Package 'a5R'

October 19, 2026

Title 'A5' Discrete Global Grid System

Version 1.0.0

Description Bindings for the ``A5 geospatial index'' <https://a5geo.org/>. 'A5' partitions the Earth's surface into pentagonal cells across 31 resolution levels using an equal-area projection onto a dodecahedron. Provides functions for indexing coordinates to cells, traversing the cell hierarchy, computing cell boundaries, and compacting/uncompacting cell sets. Powered by the 'A5' 'Rust' crate via 'extendr'.

License Apache License (>= 2)

URL https://github.com/belian-earth/a5R, https://belian-earth.github.io/a5R/

BugReports https://github.com/belian-earth/a5R/issues

Depends R (>= 4.2)

Imports cli, rlang (>= 1.1.0), units, utils, vctrs (>= 0.6.0), wk (>= 0.9.0)

Suggests arrow, grid, knitr, parallel, pillar, rmarkdown, sf, terra, testthat (>= 3.0.0), tibble, tidyr, withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/rextendr/version 0.5.0

SystemRequirements Cargo (Rust's package manager), rustc (>= 1.82.0)

Encoding UTF-8

Language en-GB

RoxygenNote 8.0.0

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Hugh Graham [aut, cre],
belian.earth [cph],
Felix Palmer [ctb, cph] (Author of the bundled 'a5' Rust crate),
Authors of the dependency Rust crates [cph] (see LICENSE.note)

Maintainer Hugh Graham <hugh@belian.earth>

Repository CRAN

Date/Publication 2026-10-19 10:20:11 UTC

Contents

a5_bbox . . . 3
a5_cell . . . 4
a5_cell_child . . . 5
a5_cell_children_range . . . 5
a5_cell_distance . . . 7
a5_cell_from_arrow . . . 8
a5_cell_list . . . 9
a5_cell_to_boundary . . . 10
a5_cell_to_boundary_xy . . . 11
a5_compaction . . . 12
a5_coordinates . . . 13
a5_covering . . . 14
a5_covering_from_arrow . . . 16
a5_covering_measures . . . 17
a5_covering_predicates . . . 18
a5_covering_stored . . . 19
a5_covering_to_supercells . . . 20
a5_errors . . . 21
a5_get_res0_cells . . . 22
a5_get_resolution . . . 22
a5_grid_disk . . . 23
a5_hierarchy . . . 24
a5_info . . . 25
a5_is_valid . . . 26
a5_line_string_to_cells . . . 27
a5_migrate . . . 28
a5_polygon_to_cells . . . 29
a5_resolution_stats . . . 31
a5_set_operations . . . 32
a5_spatial_hierarchy . . . 33
a5_spherical_cap . . . 35
a5_threads . . . 36
a5_to_arrow_table . . . 37
a5_u64_to_hex . . . 38
wk_methods . . . 39

Index **40**

a5_bbox *Bounding box of polygons with great-circle edges*

Description

Returns, for each polygon feature, the longitude/latitude box that contains its edges as A5 draws them: great circles between consecutive vertices (see `a5_polygon_to_cells()`). Such an edge bows toward the pole, so the box can extend beyond the latitudes of the vertices, which a plain vertex bounding box would miss. A ring that winds around a pole extends the box to that pole across all longitudes.

Usage

```
a5_bbox(x)
```

Arguments

x	A polygon-like geometry. One of: • Any geometry handleable by `wk::wk_handle()` (e.g. `wk::wkt()`, `wk::wkb()`, `wk::rct()`, `sf`, `sfc`) containing one or more `POLYGON` / `MULTIPOLYGON` features. • A `SpatVector` of polygons (requires the `terra` package). • A two-column numeric matrix (`cbind(lon, lat)`) of vertices, interpreted as a single outer ring. • A `data.frame` with columns `lon` and `lat`, interpreted as a single outer ring.

Details

Longitudes are continuous: a box crossing the antimeridian has `xmin` between -180 and 180 and `xmax` above 180.

Value

A `wk::rct()` vector with one box per feature (one for matrix and `data.frame` input), in WGS 84 longitude and latitude. `NA` for a missing or empty geometry.

See Also

`a5_polygon_to_cells()`

Examples

```
# an edge along 60 degrees north bows toward the pole
poly <- wk::wkt("POLYGON ((-30 50, 30 50, 30 60, -30 60, -30 50))")
a5_bbox(poly)
wk::wk_bbox(poly)
```

a5_cell *A5 Cell Index Vector*

Description

Create, test, and coerce A5 cell index vectors. Cells are stored as a record with eight raw-byte fields (b1–b8) representing the little-endian bytes of the u64 cell ID. This avoids the precision loss of floating-point storage and keeps memory compact.

Usage

```
a5_cell(x = character())

is_a5_cell(x)

as_a5_cell(x)
```

Arguments

x	A character vector of hex-encoded A5 cell IDs, or an object coercible to one.

Details

Ids are hex strings of up to 16 digits, optionally prefixed with 0x. Shorter ids, as some A5 libraries write them, are padded with leading zeros, and a string that is not a hex id is an error. a5_cell() does not check that an id encodes an A5 cell, so an a5_cell vector can hold values that are not cells. Check ids from outside sources, such as a file or a database, with a5_is_valid_cell().

Value

An a5_cell vector (a5_cell, as_a5_cell), or a logical scalar (is_a5_cell).

See Also

a5_is_valid_cell() to check that values are A5 cells, and a5_is_valid() for any a5R object; a5_u64_to_hex() to convert back to hex strings; a5_migrate() for ids written by a5R before 1.0.

Examples

```
cells <- a5_cell(c("0800000000000006", "0800000000000016"))
cells
a5_is_valid_cell(cells)

# a5_cell() does not validate: this value is not a cell
a5_is_valid_cell(a5_cell("fd00000000000000"))
```

a5_cell_child *The i-th child of each cell*

Description

Returns one descendant of each cell at a finer resolution without enumerating the others: the `i`-th element of what `a5_cell_to_children()` would return. Useful for random draws from large cells, where building the full child list to pick one element is wasteful.

Usage

```
a5_cell_child(cell, child_resolution, i)
```

Arguments

`cell`	An a5_cell vector.
`child_resolution`	Integer scalar target resolution, at or finer than every cell's own resolution.
`i`	Integer vector of 1-based child positions, recycled against `cell`. Must lie within `1:a5_get_num_children(res, child_resolution)` for each cell's resolution `res`; out of range is an error.

Value

An a5_cell vector the same length as `cell`. `NA` where `cell` or `i` is `NA`.

See Also

`a5_cell_to_children()`, `a5_get_num_children()`

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 5)
a5_cell_child(cell, child_resolution = 7, i = 1:16)
identical(a5_cell_child(cell, 7, 1:16), a5_cell_to_children(cell, 7))
```

a5_cell_children_range *Range of descendant cell ids*

Description

Returns the smallest and largest descendant of each cell at a finer resolution, without enumerating the descendants. Because the A5 index encodes position along a space-filling curve with a cell's descendants sharing its leading bits, the descendants of a cell at any resolution up to 29 occupy a contiguous range of ids among all cells at that resolution: every cell at `child_resolution` whose id lies between `lo` and `hi` inclusive is a descendant, and no descendant lies outside. This makes `BETWEEN lo AND hi` an exact filter on a store of `child_resolution` cells sorted by id. Not every integer in the range is a valid cell, and cells of other resolutions may fall inside it.

Usage

```
a5_cell_children_range(cell, child_resolution)
```

Arguments

`cell`	An a5_cell vector.
`child_resolution`	Integer scalar target resolution from 0 to 29, at or finer than every cell's own resolution.

Details

Resolution 30 is refused because the guarantee does not hold there. The index has no room for the usual face bits at that resolution, so faces use three different bit layouts whose id ranges overlap numerically, and a range can contain valid resolution-30 cells from another face. Eighteen of the sixty faces cannot be encoded at resolution 30 at all; the upstream library falls back to resolution 29 for them. Use `a5_cell_to_children()` if you need resolution-30 descendants.

Value

A data frame with a5_cell columns `lo` and `hi`, one row per input cell. `NA` where `cell` is `NA`.

See Also

`a5_cell_to_children()`, `a5_cell_to_arrow()` for exact 64-bit ids to pass to SQL.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 5)
rng <- a5_cell_children_range(cell, child_resolution = 8)
children <- vctrs::vec_sort(a5_cell_to_children(cell, child_resolution = 8))
identical(rng$lo, children[1])
identical(rng$hi, children[length(children)])
```

a5_cell_distance *Distance between cell centroids*

Description

Computes the distance between the centroids of pairs of A5 cells using the specified method.

Usage

```
a5_cell_distance(
  from,
  to,
  units = "m",
  method = c("haversine", "geodesic", "rhumb")
)
```

Arguments

from, to	a5_cell vectors (recycled to common length).
units	Character scalar specifying the distance unit (default "m"). Any unit convertible from metres via units::set_units() is accepted (e.g. "km", "mi"). If NULL, the distance is returned as a numeric vector in metres.
method	Distance calculation method. One of "haversine" (great-circle, default), "geodesic" (WGS84 ellipsoid via Karney 2013), or "rhumb" (loxodrome / constant-bearing).

Value

A units::units vector of distances.

See Also

a5_cell_to_lonlat() for cell centroids, a5_cell_area() for cell areas.

Examples

```
a <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 24)
b <- a5_lonlat_to_cell(-3.10, 55.90, resolution = 24)
a5_cell_distance(a, b)
a5_cell_distance(a, b, units = "km")
a5_cell_distance(a, b, method = "geodesic")
```

a5_cell_from_arrow *Convert between a5_cell and Arrow uint64 arrays*

Description

Losslessly convert between a5_cell vectors and Arrow `uint64` arrays. This avoids the precision loss that occurs when Arrow converts `uint64` to R's `double` (which can only represent integers exactly up to 2^53, while A5 cell IDs span the full 0–2^64 range).

Usage

```
a5_cell_from_arrow(x)

a5_cell_to_arrow(x)
```

Arguments

`x`	For `a5_cell_from_arrow()`, an Arrow `Array` or `ChunkedArray` of type `uint64`. For `a5_cell_to_arrow()`, an a5_cell vector.

Details

The conversion runs in Rust and passes one contiguous `uint64` buffer through the Arrow C data interface, so no value is ever converted to `double`. NA cells are null elements. The resulting Arrow arrays can be written directly to Parquet and read correctly by DuckDB, Python, and other Arrow-compatible tools.

Value

`a5_cell_from_arrow()` returns an a5_cell vector. `a5_cell_to_arrow()` returns an Arrow `Array` of type `uint64`.

See Also

`a5_u64_to_hex()` for converting to hex strings instead.

Examples

```
cell <- a5_lonlat_to_cell(135, 0, resolution = 10)
arr <- a5_cell_to_arrow(cell)
back <- a5_cell_from_arrow(arr)
identical(format(cell), format(back))


cells <- a5_lonlat_to_cell(c(-3.19, 135), c(55.95, 0), resolution = 10)
arr <- a5_cell_to_arrow(cells)
arr$type$ToString()
```

a5_cell_list *List of A5 cell vectors*

Description

`a5_cell_to_children()`, `a5_grid_disk()` and `a5_spherical_cap()` return an `a5_cell_list` when called with `simplify = FALSE`: a `vctrs::list_of()` holding one a5_cell vector per input, usable as a list column. It behaves as any list: `lengths()` gives the number of cells per input, `[[` extracts one element, and `tidyr::unnest()` expands it to one row per cell.

Usage

```
as_a5_cell_list(x)

## S3 method for class 'a5_cell_list'
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

`x`	An `a5_cell_list`, or for `as_a5_cell_list()` a list of a5_cell vectors.
`recursive`, `use.names`	Ignored.

Details

`unlist()` concatenates the elements into a single a5_cell vector, equivalent to `vctrs::list_unchop()`. Without this method base `unlist()` would descend into the raw byte fields of the record and return a meaningless raw vector. The `recursive` and `use.names` arguments are accepted for compatibility and ignored.

`as_a5_cell_list()` wraps a plain list of a5_cell vectors (for example the output of `lapply()`) so that `unlist()` works on it. Elements that are hex strings are converted; `NULL` elements become empty.

Value

`unlist()` returns an a5_cell vector.

Examples

```
cells <- a5_lonlat_to_cell(c(-3.19, 0), c(55.95, 0), resolution = 5)
lst <- a5_cell_to_children(cells, simplify = FALSE)
lengths(lst)
unlist(lst)

# lapply() produces a plain list, on which base unlist() would return raw
# bytes; wrap it first.
plain <- lapply(seq_along(cells), function(i) a5_cell_to_children(cells[i]))
unlist(as_a5_cell_list(plain))
```

a5_cell_to_boundary *Get cell boundary polygons*

Description

Returns the boundary of each cell as a wk::wkt() or wk::wkb() polygon geometry. Boundaries are pentagonal polygons on the WGS 84 ellipsoid.

Usage

```
a5_cell_to_boundary(
  cell,
  format = c("wkb", "wkt"),
  closed_ring = TRUE,
  segments = NULL,
  closed
)
```

Arguments

cell	An a5_cell vector.
format	Character scalar, either "wkb" (default) or "wkt".
closed_ring	Logical scalar; if TRUE (default) the ring is closed (first vertex repeated at end).
segments	Integer scalar or NULL. Number of segments per cell edge. A5 cell edges are slightly curved on the sphere: a single segment between two vertices strays from the true edge by up to about 450 m at resolution 6, 25 m at resolution 8 and 1.5 m at resolution 10. The error falls with the square of the number of segments. NULL uses the A5 default of max(1, 2^(6 - resolution)), so at resolution 6 and finer each edge is a single segment. Pass a larger value when the polygons are used for spatial tests rather than display.
closed	Deprecated: renamed closed_ring, as in the other A5 libraries.

Value

A wk_wkt or wk_wkb vector of polygon geometries with wk::wk_crs_longlat() CRS.

See Also

a5_cell_to_boundary_xy() for the same vertices as flat coordinate vectors; a5_cell_to_lonlat() for cell centroids.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 5)
a5_cell_to_boundary(cell)
a5_cell_to_boundary(cell, format = "wkt")
```

a5_cell_to_boundary_xy *Get cell boundary vertices as flat coordinate vectors*

Description

Returns the boundary vertices of every cell in one pair of coordinate vectors, with a vector of per-cell vertex counts. This skips building a geometry for each cell, so it suits drawing many cells at once, for example as a single `grid::polygonGrob()` with `id.lengths = lengths`.

Usage

```
a5_cell_to_boundary_xy(cell, closed_ring = FALSE, segments = NULL, closed)
```

Arguments

`cell`	An a5_cell vector.
`closed_ring`	Logical scalar; if `TRUE` each ring repeats its first vertex at the end. Defaults to `FALSE`, unlike `a5_cell_to_boundary()`, since polygon drawing closes rings itself.
`segments`	Integer scalar or `NULL`. Number of segments per cell edge. A5 cell edges are slightly curved on the sphere: a single segment between two vertices strays from the true edge by up to about 450 m at resolution 6, 25 m at resolution 8 and 1.5 m at resolution 10. The error falls with the square of the number of segments. `NULL` uses the A5 default of `max(1, 2^(6 - resolution))`, so at resolution 6 and finer each edge is a single segment. Pass a larger value when the polygons are used for spatial tests rather than display.
`closed`	Deprecated: renamed `closed_ring`, as in the other A5 libraries.

Details

The vertices and their order are those of `a5_cell_to_boundary()`. Longitudes within a ring are continuous, as in the other A5 libraries: a ring that crosses the antimeridian continues past 180 or -180 instead of jumping, so `x` can fall outside [-180, 180]. Shift or split such rings before drawing them on a map that wraps at the antimeridian. A ring around a pole cannot be drawn as a simple polygon in longitude and latitude.

Value

A list with three elements:

- `x`: double vector of longitudes, every cell's vertices in turn.
- `y`: double vector of latitudes, aligned with `x`.
- `lengths`: integer vector with the number of vertices of each cell, the same length as `cell`. An `NA` cell has length 0.

See Also

a5_cell_to_boundary() for polygon geometries.

Examples

```
cells <- a5_lonlat_to_cell(c(-3.19, -3.10), c(55.95, 55.90), resolution = 9)
xy <- a5_cell_to_boundary_xy(cells)
xy$lengths

# one grob for all cells
grob <- grid::polygonGrob(
  xy$x, xy$y,
  id.lengths = xy$lengths,
  default.units = "native"
)
```

---

a5_compaction *Compact and uncompact sets of cells*

Description

a5_compact() stores a set of cells as an a5_covering: every complete group of siblings is replaced by its parent, repeatedly, giving the fewest cells for the same set, and the covering records the resolution the set stands for. a5_uncompact() expands coverings back to their cells at that resolution, or at a finer one. Uncompacting a compacted set returns the original cells, sorted in curve order and without duplicates.

Usage

```
a5_compact(cell, resolution = NULL)

a5_uncompact(x, resolution = NULL, simplify = TRUE)
```

Arguments

cell	An a5_cell vector. NA cells are skipped.
resolution	Integer scalar resolution. For a5_compact(), the resolution the covering stands for: by default that of the finest cell, and never finer than it. For a5_uncompact(), the resolution to expand to: by default each covering's own, and never coarser than it.
x	An a5_covering vector, or an a5_cell vector treated as one covering (see as_a5_covering()).
simplify	Logical scalar. If TRUE (default), return one flat a5_cell vector with the cells of every covering concatenated in input order. If FALSE, return an a5_cell_list with one element per covering, suitable for a list column.

Details

Both work on the index hierarchy (see a5_hierarchy), so the coverage they preserve is the set of index descendants, not a geometric area.

Value

`a5_compact()`: an a5_covering of length one. `a5_uncompact()`: an a5_cell vector, or an a5_cell_list when `simplify = FALSE`, with each covering's cells in curve order. A missing covering contributes no cells (an empty element in the list form).

Memory

A covering can stand for far more cells than it stores: a country at resolution 18 is about a billion cells, stored as a few hundred thousand. `a5_uncompact()` refuses to build more than `.Machine$integer.max` cells; check with a5_covering printing first, and prefer the covering functions that work on compacted sets.

See Also

a5_covering, `a5_polygon_to_cells()` and `a5_grid_disk()`, which return coverings.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 5)
children <- a5_cell_to_children(cell, child_resolution = 7)
covering <- a5_compact(children) # stored as the parent
covering
a5_uncompact(covering)           # the 16 children again

# expand to a finer resolution
a5_uncompact(a5_compact(cell), resolution = 7)

# one covering per element, kept apart with simplify = FALSE
disks <- a5_grid_disk(a5_cell_to_children(cell), k = 1)
a5_uncompact(disks, simplify = FALSE)
```

---

a5_coordinates *Convert between coordinates and cells*

Description

`a5_lonlat_to_cell()` maps longitude/latitude coordinates to the cell containing each point at a resolution. `a5_cell_to_lonlat()` returns the centre point of each cell. Indexing a cell's centre at the cell's own resolution returns the cell.

Usage

```
a5_lonlat_to_cell(lon, lat, resolution)

a5_cell_to_lonlat(cell, as_dataframe = FALSE)
```

Arguments

lon	Numeric vector of longitudes in degrees.
lat	Numeric vector of latitudes in degrees, between -90 and 90. Non-finite coordinates (NA, NaN, Inf) give NA cells.
resolution	Integer scalar or vector of resolutions (0–30), recycled against lon and lat.
cell	An a5_cell vector (or character coercible to one).
as_dataframe	Logical scalar controlling the return container. When FALSE (default), centres are returned as a wk::xy() vector with WGS 84 CRS, the geographic-typed form that plugs into wk/sf pipelines. When TRUE, centres are returned as a base data.frame with columns lon and lat.

Value

- a5_lonlat_to_cell(): an a5_cell vector.
- a5_cell_to_lonlat(): a wk::xy() vector, or a data.frame with columns lon and lat when as_dataframe = TRUE.

See Also

a5_cell_to_boundary() for full cell polygons, a5_polygon_to_cells() for areas.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 5)
a5_cell_to_lonlat(cell)
a5_cell_to_lonlat(cell, as_dataframe = TRUE)

# a cell's centre indexes back to the cell
ll <- a5_cell_to_lonlat(cell, as_dataframe = TRUE)
a5_lonlat_to_cell(ll$lon, ll$lat, resolution = 5) == cell
```

---

a5_covering *A5 covering vectors*

Description

A covering is a set of A5 cells at one resolution, standing for an area of the globe, possibly in disjoint parts: the cells of a polygon, a disk or a cap. It is stored compacted. Complete groups of sibling cells are replaced by their parent, so the stored cells have mixed resolutions, and a trailing *compaction marker* records the resolution they stand for. An a5_covering vector holds one covering per element, so it fits in a data frame column, with one covering per feature or per centre cell.

Usage

```
as_a5_covering(x, ...)

## S3 method for class 'a5_covering'
as_a5_covering(x, ...)

## S3 method for class 'a5_cell'
as_a5_covering(x, resolution = NULL, ...)

## S3 method for class 'a5_cell_list'
as_a5_covering(x, resolution = NULL, ...)

## S3 method for class 'list'
as_a5_covering(x, resolution = NULL, ...)

is_a5_covering(x)

a5_covering_resolution(x)
```

Arguments

x	For `as_a5_covering()`: an a5_cell vector (one covering), an a5_cell_list (one covering per element), or a list of a5_cell vectors or of raw vectors of little-endian 64-bit values, as stored by other A5 libraries. `NULL` list elements are missing coverings. For `a5_covering_resolution()` and `is_a5_covering()`: any object.
...	Unused.
resolution	Integer scalar resolution the coverings stand for, or `NULL` (the default) for the resolution of each input's finest cell. No cell may be finer than `resolution`. Ignored for raw input that carries a compaction marker.

Details

`a5_polygon_to_cells()`, `a5_grid_disk()`, `a5_spherical_cap()` and `a5_compact()` return coverings. `as_a5_covering()` builds them from cells or from raw 64-bit values, sorting and compacting them as needed. `a5_covering_resolution()` returns the resolution each covering stands for, and `a5_uncompact()` expands coverings to their cells.

Value

`as_a5_covering()`: an `a5_covering` vector. `a5_covering_resolution()`: an integer vector, `NA` for a missing covering and -1 for an empty one without a recorded resolution. `is_a5_covering()`: a logical scalar.

Working with coverings

Use coverings through a5R functions where one fits. The stored cells are index parents (see a5_hierarchy): their outlines do not nest geometrically, so drawing them shows overlaps and gaps.

Uncompact a covering before plotting or joining it to data keyed by cell. For index range queries, a5_covering_stored() returns the stored cells.

length() counts coverings, not cells. x[i] and x[[i]] both return coverings, and c() concatenates them. Two coverings of the same cells compare equal with ==, because a5R always stores a covering in the same sorted, compacted form. A missing covering is NA, for example the disk around an NA cell; an empty covering (no cells) is not missing.

Coverings are stored with the compaction marker, as in the other A5 libraries, so a5_covering_to_arrow() writes them to a list<uint64> column that DuckDB, Python and JavaScript read without losing the resolution.

See Also

a5_uncompact(), a5_is_valid(), a5_covering_to_arrow().

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 8)
disk <- a5_grid_disk(cell, k = 3)
disk
a5_covering_resolution(disk)

# from cells: complete sibling groups are stored as their parent
kids <- a5_cell_to_children(cell, child_resolution = 10)
as_a5_covering(kids)
as_a5_covering(kids[-1])
```

---

a5_covering_from_arrow

Convert between a5_covering and Arrow list<uint64> *arrays*

Description

Losslessly convert a5_covering vectors to and from Arrow list<uint64> arrays, one list element per covering. Each element holds the covering's compacted cells followed by its compaction marker, the layout every A5 library uses, so the resolution travels with the data into Parquet, DuckDB, Python or JavaScript. A missing covering is a null element.

Usage

```
a5_covering_from_arrow(x)

a5_covering_to_arrow(x)
```

Arguments

x	For a5_covering_from_arrow(), an Arrow Array or ChunkedArray of type list<uint64>. For a5_covering_to_arrow(), an a5_covering vector.

Details

a5_covering_from_arrow() accepts coverings written by any A5 library. Each one is checked, sorted and compacted; an element without a compaction marker stands for the resolution of its finest cell.

Value

a5_covering_from_arrow() returns an a5_covering vector. a5_covering_to_arrow() returns an Arrow Array of type list<uint64>.

See Also

a5_cell_to_arrow() for single cells.

Examples

```
cells <- a5_lonlat_to_cell(c(-3.19, 135), c(55.95, 0), resolution = 10)
disks <- a5_grid_disk(cells, k = 2)
arr <- a5_covering_to_arrow(disks)
arr$type$ToString()
identical(a5_covering_from_arrow(arr), disks)
```

---

a5_covering_measures *Measure coverings*

Description

a5_count() returns the number of cells each covering stands for at its resolution: the length of its a5_uncompact() result, without building it. This differs from the number of cells stored. a5_area() returns the area of each covering. A5 cells are equal-area, so this is exact up to rounding: the count times a5_cell_area() at the covering's resolution.

Usage

```
a5_count(x)

a5_area(x, units = "m^2")
```

Arguments

x	An a5_covering vector, or an object as_a5_covering() accepts.
units	Character scalar area unit for a5_area() (default "m^2"). Any unit units::set_units() can convert to is accepted (e.g. "km^2", "ha"). If NULL, a plain numeric vector in square metres.

Details

Counts are doubles, exact up to 2^53 (about 9e15 cells); coverings at the finest resolutions can stand for more, and those counts are rounded.

Value

`a5_count()`: a numeric vector the length of `x`. `a5_area()`: a units::units vector the length of `x`, or numeric when `units = NULL`. `NA` for a missing covering.

See Also

`a5_cell_area()` for the area of a single cell.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 12)
cap <- a5_spherical_cap(cell, radius = 10000)
cap
a5_count(cap)
a5_area(cap, units = "km^2")
```

---

a5_covering_predicates

Test coverings

Description

`a5_contains()` tests whether each cell is in a covering, and `a5_overlaps()` whether two coverings share any cell. Both work on the compacted cells, without expanding them, and element by element, recycling a length-one argument.

Usage

```
a5_contains(x, cell)

a5_overlaps(x, y)
```

Arguments

`x`	An a5_covering vector, or an object `as_a5_covering()` accepts.
`cell`	An a5_cell vector, each at the resolution of its covering.
`y`	An a5_covering vector, at the same resolution as `x`.

Details

a5_contains() is a point-in-polygon test once the point is indexed at the covering's resolution: a5_contains(covering, a5_lonlat_to_cell(lon, lat, a5_covering_resolution(covering))). The cell must be at the covering's resolution, or it is an error: A5 cells do not nest geometrically, so a finer cell is not guaranteed to lie inside the coarser cell that holds it in the index. An empty covering contains no cell.

Value

A logical vector of the common size of the inputs. NA where either input is NA.

See Also

a5_set_operations.

Examples

```
poly <- wk::wkt("POLYGON ((-3.4 55.85, -3 55.85, -3 56.05, -3.4 56.05, -3.4 55.85))")
edinburgh <- a5_polygon_to_cells(poly, resolution = 12)

# point in polygon, at the covering's resolution
points <- a5_lonlat_to_cell(c(-3.19, -4.25), c(55.95, 55.86), resolution = 12)
a5_contains(edinburgh, points)

cap <- a5_spherical_cap(points, radius = 5000)
a5_overlaps(edinburgh, cap)
```

---

a5_covering_stored *Stored cells of a covering*

Description

Returns the cells a covering stores: its compacted cells, which have mixed resolutions, in curve order and without the compaction marker. Use them for index range queries. Each stored cell's descendants at the covering's resolution form one contiguous id range, given by a5_cell_children_range(), so a covering is the union of those ranges.

Usage

```
a5_covering_stored(x, simplify = TRUE)
```

Arguments

x	An a5_covering vector, or anything as_a5_covering() accepts.
simplify	Logical scalar. If TRUE (default), return one flat a5_cell vector with the stored cells of every covering in turn. If FALSE, return an a5_cell_list with one element per covering; a missing covering gives an empty element.

Details

The stored cells are index parents, not geometric ones: their outlines do not nest, so do not draw them or use them as an area. Use `a5_uncompact()` for the cells a covering stands for.

Value

An a5_cell vector, or an a5_cell_list if `simplify = FALSE`.

See Also

`a5_uncompact()` for the cells at the covering's resolution, `a5_covering_resolution()`, `a5_cell_children_range()`.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 8)
disk <- a5_grid_disk(cell, k = 3)
stored <- a5_covering_stored(disk)
a5_get_resolution(stored)

# one id range per stored cell, at the covering's resolution
a5_cell_children_range(stored, child_resolution = a5_covering_resolution(disk))
```

---

a5_covering_to_supercells

Supercells of a covering

Description

`a5_covering_to_supercells()` gives, for each covering, the cells at a coarser `resolution` that contain the centre of at least one of its cells, such as the partitions of a store that hold an area's data. It matches `a5_cell_to_supercell(a5_uncompact(x), resolution)` without duplicates, but never builds the covering's cells.

Usage

```
a5_covering_to_supercells(x, resolution)
```

Arguments

`x`	An a5_covering vector.
`resolution`	Integer scalar target resolution, at or coarser than each covering's resolution.

Value

An a5_covering vector at `resolution`, the same length as `x`. `NA` where `x` is `NA`.

See Also

`a5_cell_to_supercell()`

Examples

```
area <- a5_polygon_to_cells(
  wk::wkt("POLYGON ((-3.3 55.9, -3.1 55.9, -3.1 56, -3.3 56, -3.3 55.9))"),
  resolution = 14
)
a5_uncompact(a5_covering_to_supercells(area, 6))
```

---

a5_errors *Error classes*

Description

Errors raised by a5R have class `a5R_error`, so code can catch them apart from other errors, and most also have a class naming what went wrong:

Details

- `a5R_error_resolution`: a resolution that is missing, out of range, or on the wrong side of a cell's own resolution.
- `a5R_error_geometry`: coordinates or geometries that cannot be indexed, such as latitudes beyond 90 degrees, projected coordinates, or malformed rings.
- `a5R_error_cell`: input that cannot be read as cells or coverings, such as a string that is not a hex id.
- `a5R_error_arrow`: Arrow input of the wrong type.
- `a5R_error_size`: a result too large to build.
- `a5R_error_argument`: any other invalid argument.

Errors from the a5 Rust library carry only `a5R_error` unless a5R can tell what they concern.

Examples

```
res <- tryCatch(
  a5_lonlat_to_cell(0, 91, resolution = 5),
  a5R_error_geometry = function(e) "bad coordinates"
)
res
```

a5_get_res0_cells *Get all resolution-0 root cells*

Description

Returns the 12 root cells corresponding to the 12 faces of the dodecahedron.

Usage

```
a5_get_res0_cells()
```

Value

An a5_cell vector of length 12.

Examples

```
a5_get_res0_cells()
```

a5_get_resolution *Get the resolution of A5 cell indices*

Description

Extracts the resolution level (0–30) encoded in each cell index.

Usage

```
a5_get_resolution(cell)
```

Arguments

cell	An a5_cell vector.

Value

An integer vector of resolutions.

See Also

a5_cell_to_parent(), a5_cell_to_children()

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 10)
a5_get_resolution(cell)
```

a5_grid_disk *Cells within k hops of a cell*

Description

Returns, for each centre cell, the cells reachable within k hops, including the centre cell itself, as an a5_covering. a5_grid_disk() steps between cells that share an edge; a5_grid_disk_vertex() also steps between cells that share only a vertex, so it reaches more cells for the same k.

Usage

```
a5_grid_disk(cell, k)

a5_grid_disk_vertex(cell, k)
```

Arguments

cell	An a5_cell vector of centre cells.
k	Non-negative integer scalar, number of hops.

Value

An a5_covering vector the same length as cell, at the resolution of each centre cell. NA where cell is NA. Use a5_uncompact() for the cells.

See Also

a5_spherical_cap() for distance-based selection.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 8)
disk <- a5_grid_disk(cell, k = 1)
disk
a5_uncompact(disk)
a5_grid_disk_vertex(cell, k = 1)

cells <- a5_lonlat_to_cell(c(-3.19, 0), c(55.95, 0), resolution = 5)
a5_grid_disk(cells, k = 2) # one covering per cell
```

a5_hierarchy *Index parents and children*

Description

`a5_cell_to_parent()` returns the ancestor of each cell at a coarser resolution. `a5_cell_to_children()` returns the descendants of each cell at a finer resolution: 4 per resolution step, or 5 from resolution 0. Both follow the cell index, so they are cheap bit operations.

Usage

```
a5_cell_to_parent(cell, parent_resolution = NULL, resolution)

a5_cell_to_children(cell, child_resolution = NULL, simplify = TRUE, resolution)
```

Arguments

`cell`	An a5_cell vector.
`parent_resolution`	Integer scalar target resolution for `a5_cell_to_parent()`, or `NULL` (default) for the immediate parent. It must not be finer than any cell's resolution; `-1` gives the world cell.
`resolution`	Deprecated: renamed `parent_resolution` and `child_resolution`, as in the other A5 libraries.
`child_resolution`	Integer scalar target resolution for `a5_cell_to_children()`, or `NULL` (default) for the immediate children.
`simplify`	Logical scalar. If `TRUE` (default), return one flat a5_cell vector with the children of every input concatenated in input order; which child came from which parent is not recorded. If `FALSE`, return an a5_cell_list: a `vctrs::list_of()` of a5_cell vectors with one element per input, suitable for a list column.

Value

- `a5_cell_to_parent()`: an a5_cell vector the same length as `cell`.
- `a5_cell_to_children()`: an a5_cell vector, or an a5_cell_list when `simplify = FALSE`. An `NA` input contributes no cells (an empty element in the list form).

Index hierarchy is not spatially nested

A5 parents and children are defined by the cell index, not by geometry. A cell's index children do not tile it: their union overlaps the parent's neighbours. Measured over random cells, the index parent is not the coarse cell containing a fine cell's centre for about half of fine cells one resolution apart, falling to about 35 percent at four or more resolutions apart. Use these functions for operations on ids (compaction, id ranges, curve ordering) and a5_spatial_hierarchy when cells must nest by location.

See Also

a5_spatial_hierarchy for parents and children by location, `a5_cell_child()` for one child at a time, `a5_cell_children_range()` for the id range of all descendants, `a5_get_resolution()`.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 10)
a5_cell_to_parent(cell)
a5_cell_to_parent(cell, parent_resolution = 5)

a5_cell_to_children(cell)
cells <- a5_lonlat_to_cell(c(-3.19, 0), c(55.95, 0), resolution = 5)
a5_cell_to_children(cells, child_resolution = 7)                  # 32 cells
a5_cell_to_children(cells, child_resolution = 7, simplify = FALSE) # list of 2

# children map back to their parent
all(a5_cell_to_parent(a5_cell_to_children(cell, 12), 10) == cell)
```

`a5_info` *a5R build and session information*

Description

Returns the versions behind a5R and its current settings, as a named list that prints as a summary. The same summary, on one line, is shown when the package is attached with `library()`.

Usage

```
a5_info()
```

Details

The `index` element names the A5 cell index that a5R's ids belong to. Ids from different index versions look alike, and nothing in an id says which version wrote it (see `a5_migrate()`), so record `index` with stored cells and check it when reading them back. With Arrow and Parquet, `a5_to_arrow_table()` stores it in the schema metadata under the key `"a5_index"`, and `a5_from_arrow_table()` checks it.

Value

A list of class `a5_info` with elements:

- `version`: the a5R package version, a character scalar.
- `crate`: the version of the bundled a5 Rust crate.
- `index`: the A5 cell index version of a5R's ids, `"1.0"`.
- `threads`: the current thread count (see `a5_threads()`).
- `c_api`: the version of the C API (see a5_c_api).

Examples

```
a5_info()
a5_info()$index
```

`a5_is_valid` *Check that values are A5 cells*

Description

`a5_is_valid_cell()` checks each value against the A5 index encoding: its resolution bits must name a resolution, and its face or quintant must exist. Use it on ids from outside sources, such as a file or a database, before passing them to other functions. `a5_cell()` accepts any 16-digit hex string, so an `a5_cell` vector can hold values that are not cells.

Usage

```
a5_is_valid(x)

a5_is_valid_cell(x)
```

Arguments

`x`	An a5_cell vector or a character vector of hex strings. For `a5_is_valid()`, also an a5_cell_list or an a5_covering.

Details

`a5_is_valid()` checks any a5R object, returning one value per element:

- an a5_cell vector or hex strings: whether each value is a cell, as `a5_is_valid_cell()`.
- an a5_cell_list: whether every cell in each element is valid. An empty element is valid.
- an a5_covering: whether each covering is well formed: every stored value but a trailing compaction marker is a valid cell, no cell is finer than the covering's resolution, and the cells are sorted in curve order without overlapping. a5R always builds coverings this way; the check is for coverings assembled by hand, as the covering functions rely on it. A covering need not be fully compacted to be valid.

The world cell (id 0, resolution -1) is valid: it has no boundary, but `a5_cell_to_parent()` returns it as the parent of the resolution-0 cells. Validity says nothing about which index version an id was written with: ids from a5R before 1.0 are valid cells in a different place. See `a5_migrate()`.

Value

A logical vector the length of `x`. `NA` where `x` is `NA`; for an a5_cell_list, `NA` where an element holds `NA` and no invalid cell; for an a5_covering, `NA` for a missing covering. Hex strings are read as `a5_cell()` reads them, so ids of fewer than 16 digits and a `0x` prefix are accepted; anything else is `FALSE`.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 10)
a5_is_valid_cell(cell)
a5_is_valid_cell(c(a5_u64_to_hex(cell), "fd00000000000000", "not_a_cell", NA))

a5_is_valid(a5_cell_to_children(cell, simplify = FALSE))
```

---

a5_line_string_to_cells

Cells traced by a great-circle linestring

Description

Returns, for each linestring feature, the A5 cells at `resolution` whose pentagons the great-circle path through its waypoints crosses. Each path's cells are uncompacted and in order along it, with duplicates removed first-seen; a `MULTILINESTRING` feature's parts are traced in order into one path.

Usage

```
a5_line_string_to_cells(x, resolution, simplify = TRUE)
```

Arguments

`x`	A linestring-like geometry. One of: • Any geometry handleable by `wk::wk_handle()` containing one or more `LINESTRING` / `MULTILINESTRING` features. • A `SpatVector` of linestrings (requires the `terra` package). • A two-column numeric matrix (`cbind(lon, lat)`) of waypoints. • A `data.frame` with columns `lon` and `lat`.
`resolution`	Integer scalar target resolution (0–30).
`simplify`	Logical scalar. If `TRUE` (default), return one flat a5_cell vector with the paths of every feature concatenated in feature order; a cell crossed by several features appears once per feature. If `FALSE`, return an a5_cell_list with one path per feature, suitable for a list column.

Details

Consecutive waypoints are connected by great-circle arcs (not rhumb lines or planar segments), so antimeridian-crossing paths work transparently when written in unwrapped lon/lat. A line string with a single waypoint gives the cell containing it.

A path is an ordered sequence, so the result is a vector of cells rather than an a5_covering; use `as_a5_covering()` to treat the cells a path crosses as a set.

Matrix and `data.frame` inputs are treated as a single linestring; for multi-feature data, pass an `sf`, `sfc`, wk, or `SpatVector` geometry instead.

Value

An a5_cell vector at `resolution`, or an a5_cell_list when `simplify = FALSE`. A missing or empty geometry contributes no cells (an empty element in the list form).

See Also

`a5_polygon_to_cells()`, `a5_grid_disk()`.

Examples

```
line <- wk::wkt("LINESTRING (2.35 48.86, -0.13 51.51)")
cells <- a5_line_string_to_cells(line, resolution = 6)
cells

# One path per feature
lines <- wk::wkt(c(
  "LINESTRING (2.35 48.86, -0.13 51.51)",
  "LINESTRING (2.35 48.86, 4.83 45.76)"
))
paths <- a5_line_string_to_cells(lines, resolution = 6, simplify = FALSE)
lengths(paths)
```

---

a5_migrate *Migrate cell ids from the pre-1.0 index*

Description

Converts cell ids written by a5R before 1.0 (the a5 Rust crate 0.10 and earlier, the "v0" index) to the current, permanent index. A5 1.0 threads its space-filling curve through the cells differently, so cells keep their shape and position but get new ids. The conversion is a cheap bit operation; there is no need to re-index data from its coordinates. The exception is resolution 1, whose 60 cells changed from triangles to pentagons: each migrated resolution-1 id is the new cell containing the old cell's centre.

Usage

```
a5_migrate(cell)
```

Arguments

`cell`	An a5_cell vector of ids in the v0 index.

Details

A v0 id is still a valid cell in the current index, but in a different place, and nothing in the id records which version wrote it. Migrate stored ids exactly once, as soon as they are read: migrating current ids, or migrating twice, gives wrong cells without any error.

Upstream will retire this conversion in a future version.

Value

An a5_cell vector of the same cells in the current index, the same length as `cell`. `NA` where `cell` is `NA`.

See Also

`a5_cell_from_arrow()` to read stored ids.

Examples

```
# A resolution-8 cell id from a5R 0.6.0
v0 <- a5_cell("59c4780000000000")
a5_migrate(v0)
```

`a5_polygon_to_cells` *Cells inside or overlapping a polygon*

Description

Returns, for each polygon feature, an a5_covering of the A5 cells at `resolution` that belong to it. By default a cell belongs when its centre lies inside the polygon; set `containment = "overlapping"` to also include every cell that touches the polygon boundary, giving gap-free coverage.

Usage

```
a5_polygon_to_cells(x, resolution, containment = c("centre", "overlapping"))
```

Arguments

`x`	A polygon-like geometry. One of: • Any geometry handleable by `wk::wk_handle()` (e.g. `wk::wkt()`, `wk::wkb()`, `wk::rct()`, `sf`, `sfc`) containing one or more `POLYGON` / `MULTIPOLYGON` features. • A `SpatVector` of polygons (requires the `terra` package). • A two-column numeric matrix (`cbind(lon, lat)`) of vertices, interpreted as a single outer ring. • A `data.frame` with columns `lon` and `lat`, interpreted as a single outer ring.
`resolution`	Integer scalar target resolution (0-30).
`containment`	Character scalar selecting which cells to include. `"centre"` (the default; `"center"` also works) includes a cell when its centre lies inside the polygon. `"overlapping"` additionally includes every cell that overlaps the polygon boundary, so the result fully covers the polygon. The overlapping set is a superset of the centre set.

Details

The result has one covering per feature, so it can be added as a column: `sf_data$cells <- a5_polygon_to_cells(sf_data, 10)`. A feature's parts (a `MULTIPOLYGON`) and holes are combined into its covering, with hole interiors excluded. To combine features into one covering, use `a5_union()`. Coverings are compacted; use `a5_uncompact()` for the cells at `resolution`.

With `containment = "centre"`, membership is determined by **centre-point containment**: a cell is included if its centroid lies inside the polygon, with hole interiors excluded. Cells straddling the boundary whose centre falls outside are dropped, so the union of the cells does not fully cover the polygon.

With `containment = "overlapping"`, every cell that contains any point of the polygon boundary is kept as well, so the returned cells cover the polygon without gaps. Hole boundaries count as boundary, so cells straddling a hole edge are included while the hole interior is still excluded.

Coordinates must be WGS 84 longitude/latitude in degrees. Rings are closed automatically; a trailing duplicate vertex is dropped if present. An outer ring with fewer than 3 distinct vertices gives an empty covering, and a hole with fewer than 3 is ignored, as in the other A5 libraries.

Polygon edges are great-circle arcs, the shortest path on the sphere between consecutive vertices, as in the other A5 libraries and in s2. An edge between two points on the same parallel therefore bows toward the pole: a 20 degree edge along 10 degrees north peaks at 10.15 degrees north, and a 1 degree edge at that latitude bows by about 40 m. An edge whose ends are antipodal has no single great circle and gives an undefined result. Where edges should follow parallels, as in GeoJSON (RFC 7946 defines its edges as straight in longitude and latitude), add vertices along them first.

A `wk::rct()` box is a polygon with great-circle edges like any other, with two exceptions: a box covering the whole globe (`wk::rct(-180, -90, 180, 90)`) gives every cell, and a box reaching from pole to pole keeps its west and east edges on their meridians. Any other box spanning 180 degrees of longitude or more is an error, because its top and bottom edges would run the short way round the globe.

Rings must not cross themselves: a ring with two edges that cross, such as a bow tie, is an error of class `a5R_error_geometry`. Repair such rings first, for example with `sf::st_make_valid()`. Because edges are great circles, a ring that is valid in longitude and latitude can still cross itself on the sphere; `s2::s2_is_valid()` checks rings with the same edge model. Rings that only touch themselves, and holes that cross the outer ring, are not checked.

Where no A5 cell centroids at the specified `resolution` fall within the geometry, its covering is empty.

Matrix and `data.frame` inputs are treated as a single ring; for multi-feature data or polygons with holes, pass an `sf`, `sfc`, wk, or `SpatVector` geometry instead.

Value

An a5_covering vector with one element per feature (one for matrix and `data.frame` input). `NA` for a missing geometry; an empty covering for an empty geometry or one too small to hold a cell centre.

See Also

`a5_line_string_to_cells()`, `a5_uncompact()`.

Examples

```
poly <- wk::wkt(
  "POLYGON ((-3.3 55.9, -3.1 55.9, -3.1 56, -3.3 56, -3.3 55.9))"
)
covering <- a5_polygon_to_cells(poly, resolution = 8)
covering
a5_uncompact(covering)

# Gap-free coverage: every cell touching the polygon
a5_polygon_to_cells(poly, resolution = 8, containment = "overlapping")

# One covering per feature
polys <- wk::wkt(c(
  "POLYGON ((-3.3 55.9, -3.1 55.9, -3.1 56, -3.3 56, -3.3 55.9))",
  NA,
  "POLYGON ((-4.3 55.8, -4.2 55.8, -4.2 55.9, -4.3 55.9, -4.3 55.8))"
))
a5_polygon_to_cells(polys, resolution = 8)
```

---

a5_resolution_stats *Cell size and counts by resolution*

Description

Properties of the grid at a resolution, independent of any particular cell:

Usage

```
a5_cell_area(resolution, units = "m^2")

a5_cell_edge_length_avg(resolution, units = "m")

a5_get_num_cells(resolution)

a5_get_num_children(parent_resolution, child_resolution)
```

Arguments

`resolution`	Integer vector of resolutions (0–30). A scalar for `a5_get_num_cells()`.
`units`	Character scalar giving the output unit: an area unit for `a5_cell_area()` (default `"m^2"`), a length unit for `a5_cell_edge_length_avg()` (default `"m"`). Any unit `units::set_units()` can convert to is accepted (e.g. `"km^2"`, `"ha"`, `"km"`, `"mi"`). If `NULL`, a plain numeric vector in square metres or metres is returned.
`parent_resolution, child_resolution`	Integer scalars (0–30), with `child_resolution` at or finer than `parent_resolution`.

Details

- `a5_cell_area()`: the area of one cell. A5 is an equal-area grid, so every cell at a resolution has the same area.
- `a5_cell_edge_length_avg()`: the average cell edge length. Individual edges vary from this by roughly +/-10%, depending on the cell's shape and position; use `a5_cell_to_boundary()` to measure a specific cell.
- `a5_get_num_cells()`: the number of cells covering the globe.
- `a5_get_num_children()`: the number of descendants each cell has at a finer resolution, as returned by `a5_cell_to_children()`.

Use the first two to choose a resolution for a target cell size.

Value

- `a5_cell_area()`, `a5_cell_edge_length_avg()`: a units::units vector the length of `resolution`, or numeric when `units = NULL`.
- `a5_get_num_cells()`, `a5_get_num_children()`: a numeric scalar. Counts are doubles because they can exceed R's integer range.

Examples

```
a5_cell_area(0:5)
a5_cell_area(5, units = "km^2")
a5_cell_edge_length_avg(10, units = "km")

a5_get_num_cells(10)
a5_get_num_children(5, 8) # 4^3 = 64

# cells at a resolution times their area covers the globe
a5_get_num_cells(10) * a5_cell_area(10, units = "km^2")
```

---

a5_set_operations *Combine coverings*

Description

Set operations on a5_covering vectors, computed on the compacted cells without expanding them, so they stay fast for coverings that stand for billions of cells:

Usage

```
a5_union(x, y = NULL)

a5_intersect(x, y)

a5_difference(x, y)
```

Arguments

x, y	a5_covering vectors, or objects as_a5_covering() accepts. Sizes must be compatible under vctrs recycling rules.

Details

- a5_union(): cells in either covering. With y omitted, the union of every covering in x, as one covering: for example the coverings of all features from a5_polygon_to_cells().
- a5_intersect(): cells in both coverings.
- a5_difference(): cells in x but not in y.

The operations work element by element, recycling a length-one argument, so a5_intersect(countries, cap) intersects every country with one cap. Both coverings of a pair must be at the same resolution: A5 cells do not nest geometrically across resolutions, so combining sets at different resolutions has no meaning, and it is an error. An empty covering with no recorded resolution, as a5_compact() returns for no cells, combines with any covering as the empty set.

Value

An a5_covering vector of the common size of x and y (size one for a5_union(x)). NA where either input is NA; a5_union(x) skips NA elements.

See Also

a5_contains(), a5_overlaps() and a5_count() to query coverings.

Examples

```
edinburgh <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 12)
glasgow <- a5_lonlat_to_cell(-4.25, 55.86, resolution = 12)
caps <- a5_spherical_cap(c(edinburgh, glasgow), radius = 40000)

a5_union(caps[1], caps[2])
a5_intersect(caps[1], caps[2])
a5_difference(caps[1], caps[2])

# the union of every element
a5_union(caps)
```

---

a5_spatial_hierarchy *Supercells and subcells: parents and children by location*

Description

a5_cell_to_supercell() returns, for each cell, the cell at a coarser resolution that contains its centre. a5_cell_to_subcells() returns, for each cell, the cells at a finer resolution whose centres lie inside it, as an a5_covering. These are the location-based counterparts of a5_cell_to_parent() and a5_cell_to_children(), which follow the index and often cross into neighbouring cells (see a5_hierarchy).

Usage

```
a5_cell_to_supercell(cell, resolution = NULL)

a5_cell_to_subcells(cell, resolution = NULL)
```

Arguments

`cell`	An a5_cell vector.
`resolution`	Integer scalar target resolution, or `NULL` for one step: one resolution coarser for `a5_cell_to_supercell()`, one finer for `a5_cell_to_subcells()`. Subcells require a resolution at or finer than every cell's own.

Details

The two functions define an exact partition. Every fine cell has exactly one supercell at each coarser resolution, and the subcells of a coarse cell are exactly the fine cells whose supercell it is, so the subcells of cells that tile an area tile it too, with no cell counted twice. A centre on a shared edge goes to the cell `a5_lonlat_to_cell()` assigns it to. Aggregating fine values by supercell therefore attributes each value to the coarse cell it lies in.

Value

- `a5_cell_to_supercell()`: an a5_cell vector the same length as `cell`. `NA` where `cell` is `NA`, where `resolution` is finer than the cell, or for a resolution-0 cell with `resolution = NULL`. A cell at `resolution` is returned unchanged.
- `a5_cell_to_subcells()`: an a5_covering vector the same length as `cell`. `NA` where `cell` is `NA`.

Supercell

The result equals `a5_lonlat_to_cell(a5_cell_to_lonlat(cell), resolution)`, computed without converting the centre to degrees. It takes about half the time of `a5_lonlat_to_cell()`, but far longer than `a5_cell_to_parent()`, which is a bit operation.

Subcells

Counts vary from cell to cell and average `4^d` for a resolution difference `d`, or `5 * 4^(d - 1)` from resolution 0. The covering is compacted, so it stores far fewer cells than it stands for at large differences; use `a5_uncompact()` for the cells, and never draw the stored cells, which are index parents whose outlines extend beyond the coarse cell. Resolution 30 covers only part of the globe, so a cell whose subcells reach past it has all its subcells given at resolution 29.

See Also

a5_hierarchy for parents and children by index, `vignette("spatial-hierarchy")`.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 18)
a5_cell_to_supercell(cell, resolution = 15)
a5_cell_to_parent(cell, parent_resolution = 15) # may differ

coarse <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 10)
sub <- a5_cell_to_subcells(coarse, resolution = 12)
sub
kids <- a5_uncompact(sub)
all(a5_cell_to_supercell(kids, resolution = 10) == coarse)
```

---

a5_spherical_cap *Cells within a great-circle radius*

Description

Returns, for each centre cell, the cells whose centres fall within a great-circle distance of its centre, as an a5_covering.

Usage

```
a5_spherical_cap(cell, radius, units = "m")
```

Arguments

`cell`	An a5_cell vector of centre cells.
`radius`	Non-negative numeric scalar, great-circle radius in `units`, or a units object of length. It is measured on the sphere A5 uses internally, so membership can differ by a fraction of a percent of `radius` from distances given by `a5_cell_distance()`.
`units`	Character scalar, the length unit of a plain numeric `radius`: `"m"` (the default) or any unit convertible from metres, such as `"km"`.

Value

An a5_covering vector the same length as `cell`, at the resolution of each centre cell. `NA` where `cell` is `NA`. Use `a5_uncompact()` for the cells.

See Also

`a5_grid_disk()` for hop-based selection.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 8)
cap <- a5_spherical_cap(cell, radius = 1000)
cap
a5_uncompact(cap)
```

a5_threads *Get or set the number of threads used by a5R*

Description

Controls the number of threads used for parallel processing in vectorised functions. One thread (the default) runs sequentially with no overhead; more run in parallel via rayon. As with `options()` and `par()`, `a5_threads()` reports the current count and `a5_threads(n)` sets it.

Usage

```
a5_threads(n = NULL)

## S3 method for class 'a5_threads'
with(data, expr, ...)
```

Arguments

`n`	`NULL` (the default) to report the current count, or an integer scalar of at least 1 to set it. The value returned by an earlier `a5_threads(n)` call also works.
`data`	The value of `a5_threads(n)`.
`expr`	Code to run with that thread count.
`...`	Unused.

Details

`a5_threads(n)` returns the previous thread count, so a change can be scoped to a block of code with `with()`, as `mirai::daemons()` is: `with(a5_threads(4), expr)` runs `expr` on four threads and then restores the previous count, even if `expr` fails.

The count can also be set before the package loads, with the option `a5R.threads` or the environment variable `A5R_NUM_THREADS`.

A forked worker, as created by `parallel::mclapply()` or `future::multicore`, inherits the count but not the threads, so a5R runs on one thread there until the worker calls `a5_threads()` itself. Separate-process workers, as from `mirai` or `future::multisession`, start at one thread like any new session.

Value

`a5_threads()`: the current count, an integer scalar. `a5_threads(n)`: the previous count, invisibly, as an `a5_threads` object. `with()`: the value of `expr`.

Examples

```
a5_threads()

cells <- a5_lonlat_to_cell(runif(1000, -180, 180), runif(1000, -80, 80), 10)
b <- with(a5_threads(2), a5_cell_to_boundary(cells))
a5_threads()
```

---

a5_to_arrow_table *Convert data frames with cell columns to and from Arrow tables*

Description

`a5_to_arrow_table()` converts a data frame to an Arrow `Table`, writing a5_cell columns as `uint64` and a5_covering columns as `list<uint64>` (see `a5_cell_to_arrow()` and `a5_covering_to_arrow()`), and records the A5 index version (`a5_info()$index`) in the schema metadata under `"a5_index"`. `a5_from_arrow_table()` reverses it, turning `uint64` columns into cells and `list<uint64>` columns into coverings.

Usage

```
a5_to_arrow_table(x)

a5_from_arrow_table(x, cells = NULL, coverings = NULL)
```

Arguments

`x`	For `a5_to_arrow_table()`, a data frame. For `a5_from_arrow_table()`, an Arrow `Table`, `RecordBatch`, or anything `arrow::as_arrow_table()` accepts, such as a query result.
`cells, coverings`	Names of the columns to read as cells and as coverings, or `NULL` (the default) for every `uint64` and every `list<uint64>` column respectively.

Details

Ids stay exact throughout, never passing through doubles or text, so the pair suits Parquet files and DuckDB queries. A table whose metadata records a different index version is an error, since its ids would name other cells; convert such ids with `a5_migrate()`.

Value

`a5_to_arrow_table()`: an Arrow `Table`. `a5_from_arrow_table()`: a data frame.

See Also

`a5_info()`

Examples

```
cells <- a5_lonlat_to_cell(c(-3.19, 135), c(55.95, 0), resolution = 10)
df <- data.frame(cell = cells, value = c(1.5, 2.5))
df$disk <- a5_grid_disk(cells, k = 1)
tbl <- a5_to_arrow_table(df)
tbl$schema
back <- a5_from_arrow_table(tbl)
identical(back$cell, df$cell)
```

---

a5_u64_to_hex *Coerce between hex strings and A5 cell vectors*

Description

`a5_u64_to_hex()` converts an a5_cell vector to 16-character zero-padded hex strings. `a5_hex_to_u64()` converts hex strings to an a5_cell vector.

Usage

```
a5_u64_to_hex(x)

a5_hex_to_u64(x)
```

Arguments

`x`	For `a5_u64_to_hex()`, an a5_cell vector (or object coercible to one). For `a5_hex_to_u64()`, a character vector of hex-encoded cell IDs.

Details

These are named to match `u64_to_hex` / `hex_to_u64` in the upstream Python, JavaScript, and DuckDB A5 bindings. In those languages the functions convert between a native 64-bit unsigned integer and its hex representation. Because R has no native `uint64` type, `a5_u64_to_hex()` accepts an a5_cell (which stores the `u64` internally as eight raw bytes) instead of a bare integer.

Value

`a5_u64_to_hex()` returns a character vector. `a5_hex_to_u64()` returns an a5_cell vector.

See Also

`a5_cell_from_arrow()` and `a5_cell_to_arrow()` for lossless conversion between a5_cell and Arrow `uint64` arrays.

Examples

```
cell <- a5_lonlat_to_cell(-3.19, 55.95, resolution = 5)
hex <- a5_u64_to_hex(cell)
hex

a5_hex_to_u64(hex)
```

`wk_methods` *wk methods for a5_cell*

Description

Integration with the wk geometry framework. Allows `a5_cell` vectors to be handled as geometry (via their boundary polygons) and to report their CRS.

Usage

```
## S3 method for class 'a5_cell'
wk_handle(handleable, handler, ...)

## S3 method for class 'a5_cell'
wk_crs(x)
```

Arguments

`handleable, x`	An a5_cell vector.
`handler`	A wk handler.
`...`	Passed to underlying methods.

Value

- `wk_handle()`: the result of the handler.
- `wk_crs()`: a wk::wk_crs object (WGS 84 lon/lat).

Index

a5_area *(a5_covering_measures)*, 17
a5_bbox, 3
a5_c_api, *25*
a5_cell, 4, *5–15*, *18–20*, *22–24*, *26–29*, *34, 35*, *37–39*
a5_cell(), *26*
a5_cell_area *(a5_resolution_stats)*, 31
a5_cell_area(), *7*, *17, 18*
a5_cell_child, 5
a5_cell_child(), *25*
a5_cell_children_range, 5
a5_cell_children_range(), *19, 20*, *25*
a5_cell_distance, 7
a5_cell_distance(), *35*
a5_cell_edge_length_avg *(a5_resolution_stats)*, 31
a5_cell_from_arrow, 8
a5_cell_from_arrow(), *29*, *38*
a5_cell_list, 9, *12, 13*, *15*, *19, 20*, *24*, *26–28*
a5_cell_to_arrow *(a5_cell_from_arrow)*, 8
a5_cell_to_arrow(), *6*, *17*, *37, 38*
a5_cell_to_boundary, 10
a5_cell_to_boundary(), *11, 12*, *14*, *32*
a5_cell_to_boundary_xy, 11
a5_cell_to_boundary_xy(), *10*
a5_cell_to_children *(a5_hierarchy)*, 24
a5_cell_to_children(), *5, 6*, *22*, *32, 33*
a5_cell_to_lonlat *(a5_coordinates)*, 13
a5_cell_to_lonlat(), *7*, *10*
a5_cell_to_parent *(a5_hierarchy)*, 24
a5_cell_to_parent(), *22*, *26*, *33, 34*
a5_cell_to_subcells *(a5_spatial_hierarchy)*, 33
a5_cell_to_supercell *(a5_spatial_hierarchy)*, 33
a5_cell_to_supercell(), *20*
a5_compact *(a5_compaction)*, 12
a5_compact(), *15*
a5_compaction, 12
a5_contains *(a5_covering_predicates)*, 18
a5_contains(), *33*
a5_coordinates, 13
a5_count *(a5_covering_measures)*, 17
a5_count(), *33*
a5_covering, *12, 13*, 14, *16–20*, *23*, *26, 27*, *29, 30*, *32–35*, *37*
a5_covering_from_arrow, 16
a5_covering_measures, 17
a5_covering_predicates, 18
a5_covering_resolution *(a5_covering)*, 14
a5_covering_resolution(), *20*
a5_covering_stored, 19
a5_covering_stored(), *16*
a5_covering_to_arrow *(a5_covering_from_arrow)*, 16
a5_covering_to_arrow(), *16*, *37*
a5_covering_to_supercells, 20
a5_difference *(a5_set_operations)*, 32
a5_errors, 21
a5_from_arrow_table *(a5_to_arrow_table)*, 37
a5_from_arrow_table(), *25*
a5_get_num_cells *(a5_resolution_stats)*, 31
a5_get_num_children *(a5_resolution_stats)*, 31
a5_get_num_children(), *5*
a5_get_res0_cells, 22
a5_get_resolution, 22
a5_get_resolution(), *25*
a5_grid_disk, 23
a5_grid_disk(), *13*, *15*, *28*, *35*
a5_grid_disk_vertex *(a5_grid_disk)*, 23
a5_hex_to_u64 *(a5_u64_to_hex)*, 38
a5_hierarchy, *13*, *15*, 24, *33, 34*
a5_info, 25
a5_info(), *37*
a5_intersect *(a5_set_operations)*, 32

a5_is_valid, 26
a5_is_valid(), *4*, *16*
a5_is_valid_cell *(*a5_is_valid*)*, 26
a5_is_valid_cell(), *4*
a5_line_string_to_cells, 27
a5_line_string_to_cells(), *30*
a5_lonlat_to_cell *(*a5_coordinates*)*, 13
a5_lonlat_to_cell(), *34*
a5_migrate, 28
a5_migrate(), *4*, *25, 26*, *37*
a5_overlaps *(*a5_covering_predicates*)*, 18
a5_overlaps(), *33*
a5_polygon_to_cells, 29
a5_polygon_to_cells(), *3*, *13–15*, *28*, *33*
a5_resolution_stats, 31
a5_set_operations, *19*, 32
a5_spatial_hierarchy, *24, 25*, 33
a5_spherical_cap, 35
a5_spherical_cap(), *15*, *23*
a5_threads, 36
a5_threads(), *25*
a5_to_arrow_table, 37
a5_to_arrow_table(), *25*
a5_u64_to_hex, 38
a5_u64_to_hex(), *4*, *8*
a5_uncompact *(*a5_compaction*)*, 12
a5_uncompact(), *15–17*, *20*, *23*, *30*, *34, 35*
a5_union *(*a5_set_operations*)*, 32
a5_union(), *30*
arrow::as_arrow_table(), *37*
as_a5_cell *(*a5_cell*)*, 4
as_a5_cell_list *(*a5_cell_list*)*, 9
as_a5_covering *(*a5_covering*)*, 14
as_a5_covering(), *19*, *27*

grid::polygonGrob(), *11*

is_a5_cell *(*a5_cell*)*, 4
is_a5_covering *(*a5_covering*)*, 14

library(), *25*

options(), *36*

par(), *36*
parallel::mclapply(), *36*

units, *35*
units::set_units(), *7*, *17*, *31*
units::units, *7*, *18*, *32*

unlist.a5_cell_list *(*a5_cell_list*)*, 9

vctrs::list_of(), *9*, *24*
vctrs::list_unchop(), *9*

with(), *36*
with.a5_threads *(*a5_threads*)*, 36
wk, *39*
wk handler, *39*
wk::rct(), *3*, *29, 30*
wk::wk_crs, *39*
wk::wk_handle(), *3*, *27*, *29*
wk::wkb(), *3*, *10*, *29*
wk::wkt(), *3*, *10*, *29*
wk::xy(), *14*
wk_crs.a5_cell *(*wk_methods*)*, 39
wk_handle.a5_cell *(*wk_methods*)*, 39
wk_methods, 39