrd = 0L)
```

Arguments

`XY`	A two-column matrix giving the coordinates of a polygon.
`extent`	a vector with four numeric values giving the extent of the raster. By default, values are determined to minimally cover the polygon.
`k`	an integer value giving the number of pixels per unit (i.e., the inverse of the resolution of the raster). The resolution is the same in both directions.
`value`	the value given to the pixels inside the polygon (converted to integer).
`backgrd`	idem for the pixels outside the polygon.

Details

The mask is returned as a matrix which is filled rowwise (in agreement with the convention used in rasters) and can be input into functions in **terra** (e.g., `rast()`).

`polygon2mask` does basically the same operation than `terra::rasterize()` but is faster and can produce a vector for masking raster data.

The output matrix is row-filled, unlike matrices in R which are column-filled. It should be transposed before passed to `terra::rast()`, or its `dim` attribute can be ignored if used as a mask to a raster (which is also usually row-filled).

Value

a matrix stored as integers; the dimensions of this matrix give the size of the raster.

Note

The code is still in development. The version in **tigers** 0.1-3.3 has larger C buffers which should make the overall code more stable.

Author(s)

Emmanuel Paradis

References

Nievergelt, J. and Preparata, F. P. (1982) Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, **25**, 739–747. <doi:10.1145/358656.358681>.

Examples

```
## from ?chullPolygon:
XY <- rbind(c(0, 0),
            c(1, 0),
            c(.25, .25),
            c(.5, .5),
            c(1.2, .8),
            c(1, .78),
            c(0, 1))

layout(matrix(1:9, 3, 3, TRUE))
k <- 2
for (i in 1:9) {
    msk <- polygon2mask(XY, k = k)
    d <- dim(msk)
    image(1:d[1], 1:d[2], msk)
    dm <- paste(d, collapse = "x")
    title(paste("k =", k, ", dim =", dm))
    k <- k * 2
}

layout(1)
```

`polygonOverlap` *Decomposition and Overlap of Polygons*

Description

`decomposePolygon` decomposes a polygon into convex subpolygons.

`polygonOverlap` finds the intersection of two polygons.

Usage

```
decomposePolygon(x, y = NULL, method = 1, quiet = FALSE)
polygonOverlap(A, B)
```

Arguments

`x, y`	the coordinates of the points given in the usual way in R.
`method`	the method used for triangulation (see `triangulate`).
`quiet`	if the polygon is convex, a warning message is issued unless this option is switched to `TRUE`.
`A, B`	two two-column matrices giving the coordinates of two polygons.

Details

Both functions require the polygons to be in counterclockwise order (which is checked and arranged internally if needed).

The method in `decomposePolygon` is from Hertel and Mehlhorn (1983).

The method in `polygonOverlap` is based on first decomposing the two polygons into convex subpolygons, then computing their intersections with `convexPolygonOverlap`. The results is a list of polygons. A different algorithm is sketched in Chamberlain and Duquette (2007).

Value

`decomposePolygon` returns a two-column matrix with integers where each row gives the indices of two vertices of the input polygon defining a diagonal; the set of these diagonals define convex subpolygons.

`polygonOverlap` returns a list of polygons each defined by a two-column numeric matrix giving the coordinates of the vertices.

Note

These two functions are still in development.

Author(s)

Emmanuel Paradis

References

Chamberlain, R. G. and Duquette, W. H. (2007) Some algorithms for polygons on a sphere. JPL Open Repository. <doi:2014/41271>

Hertel, S. and Mehlhorn, K. (1983) Fast triangulation of simple polygons. In: *Foundations of Computation Theory.* Ed. Karpinski, M. Springer, Berlin, pp. 207–218. <doi:10.1007/3-540-12689-9_105>

See Also

`convexPolygonOverlap`, `is.clockwise`

Examples

```
## same polygon than in ?triangulate
XY <- rbind(c(0, 0), c(1, 0), c(.25, .25), c(.5, .5),
            c(1.2, .8), c(1, .78), c(0, 1))
decomposePolygon(XY) # similar to the output of triangulate()
## "lift up" one vertex:
XYb <- XY
XYb[6, 2] <- 1.2
decomposePolygon(XYb) # one diagonal less

## A is concave, B is convex:
A <- rbind(c(0, 1.5), c(2, 1), c(0.5, 1.5), c(2, 2))
B <- rbind(c(1, 0), c(3, 0), c(3, 3), c(1, 3))
AB <- polygonOverlap(A, B)
plot(rbind(A, B), , "n", asp = 1)
polygon(A)
polygon(B)
lapply(AB, polygon, col = "gold")
```

`random_point_in_triangle` *Random Points in Triangle*

Description

Generates random points inside a triangle using Osada et al.'s (2002, Sect. 4.2) method.

Usage

```
random_point_in_triangle(n, X, rfun1 = runif, rfun2 = runif)
rpit(n, X, rfun1 = runif, rfun2 = runif)
```

Arguments

`n`	an integer giving the number of points to generate.
`X`	a numeric matrix with 3 rows and 2 columns giving the coordinates of the triangle.
`rfun1`	a function generating random values in [0,1]. By default, the values are generated under a uniform distribution.
`rfun2`	same as the previous argument (see details).

Details

By default, the points are uniformly distributed in the triangle. The `Beta` function offers an interesting alternative to generate points concentrated in a specific part of the triangle (see examples).

Value

A numeric matrix with n rows and two columns giving the coordinates of the points.

Author(s)

Emmanuel Paradis

References

Osada, R., Funkhouser, T., Chazelle, B., and Dobkin, D. (2002) Shape distributions. *ACM Transactions on Graphics*, **21**, 807–832. <doi:10.1145/571647.571648>

Examples

```
## a random triangle in [0,1]^2:
P <- matrix(runif(6), 3, 2)

## n points uniformly distributed in the triangle P:
n <- 10000
x <- rpit(n, P)

layout(matrix(1:2, 1))

plot(P, type = "n", asp = 1)
polygon(P, col = "yellow", border = NA)
points(x, pch = ".", col = "blue")

## using Beta distributions:
foo <- function(n) rbeta(n, 1, 10)
bar <- function(n) rbeta(n, 1, 1)
y <- rpit(n, P, foo, bar)

plot(P, type = "n", asp = 1)
polygon(P, col = "yellow", border = NA)
points(y, pch = ".", col = "blue")

layout(1)
```

`redundantVertices` *Redundant Vertices in a Polygon*

Description

Tests and optionally correct for redundant vertices in a polygon.

The other functions test some features of a polygon.

`revPolygon()` reverses the order of the vertices (i.e., swiching between clockwise and counter-clockwise orders).

Usage

```
redundantVertices(x, tol = 1e-8, check.only = FALSE, colinear = TRUE)
is.clockwise(x)
is.convex(x)
is.open(x)
revPolygon(x, copy = TRUE)
```

Arguments

`x`	a two-column matrix.
`tol`	the tolerance to consider two vertices identical.
`check.only`	a logical value.
`colinear`	a logical value.
`copy`	by default, a new polygon is created; if `FALSE`, the vertex order is reversed within the same object.

Details

If `check.only` is `TRUE`, the first function prints the diagnostics and nothing is returned. Otherwise, the possibly corrected matrix is returned.

Two types of redundant vertices are considered: those that are close to another, and those which are colinear. If the option `colinear = FALSE` is set, only the first type is considered. Colinearity is checked only in contiguous (triplet) vertices.

Value

`redundantVertices` returns a two-column numeric matrix, or nothing if `check.only = TRUE` (the diagnostics are printed in the console).

`is.clockwise`, `is.convex`, and `is.open` return a single logical value.

`revPolygon` returns by default a two-column numeric matrix, or nothing if `copy = FALSE` (the first argument is modified).

Author(s)

Emmanuel Paradis

References

The method for `is.clockwise` is from:

`https://en.wikipedia.org/wiki/Curve_orientation`

See Also

`haveOverlap`

RMA — *Reduced Major Axis*

Description

Computes the coefficients of the reduced major axis (RMA) of a set of points.

Usage

```
RMA(x, y = NULL)
```

Arguments

`x, y`	the coordinates of the points given in the usual way in R.

Details

The RMA is found by solving a polynomial equation of degree two, so there are actually two solutions which are both returned. It is usually straightforward to find the appropriate solution.

Value

a matrix with two rows and two columns named alpha and beta for the intercepts and slopes, respectively.

Author(s)

Emmanuel Paradis

References

https://mathworld.wolfram.com/LeastSquaresFittingPerpendicularOffsets.html

Examples

```
x <- 1:1000
y <- x + rnorm(1000, 5)
RMA(x, y) # same than RMA(cbind(x, y))
```

`rose` *Draw a Compass*

Description

Draw a compass on a map or a plot.

Usage

```
rose(x, y, size = 1, width = size/4, cols = c("grey10", "white"),
     labels = c("N", "S", "E", "W"), offset = 0, ...)
```

Arguments

`x, y`	coordinates of the center of the compass given as two vectors of length one (in user coordinates).
`size`	the length of the needles (in the same units than the user coordinates).
`width`	the width of the needles at the bottom; by default, one fourth of the previous.
`cols`	the colours for each side of the needles.
`labels`	the text printed at the tips of the needles.
`offset`	the space between the labels and the tips of the needles.
`...`	further arguments passed to `text` to format the labels.

Details

To not print the directions, set `labels = rep("", 4)`.

Author(s)

Emmanuel Paradis

`triangulate` *Triangulate a Polygon*

Description

Performs the decomposition of a polygon into triangles.

Usage

```
triangulate(x, y = NULL, method = 1)
```

Arguments

`x, y`	the coordinates of the points given in the usual way in R.
`method`	an integer between 1 and 4 specifying the triangulation method.

Details

The following methods are available:

- 1: the triangles are created in successive order from the first appropriate angle (i.e., an ear) encountered in the polygon.
- 2: the triangles are created to favour thin triangles.
- 3: the triangles are created to favour fat triangles.
- 4: the triangles are created to favour regular-looking triangles based on their determinant.

These methods have different requirements: method 1 needs the polygon to be closed, whereas the other methods need it to be open; method 2 needs the polygon to be in counterclockwise order, and method 3 needs it to be in clockwise order (the other methods are not sensitive to this order). These requirements are checked before performing the triangulation and the polygon is changed internally (without warning since the original polygon is not modified) if necessary.

Value

a three-column matrix giving the indices of the vertices in each triangle (i.e., each row a is a triangle).

Note

The internal codes need to be checked and tested again.

Author(s)

Emmanuel Paradis

References

Toussaint, G. (1991) Efficient triangulation of simple polygons. *Visual Computer*, **7**, 280–295. <doi:10.1007/BF01905693>

Examples

```
XY <- rbind(c(0, 0),
            c(1, 0),
            c(.25, .25),
            c(.5, .5),
            c(1.2, .8),
            c(1, .78),
            c(0, 1))
(tri <- triangulate(XY))
plot(XY, type = "n", asp = 1)
```

```
for (i in 1:nrow(tri))
    polygon(XY[tri[i, ], ], border = "white", col = "green", lwd = 2)
polygon(XY, lwd = 4, border = "lightgrey")
text(XY, labels = 1:nrow(XY), cex = 1.2)
```

trilinear2Cartesian *Trilinear Coordinates*

Description

trilinear2Cartesian calculates the coordinates of a point inside a triangle given three values interpreted as proportions.

Cartesian2trilinear does the reverse operation.

Usage

```
trilinear2Cartesian(p, X)
Cartesian2trilinear(xy, X)
```

Arguments

p	a vector with three numeric values (see details).
X	a numeric matrix with 3 rows and 2 columns giving the coordinates of the triangle.
xy	a vector with two numeric values (Cartesian coordinates).

Details

The values in p do not need to sum to one since they are scaled internally.

The triangle defined by X can be of any type. The coordinates returned by trilinear2Cartesian is always inside the triangle.

Cartesian2trilinear does not check if xy is inside the triangle.

Value

trilinear2Cartesian returns a numeric matrix with a single row and two columns giving the coordinates of the point.

Cartesian2trilinear returns a numeric matrix with a single row and three columns.

Author(s)

Emmanuel Paradis

References

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

Examples

```
## rectangular triangle (counterclockwise):
X <- rbind(c(0, 0), c(0, 1), c(1, 0))
plot(X, , "n", asp = 1)
polygon(X)

h <- sqrt(2) # hypothenuse length

points(trilinear2Cartesian(c(1, 1, 1), X)) # incenter
points(trilinear2Cartesian(c(1, h, h), X), pch = 2) # centroid
points(trilinear2Cartesian(c(h, 1, 1), X), pch = 3) # symmedian point
## the 3 midpoints:
points(trilinear2Cartesian(c(0, h, h), X), pch = 7)
points(trilinear2Cartesian(c(1, 0, h), X), pch = 7)
points(trilinear2Cartesian(c(1, h, 0), X), pch = 7)

legend("topright", ,
       c("incenter", "centroid", "symmedian point", "midpoints"),
       pch = c(1:3, 7))

f <- c(0.1, 0.3, 0.6)
o <- trilinear2Cartesian(f, X)
p <- Cartesian2trilinear(o, X)
p - f # < 1e-15
stopifnot(all.equal(as.vector(p), f))
```

`wl2col` *Wavelengths to Colours*

Description

Conversion from wavelengths to colours.

Usage

```
wl2col(x, gamma = 0.8, RGB = FALSE)
spectrum2col(spec, RGB = FALSE, no.warn = TRUE, color.system = 3)
BlackBodySpectrum(x, Temp = 300)
```

Arguments

`x`	a numeric vector with wavelengths in nanometers (nm).
`gamma`	parameter for correcting the transitions. If `gamma = 1`, then the transitions are linear (see examples).
`RGB`	a logical value. By default, colours (in HTML code) are returned. If `RGB = TRUE`, a matrix with three columns is returned.
`spec`	a numeric vector with 81 values giving the (relative) intensity of the different wave lengths between 380 nm and 780 nm (see examples).

`no.warn`	a logical value. If `TRUE` and some approximate calculations were performed in the C routine, a warning message is issued.
`color.system`	a single integer between 1 and 6 specifying the colour system (see details).
`Temp`	temperature in Kelvins (K) of the black body.

Details

Computations are mainly performed by C and Fortran codes (see References).

The argument `spec` gives the (relative) intensity of visible light between 380 nm and 780 nm in intervals with a bandwith of 5 nm (i.e., [380–385], [385–390], . . . , [775-780]). The returned value is the perceived colour of the given spectrum. It could happen that some calculations were approximate which is done silently unless `no.warn = FALSE`.

The six colour systems are: (1) NTSC, (2) EBU (PAL/SECAM), (3) SMPTE, (4) HDTV, (5) CIE, and (6) CIE REC 709.

`BlackBodySpectrum` calculates the emittance at specified wavelength(s) of a black body of temperature `Temp` using Planck's law.

Value

`wl2col` and `spectrum2col` return by default a vector of mode character with colours in HTML code. If `RGB = TRUE`, they return a matrix with the values (between 0 and 1) of red, green, and blue arranged in a three-column matrix. If the input `x` has names, these are used in the returned object (as names or rownames).

`BlackBodySpectrum` returns a numeric vector.

Author(s)

Emmanuel Paradis, John Walker, Dan Bruton

References

Bruton, D. (1996) Approximate RGB values for visible wavelengths. http://www.physics.sfasu.edu/astro/color/spectra.html

Planck, M. (1901) Ueber das Gesetz der Energieverteilung im Normalspectrum. *Annalen der Physik*, **309**, 553–563. (English translation: http://web.ihep.su/dbserv/compas/src/planck01/eng.pdf)

Walker, J. (1996) Color rendering of spectra. https://www.fourmilab.ch/documents/specrend/

Examples

```
wl <- 370:790
COLS <- c("red", "green", "blue")
if (interactive()) layout(matrix(1:3, 3))
matplot(wl, wl2col(wl, , TRUE), "l", col = COLS, lty = 1, lwd = 3)
title("gamma = 0.8 (default)")
matplot(wl, wl2col(wl, 1, TRUE), "l", col = COLS, lty = 1, lwd = 3)
title("gamma = 1")
matplot(wl, wl2col(wl, 1/3, TRUE), "l", col = COLS, lty = 1, lwd = 3)
```

```
title("gamma = 1/3")
layout(1)

spec <- numeric(81)
spec[2] <- 1
names(spec) <- seq(380, 780, 5)
sapply(1:6, function(i) spectrum2col(spec, TRUE, color.system = i))

WL <- 380:780
xlab <- "Wavelength (nm)"
ylab <- expression("Emittance (W."*m^{-2}*")")
plot(WL, BlackBodySpectrum(WL, 306), type = "l", xlab = xlab,
     ylab =ylab, log = "")
lines(WL, BlackBodySpectrum(WL, 303), lty = 2)
legend("topleft", legend = paste(c(306, 303) - 273, "degrees C"), lty = 1:2)


## vector of wavelengths:
wl <- seq(382.5, by = 5, length.out = 81)
spectrum2col(BlackBodySpectrum(wl, 310), TRUE)
spectrum2col(BlackBodySpectrum(wl, 3100), TRUE)
spectrum2col(BlackBodySpectrum(wl, 31000), TRUE)

wl <- 10:1e5
col <- wl2col(wl)
plot(wl, BlackBodySpectrum(wl, 6000), "n", log = "xy", ylim = c(1, 1e14),
     xaxs = "i", xlab = xlab, ylab = ylab)
s <- col != "#000000" # do not show the black lines
abline(v = wl[s], col = col[s])
lines(wl, BlackBodySpectrum(wl, 300))
lines(wl, BlackBodySpectrum(wl, 3000))
lines(wl, BlackBodySpectrum(wl, 6000))
text(c(3000, 240, 200), c(5e6, 6e10, 5e13), paste(c(300, 3000, 6000), "K"))
```

Index