

Package: meshcore (via r-universe)

October 15, 2026

Title One Verb Set for Lattice Meshes: Rasters and Discrete Global
Grids

Version 0.0.0.9000

Description Prototype for the mesh core (working name). A raster
(GRID) and a discrete global grid (CELL) are both meshes whose
vertex, edge and boundary tables are computed from cell ids, so
no coincident vertices are ever merged. The same verbs work on
both, and arrays join to either model through a keyed dimension
(cell id). Vocabulary follows UGRID, xdggs, uxarray and OGC API
DGGs; vertex and segment tables follow wkpool.

License MIT + file LICENSE

Encoding UTF-8

Imports jsonlite, sf, wk

Suggests testthat (>= 3.0.0), wkpool, cdtr, laridae

Remotes hypertidy/wkpool, hypertidy/cdtr, hypertidy/laridae

Config/testthat/edition 3

RoxygenNote 7.3.1

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev libssl-dev libproj-
dev libsqlite3-dev libudunits2-dev

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

Date/Publication 2026-10-14 05:19:49 UTC

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

RemoteRef HEAD

RemoteSha 7e577dc43cf93a3cbfbecbc5794d7fd4e789ad86

Contents

as_wk	2
GRID	2
hpx_nest2fxy	4
keyed_array	4

long_join	5
mesh_counts	6
n_cells	6
tri_form	6
verbs	8

Index	10
--------------	-----------

as_wk	<i>Geometry and wkpool output</i>
-------	-----------------------------------

Description

‘as_wk()’ returns one polygon per cell (wk ‘wk_polygon’), built from ‘boundaries()’ and ‘vertices()’. ‘as_wkpool()’ hands the mesh to wkpool as a pool of the model’s vertices and one closed path of segments per cell, with ‘.vx’ renumbered 1..n (the lattice key is kept as ‘.key’). The pool is already merged: no ‘merge_coincident()’ needed.

Usage

```
as_wk(x)

as_wkpool(x)
```

Arguments

x A model.

GRID	<i>Mesh models</i>
------	--------------------

Description

‘GRID()’ is a regular raster: a lattice of ‘ncol * nrow’ square-ish cells over an extent. ‘CELL()’ is a set of cells of a discrete global grid (DGGS) or other cell lattice, identified by integer cell ids.

Usage

```
GRID(x = NULL, dim = NULL, extent = NULL, crs = NA_character_)

CELL(
  x = NULL,
  grid_name = c("healpix", "hexlattice"),
  level = NULL,
  indexing_scheme = "nested",
  radius = 1,
  origin = c(0, 0),
```

```

ncol = NULL,
nrow = NULL,
orientation = c("flat", "pointy"),
crs = NA_character_
)

```

Arguments

x	For 'GRID()', a raster data source readable by GDAL, or 'NULL' to give 'dim' and 'extent'. For 'CELL()', a vector of cell ids, a multidimensional data source (e.g. Zarr) holding a 'cell_ids' array with xdggs attributes, or 'NULL' for every cell at 'level'.
dim	'c(ncol, nrow)'.
extent	'c(xmin, xmax, ymin, ymax)'.
crs	Carried, not interpreted.
grid_name	"healpix" or "hexlattice" (a planar flat-topped hexagon lattice).
level	Refinement level (HEALPix 'nside = 2^level').
indexing_scheme	Only "nested" for HEALPix.
radius, origin, ncol, nrow	hexlattice parameters: hexagon circumradius, the coordinate of lattice cell (0, 0) centre, and the lattice size in columns and rows (cell id = row * ncol + col).
orientation	hexlattice "flat" topped, or "pointy" topped (the flat lattice with x and y exchanged, as sf::st_make_grid() makes by default; "columns" then run along y).

Details

Both are meshes whose tables (cells, vertices, edges, boundaries) are computed on demand from cell ids. Vertex identity is by construction: every vertex has an integer lattice key, so coincident vertices are never found by comparing coordinates. This is recorded as 'rung = "lattice"', a rung above "exact" on the vertex identity ladder (exact float, snapped, noded, exact predicates).

Cell ids are 0-based, as in GDAL pixel offsets, HEALPix and xdggs. GRID cell ids run row-major from the top-left cell (GDAL order).

Value

A model object (a list with 'meta' and 'cell').

hpx_nest2fxy	<i>HEALPix nested id to face and within-face coordinates</i>
--------------	--

Description

HEALPix nested id to face and within-face coordinates

Usage

```
hpx_nest2fxy(id, level)
```

```
hpx_fxy2nest(face, ix, iy, level)
```

Arguments

id	Cell ids (0-based, nested).
level	Refinement level, 'nside = 2^level'.
face, ix, iy	Base face (0..11) and within-face coordinates.

Value

'hpx_nest2fxy()' a list of 'face', 'ix', 'iy'; 'hpx_fxy2nest()' cell ids.

keyed_array	<i>Arrays with a keyed dimension, and joining them to models</i>
-------------	--

Description

A keyed array is an array plus the name of the dimension(s) that index model cells. For a raster the key is computed from the 'x' and 'y' dimensions (cell id = row * ncol + col). For a DGGS the key is a coordinate variable of cell ids on the 'cells' dimension (the xdggs convention: 'cell_ids' with 'grid_name', 'level', 'indexing_scheme' attributes).

Usage

```
keyed_array(values, dims, key_dim, key = NULL, coords = list())
```

```
read_array(dsn, name = NULL)
```

```
join_array(x, a, ...)
```

Arguments

values	An R array, fastest-varying dimension first.
dims	Named dimension sizes, in the same order as ‘dim(values)’.
key_dim	Name(s) of the keyed dimension(s).
key	Cell ids along ‘key_dim’, or ‘NULL’ when the key is computed.
coords	Named list of coordinate values for the other dimensions.
dsn	Data source.
name	Array name for multidimensional sources.
x	A model.
a	A keyed array.
...	Unused.

Details

‘read_array()’ reads a classic raster (e.g. GeoTIFF) or an array from a GDAL multidimensional source (e.g. Zarr) through GDAL, via sf.

‘join_array()’ returns the model’s cells in long form, one row per cell and per combination of the array’s other dimensions, with a ‘value’ column. The join is by cell id only: no geometry is involved.

long_join	<i>Join helper for join_array() methods</i>
-----------	---

Description

For writing ‘join_array()’ methods in other packages.

Usage

```
long_join(cl, idx, m, a)
```

Arguments

cl	The model’s ‘cells()’ table.
idx	For each row of ‘cl’, the position along the array’s keyed dimension (‘NA’ for none).
m	The array values as a matrix: one row per position along the keyed dimension, one column per combination of the other dimensions.
a	The keyed array (for its other dimensions and coordinates).

Value

‘cl’ in long form with the other dimensions and ‘value’.

mesh_counts	<i>Counts of cells, vertices and edges</i>
-------------	--

Description

Counts of cells, vertices and edges

Usage

mesh_counts(x)

Arguments

x A model.

n_cells	<i>Number of cells in a model</i>
---------	-----------------------------------

Description

Number of cells in a model

Usage

n_cells(x)

Arguments

x A model.

tri_form	<i>Triangle form (mesh spec v0.1)</i>
----------	---------------------------------------

Description

‘tri_form()’ normalises the output of a triangulation engine to the triangle form of mesh spec v0.1, so neither engine has to change before meshcore can use it. ‘tri_mesh()’ runs an engine and returns the form in one call.

Usage

```

tri_form(x, erase = NULL, crs = NA_character_)

tri_mesh(
  x,
  y,
  s0 = NULL,
  s1 = NULL,
  PA = NULL,
  engine = c("cdtr", "laridae"),
  max_area = NULL,
  min_angle = NULL,
  erase = c("outer", "holes", "hull"),
  crs = NA_character_,
  ...
)

```

Arguments

x	For 'tri_form()', the result of [cdtr::cdt_triangulate()] (or [cdtr::cdt_triangulate_attr()]), of [laridae::lari_triangulate()] or [laridae::lari_tables()], or a laridae mesh handle. For 'tri_mesh()', vertex x coordinates.
erase	The erase choice the engine was asked for ("outer", "holes" or "hull"). Taken from the engine output when it records it (laridae); cdtr does not, so give it here, or it is recorded as 'NA'. For a laridae mesh handle, the triangles to return.
crs	Carried in the 'meshspec' attribute, not interpreted.
y	Vertex y coordinates.
s0, s1	1-based indices into 'x', 'y' of constraint segment ends, or 'NULL'.
PA	Optional numeric matrix of per-vertex attributes (one row per input vertex, named columns), interpolated onto new vertices.
engine	"cdtr" (the default) or "laridae". laridae gives the constraint-to-input map, and refines where the domain is the convex hull.
max_area, min_angle	Refinement targets, or 'NULL' for none.
...	Passed to the engine.

Details

The form is a named list of four data frames:

* 'vertices': 'x', 'y', any attribute columns, 'origin' (a factor with the spec dictionary 'input', 'crossing', 'steiner', 'grid', 'lattice') and, when the engine has stable ids, 'id'. Input vertices come first, in input order (after duplicates are merged). * 'triangles': 'v0', 'v1', 'v2' (1-based rows of 'vertices', wound counter-clockwise in xy), 'depth' (constraint crossings from outside, counting a doubled boundary once per copy) and, when available, 'depth_distinct' (a doubled boundary counted once). * 'constraints': 'v0', 'v1' (1-based rows of 'vertices', 'v0 < v1'), the mesh edges lying on input segments, and, when the engine provides them, 'segment' (the lowest input segment covering the

edge) and ‘count’ (how many input segments cover it). `laridae` calls this table ‘segments’ and `cdtr` calls it ‘S’; ‘`tri_form()`’ renames both to the spec name (Decision 7). * ‘report’: one row recording ‘backend’, ‘package’ (engine version), ‘erase’, ‘refined’, ‘min_edge_length’ and the row counts.

Row references are 1-based, as the spec requires in R; they become 0-based only when the form is written to Arrow. The list carries attributes ‘meshspec’ (‘version’, ‘form’, ‘crs’), ‘input_map’ (the vertex row of each input point, so features can be followed into the mesh) and ‘unrefined’ (the engine’s own refinement report, unchanged).

Value

A list of class ‘`meshcore_tri`’ with ‘vertices’, ‘triangles’, ‘constraints’ and ‘report’.

Examples

```
if (requireNamespace("cdtr", quietly = TRUE)) {
  ## 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)
  s0 <- 1:8; s1 <- c(2, 3, 4, 1, 6, 7, 8, 5)
  m <- tri_mesh(x, y, s0, s1, erase = "holes", max_area = 0.01)
  m
  ## the same from an engine result
  tri_form(cdtr::cdt_triangulate(x, y, s0, s1, erase = "holes"), erase = "holes")
}
```

verbs

The verb set

Description

The same calls work on every model.

Usage

```
cells(x, ...)

boundaries(x, ...)

vertices(x, ...)

face_edge(x, ...)

edges(x, ...)

neighbours(x, type = c("edge", "vertex"), ...)

parent(x, n = 1, ...)

children(x, n = 1, ...)
```

Arguments

<code>x</code>	A model from <code>[GRID()]</code> or <code>[CELL()]</code> .
<code>...</code>	Unused.
<code>type</code>	<code>"edge"</code> or <code>"vertex"</code> adjacency.
<code>n</code>	Number of levels to move.

Details

* `'cells()'`: one row per cell: `'cell'` id and centre `'x'`, `'y'` (longitude and latitude for HEALPix). xdggs `'cell_centers'`; OGC API DGGs zones. * `'boundaries()'`: one row per cell corner, counter-clockwise: `'cell'`, `'corner'`, `'vx'`. UGRID `'face_node_connectivity'` in long form. * `'vertices()'`: one row per unique vertex: `'vx'` (its lattice key), `'x'`, `'y'`. UGRID nodes; wkpool's vertex pool. * `'edges()'`: one row per unique undirected edge: `'edge'`, `'vx0'`, `'vx1'`. UGRID `'edge_node_connectivity'`; wkpool segments. * `'face_edge()'`: one row per cell side: `'cell'`, `'corner'`, `'edge'`. UGRID `'face_edge_connectivity'`. * `'neighbours()'`: pairs of cells sharing an edge (`'type = "edge"'`) or a vertex (`'type = "vertex"'`). UGRID `'face_face_connectivity'`. * `'parent()'`, `'children()'`: the hierarchy. OGC API DGGs parent zones and sub-zones; xdggs `'zoom_to'`.
`'vertices()'`, `'edges()'`, `'face_edge()'` and `'neighbours()'` are derived from `'boundaries()'` by integer keys alone, so they are the same code for every model. Only `'cells()'` and `'boundaries()'` (and the hierarchy) are written per model.

Index

`as_wk`, [2](#)
`as_wkpool (as_wk)`, [2](#)

`boundaries (verbs)`, [8](#)

`CELL (GRID)`, [2](#)
`cells (verbs)`, [8](#)
`children (verbs)`, [8](#)

`edges (verbs)`, [8](#)

`face_edge (verbs)`, [8](#)

`GRID`, [2](#)

`hpx_fxy2nest (hpx_nest2fxy)`, [4](#)
`hpx_nest2fxy`, [4](#)

`join_array (keyed_array)`, [4](#)

`keyed_array`, [4](#)

`long_join`, [5](#)

`mesh_counts`, [6](#)

`n_cells`, [6](#)
`neighbours (verbs)`, [8](#)

`parent (verbs)`, [8](#)

`read_array (keyed_array)`, [4](#)

`tri_form`, [6](#)
`tri_mesh (tri_form)`, [6](#)

`verbs`, [8](#)
`vertices (verbs)`, [8](#)