Package 'cdtr'

November 18, 2026

Title Constrained Delaunay Triangulation with Refinement

Version 0.2.0

Description Binding to the 'CDT' header-only C++ library: constrained and conforming Delaunay triangulation, Ruppert refinement by area or minimum angle (Ruppert (1995) <doi:10.1006/jagm.1995.1021>), resolution of crossing constraints, and topological region depth per triangle. Also returns the provenance of each vertex and the input segments behind each constraint edge, so that the coverage of overlapping polygons and lines can be found from the mesh. The input and output follow the planar straight line graph form used by 'RTriangle'.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

LinkingTo cpp11

Suggests bench, knitr, rmarkdown, RTriangle, sf, testthat (>= 3.0.0), wk, wkpool (>= 0.5.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

URL https://github.com/hypertidy/cdtr, https://hypertidy.github.io/cdtr/

BugReports https://github.com/hypertidy/cdtr/issues

NeedsCompilation yes

Author Michael Sumner [aut, cre],
Leica Geosystems Technology AB [cph] (Copyright holder of the bundled 'CDT' C++ library),
CDT Contributors [ctb, cph] ('CDT' C++ library, https://github.com/artem-ogre/CDT),
William C. Lenthe [ctb, cph] (Robust geometric predicates (predicates.h))

Maintainer Michael Sumner <mdsumner@gmail.com>

Repository CRAN

Date/Publication 2026-11-18 10:00:15 UTC

Contents

cdt_edges 2
cdt_interpolate 3
cdt_locate 3
cdt_pslg 4
cdt_quality 5
cdt_triangulate 6
cdt_triangulate_attr 9
cdt_winding 10
default_min_edge_length 11

Index **13**

`cdt_edges` *Unique edges of a triangulation*

Description

The index pairs of every triangle edge once, for plotting with `graphics::segments()` or for building an edge table.

Usage

```
cdt_edges(x)
```

Arguments

`x`	result of `cdt_triangulate()`

Value

integer matrix (2 columns) of 1-based vertex indices

Examples

```
r <- cdt_triangulate(c(0, 1, 1, 0, 0.5), c(0, 0, 1, 1, 0.5))
e <- cdt_edges(r)
plot(r$P, asp = 1)
segments(r$P[e[, 1], 1], r$P[e[, 1], 2], r$P[e[, 2], 1], r$P[e[, 2], 2])
```

cdt_interpolate *Interpolate per-vertex attributes onto points*

Description

Linear (barycentric) interpolation of a matrix of per-vertex values from a triangle mesh onto query points. Points in no triangle take the nearest vertex's value. Used to carry attributes such as z onto Steiner vertices after refinement, matching the `PA` behaviour of RTriangle.

Usage

```
cdt_interpolate(P, T, A, xq, yq)
```

Arguments

`P`	vertex coordinates (n x 2)
`T`	triangle vertex indices (m x 3, 1-based)
`A`	per-vertex attribute matrix (n x k) or vector
`xq, yq`	query coordinates

Value

a length(xq) x k matrix (column names kept from `A`)

Examples

```
## two triangles on the unit square, z = x + y at the corners
P <- cbind(c(0, 1, 1, 0), c(0, 0, 1, 1))
T <- rbind(c(1, 2, 3), c(1, 3, 4))
A <- cbind(z = P[, 1] + P[, 2])
## a linear field is reproduced exactly inside; (2, 2) is outside
cdt_interpolate(P, T, A, xq = c(0.5, 0.25, 2), yq = c(0.5, 0.1, 2))
```

cdt_locate *Locate points in a triangle mesh*

Description

Find the triangle that contains each query point. This is the point locator inside `cdt_interpolate()` (a uniform grid of triangle buckets), with exact orientation tests from the bundled robust predicates, so a point is never lost to rounding between two triangles. With a triangle to feature table it answers point-in-polygon for many points at once.

Usage

```
cdt_locate(P, T, xq, yq, barycentric = FALSE)
```

Arguments

P	vertex coordinates (n x 2)
T	triangle vertex indices (m x 3, 1-based)
xq, yq	query coordinates
barycentric	if TRUE, return a data frame with the barycentric coordinates, rather than only the triangle index

Details

A point on an edge or vertex shared by several triangles is given to the lowest triangle index that contains it. Points outside every triangle, and NA points, give NA. Triangles may be in either orientation, and degenerate (zero area) triangles contain no point.

With `barycentric = TRUE` the barycentric coordinates of each point in its triangle are returned as well. They are the weights of the triangle's three vertices (in the order given in T) that reproduce the point, so `b1 * P[T[i, 1], ] + b2 * P[T[i, 2], ] + b3 * P[T[i, 3], ]` is the point, and any per-vertex value is interpolated by the same weights, as `cdt_interpolate()` does. The weights are not negative and sum to 1; they are computed in double precision, with rounding error clamped, so a point on an edge has a weight of exactly 0 for the opposite vertex.

Value

by default an integer vector, `length(xq)`, of 1-based row indices into T (NA outside the mesh). With `barycentric = TRUE`, a data frame with one row per query point and columns `triangle` (the same index), and b1, b2, b3, which are NA where `triangle` is.

Examples

```
P <- cbind(c(0, 1, 1, 0), c(0, 0, 1, 1))
T <- rbind(c(1, 2, 3), c(1, 3, 4))
## (0.5, 0.5) is on the shared diagonal: the lower triangle id wins
cdt_locate(P, T, xq = c(0.75, 0.25, 0.5, 2), yq = c(0.25, 0.75, 0.5, 2))

## the weights of each triangle's vertices
loc <- cdt_locate(P, T, xq = c(0.75, 0.25, 0.5), yq = c(0.25, 0.75, 0.5),
                  barycentric = TRUE)
loc
## a value known at the vertices, interpolated with those weights
z <- c(0, 10, 20, 10)
rowSums(as.matrix(loc[c("b1", "b2", "b3")]) * matrix(z[T[loc$triangle, ]], ncol = 3))
```

---

cdt_pslg *Triangulate an RTriangle pslg with the CDT backend*

Description

Convenience for input in the shape of `RTriangle::pslg()`: a list with P, a two-column matrix of vertex coordinates, and S, a two-column matrix of 1-based segment vertex indices.

Usage

```
cdt_pslg(p, ...)
```

Arguments

p	a pslg (list with P and S, and optionally H, a two-column matrix of hole points, passed as `holes` unless `holes` is given in `...`)
...	passed to `cdt_triangulate()`

Value

the list returned by `cdt_triangulate()`

Examples

```
p <- list(P = cbind(c(0, 2, 2, 1, 0), c(0, 0, 2, 1, 2)),
          S = cbind(1:5, c(2:5, 1)))
r <- cdt_pslg(p, max_area = 0.05)
nrow(r$T)
plot(r$P, asp = 1, pch = ".")
polygon(r$P[t(cbind(r$T, NA)), ])
```

---

cdt_quality *Triangle quality measures*

Description

Per-triangle area, angles, edge lengths and radius-edge ratio, for judging a mesh or checking that refinement met its targets.

Usage

```
cdt_quality(x, T = NULL)
```

Arguments

x	a result of `cdt_triangulate()` (a list with `P` and `T`), or a two-column matrix of vertex coordinates, in which case `T` is needed
T	triangle vertex indices (m x 3, 1-based), when `x` is a matrix

Value

a data frame with one row per triangle and columns `area` (always non-negative), `min_angle` and `max_angle` (degrees), `min_edge` and `max_edge`, and `radius_edge` (circumradius over shortest edge; the Ruppert bound on it is about 1.41 for a 20.7 degree minimum angle, and `Inf` for a degenerate triangle). Use `summary()` on the result for the distributions.

Examples

```
## an irregular polygon with a sharp corner, refined by area and angle
x <- c(0, 2, 2, 1, 0)
y <- c(0, 0, 2, 0.4, 2)
r <- cdt_triangulate(x, y, 1:5, c(2:5, 1), max_area = 0.05, min_angle = 25)
q <- cdt_quality(r)
summary(q[c("area", "min_angle", "radius_edge")])
## both refinement targets were met
c(max_area = max(q$area), min_angle = min(q$min_angle))
```

`cdt_triangulate` *Constrained Delaunay triangulation with refinement (CDT backend)*

Description

Thin binding to the artem-ogre/CDT library. Input mirrors RTriangle::pslg: vertices as x/y and constraint segments as index pairs into the vertices.

Usage

```
cdt_triangulate(
  x,
  y,
  s0 = NULL,
  s1 = NULL,
  max_area = NULL,
  min_angle = NULL,
  max_steiner = Inf,
  min_edge_length = NULL,
  min_edge_frac = 0.3,
  area_edge_frac = 0.25,
  conforming = FALSE,
  erase = c("outer", "hull", "holes"),
  intersect = c("resolve", "error", "ignore"),
  angle_first = FALSE,
  depth_distinct = FALSE,
  edge_sources = FALSE,
  holes = NULL
)
```

Arguments

x, y	vertex coordinates
s0, s1	1-based vertex indices of constraint segment start/end (may be NULL)
max_area	maximum triangle area (Ruppert refinement by area), NULL for none
min_angle	minimum triangle angle in degrees, NULL for none

`max_steiner` budget of Steiner points to insert (Inf for unlimited)

`min_edge_length`
: refinement does not split edges or triangles already shorter than this. NULL (the default) chooses a value from the input and the targets (see `default_min_edge_length()`): a fraction of the median constraint segment length, capped so that it never blocks the `max_area` target. This stops the refiner cascading into sharp input corners. Use 0 to never give up. A larger value gives a coarser mesh, since it also stops the splitting of encroached constraint segments (see Details).

`min_edge_frac` fraction of the median constraint segment length used when `min_edge_length` is NULL

`area_edge_frac` fraction of `sqrt(max_area)` that caps the default floor

`conforming` if TRUE use conforming (Steiner on segments) instead of constrained

`erase` one of "outer" (drop triangles outside the segment-bounded region, as RTriangle's `-p` does), "hull" (keep the convex hull), "holes" (also drop even-depth regions). With no segments there is no boundary, so "hull" is used. Note that `"outer"` keeps the triangles inside holes (they have even `depth`), so use `"holes"` when the triangles are to represent polygon area, for example to build a triangle pool or sum areas.

`intersect` how crossing constraints are handled: "resolve" inserts the crossing point, "error" fails, "ignore" skips the check

`angle_first` when both criteria are given, refine by angle before area (default is area then angle; the two orderings differ little)

`depth_distinct` also return `depth_distinct`, a per-triangle depth that counts a doubled (overlapping) boundary once rather than once per copy. The default `depth` counts per copy; the two differ only where the only path from outside a triangle crosses a shared, doubled boundary.

`edge_sources` also return `edge_sources`, a data frame with one row per fixed edge and input segment covering it (see Value)

`holes` optional hole points, a two-column matrix or a list or data frame with `x` and `y`. The region around each point that is bounded by constraint segments is erased, as with the `H` of an `RTriangle::pslg()`. This is in addition to `erase`. A point in no bounded region (outside every ring, or with no segments) erases nothing, and a point exactly on a segment is not supported. Holes are applied after the topological `depth` is computed, so `depth` still describes the unerased mesh.

Details

Refinement follows Ruppert's algorithm, which protects the constraint segments: a segment is split whenever a vertex lies inside the circle on its diameter (it is "encroached"), and this applies to refinement by `max_area` alone as well as by `min_angle`. A mesh refined by `max_area` therefore has more vertices along the constraints than one from a refiner that only splits large triangles, such as Triangle with `a` but no `q`. On North Carolina's counties and anglr's Tasmanian cadastre and contours, cdtr adds 1.2 to 1.3 times as many vertices as Triangle with an area bound alone, with 40 to 62% of them on the constraints against 11 to 14%, and on the two polygon layers fewer triangles with a minimum angle under 5 degrees (see `inst/benchmarks/bench5.R`).

`min_edge_length` is the control for this: edges shorter than it are not split. Raising it gives fewer vertices: on those inputs a floor of `sqrt(max_area)` gives 10 to 12% fewer than the default. The `unrefined` component of the result counts what was left undone, so check it and the largest area after raising the floor.

Value

list with P (vertices), T (1-based triangle indices, wound counter-clockwise / positive signed area), S (fixed edges), depth (per-triangle constraint layer depth: 0 outside, odd inside, even hole; a doubled shared boundary is counted once per copy), origin (a factor giving each vertex's provenance: "input", "crossing" or "steiner"), n_input (vertices after dedupe, before Steiner), n_crossing (vertices added resolving crossing constraints), input_map (input vertex -> deduped vertex index), min_edge_length (the value used), and unrefined, a data frame with one row per refinement pass counting what CDT could not refine (short edges, sharp fixed corners, and so on). When `depth_distinct = TRUE` the list also has depth_distinct (a doubled boundary counted once). When `edge_sources = TRUE` the list also has edge_sources, a data frame with columns edge (row in S), `segment` (position of the input segment in `s0`, `s1`) and `forward` (TRUE when the input segment runs the same way as the S row, from its first to its second vertex). Every input segment covering an edge has its own row, through crossings, splits and refinement, so an edge shared by k input segments (duplicates included) has k rows. Zero-length input segments have no rows.

Examples

```
## unit square with a square hole: two closed rings of 4 vertices
x <- c(0, 1, 1, 0, 0.25, 0.75, 0.75, 0.25)
y <- c(0, 0, 1, 1, 0.25, 0.25, 0.75, 0.75)
s0 <- c(1, 2, 3, 4, 5, 6, 7, 8)
s1 <- c(2, 3, 4, 1, 6, 7, 8, 5)

## constrained triangulation, hole triangles kept with even depth
r <- cdt_triangulate(x, y, s0, s1)
table(r$depth)

## drop the hole and refine by area and minimum angle
r2 <- cdt_triangulate(x, y, s0, s1, erase = "holes",
                      max_area = 0.005, min_angle = 25)
table(r2$origin)
plot(r2$P, asp = 1, pch = ".")
polygon(r2$P[t(cbind(r2$T, NA)), ], col = "grey90")

## erase the region around a point instead (like RTriangle's pslg H)
r5 <- cdt_triangulate(x, y, s0, s1, holes = cbind(0.5, 0.5))
table(r5$depth)

## which input segments make up each fixed edge
r4 <- cdt_triangulate(x, y, s0, s1, max_area = 0.05, edge_sources = TRUE)
head(r4$edge_sources)

## points only: Delaunay triangulation of the convex hull
set.seed(1)
r3 <- cdt_triangulate(runif(30), runif(30))
```

```
nrow(r3$T)
```

`cdt_triangulate_attr` *Triangulate with per-vertex attributes carried onto new vertices*

Description

`cdt_triangulate()` plus a `PA` matrix of per-input-vertex attributes (z, m, t ...), returned as `$PA` aligned with the output vertices. Input vertices keep their values; Steiner vertices get values interpolated from the unrefined constrained triangulation of the input.

Usage

```
cdt_triangulate_attr(x, y, s0 = NULL, s1 = NULL, PA = NULL, ...)
```

Arguments

`x`, `y`	vertex coordinates
`s0`, `s1`	1-based vertex indices of constraint segment start/end (may be NULL)
`PA`	numeric matrix with one row per input vertex (or NULL)
`...`	passed to `cdt_triangulate()`

Value

the list returned by `cdt_triangulate()`, plus `PA`, a matrix with one row per output vertex and the columns of the input `PA` (absent when `PA = NULL`)

Examples

```
x <- c(0, 1, 1, 0)
y <- c(0, 0, 1, 1)
## a sloping surface: z rises with x
r <- cdt_triangulate_attr(x, y, 1:4, c(2:4, 1),
                          PA = cbind(z = 10 * x), max_area = 0.02)
## Steiner vertices carry interpolated z
head(cbind(r$P, r$PA)[r$origin == "steiner", , drop = FALSE])
all.equal(r$PA[, "z"], 10 * r$P[, 1])
```

cdt_winding *Winding number of each input feature in every triangle*

Description

For a triangulation of several overlapping inputs, find which inputs cover each triangle. Triangles are walked from a starting point known to be covered by nothing. Stepping across an ordinary edge changes nothing; stepping across a fixed edge changes the winding number of each feature covering it by plus or minus one, according to the direction of its input segment (see `edge_sources` in `cdt_triangulate()`). Only the mesh is used, with no point-in-polygon test.

Usage

```
cdt_winding(x, owner, outside, n_features = max(owner))
```

Arguments

`x`	a result of `cdt_triangulate()` made with `edge_sources = TRUE`
`owner`	the feature of each input segment, an integer vector with one element for each segment passed as `s0`, `s1`. Zero means the segment is not counted, for a frame or any other construction line. Features are `1` to `n_features`.
`outside`	a point `c(x, y)` in the mesh, covered by no feature
`n_features`	the number of features, by default the largest in `owner`

Details

Outer rings should run counter-clockwise and holes clockwise, which makes the winding number 1 inside the feature (a clockwise outer ring gives -1). Overlapping and shared edges are handled, and refinement does not matter, as `edge_sources` follows each input through splits.

The mesh needs a region covered by nothing to start from, for example a frame around the inputs (give it feature 0, or a feature that you ignore), and the walk must be able to reach every triangle. Use `erase = "hull"`, or the default `"outer"` with a frame. With `erase = "holes"` the triangles are not connected through the erased regions, so some are not reached.

Value

an integer matrix with one row per triangle (the rows of `x$T`) and one column per feature, giving its winding number there. Triangles the walk cannot reach are `NA`, with a warning.

Examples

```
## two overlapping squares, in a frame (feature 0)
sq <- function(x0, y0, x1, y1) rbind(c(x0, y0), c(x1, y0), c(x1, y1), c(x0, y1))
rings <- list(sq(0, 0, 2, 2), sq(1, 1, 3, 3), sq(-1, -1, 4, 4))
feature <- c(1, 2, 0)
n <- vapply(rings, nrow, 1L)
first <- cumsum(c(0L, n[-length(n)]))
```

```
xy <- do.call(rbind, rings)
s0 <- unlist(lapply(seq_along(n), function(k) first[k] + 1:4))
s1 <- unlist(lapply(seq_along(n), function(k) first[k] + c(2:4, 1L)))
r <- cdt_triangulate(xy[, 1], xy[, 2], s0, s1, max_area = 0.5,
                     edge_sources = TRUE)
w <- cdt_winding(r, owner = rep(feature, n), outside = c(-0.5, -0.5))
## triangles by the features covering them
table(feature_1 = w[, 1], feature_2 = w[, 2])
```

`default_min_edge_length` *Default refinement edge-length floor*

Description

`min_edge_frac` times the median constraint segment length, which keeps the refiner from cascading into sharp input corners, capped at `area_edge_frac * sqrt(max_area)` so the floor never blocks the area target (a triangle of area A has edges of order sqrt(A)). With no segments only the area cap applies; with neither, 0.

Usage

```
default_min_edge_length(
  x,
  y,
  s0,
  s1,
  max_area = NULL,
  min_edge_frac = 0.3,
  area_edge_frac = 0.25
)
```

Arguments

`x`, `y`	vertex coordinates
`s0`, `s1`	1-based vertex indices of constraint segment start/end (may be NULL)
`max_area`	maximum triangle area (Ruppert refinement by area), NULL for none
`min_edge_frac`	fraction of the median constraint segment length used when `min_edge_length` is NULL
`area_edge_frac`	fraction of `sqrt(max_area)` that caps the default floor

Value

a single non-negative number, the edge-length floor that `cdt_triangulate()` uses when `min_edge_length = NULL`

Examples

```
x <- c(0, 1, 1, 0)
y <- c(0, 0, 1, 1)
s0 <- 1:4
s1 <- c(2:4, 1)
## 0.3 times the median segment length
default_min_edge_length(x, y, s0, s1)
## a small area target caps the floor at 0.25 * sqrt(max_area)
default_min_edge_length(x, y, s0, s1, max_area = 0.01)
## no segments and no area target: no floor
default_min_edge_length(x, y, integer(0), integer(0))
```

Index

cdt_edges, 2
cdt_interpolate, 3
cdt_interpolate(), *3, 4*
cdt_locate, 3
cdt_pslg, 4
cdt_quality, 5
cdt_triangulate, 6
cdt_triangulate(), *2*, *5*, *9–11*
cdt_triangulate_attr, 9
cdt_winding, 10

default_min_edge_length, 11
default_min_edge_length(), *7*

graphics::segments(), *2*

summary(), *5*