Package 'voronoifortune'

January 8, 2025

Version 1.0

Date 2025-01-07

Title Voronoi Tessellation by Fortune Algorithm

Description Fortune's (1987, <doi:10.1007/BF01840357>) algorithm is a very efficient method to perform Voronoi tessellation and Delaunay triangulation. This package is a port of the original code published in the early 1990's by Steven Fortune.

License GPL-3

URL https://github.com/emmanuelparadis/voronoifortune

BugReports https://github.com/emmanuelparadis/voronoifortune/issues

NeedsCompilation yes

Author Emmanuel Paradis [aut, cre, cph]
(<https://orcid.org/0000-0003-3092-2199>),
Steven Fortune [aut, ctb],
Plan 9 Foundation [cph]

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

Repository CRAN

Date/Publication 2025-01-08 15:00:01 UTC

Contents

voronoifortune-package *Fortune Algorithm for Voronoi Diagrams*

Description

voronoifortune is a port to R of Fortune's algorithm, a very fast method to compute simultaneously the Delaunay triangulaton and the Voronoi tessellation for a set of sites (points). The algorithm scales linearly with the number of points whereas most other algorithms scale polynomially.

A Voronoi tessellation defines for each site a cell (or region) so that all points included in the cell are not nearer to any other sites (distances are meant to be Euclidean distances).

A Delaunay triangulaton defines a set of triangles with vertices at the sites so that the circle circumscribed to a given triangle does not include any other site; see an annotated example in voronoi including a test of this property.

See the GitHub repository of the present package (link in the DESCRIPTION file) for some notes on porting this code to R.

Author(s)

Emmanuel Paradis, Steven Fortune

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

References

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

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

plot.voronoi *Plot Voronoi Tessellation and Delaunay Triangulation*

Description

Plot Voronoi tessellation (VT) and Delaunay triangulation (DT).

Usage

```
## S3 method for class 'voronoi'
plot(x, delaunay = TRUE, voronoi = TRUE,
     X = NULL, add = FALSE, asp = 1,
     col.delaunay = "red", col.voronoi = "blue", ...)
```

Arguments

x	an object of class "voronoi".
delaunay	a logical value: draw the DT?
voronoi	a logical value: draw the VT?
X	a two-column matrix with the original data (site coordinates).
add	a logical value. By default, a new plot is made. Setting add = TRUE is useful to add the tessellation and/or the triangulation on, e.g., a map.
asp	a numeric value. By default, both axes are scaled similarly (isometry). Use asp = NULL to have independent scaling on both axes (like a standard plot in R).
col.delaunay, col.voronoi	the colours used for the segments.
...	other arguments passed to plot.default.

Details

The ... argument makes plotting very flexible.

The default procedure is to first draw the DT, setting the limits of the axes according to the data, and then to draw the VT. Playing with the different options can change the order these two are drawn.

The Fortune algorithm often adds some vertices which are far from the data points (sites); so if delaunay = FALSE, the scales are likely to extend much more than the default.

The infinite edges of the VT are drawn as dashed lines.

Value

(NULL).

Author(s)

Emmanuel Paradis

See Also

voronoi

Examples

```
X <- matrix(runif(200), 100, 2)
res <- voronoi(X)
plot(res)
plot(res, delaunay = FALSE)

dat <- matrix(runif(40), 20, 2)
tess.dat <- voronoi(dat)
op <- par(mar = rep(0, 4))
## pass the data with the X argument:
plot(tess.dat, X = dat, pch = ".", axes = FALSE, ann = FALSE,
     xlim = c(-1, 2), ylim = c(-1, 2))
```

```
legend("topleft", , c("Delaunay triangulation", "Voronoi tessellation"),
       lty = 1, col = c("red", "blue"), bty = "n")
par(op)
```

---

voronoi *Voronoi Tessellation and Delaunay Triangulation*

Description

Voronoi tessellation and Delaunay triangulation are performed simultaneously with the Fortune (1987) algorithm.

Usage

```
voronoi(X, sorted = FALSE, debug = FALSE)
## S3 method for class 'voronoi'
print(x, ...)
```

Arguments

X	a two-column matrix with the coordinates.
sorted	a logical: are the coordinates already sorted in increasing order first by y, then by x? (See order.) If TRUE, this shortens the running time.
debug	a logical: if TRUE, some details of the computation are printed for debugging.
x	an object of class "voronoi".
...	(unused).

Value

voronoi() returns an object of class "voronoi" with four elements:

Triplets	a three-column matrix of integers giving the triangles of the Delaunay triangulation (indices from the original data);
Vertices	a two-column matrix of reals giving the coordinates of the vertices of the Voronoi tessellation;
Edges	a two-column matrix of integers giving the edges of the tessellation (row indices of the vertices in the previous matrix);
Lines	a description of the previous edges (some of them are semi-infinite indicated by a 0 in the matrix Edges; this is used to draw the dashed lines by plot.voronoi).

Author(s)

Emmanuel Paradis, Steven Fortune

References

Fortune, S. (1987) A sweepline algorithm for Voronoi diagrams. *Algorithmica*, **2**, 153–174. doi:10.1007/BF01840357.

See Also

plot.voronoi

Examples

```
n <- 100L
xx <- runif(n)
yy <- runif(n)
X <- cbind(xx, yy)
(res <- voronoi(X))
str(res)

### show the circumcircle of each Delaunay triangle ###

n <- 10L
X <- cbind(runif(n), runif(n))
res <- voronoi(X)

## if 12 windows not enough par(ask) is set to TRUE
op <- par(mfrow = c(3, 4), ask = interactive())
## to show the circle as disk:
col <- rgb(.5, .5, 1, .3)

## for each triangle:
for (i in 1:nrow(res$Triplets)) {
    plot(res, X = X, voronoi = FALSE, main = i)
    ## get the 3 vertices of the triangle:
    P <- res$Triplets[i, ]
    ## center the coordinates on the 1st vertex:
    ## (B and C are the new coordinates of the 2nd
    ## and 3rd vertices)
    A <- X[P[1], ]
    B <- X[P[2], ] - A
    C <- X[P[3], ] - A
    ## the coordinates of the center of the circumcircle are:
    ## (https://en.wikipedia.org/wiki/Circumcircle)
    cr <- c(C[2] * sum(B^2) - B[2] * sum(C^2),
            B[1] * sum(C^2) - C[1] * sum(B^2))
    cr <- cr / (2 * (B[1] * C[2] - B[2] * C[1]))
    ## since the circle intersects with the new origin,
    ## the radius is simply:
    r <- sqrt(sum(cr^2))
    ## translate back to the original coordinate system:
    cr <- cr + A
    ## draw the circumcircle:
    symbols(cr[1], cr[2], circles = r, inches = FALSE, add = TRUE,
            fg = col, bg = col)
```

```
    ## test numerically that no points are inside the circumcircle:
    ## 1/ build a matrix with the coordinates of the center
    ## and all the vertices:
    M <- rbind(cr, X)
    ## 2/ compute the Euclidean distances, we keep only the n first
    ## values which are the distances from the center to the n vertices:
    Dis <- dist(M)[1:n]
    ## 3/ all other points than the 3 in P should be farther
    ## to the center:
    stopifnot(all(Dis[-P] > r))
}

par(op)
```

Index