

Package: laridae (via r-universe)

October 15, 2026

Type Package

Title Constrained Triangulation and Meshing with CGAL

Version 0.1.0

Description A persistent constrained Delaunay triangulation handle backed by the 'CGAL' (Computational Geometry Algorithms Library) classes 'Constrained_triangulation_plus_2' and 'Delaunay_mesher_2' (The CGAL Project (2026) <https://doc.cgal.org/6.2.1/Manual/packages.html>). Points and constraint segments can be inserted and removed at any time, crossing constraints are resolved by inserting the crossing, and the mesh can be refined by minimum angle, maximum area, or a sizing field that varies with position. Input and output follow the vertices / segments / triangles contract shared with 'cdtr', including per-triangle constraint depth.

License GPL (>= 3)

Language en-US

Encoding UTF-8

LinkingTo cpp11, Rcpp, RcppCGAL, BH

Imports Rcpp

SystemRequirements C++17

Suggests bench, knitr, rmarkdown, sf, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Roxygen list(markdown = TRUE)

URL <https://github.com/hypertidy/laridae>

BugReports <https://github.com/hypertidy/laridae/issues>

Config/roxygen2/version 8.1.0

Repository <https://hypertidy.r-universe.dev>

Date/Publication 2026-10-12 11:57:47 UTC

RemoteUrl <https://github.com/hypertidy/laridae>

RemoteRef HEAD
RemoteSha 00b8a181d5ff14b349fb0c4355e96c398422ac6a

Contents

default_min_edge_length	2
lari_add_points	3
lari_counts	4
lari_new	5
lari_refine	6
lari_step	7
lari_tables	8
lari_triangulate	9

Index 11

default_min_edge_length
<i>Default refinement edge-length floor</i>

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. Identical to cdtr::default_min_edge_length().

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 segment end indices into x/y |
| max_area | the area target, or NULL |
| min_edge_frac | fraction of the median segment length |
| area_edge_frac | fraction of sqrt(max_area) |

Value

a single number

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))
```

lari_add_points	<i>Edit a mesh</i>
-----------------	--------------------

Description

lari_add_points() inserts points (a point on a constraint splits it) and returns their vertex ids. lari_add_segments() inserts constraints between existing vertex ids, inserting the crossing point where they cross existing constraints, and returns the new segment ids. lari_remove_points() removes vertices, along with any segment that ends at them. lari_remove_segments() removes constraints by their end vertex ids (or by segment id with id); crossing vertices that no longer lie on any constraint are removed too, so adding and then removing a segment leaves the tables as they were.

Usage

```
lari_add_points(m, x, y, PA = NULL)

lari_add_segments(m, s0, s1)

lari_remove_points(m, ids)

lari_remove_segments(m, s0 = NULL, s1 = NULL, id = NULL)
```

Arguments

m	a mesh from lari_new()
x, y	point coordinates
PA	attribute matrix, one row per point, with the mesh's columns
s0, s1	vertex ids of segment ends
ids	vertex ids
id	segment ids (alternative to s0/s1)

Details

Any edit discards the mesher state: the next `lari_refine()` or `lari_step()` starts a new mesher on the edited triangulation (which only finds bad triangles where the edit made them).

Value

`lari_add_points()`: integer vertex ids. `lari_add_segments()`: integer segment ids (NA for a zero-length segment). The removers return the number removed, invisibly.

Examples

```
m <- lari_new(c(0, 1, 1, 0), c(0, 0, 1, 1))
ids <- lari_add_points(m, 0.5, 0.5)
lari_add_segments(m, c(1, 2), c(3, 4))    ## crossing diagonals
lari_vertices(m)
lari_remove_segments(m, 2, 4)
lari_vertices(m)
```

`lari_counts`

Mesh counts

Description

Vertex counts by origin, face and segment counts, and the report from the last refinement: bad triangles still failing a criterion, and of those how many have a short edge (below the floor), touch a sharp constrained corner, or have their circumcentre outside the meshed domain; inserted vertices, whether the max_steiner budget stopped it, and stalled (the mesher stopped making progress; should be 0).

Usage

```
lari_counts(m)
```

Arguments

`m` a mesh from `lari_new()`

Value

a list

Examples

```
## a square with a square hole: two closed rings of segments
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)
m <- lari_new(x, y, s0 = 1:8, s1 = c(2, 3, 4, 1, 6, 7, 8, 5))
lari_counts(m)$steiner
lari_refine(m, max_area = 0.01, min_angle = 25)
lari_counts(m)$steiner
```

lari_new	Create a mesh handle
----------	----------------------

Description

A laridae mesh is a persistent constrained Delaunay triangulation (CGAL's `Constrained_triangulation_plus_2` with exact predicates). Points and constraint segments can be added and removed at any time; crossing constraints are resolved by inserting the crossing point, and the constraint hierarchy maps every output edge back to the input segment it came from.

Usage

```
lari_new(x = NULL, y = NULL, s0 = NULL, s1 = NULL, PA = NULL)
```

Arguments

x, y	vertex coordinates (may be omitted for an empty mesh)
s0, s1	1-based indices into x/y of constraint segment start and end (may be NULL)
PA	optional numeric matrix (or data frame) of per-vertex attributes, one row per vertex, e.g. <code>cbind(z_ = z)</code> . New vertices (crossings, Steiner points) get linearly interpolated values. The attribute columns are fixed when the mesh is created.

Details

Vertex ids are stable for the life of the handle: removing a vertex leaves a tombstone and its id is never reused. Duplicate input points share one id (the first point's attributes are kept).

The handle is an external pointer: it does not survive saving and reloading an R session.

Value

a `laridae_mesh` handle; `attr(m, "ids")` holds the vertex id of each input row and `attr(m, "segment_ids")` the id of each input segment

Examples

```
## a square with a square hole
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)
m <- lari_new(x, y, 1:8, c(2, 3, 4, 1, 6, 7, 8, 5))
lari_triangles(m)
lari_refine(m, max_area = 0.01, min_angle = 25)
lari_counts(m)
```

lari_refine

*Refine a mesh***Description**

Ruppert-style Delaunay refinement with CGAL's Delaunay_mesher_2, by minimum angle, maximum area, and a sizing field. The mesher is kept on the handle so `lari_step()` can continue it one point at a time; any edit to the mesh discards it.

Usage

```
lari_refine(
    m,
    min_angle = NULL,
    max_area = NULL,
    size = NULL,
    max_steiner = Inf,
    min_edge_length = NULL,
    seeds = NULL,
    erase = c("outer", "holes", "hull")
)
```

Arguments

<code>m</code>	a mesh from <code>lari_new()</code>
<code>min_angle</code>	minimum triangle angle in degrees, NULL for none. Above about 20.7 degrees termination is not guaranteed by theory; the floor keeps it finite.
<code>max_area</code>	maximum triangle area, NULL for none
<code>size</code>	a sizing field giving the local maximum triangle area: a function(x, y) (called with scalars at each triangle centroid), or a grid as <code>list(x, y, z)</code> in <code>graphics::image()</code> convention (x and y are cell centres, z is length(x) by length(y); nearest cell, NA is unbounded). Combined with <code>max_area</code> by taking the smaller bound.
<code>max_steiner</code>	budget of new vertices for this call (Inf for none)
<code>min_edge_length</code>	the edge-length floor; NULL chooses <code>default_min_edge_length()</code> from the current segments and the area bound, taking the smaller of <code>max_area</code> and the finest area the sizing field asks for so a sizing field is honoured rather than capped; 0 never gives up
<code>seeds</code>	optional points (two-column matrix or list with x, y) marking regions NOT to mesh, CGAL style; when given they replace erase
<code>erase</code>	which region is meshed: "outer" (everything inside the outermost constraints, depth > 0), "holes" (odd depth only), or "hull"

Details

The refiner never splits a triangle whose shortest edge is below `min_edge_length`: that floor is what stops refinement cascading into sharp input corners. Whatever could not be refined is reported by `lari_counts()` (unrefined) and returned here as a data frame.

Value

the unrefined report as a one-row data frame, invisibly

Examples

```
x <- c(0, 1, 1, 0); y <- c(0, 0, 1, 1)
m <- lari_new(x, y, 1:4, c(2, 3, 4, 1))
lari_refine(m, max_area = 0.01)
## a sizing field: small triangles near the origin
m2 <- lari_new(x, y, 1:4, c(2, 3, 4, 1))
lari_refine(m2, size = function(x, y) 0.0005 + 0.02 * (x^2 + y^2))
nrow(lari_triangles(m2))
```

lari_step

Step the mesher

Description

Insert up to `n` refinement points with CGAL's step-by-step mesher, using the criteria from the last `lari_refine()` call. After an edit the mesher is rebuilt on the edited mesh, so stepping refines only what the edit spoiled.

Usage

```
lari_step(m, n = 1L)
```

Arguments

<code>m</code>	a mesh from <code>lari_new()</code>
<code>n</code>	maximum number of new vertices

Value

the number of vertices inserted

Examples

```
m <- lari_new(c(0, 1, 1, 0), c(0, 0, 1, 1), 1:4, c(2, 3, 4, 1))
lari_refine(m, max_area = 0.01, max_steiner = 0)
lari_step(m, 5)
lari_counts(m)$steiner
```

lari_tables

*Mesh tables***Description**

The three output tables of the shared contract.

Usage

```
lari_tables(m, erase = c("outer", "holes", "hull"))

lari_vertices(m)

lari_triangles(m, erase = c("outer", "holes", "hull"))

lari_segments(m)

lari_depth(m, erase = c("outer", "holes", "hull"))
```

Arguments

<code>m</code>	a mesh from <code>lari_new()</code>
<code>erase</code>	which triangles to return: "outer" (depth > 0, what Triangle's -p keeps), "holes" (odd depth), "hull" (all). With no segments every triangle is returned.

Details

- vertices: x, y, the attribute columns, origin ("input", "crossing", "steiner") and the stable vertex id
- triangles: v0, v1, v2 (1-based rows of the vertices table), depth and depth_distinct
- segments: constraint edges v0, v1 (rows of the vertices table), the input segment id they came from (the lowest, when several coincide) and count, the number of input segments covering the edge

depth is the constraint layer: the number of constraint crossings on the way in from outside the convex hull, counting an edge covered by k input segments k times. depth_distinct is the same count but a doubled boundary (an edge covered by several coincident segments) is crossed once, so it ignores the count weighting. For nested rings odd is inside and even is a hole; for a coverage it orders regions by distance from the boundary.

`lari_tables()` also returns a one-row report data frame recording the provenance of the tables: backend ("laridae"), the laridae package and cgal versions, the erase region, whether the mesh was refined, the min_edge_length floor, and the last refinement's unrefined counts (bad, short_edge, sharp_corner, outside, inserted, budget_hit, stalled), all NA when the mesh has not been refined.

Row indices are only valid for one snapshot of the tables: use id to follow vertices across edits.

Value

a data frame; `lari_tables()` returns the three tables and the report in a list, `lari_depth()` the depth column of the triangles

Examples

```
m <- lari_new(c(0, 1, 1, 0, 0.25, 0.75, 0.75, 0.25),
              c(0, 0, 1, 1, 0.25, 0.25, 0.75, 0.75),
              1:8, c(2, 3, 4, 1, 6, 7, 8, 5))
table(lari_depth(m))
lari_segments(m)
```

<code>lari_triangulate</code>	<i>One-shot constrained triangulation</i>
-------------------------------	---

Description

Build a mesh, optionally refine it, and return the tables: the same shape as `cdtr`'s one-shot call, for use as a drop-in backend.

Usage

```
lari_triangulate(
  x,
  y,
  s0 = NULL,
  s1 = NULL,
  PA = NULL,
  max_area = NULL,
  min_angle = NULL,
  min_edge_length = NULL,
  erase = c("outer", "holes", "hull"),
  ...
)
```

Arguments

<code>x, y</code>	vertex coordinates (may be omitted for an empty mesh)
<code>s0, s1</code>	1-based indices into <code>x/y</code> of constraint segment start and end (may be <code>NULL</code>)
<code>PA</code>	optional numeric matrix (or data frame) of per-vertex attributes, one row per vertex, e.g. <code>cbind(z_ = z)</code> . New vertices (crossings, Steiner points) get linearly interpolated values. The attribute columns are fixed when the mesh is created.
<code>max_area</code>	maximum triangle area, <code>NULL</code> for none
<code>min_angle</code>	minimum triangle angle in degrees, <code>NULL</code> for none. Above about 20.7 degrees termination is not guaranteed by theory; the floor keeps it finite.

min_edge_length	the edge-length floor; NULL chooses <code>default_min_edge_length()</code> from the current segments and the area bound, taking the smaller of max_area and the finest area the sizing field asks for so a sizing field is honoured rather than capped; 0 never gives up
erase	which region is meshed: "outer" (everything inside the outermost constraints, depth > 0), "holes" (odd depth only), or "hull"
...	passed to <code>lari_refine()</code> (size, max_steiner, seeds)

Value

a list with vertices, triangles, segments, a one-row provenance report (see `lari_tables()`), unrefined (NULL when no refinement was asked for) and min_edge_length

Examples

```
r <- lari_triangulate(c(0, 1, 1, 0), c(0, 0, 1, 1), 1:4, c(2, 3, 4, 1), max_area = 0.05)
str(r)
```

Index

`default_min_edge_length`, [2](#)
`default_min_edge_length()`, [6](#), [10](#)

`graphics::image()`, [6](#)

`lari_add_points`, [3](#)
`lari_add_segments(lari_add_points)`, [3](#)
`lari_counts`, [4](#)
`lari_counts()`, [7](#)
`lari_depth(lari_tables)`, [8](#)
`lari_new`, [5](#)
`lari_new()`, [3](#), [4](#), [6–8](#)
`lari_refine`, [6](#)
`lari_refine()`, [4](#), [7](#), [10](#)
`lari_remove_points(lari_add_points)`, [3](#)
`lari_remove_segments(lari_add_points)`,
 [3](#)
`lari_segments(lari_tables)`, [8](#)
`lari_step`, [7](#)
`lari_step()`, [4](#), [6](#)
`lari_tables`, [8](#)
`lari_tables()`, [10](#)
`lari_triangles(lari_tables)`, [8](#)
`lari_triangulate`, [9](#)
`lari_vertices(lari_tables)`, [8](#)