Package ‘lawn’

February 1, 2019

Title Client for 'Turfjs' for 'Geospatial' Analysis

Description Client for 'Turfjs' (<http://turfjs.org>) for
'geospatial' analysis. The package revolves around using 'GeoJSON'
data. Functions are included for creating 'GeoJSON' data objects,
measuring aspects of 'GeoJSON', and combining, transforming,
and creating random 'GeoJSON' data objects.

Type Package

Version 0.5.0

License MIT + file LICENSE

URL `https://github.com/ropensci/lawn`

BugReports `https://github.com/ropensci/lawn/issues`

LazyData true

VignetteBuilder knitr

Encoding UTF-8

Imports V8, jsonlite, magrittr

Suggests roxygen2 (>= 6.1.1), testthat, knitr, rmarkdown, leaflet

Enhances maps, geojsonio

RoxygenNote 6.1.1

NeedsCompilation no

Author Scott Chamberlain [aut, cre],
Jeff Hollister [aut],
Morgan Herlocker [cph]

Maintainer Scott Chamberlain <`myrmecocystus@gmail.com`>

Repository CRAN

Date/Publication 2019-02-01 05:33:25 UTC

R topics documented:

`lawn-package` *R client for turf.js for geospatial analysis*

Description

turf.js uses GeoJSON for all geographic data, and expects the data to be standard WGS84 longitude,latitude coordinates. See `http://geojson.io/` for a tool to easily create GeoJSON in a browser.

Author(s)

Scott Chamberlain (<myrmecocystus@gmail.com>)

Jeff Hollister (<hollister.jeff@epa.gov>)

See Also

lawn-defunct

`as.feature` *Coerce character strings or JSON to GeoJSON Feature*

Description

Coerce character strings or JSON to GeoJSON Feature

Usage

```
as.feature(x, ...)
```

Arguments

`x`	a character string or json class with a GeoJSON object, any of feature, point, multipoint, linestring, multilinestring, polygon, or multipolygon. featurecollection and geometrycollection simply returned without alteration
`...`	ignored

Value

a `feature` class object

Examples

```
poly <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [105.818939,21.004714],
      [105.818939,21.061754],
      [105.890007,21.061754],
      [105.890007,21.004714],
      [105.818939,21.004714]
     ]]
  }
}'
as.feature(poly)

pt <- '{"type":"Point","coordinates":[-75.343,39.984]}'
as.feature(pt)

line <- '{
  "type": "LineString",
  "coordinates": [
    [-77.031669, 38.878605],
    [-77.029609, 38.881946],
    [-77.020339, 38.884084],
    [-77.025661, 38.885821],
    [-77.021884, 38.889563],
    [-77.019824, 38.892368]
    ]
}'
as.feature(line)

# returns self if no match - note "Points" is not a GeoJSON type
pt <- '{"type":"Points","coordinates":[-75.343,39.984]}'
as.feature(pt)
```

`as_feature` *Convert a FeatureCollection to a Feature*

Description

Convert a FeatureCollection to a Feature

Usage

```
as_feature(x)
```

Arguments

x	A data-FeatureCollection.

Details

If there are more than one feature within the featurecollection, each feature is split out into a separate feature, returned in a list. Each feature is assigned a class matching it's GeoJSON data type (e.g., point, polygon, linestring).

See Also

as.feature , which is similarly named, but has a different purpose

Examples

```
as_feature(lawn_random())
# as_feature(lawn_random("polygons"))
```

---

data-types *Description of GeoJSON data types*

Description

GeoJSON types based on https://tools.ietf.org/html/rfc7946

GeoJSON object

GeoJSON always consists of a single object. This object (referred to as the GeoJSON object below) represents a geometry, feature, or collection of features.

- The GeoJSON object may have any number of members (name/value pairs).
- The GeoJSON object must have a member with the name "type". This member's value is a string that determines the type of the GeoJSON object.
- The value of the type member must be one of: "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", "MultiPolygon", "GeometryCollection", "Feature", or "FeatureCollection". The case of the type member values must be as shown here.
- A GeoJSON object may have an optional "crs" member, the value of which must be a coordinate reference system object (see 3. Coordinate Reference System Objects).
- A GeoJSON object may have a "bbox" member, the value of which must be a bounding box array (see 4. Bounding Boxes).

Geometry

A Geometry object represents points, curves, and surfaces in coordinate space. Every Geometry object is a GeoJSON object no matter where it occurs in a GeoJSON text.

- The value of a Geometry object's "type" member MUST be one of the seven geometry types (see Section 1.4).
- A GeoJSON Geometry object of any type other than "GeometryCollection" has a member with the name "coordinates". The value of the "coordinates" member is an array. The structure of the elements in this array is determined by the type of geometry. GeoJSON processors MAY interpret Geometry objects with empty "coordinates" arrays as null objects.

Point

For type "Point", the "coordinates" member must be a single position.

Example JSON: `{ "type": "Point", "coordinates": [100.0, 0.0] }`

In `lawn`: `lawn_point(c(1, 2))`

See: lawn_point

MultiPoint

For type "MultiPoint", the "coordinates" member must be an array of positions.

Example JSON: `{ "type": "MultiPoint", "coordinates": [ [100.0, 0.0],[101.0, 1.0] ] }`

See: lawn_multipoint

Polygon

For type "Polygon", the "coordinates" member must be an array of LinearRing coordinate arrays. For Polygons with multiple rings, the first must be the exterior ring and any others must be interior rings or holes.

Example JSON: `{ "type": "Polygon", "coordinates": [  [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.`

In `lawn`: `lawn_polygon(list(list(c(-2, 52), c(-3, 54), c(-2, 53),c(-2, 52))))`

See: lawn_polygon

MultiPolygon

For type "MultiPolygon", the "coordinates" member must be an array of Polygon coordinate arrays.

Example JSON:

```
{ "type": "MultiPolygon", "coordinates": [   [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0],
] }
```

See: lawn_multipolygon

LineString

For type "LineString", the "coordinates" member must be an array of two or more positions. A LinearRing is closed LineString with 4 or more positions. The first and last positions are equivalent (they represent equivalent points). Though a LinearRing is not explicitly represented as a GeoJSON geometry type, it is referred to in the Polygon geometry type definition.

Example JSON: `{ "type": "LineString", "coordinates": [ [100.0, 0.0],[101.0, 1.0] ] }`

In lawn: `lawn_linestring(list(c(-2, 52), c(-3, 54), c(-2, 53)))`

See: lawn_linestring

MultiLineString

For type "MultiLineString", the "coordinates" member must be an array of LineString coordinate arrays.

Example JSON: `{  "type": "MultiLineString", "coordinates": [     [[ -105, 39 ], [ -105, 39 ]],     [[ -1`
`] }`

See: lawn_multilinestring

Feature

A GeoJSON object with the type "Feature" is a feature object:

- A feature object must have a member with the name "geometry". The value of the geometry member is a geometry object as defined above or a JSON null value.
- A feature object must have a member with the name "properties". The value of the properties member is an object (any JSON object or a JSON null value).
- If a feature has a commonly used identifier, that identifier should be included as a member of the feature object with the name "id".

See: lawn_feature

FeatureCollection

A GeoJSON object with the type "FeatureCollection" is a feature collection object. An object of type "FeatureCollection" must have a member with the name "features". The value corresponding to "features" is an array. Each element in the array is a feature object as defined above.

In lawn: `lawn_featurecollection(lawn_point(c(-75, 39)))`

See: lawn_featurecollection

GeometryCollection

Each element in the geometries array of a GeometryCollection is one of the geometry objects described above.

Example JSON: `{    "type": "GeometryCollection",    "geometries": [ {     "type": "Point",     "coordina`
`}, {     "type": "LineString",    "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]`
`}  ] }`

See: lawn_geometrycollection

georandom *Return a FeatureCollection with N number of features with random coordinates*

Description

Return a FeatureCollection with N number of features with random coordinates

Usage

```
gr_point(n = 10, bbox = NULL)

gr_position(bbox = NULL)

gr_polygon(n = 1, vertices = 10, max_radial_length = 10,
  bbox = NULL)
```

Arguments

n	(integer) Number of features to create. Default: 10 (points), 1 (polygons)
bbox	(numeric) A bounding box of length 4, of the form west, south, east, north order. By default, no bounding box is passed in.
vertices	(integer) Number coordinates each Polygon will contain. Default: 10
max_radial_length	(integer) Maximum number of decimal degrees latitude or longitude that a vertex can reach out of the center of the Polygon. Default: 10

Details

These functions create either random points, polygons, or positions (single long/lat coordinate pairs).

Value

A data-FeatureCollection for point and polygon, or numeric vector for position.

References

https://github.com/mapbox/geojson-random

See Also

lawn_random

Examples

```
# Random points
gr_point(5)
gr_point(10)
gr_point(1000)
## with bounding box
gr_point(5, c(50, 50, 60, 60))

# Random positions
gr_position()
## with bounding box
gr_position(c(0, 0, 10, 10))

# Random polygons
## number of polygons, default is 1 polygon
gr_polygon()
gr_polygon(5)
## number of vertices, 3 vs. 100
gr_polygon(1, 3)
gr_polygon(1, 100)
## max radial length, compare the following three
gr_polygon(1, 10, 5)
gr_polygon(1, 10, 30)
gr_polygon(1, 10, 100)
## use a bounding box
gr_polygon(1, 5, 5, c(50, 50, 60, 60))
```

---

lawn-defunct *Defunct functions in* **lawn**

Description

- lawn_size: Function removed. The `size` method in turf.js has been removed. See https://github.com/Turfjs/turf/issues/306
- lawn_reclass: Function removed. The `reclass` method in turf.js has been removed. See https://github.com/Turfjs/turf/issues/306
- lawn_jenks: Function removed. The `jenks` method in turf.js has been removed. See https://github.com/Turfjs/turf/issues/306
- lawn_quantile: Function removed. The `quantile` method in turf.js has been removed. See https://github.com/Turfjs/turf/issues/306
- lawn_aggregate: Function removed. The `aggregate` method in turf.js has been removed. See https://github.com/Turfjs/turf/issues/306

`lawn_along` *Get a point at a distance along a line*

Description

Takes a data-LineString and returns a data-Point at a specified distance along the line.

Usage

```
lawn_along(line, distance, units, lint = FALSE)
```

Arguments

`line`	An input data-LineString.
`distance`	Distance along the line.
`units`	Units for the `distance` argument. Can be degrees, radians, miles, or kilometers.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-Point distance units along the line.

See Also

Other measurements: `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
pts <- '[
   [-21.964416, 64.148203],
   [-21.956176, 64.141316],
   [-21.93901, 64.135924],
   [-21.927337, 64.136673]
]'
lawn_along(lawn_linestring(pts), 1, 'miles')

line <- '{
"type": "Feature",
"properties": {},
"geometry": {
  "type": "LineString",
  "coordinates": [
    [-77.031669, 38.878605],
    [-77.029609, 38.881946],
```

```
    [-77.020339, 38.884084],
    [-77.025661, 38.885821],
    [-77.021884, 38.889563],
    [-77.019824, 38.892368]
    ]
 }
}'
lawn_along(line, distance = 1, units = 'miles')
## Not run:
lawn_along(lawn_linestring(pts), 1, 'miles') %>% view
res <- lawn_along(lawn_linestring(pts), 1, 'miles')
lawn_featurecollection(list(res, lawn_linestring(pts))) %>% view

## End(Not run)
```

`lawn_area` *Calculate the area of a polygon or group of polygons*

Description

Calculate the area of a polygon or group of polygons

Usage

```
lawn_area(input, lint = FALSE)
```

Arguments

`input`	A data-Feature or data-FeatureCollection of polygons
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A numeric in square meters

See Also

Other measurements: `lawn_along`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
lawn_area(lawn_data$poly)
lawn_area(lawn_data$multipoly)
```

lawn_average — *Average of a field among points within polygons*

Description

Calculate the average value of a field for a set of data-Points within a set of data-Polygons

Usage

```
lawn_average(polygons, points, in_field, out_field = "average",
  lint = FALSE)
```

Arguments

polygons	A data-FeatureCollection of data-Polygon's
points	A data-FeatureCollection of data-Point's
in_field	(character) The field in the points feature from which to pull values to average.
out_field	(character) The field in polygons to put results of the averages.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

Polygons with the value of out_field set to the calculated averages

See Also

Other aggregations: `lawn_collect`, `lawn_count`, `lawn_deviation`, `lawn_max`, `lawn_median`, `lawn_min`, `lawn_sum`, `lawn_variance`

Examples

```
## Not run:
# using data in the package
cat(lawn_data$points_average)
cat(lawn_data$polygons_average)
lawn_average(polygons = lawn_data$polygons_average,
             points = lawn_data$points_average, 'population')

## End(Not run)
```

lawn_bbox *Make a bounding box from a polygon*

Description

Takes a polygon data-Polygon and returns a bbox

Usage

```
lawn_bbox(x, lint = FALSE)
```

Arguments

x	A FeatureCollection of data-Polygon features.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A bounding box.

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
bbox <- c(0, 0, 10, 10)
lawn_bbox(lawn_bbox_polygon(bbox))
```

lawn_bbox_polygon *Make a polygon from a bounding box*

Description

Takes a bbox and returns an equivalent polygon data-Polygon.

Usage

```
lawn_bbox_polygon(bbox)
```

Arguments

bbox	An array of bounding box coordinates in the form: `[xLow, yLow, xHigh, yHigh]`.

Value

A data-Polygon representation of the bounding box.

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
bbox <- c(0, 0, 10, 10)
lawn_bbox_polygon(bbox)
## Not run:
lawn_bbox_polygon(bbox) %>% view
lawn_bbox_polygon(c(1, 3, 5, 50)) %>% view

## End(Not run)
```

`lawn_bearing` *Get geographic bearing between two points*

Description

Takes two data-Point's and finds the geographic bearing between them.

Usage

```
lawn_bearing(start, end, lint = FALSE)
```

Arguments

`start`	Starting data-Feature with a single data-Point
`end`	Ending data-Feature with a single data-Point
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A numeric value of the bearing in degrees.

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
start <- '{
 "type": "Feature",
 "properties": {
   "marker-color": "#f00"
 },
 "geometry": {
   "type": "Point",
   "coordinates": [-75.343, 39.984]
 }
}'

end <- '{
  "type": "Feature",
  "properties": {
    "marker-color": "#0f0"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [-75.534, 39.123]
  }
}'
lawn_bearing(start, end)
```

`lawn_bezier` *Curve a linestring*

Description

Takes a data-LineString and returns a curved version by applying a Bezier spline algorithm.

Usage

```
lawn_bezier(line, resolution = 10000L, sharpness = 0.85,
  lint = FALSE)
```

Arguments

`line`	A data-Feature with a single data-LineString
`resolution`	Time in milliseconds between points
`sharpness`	A measure of how curvy the path should be between splines
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-LineString curved line.

See Also

Other transformations: lawn_buffer, lawn_concave, lawn_convex, lawn_difference, lawn_intersect, lawn_merge, lawn_simplify, lawn_union

Examples

```
pts <- '[
   [-21.964416, 64.148203],
   [-21.956176, 64.141316],
   [-21.93901, 64.135924],
   [-21.927337, 64.136673]
]'
lawn_bezier(lawn_linestring(pts))
lawn_bezier(lawn_linestring(pts), 9000L)
lawn_bezier(lawn_linestring(pts), 9000L, 0.65)
## Not run:
lawn_bezier(lawn_linestring(pts)) %>% view
lawn_featurecollection(list(lawn_linestring(pts),
  lawn_bezier(lawn_linestring(pts)))) %>% view

## End(Not run)
```

---

lawn_boolean_clockwise *Boolean clockwise*

Description

Boolean clockwise

Usage

```
lawn_boolean_clockwise(line, lint = FALSE)
```

Arguments

line	line data-Feature<(data-LineString)>
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a logical (TRUE/FALSE)

See Also

Other boolean functions: lawn_boolean_contains, lawn_boolean_crosses, lawn_boolean_disjoint, lawn_boolean_overlap, lawn_boolean_pointonline, lawn_boolean_within

Examples

```
l1 <- '[[0,0],[1,1],[1,0],[0,0]]'
l2 <- '[[0,0],[1,0],[1,1],[0,0]]'
lawn_boolean_clockwise(lawn_linestring(l1))
lawn_boolean_clockwise(lawn_linestring(l2))
```

---

lawn_boolean_contains *Boolean contains*

Description

Boolean contains

Usage

```
lawn_boolean_contains(feature1, feature2, lint = FALSE)
```

Arguments

`feature1, feature2`
: any data-Geometry/data-Feature objects

`lint`
: (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a logical (`TRUE`/`FALSE`)

See Also

Other boolean functions: `lawn_boolean_clockwise`, `lawn_boolean_crosses`, `lawn_boolean_disjoint`, `lawn_boolean_overlap`, `lawn_boolean_pointonline`, `lawn_boolean_within`

Examples

```
l1 <- '[[1, 1], [1, 2], [1, 3], [1, 4]]'
pt1 <- '[1, 2]'
lawn_boolean_contains(feature1=lawn_linestring(l1), feature2=lawn_point(pt1))
```

lawn_boolean_crosses *Boolean crosses*

Description

Boolean crosses

Usage

```
lawn_boolean_crosses(feature1, feature2, lint = FALSE)
```

Arguments

`feature1, feature2`
: any data-Geometry/data-Feature objects

`lint`
: (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a logical (`TRUE`/`FALSE`)

See Also

Other boolean functions: `lawn_boolean_clockwise`, `lawn_boolean_contains`, `lawn_boolean_disjoint`, `lawn_boolean_overlap`, `lawn_boolean_pointonline`, `lawn_boolean_within`

Examples

```
l1 <- '[[-2, 2], [4, 2]]'
l2 <- '[[1, 1], [1, 2], [1, 3], [1, 4]]'
lawn_boolean_crosses(lawn_linestring(l1), lawn_linestring(l2))
```

lawn_boolean_disjoint *Boolean crosses*

Description

Boolean crosses

Usage

```
lawn_boolean_disjoint(feature1, feature2, lint = FALSE)
```

Arguments

`feature1, feature2`
: any data-Geometry/data-Feature objects

`lint`
: (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a logical (`TRUE`/`FALSE`)

See Also

Other boolean functions: `lawn_boolean_clockwise`, `lawn_boolean_contains`, `lawn_boolean_crosses`, `lawn_boolean_overlap`, `lawn_boolean_pointonline`, `lawn_boolean_within`

Examples

```
pt1 <- '[2, 2]'
l1 <- '[[1, 1], [1, 2], [1, 3], [1, 4]]'
lawn_boolean_disjoint(lawn_point(pt1), lawn_linestring(l1))
```

`lawn_boolean_overlap` *Boolean overlap*

Description

Boolean overlap

Usage

```
lawn_boolean_overlap(feature1, feature2, lint = FALSE)
```

Arguments

`feature1, feature2`
: any data-Geometry/data-Feature objects

`lint`
: (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a logical (`TRUE`/`FALSE`)

See Also

Other boolean functions: lawn_boolean_clockwise, lawn_boolean_contains, lawn_boolean_crosses, lawn_boolean_disjoint, lawn_boolean_pointonline, lawn_boolean_within

Examples

```
poly1 <- "[[[0,0],[0,5],[5,5],[5,0],[0,0]]]"
poly2 <- "[[[1,1],[1,6],[6,6],[6,1],[1,1]]]"
poly3 <- "[[[10,10],[10,15],[15,15],[15,10],[10,10]]]"
lawn_boolean_overlap(lawn_polygon(poly1), lawn_polygon(poly2))
lawn_boolean_overlap(lawn_polygon(poly2), lawn_polygon(poly3))
```

---

lawn_boolean_pointonline

Boolean overlap

Description

Boolean overlap

Usage

```
lawn_boolean_pointonline(point, linestring, ignoreEndVertices = FALSE,
  lint = FALSE)
```

Arguments

point	any data-Geometry/data-Feature
linestring	any data-Geometry/data-Feature
ignoreEndVertices	(logical) whether to ignore the start and end vertices. Default: ‘FALSE
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a logical (`TRUE`/`FALSE`)

See Also

Other boolean functions: lawn_boolean_clockwise, lawn_boolean_contains, lawn_boolean_crosses, lawn_boolean_disjoint, lawn_boolean_overlap, lawn_boolean_within

Examples

```
l1 <- "[[-1, -1],[1, 1],[1.5, 2.2]]"
lawn_boolean_pointonline(lawn_point("[0, 0]"), lawn_linestring(l1))
```

lawn_boolean_within *Boolean within*

Description

returns TRUE if the first geometry is completely within the second geometry

Usage

```
lawn_boolean_within(feature1, feature2, lint = FALSE)
```

Arguments

feature1, feature2	any data-Geometry/data-Feature objects
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a logical (TRUE/FALSE)

See Also

Other boolean functions: lawn_boolean_clockwise, lawn_boolean_contains, lawn_boolean_crosses, lawn_boolean_disjoint, lawn_boolean_overlap, lawn_boolean_pointonline

Examples

```
pt1 <- '[1, 2]'
l1 <- '[[1, 1], [1, 2], [1, 3], [1, 4]]'
lawn_boolean_within(lawn_point(pt1), lawn_linestring(l1))
```

lawn_buffer *Buffer a feature*

Description

Calculates a buffer for input features for a given radius.

Usage

```
lawn_buffer(input, dist, units = "kilometers", lint = FALSE)
```

Arguments

`input`	A data-Feature or data-FeatureCollection
`dist`	(integer/numeric) Distance used to buffer the input.
`units`	(character) Units of the `dist` argument. Can be miles, feet, kilometers (default), meters, or degrees.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Author(s)

Jeff Hollister `<hollister.jeff@epa.gov>`

See Also

Other transformations: `lawn_bezier`, `lawn_concave`, `lawn_convex`, `lawn_difference`, `lawn_intersect`, `lawn_merge`, `lawn_simplify`, `lawn_union`

Examples

```
# From a Point
pt <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
    "type": "Point",
    "coordinates": [-90.548630, 14.616599]
  }
}'
lawn_buffer(pt, 5)

# From a FeatureCollection
dat <- lawn_random(n = 100)
lawn_buffer(dat, 100)

# From a Feature
dat <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
     "type": "Polygon",
     "coordinates": [[
       [-112.072391,46.586591],
       [-112.072391,46.61761],
       [-112.028102,46.61761],
       [-112.028102,46.586591],
       [-112.072391,46.586591]
     ]]
   }
}'
```

```
lawn_buffer(dat, 1, "miles")

# buffer a point
lawn_buffer(lawn_point(c(-74.50,40)), 100, "meters")
```

---

lawn_center *Get center point*

Description

Takes a data-FeatureCollection and returns the absolute center point of all features.

Usage

```
lawn_center(features, properties = NULL, lint = FALSE)
```

Arguments

features	Input features, as a data-Feature or data-FeatureCollection
properties	A list of properties. Default: `NULL`
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-Point feature at the absolute center point of all input features.

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
lawn_center(lawn_data$points_average)
lawn_center(lawn_data$points_average, properties = list(
  foo = "bar", hello = "world"))
## Not run:
lawn_center(lawn_data$points_average) %>% view
lawn_featurecollection(lawn_data$points_average) %>% view
lawn_center(lawn_data$points_average) %>% view

## End(Not run)
```

lawn_center_of_mass *Center of mass*

Description

Takes a data-Feature or a data-FeatureCollection and returns its center of mass using formula https://en.wikipedia.org/wiki/Centroid#Centroid_of_polygon

Usage

```
lawn_center_of_mass(x, lint = FALSE)
```

Arguments

x	a data-Feature or data-FeatureCollection
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a data-Feature<(data-Point)>

See Also

Other measurements: lawn_along, lawn_area, lawn_bbox_polygon, lawn_bbox, lawn_bearing, lawn_center, lawn_centroid, lawn_destination, lawn_distance, lawn_envelope, lawn_extent, lawn_line_distance, lawn_midpoint, lawn_point_on_surface, lawn_pt2line_distance, lawn_square

Examples

```
x <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
     "type": "Polygon",
     "coordinates": [[
       [-112.072391,46.586591],
       [-112.072391,46.61761],
       [-112.028102,46.61761],
       [-112.028102,46.586591],
       [-112.072391,46.586591]
     ]]
   }
}'
lawn_center_of_mass(x)

lawn_center_of_mass(lawn_data$polygons_average)
```

lawn_centroid *Centroid*

Description

Takes one or more features and calculates the centroid using the arithmetic mean of all vertices. This lessens the effect of small islands and artifacts when calculating the centroid of a set of polygons.

Usage

```
lawn_centroid(features, properties = NULL, lint = FALSE)
```

Arguments

features	Input features, as a data-Feature or data-FeatureCollection
properties	A list of properties. Default: NULL
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a data-Feature<(data-Point)> - centroid of the input features

See Also

Other measurements: lawn_along, lawn_area, lawn_bbox_polygon, lawn_bbox, lawn_bearing, lawn_center_of_mass, lawn_center, lawn_destination, lawn_distance, lawn_envelope, lawn_extent, lawn_line_distance, lawn_midpoint, lawn_point_on_surface, lawn_pt2line_distance, lawn_square

Examples

```
poly <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [105.818939,21.004714],
      [105.818939,21.061754],
      [105.890007,21.061754],
      [105.890007,21.004714],
      [105.818939,21.004714]
     ]]
  }
}'
lawn_centroid(features = poly)
lawn_centroid(features = as.feature(poly))
lawn_centroid(features = poly, properties = list(foo = "bar"))
```

`lawn_circle` *circle*

Description

Takes a data-Point and calculates the circle polygon given a radius in degrees, radians, miles, or kilometers; and steps for precision

Usage

```
lawn_circle(center, radius, steps = FALSE, units = "kilometers",
  lint = FALSE)
```

Arguments

`center`	The center, a data-Feature<(data-Point)>
`radius`	(integer) Radius of the circle.
`steps`	(integer) Number of steps.
`units`	(character) Miles, kilometers (default), degrees, or radians
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-Feature<(data-Polygon)>

See Also

Other assertions: `lawn_dissolve`, `lawn_tesselate`

Examples

```
pt <- '{
  "type": "Feature",
  "properties": {
    "marker-color": "#0f0"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [-75.343, 39.984]
  }
}'

lawn_circle(pt, radius = 5, steps = 10)
## Not run:
lawn_circle(pt, radius = 5, steps = 10) %>% view
lawn_circle(pt, radius = 4, steps = 10) %>% view
```

```
lawn_circle(pt, radius = 3, steps = 10) %>% view
lawn_circle(pt, radius = 10, steps = 10) %>% view
lawn_circle(pt, radius = 5, steps = 5) %>% view
lawn_circle(pt, radius = 5, steps = 4) %>% view

## End(Not run)
```

---

lawn_collect *Collect method*

Description

Given an inProperty on points and an outProperty for polygons, this finds every point that lies within each polygon, collects the inProperty values from those points, and adds them as an array to outProperty on the polygon.

Usage

```
lawn_collect(polygons, points, in_field, out_field, lint = FALSE)
```

Arguments

`polygons`	a `data-FeatureCollection` of `data-Polygon` features
`points`	a `data-FeatureCollection` of `data-Point` features
`in_field`	(character) the field in input data to analyze
`out_field`	(character) the field in which to store results
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A FeatureCollection of data-Polygon features with properties listed as `out_field`

Author(s)

Jeff Hollister `<hollister.jeff@epa.gov>`

See Also

Other aggregations: `lawn_average`, `lawn_count`, `lawn_deviation`, `lawn_max`, `lawn_median`, `lawn_min`, `lawn_sum`, `lawn_variance`

Examples

```
ex_polys <- lawn_data$polygons_aggregate
ex_pts <- lawn_data$points_aggregate
res <- lawn_collect(ex_polys, ex_pts, 'population', 'stuff')
res$type
res$features
res$features$properties

## Not run:
lawn_collect(ex_polys, ex_pts, 'population', 'stuff') %>% view

## End(Not run)
```

---

lawn_collectionof *Enforce expectations about types of FeatureCollection inputs*

Description

Enforce expectations about types of FeatureCollection inputs

Usage

```
lawn_collectionof(x, type, name, lint = FALSE)
```

Arguments

`x`	a data-FeatureCollection for which features will be judged. required
`type`	(character) expected GeoJSON type. required.
`name`	(character) name of calling function. required.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

nothing if no problems - error message if a problem

See Also

Other invariant: `lawn_featureof`, `lawn_geosjontype`

Examples

```
# all okay
cat(lawn_data$points_count)
lawn_collectionof(lawn_data$points_count, 'Point', 'stuff')

# error
# lawn_collectionof(lawn_data$points_count, 'Polygon', 'stuff')
```

`lawn_combine` *Combine singular features into plural versions*

Description

Combines a FeatureCollection of Point, LineString, or Polygon features into MultiPoint, MultiLineString, or MultiPolygon features.

Usage

```
lawn_combine(fc, lint = FALSE)
```

Arguments

`fc`	A data-FeatureCollection of any type.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Examples

```
# combine points
fc1 <- '{
 "type": "FeatureCollection",
 "features": [
   {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [19.026432, 47.49134]
     }
   }, {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [19.074497, 47.509548]
     }
   }
 ]
}'
lawn_combine(fc1)

# combine linestrings
fc2 <- '{
 "type": "FeatureCollection",
 "features": [
   {
```

```
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [-21.964416, 64.148203],
          [-21.956176, 64.141316],
          [-21.93901, 64.135924],
          [-21.927337, 64.136673]
        ]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [-21.929054, 64.127985],
          [-21.912918, 64.134726],
          [-21.916007, 64.141016],
          [-21.930084, 64.14446]
        ]
      }
    }
  ]
}'
lawn_combine(fc2)
## Not run:
fc1 %>% view
lawn_combine(fc1) %>% view
fc2 %>% view
lawn_combine(fc2) %>% view

## End(Not run)
```

lawn_concave *Concave hull polygon*

Description

Takes a set of data-Point's and returns a concave hull polygon. Internally, this implements a Monotone chain algorithm

Usage

```
lawn_concave(points, maxEdge = 1, units = "miles", lint = FALSE)
```

Arguments

points	Input points in a data-FeatureCollection.
maxEdge	The size of an edge necessary for part of the hull to become concave (in miles).
units	Used for maxEdge distance (miles (default) or kilometers).
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a concave hull data-Polygon

See Also

Other transformations: `lawn_bezier`, `lawn_buffer`, `lawn_convex`, `lawn_difference`, `lawn_intersect`, `lawn_merge`, `lawn_simplify`, `lawn_union`

Examples

```
## Not run:
points <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.601226, 44.642643]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.591442, 44.651436]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.580799, 44.648749]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.573589, 44.641788]
```

```
        }
      }, {
        "type": "Feature",
        "properties": {},
        "geometry": {
          "type": "Point",
          "coordinates": [-63.587665, 44.64533]
        }
      }, {
        "type": "Feature",
        "properties": {},
        "geometry": {
          "type": "Point",
          "coordinates": [-63.595218, 44.64765]
        }
      }
      ]
}'
lawn_concave(points, 1)

lawn_concave(points) %>% view

## End(Not run)
```

`lawn_convex` *Convex hull polygon*

Description

Takes a set of data-Point's and returns a convex hull polygon. Internally, this uses the convex-hull module that implements a Monotone chain hull

Usage

```
lawn_convex(input, lint = FALSE)
```

Arguments

`input`	Input points in a data-FeatureCollection.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a convex hull data-Polygon

See Also

Other transformations: lawn_bezier, lawn_buffer, lawn_concave, lawn_difference, lawn_intersect, lawn_merge, lawn_simplify, lawn_union

Examples

```
points <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.601226, 44.642643]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.591442, 44.651436]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.580799, 44.648749]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.573589, 44.641788]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.587665, 44.64533]
      }
    }, {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Point",
        "coordinates": [-63.595218, 44.64765]
      }
    }
```

```
    ]
}'
lawn_convex(points)
## Not run:
lawn_convex(points) %>% view

## End(Not run)
```

`lawn_coordall` *Get all coordinates from any GeoJSON object, returning an array of coordinate arrays.*

Description

Takes any data-GeoJSON and returns an array of coordinate arrays

Usage

```
lawn_coordall(x, lint = FALSE)
```

Arguments

`x`	any data-GeoJSON object
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

matrix of coordinates, where each row in the matrix is a coordinate pair

Examples

```
lawn_point(c(-74.5, 40)) %>% lawn_coordall()

rings <- list(list(
   c(-2.275543, 53.464547),
   c(-2.275543, 53.489271),
   c(-2.215118, 53.489271),
   c(-2.215118, 53.464547),
   c(-2.275543, 53.464547)
))
lawn_polygon(rings) %>% lawn_coordall()
```

lawn_coordeach *Iterate over property objects in any GeoJSON object*

Description

Iterate over property objects in any GeoJSON object

Usage

```
lawn_coordeach(x, fun = NULL, excludeWrapCoord = FALSE, lint = FALSE)
```

Arguments

x	any data-GeoJSON object
fun	(character) a Javascript function. if not given, returns self
excludeWrapCoord	(logical) whether or not to include the final coordinate of LinearRings that wraps the ring in its iteration.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

matrix of coordinates, where each row in the matrix is a coordinate pair

Examples

```
x <- "{ type: 'Point', coordinates: [10, 50] }"

# don't apply any function, identity essentially
lawn_coordeach(x)

# appply a function callback
lawn_coordeach(x, "z.length === 2")
lawn_coordeach(lawn_data$points_count, "z.length === 2")

z <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "population": 200,
        "name": "things"
      },
      "geometry": {
        "type": "Point",
```

```
        "coordinates": [-112.0372, 46.608058]
      }
    }, {
      "type": "Feature",
      "properties": {
        "population": 600,
        "name": "stuff"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-112.045955, 46.596264]
      }
    }
    ]
}'
lawn_coordeach(z)
lawn_coordeach(z, "z.reduce(function(a, b) { return a + b; }, 0)")
```

`lawn_count` *Count number of points within polygons*

Description

Calculates the number of data-Point's that fall within the set of data-Polygon's

Usage

```
lawn_count(polygons, points, in_field, out_field = "count",
  lint = FALSE)
```

Arguments

`polygons`	a `data-FeatureCollection` of `data-Polygon` features
`points`	a `data-FeatureCollection` of `data-Point` features
`in_field`	(character) the field in input data to analyze
`out_field`	(character) the field in which to store results
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-FeatureCollection

See Also

Other aggregations: `lawn_average`, `lawn_collect`, `lawn_deviation`, `lawn_max`, `lawn_median`, `lawn_min`, `lawn_sum`, `lawn_variance`

Examples

```
## Not run:
# using data in the package
cat(lawn_data$points_count)
cat(lawn_data$polygons_count)
lawn_count(lawn_data$polygons_count, lawn_data$points_count, 'population')

## End(Not run)
```

---

lawn_data *Data for use in examples*

Description

Data for use in examples

Format

A list of character strings of points or polygons in FeatureCollection or Feature Geojson formats.

Details

The data objects included in the list, accessible by name

- filter_features - FeatureCollection of points
- points_average - FeatureCollection of points
- polygons_average - FeatureCollection of polygons
- points_count - FeatureCollection of points
- polygons_count - FeatureCollection of polygons
- points_within - FeatureCollection of points
- polygons_within - FeatureCollection of polygons
- poly - Feature of a single 1 degree by 1 degree polygon
- multipoly - FeatureCollection of two 1 degree by 1 degree polygons
- polygons_aggregate - FeatureCollection of Polygons from turf.js examples
- points_aggregate - FeatureCollection of Points from turf.js examples

`lawn_destination` *Calculate destination point*

Description

Takes a data-Point and calculates the location of a destination point given a distance in degrees, radians, miles, or kilometers; and bearing in degrees. Uses the Haversine formula to account for global curvature.

Usage

```
lawn_destination(start, distance, bearing, units, lint = FALSE)
```

Arguments

`start`	Starting point, a data-Feature<data-Point>
`distance`	Distance from the starting point.
`bearing`	Ranging from -180 to 180.
`units`	Miles, kilometers, degrees, or radians.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

the calculated destination, a data-Feature<data-Point>

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
pt <- '{
  "type": "Feature",
  "properties": {
    "marker-color": "#0f0"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [-75.343, 39.984]
  }
}'
lawn_destination(pt, 50, 90, "miles")
lawn_destination(pt, 100, 90, "miles")
```

```
lawn_destination(pt, 2, 45, "kilometers")
lawn_destination(pt, 2, 30, "degrees")
## Not run:
pt %>% view
lawn_destination(pt, 200, 90, "miles") %>% view

## End(Not run)
```

`lawn_deviation` *Standard deviation of a field among points within polygons*

Description

Calculates the population standard deviation (i.e. denominator = n, not n-1) of values from data-Point's within a set of data-Polygon's

Usage

```
lawn_deviation(polygons, points, in_field, out_field = "deviation",
  lint = FALSE)
```

Arguments

polygons	Polygon(s) (data-FeatureCollection<(data-Polygon)>) defining area to aggregate
points	Points (data-FeatureCollection<(data-Point)>) with values to aggregate
in_field	Character for the name of the field on `pts` on which you wish to perform the aggregation.
out_field	Character for the name of the field on the output polygon FeatureCollection that will store the resultant value.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

polygons with appended field representing deviation, as a data-FeatureCollection

Author(s)

Jeff Hollister <hollister.jeff@epa.gov>

See Also

Other aggregations: `lawn_average`, `lawn_collect`, `lawn_count`, `lawn_max`, `lawn_median`, `lawn_min`, `lawn_sum`, `lawn_variance`

Examples

```
## Not run:
ex_polys <- lawn_data$polygons_aggregate
ex_pts <- lawn_data$points_aggregate
lawn_deviation(ex_polys, ex_pts, "population")

## End(Not run)
```

`lawn_difference` *Difference*

Description

Finds the difference between two data-Polygon's by clipping the second polygon from the first.

Usage

```
lawn_difference(poly1, poly2, lint = FALSE)
```

Arguments

`poly1`	A data-Feature<(data-Polygon)> feature
`poly2`	data-Feature<(data-Polygon)> to erase from poly1
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-Feature<(data-Polygon)> feature showing the area of poly1 excluding the area of poly2

See Also

Other transformations: `lawn_bezier`, `lawn_buffer`, `lawn_concave`, `lawn_convex`, `lawn_intersect`, `lawn_merge`, `lawn_simplify`, `lawn_union`

Examples

```
## Not run:
# skipping on cran
poly1 <- '{
 "type": "Feature",
  "properties": {
    "fill": "#0f0"
  },
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
```

```
      [-46.738586, -23.596711],
      [-46.738586, -23.458207],
      [-46.560058, -23.458207],
      [-46.560058, -23.596711],
      [-46.738586, -23.596711]
    ]]
  }
}'

poly2 <- '{
  "type": "Feature",
  "properties": {
    "fill": "#00f"
  },
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [-46.650009, -23.631314],
      [-46.650009, -23.5237],
      [-46.509246, -23.5237],
      [-46.509246, -23.631314],
      [-46.650009, -23.631314]
    ]]
  }
}'
lawn_difference(poly1, poly2)

## End(Not run)
## Not run:
lawn_featurecollection(list(poly1, poly2)) %>% view
lawn_difference(poly1, poly2) %>% view
fc <- lawn_featurecollection(list(
   lawn_polygon(jsonlite::fromJSON(poly1)$geometry$coordinates),
   lawn_polygon(jsonlite::fromJSON(poly2)$geometry$coordinates)
))
view(fc)

## End(Not run)
```

`lawn_dissolve` *Dissolves a FeatureCollection of polygons based on a property. Note that multipart features within the collection are not supported*

Description

Dissolves a FeatureCollection of polygons based on a property. Note that multipart features within the collection are not supported

Usage

```
lawn_dissolve(features, key, lint = FALSE)
```

Arguments

`features`	A data-FeatureCollection<(data-Polygon)>
`key`	(character) The property on which to filter
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-FeatureCollection<(data-Polygon)> containing the dissolved polygons

See Also

Other assertions: `lawn_circle`, `lawn_tesselate`

Examples

```
cat(lawn_data$filter_features)
x <- '{
 "type": "FeatureCollection",
 "features": [
  {
    "type": "Feature",
    "properties": {
      "combine": "yes"
    },
    "geometry": {
      "type": "Polygon",
      "coordinates": [[[0, 0], [0, 1], [1, 1], [1, 0], [0, 0]]]
    }
  },
  {
    "type": "Feature",
    "properties": {
      "combine": "yes"
    },
    "geometry": {
      "type": "Polygon",
      "coordinates": [[[0, -1], [0, 0], [1, 0], [1, -1], [0,-1]]]
    }
  },
  {
    "type": "Feature",
    "properties": {
      "combine": "no"
    },
    "geometry": {
      "type": "Polygon",
      "coordinates": [[[1,-1],[1, 0], [2, 0], [2, -1], [1, -1]]]
    }
  }
```

```
  ]
}'
lawn_dissolve(x, key = 'combine')
```

`lawn_distance` *Distance between two points*

Description

Calculates the distance between two data-Points in degress, radians, miles, or kilometers. Uses the Haversine formula to account for global curvature.

Usage

```
lawn_distance(from, to, units = "kilometers", lint = FALSE)
```

Arguments

`from`	Origin data-Feature<(data-Point)>
`to`	Destination data-Feature<(data-Point)>
`units`	(character) Can be degrees, radians, miles, or kilometers (default).
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

Single numeric value

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
from <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Point",
   "coordinates": [-75.343, 39.984]
 }
}'
to <- '{
  "type": "Feature",
  "properties": {},
```

```
  "geometry": {
    "type": "Point",
    "coordinates": [-75.534, 39.123]
  }
}'
lawn_distance(from, to)
```

`lawn_envelope` *Calculate envelope around features*

Description

Takes any number of features and returns a rectangular data-Polygon that encompasses all vertices.

Usage

```
lawn_envelope(fc, lint = FALSE)
```

Arguments

`fc`	A data-Feature or data-FeatureCollection
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a rectangular data-Feature<(data-Polygon)> that encompasses all vertices

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
fc <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "name": "Location A"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-75.343, 39.984]
      }
```

```
    }, {
      "type": "Feature",
      "properties": {
        "name": "Location B"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-75.833, 39.284]
      }
    }, {
      "type": "Feature",
      "properties": {
        "name": "Location C"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-75.534, 39.123]
      }
    }
  ]
}'
lawn_envelope(fc)
## Not run:
fc %>% view
lawn_envelope(fc) %>% view

## End(Not run)
```

`lawn_explode` *Explode vertices to points*

Description

Takes a feature or set of features and returns all positions as points

Usage

```
lawn_explode(input, lint = FALSE)
```

Arguments

`input`	data-Feature or data-FeatureCollection
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-FeatureCollection of points

Examples

```
poly <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Polygon",
   "coordinates": [[
     [177.434692, -17.77517],
     [177.402076, -17.779093],
     [177.38079, -17.803937],
     [177.40242, -17.826164],
     [177.438468, -17.824857],
     [177.454948, -17.796746],
     [177.434692, -17.77517]
    ]]
 }
}'
lawn_explode(poly)
## Not run:
lawn_data$polygons_average %>% view
lawn_explode(lawn_data$polygons_average) %>% view
lawn_data$polygons_within %>% view
lawn_explode(lawn_data$polygons_within) %>% view

## End(Not run)
```

`lawn_extent` *Get a bounding box*

Description

Calculates the extent of all input features in a FeatureCollection, and returns a bounding box. The returned bounding box is of the form (west, south, east, north).

Usage

```
lawn_extent(input, lint = FALSE)
```

Arguments

`input`	A data-Feature or data-FeatureCollection
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A bounding box, numeric vector of length 4, in `[ minX, minY, maxX, maxY ]` order

See Also

Other measurements: lawn_along, lawn_area, lawn_bbox_polygon, lawn_bbox, lawn_bearing, lawn_center_of_mass, lawn_center, lawn_centroid, lawn_destination, lawn_distance, lawn_envelope, lawn_line_distance, lawn_midpoint, lawn_point_on_surface, lawn_pt2line_distance, lawn_square

Examples

```
# From a FeatureCollection
cat(lawn_data$points_average)
lawn_extent(lawn_data$points_average)

# From a Feature
dat <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
     "type": "Polygon",
     "coordinates": [[
       [-112.072391,46.586591],
       [-112.072391,46.61761],
       [-112.028102,46.61761],
       [-112.028102,46.586591],
       [-112.072391,46.586591]
     ]]
   }
}'
lawn_extent(dat)
```

---

lawn_feature *Create a Feature*

Description

Create a Feature

Usage

```
lawn_feature(geometry, properties = c(), lint = FALSE)
```

Arguments

geometry	(character/json) Any geojson geometry.
properties	(list) list of properties, must be named
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

See Also

Other data functions: lawn_featurecollection, lawn_filter, lawn_geometrycollection, lawn_linestring, lawn_multilinestring, lawn_multipoint, lawn_multipolygon, lawn_point, lawn_polygon, lawn_random, lawn_remove, lawn_sample

Examples

```
## Not run:
# points
## single point
pt <- '{"type":"Point","coordinates":[-75.343,39.984]}'
lawn_feature(pt)

## with properties
lawn_feature(pt, properties = list(foo = "bar"))

## many points in a list
pts <- list(
   lawn_point(c(-75.343, 39.984))$geometry,
   lawn_point(c(-75.833, 39.284))$geometry,
   lawn_point(c(-75.534, 39.123))$geometry
)
lapply(pts, lawn_feature)

## End(Not run)
```

---

lawn_featurecollection *Create a FeatureCollection*

Description

Create a FeatureCollection

Usage

```
lawn_featurecollection(features)
```

Arguments

features	Input features, can be json as json or character class, or a point, polygon, linestring, or centroid class, or many of those things in a list.

See Also

Other data functions: lawn_feature, lawn_filter, lawn_geometrycollection, lawn_linestring, lawn_multilinestring, lawn_multipoint, lawn_multipolygon, lawn_point, lawn_polygon, lawn_random, lawn_remove, lawn_sample

Examples

```
## Not run:
# points
## single point
pt <- lawn_point(c(-75.343, 39.984), properties = list(name = 'Location A'))
lawn_featurecollection(pt)

## many points in a list
features <- list(
   lawn_point(c(-75.343, 39.984), properties = list(name = 'Location A')),
   lawn_point(c(-75.833, 39.284), properties = list(name = 'Location B')),
   lawn_point(c(-75.534, 39.123), properties = list(name = 'Location C'))
)
lawn_featurecollection(features)

# polygons
rings <- list(list(
   c(-2.275543, 53.464547),
   c(-2.275543, 53.489271),
   c(-2.215118, 53.489271),
   c(-2.215118, 53.464547),
   c(-2.275543, 53.464547)
))
## single polygon
lawn_featurecollection(lawn_polygon(rings))

## many polygons in a list
rings2 <- list(list(
   c(-2.775543, 54.464547),
   c(-2.775543, 54.489271),
   c(-2.245118, 54.489271),
   c(-2.245118, 54.464547),
   c(-2.775543, 54.464547)
))
features <- list(
   lawn_polygon(rings, properties = list(name = 'poly1', population = 400)),
   lawn_polygon(rings2, properties = list(name = 'poly2', population = 5000))
)
lawn_featurecollection(features)

# linestrings
pts1 <- list(
   c(-2.364416, 53.448203),
   c(-2.356176, 53.441316),
   c(-2.33901, 53.435924),
   c(-2.327337, 53.436673)
)
## single linestring
lawn_featurecollection(lawn_linestring(pts1))

## many linestring's in a list
pts2 <- rapply(pts1, function(x) x+0.1, how = "list")
```

```
features <- list(
   lawn_linestring(pts1, properties = list(name = 'line1', distance = 145)),
   lawn_linestring(pts2, properties = list(name = 'line2', distance = 145))
)
lawn_featurecollection(features)

# mixed feature set: polygon, linestring, and point
features <- list(
   lawn_polygon(rings, properties = list(name = 'poly1', population = 400)),
   lawn_linestring(pts1, properties = list(name = 'line1', distance = 145)),
   lawn_point(c(-2.25, 53.479271), properties = list(name = 'Location A'))
)
lawn_featurecollection(features)

# Return self if a featurecollection class passed
res <- lawn_featurecollection(features)
lawn_featurecollection(res)

# json featurecollection passed in
library("jsonlite")
str <- toJSON(unclass(res))
lawn_featurecollection(str)

# from a centroid object
poly <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [105.818939,21.004714],
      [105.818939,21.061754],
      [105.890007,21.061754],
      [105.890007,21.004714],
      [105.818939,21.004714]
     ]]
  }
}'
cent <- lawn_centroid(poly)
lawn_featurecollection(cent)

# from a feature
pt <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Point",
   "coordinates": [-90.548630, 14.616599]
  }
}'
x <- lawn_buffer(pt, 5)
lawn_featurecollection(x)
```

```
# From a geo_list object from geojsonio package
# library("geojsonio")
# vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0), c(100.0,1.0),
#   c(100.0,0.0))
# x <- geojson_list(vecs, geometry="polygon")
# lawn_featurecollection(x)

## End(Not run)
```

---

lawn_featureeach *Iterate over features in any GeoJSON object*

Description

Iterate over features in any GeoJSON object

Usage

```
lawn_featureeach(x, fun = NULL, lint = FALSE)
```

Arguments

x	any data-GeoJSON object
fun	a Javascript function. if not given, returns self
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

matrix of coordinates, where each row in the matrix is a coordinate pair

Examples

```
x <- "{ type: 'Feature', geometry: null, properties: { foo: 1, bar: 3 } }"

# don't apply any function, identity essentially
lawn_featureeach(x)

lawn_featureeach(lawn_data$points_count)

# appply a function callback
lawn_featureeach(lawn_data$points_count, "z.geometry")
lawn_featureeach(lawn_data$points_count, "z.geometry.type")
lawn_featureeach(lawn_data$points_count, "z.properties")
lawn_featureeach(lawn_data$points_count, "z.properties.population")
```

lawn_featureof *Enforce expectations about types of Feature inputs*

Description

Enforce expectations about types of Feature inputs

Usage

```
lawn_featureof(x, type, name, lint = FALSE)
```

Arguments

x	a data-Feature with an expected geometry type. required.
type	(character) expected GeoJSON type. required.
name	(character) name of calling function. required.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

nothing if no problems - error message if a problem

See Also

Other invariant: `lawn_collectionof`, `lawn_geosjontype`

Examples

```
# all okay
x <- "{ type: 'Feature', properties: {}, geometry: { type: 'Point',
  coordinates: [10, 50] } }"
lawn_featureof(x, 'Point', 'foobar')

# error
# lawn_featureof(x, 'MultiPoint', 'foobar')
```

lawn_filter *Filter a FeatureCollection by a given property and value*

Description

Filter a FeatureCollection by a given property and value

Usage

```
lawn_filter(features, key, value, lint = FALSE)
```

Arguments

features	A data-FeatureCollection
key	(character) The property on which to filter.
value	(character) The value of that property on which to filter.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

S filtered data-FeatureCollection with only features that match input key and value.

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_random`, `lawn_remove`, `lawn_sample`

Examples

```
cat(lawn_data$filter_features)
lawn_filter(features = lawn_data$filter_features, key = 'species',
  value = 'oak')
lawn_filter(lawn_data$filter_features, 'species', 'maple')
lawn_filter(lawn_data$filter_features, 'species', 'redwood')
```

lawn_flatten *Flatten*

Description

Flattens any GeoJSON to a FeatureCollection

Usage

```
lawn_flatten(x, lint = FALSE)
```

Arguments

x	any valid GeoJSON with multi-geometry data-Feature's
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a data-FeatureCollection

See Also

Other misc: lawn_truncate

Examples

```
x <- '{"type":"MultiPolygon","coordinates":[
  [[[102,2],[103,2],[103,3],[102,3],[102,2]]],
  [[[100,0],[101,0],[101,1],[100,1],[100,0]],
  [[100.2,0.2],[100.2,0.8],[100.8,0.8],[100.8,0.2],[100.2,0.2]]]
 ]
}'
lawn_flatten(x)
lawn_flatten(x, TRUE)
```

lawn_flip *Flip x,y to y,x, and vice versa*

Description

Flip x,y to y,x, and vice versa

Usage

```
lawn_flip(input, lint = FALSE)
```

Arguments

input	data-Feature or data-FeatureCollection
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

A data-Feature or data-FeatureCollection

Examples

```
# a point
serbia <- '{
  "type": "Feature",
  "properties": {"color": "red"},
  "geometry": {
    "type": "Point",
    "coordinates": [20.566406, 43.421008]
   }
}'
lawn_flip(serbia)

# a featurecollection
pts <- lawn_random("points")
lawn_flip(pts)
## Not run:
lawn_data$points_average %>% view
lawn_flip(lawn_data$points_average) %>% view
lawn_data$polygons_average %>% view
lawn_flip(lawn_data$polygons_average) %>% view

## End(Not run)
```

---

lawn_geometrycollection

Create a geometrycollection

Description

Create a geometrycollection

Usage

```
lawn_geometrycollection(coordinates, properties = NULL)
```

Arguments

`coordinates`	A list of GeoJSON geometries, or in json.
`properties`	A list of properties.

Value

A data-GeometryCollection feature.

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_random`, `lawn_remove`, `lawn_sample`

Examples

```
x <- list(
  list(
    type = "Point",
    coordinates = list(
     list(100, 0)
    )
  ),
  list(
    type = "LineString",
    coordinates = list(
     list(100, 0),
     list(102, 1)
    )
  )
)
lawn_geometrycollection(x)
lawn_geometrycollection(x,
  properties = list(city = 'Los Angeles', population = 400))

x <- '[
   {
     "type": "Point",
     "coordinates": [100.0, 0.0]
   },
   {
     "type": "LineString",
     "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
   }
]'
lawn_geometrycollection(x)
```

lawn_geosjontype — *Enforce expectations about types of GeoJSON objects.*

Description

Enforce expectations about types of GeoJSON objects.

Usage

```
lawn_geosjontype(x, type, name, lint = FALSE)
```

Arguments

`x`	value of any data-GeoJSON object. required.
`type`	expected GeoJSON type. required.
`name`	name of calling function. required.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

nothing if no problems - error message if a problem

See Also

Other invariant: `lawn_collectionof`, `lawn_featureof`

Examples

```
# all okay
x <- "{ type: 'Point', coordinates: [10, 50] }"
lawn_geosjontype(x, 'Point', 'fooBar')

# error
# lawn_geosjontype(x, 'Polygon', 'fooBar')
```

lawn_getcoord — *Unwrap a coordinate from a Feature with a Point geometry, or a single coordinate.*

Description

Unwrap a coordinate from a Feature with a Point geometry, or a single coordinate.

Usage

```
lawn_getcoord(x, lint = FALSE)
```

Arguments

x	any data-GeoJSON object
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

matrix of coordinates, where each row in the matrix is a coordinate pair

Examples

```
x <- "{ type: 'Point', coordinates: [10, 50] }"
lawn_getcoord(x)

library(jsonlite)
x <- fromJSON(lawn_data$points_count, FALSE)$features
lawn_getcoord(x[[1]])
lawn_getcoord(x[[2]])
lawn_getcoord(x[[1]]$geometry)
lawn_getcoord(x[[1]]$geometry$coordinates)

# fails
# lawn_getcoord(x[[1]]$geometry$coordinates[[1]])
```

lawn_hex_grid — *Create a HexGrid*

Description

Takes a bounding box and a cell size in degrees and returns a data-FeatureCollection of flat-topped hexagons (data-Polygon features) aligned in an "odd-q" vertical grid as described in Hexagonal Grids http://www.redblobgames.com/grids/hexagons/

Usage

```
lawn_hex_grid(extent, cellWidth, units)
```

Arguments

extent	(numeric) Extent in [minX, minY, maxX, maxY] order.
cellWidth	(integer) Width of each cell.
units	(character) Units to use for cellWidth, one of 'miles' or 'kilometers'.

Value

A data-FeatureCollection grid of points.

See Also

Other interpolation: lawn_isolines, lawn_planepoint, lawn_point_grid, lawn_square_grid, lawn_tin, lawn_triangle_grid

Examples

```
lawn_hex_grid(c(-96,31,-84,40), 50, 'miles')
lawn_hex_grid(c(-96,31,-84,40), 30, 'miles')
```

---

lawn_idw *IDW*

Description

Takes a FeatureCollection of points with known value, a power parameter, a cell depth, a unit of measurement and returns a FeatureCollection of polygons in a square-grid with an interpolated value property "IDW" for each grid cell. It finds application when in need of creating a continuous surface (i.e. rainfall, temperature, chemical dispersion surface...) from a set of spatially scattered points.

Usage

```
lawn_idw(controlPoints, valueField, b, cellWidth, units = "kilometers",
  lint = FALSE)
```

Arguments

controlPoints	A data-FeatureCollection, Sampled points with known value
valueField	(character) GeoJSON field containing the known value to interpolate on
b	(integer) Exponent regulating the distance-decay weighting
cellWidth	(integer) The distance across each cell

units (character) used in calculating cellSize, can be degrees, radians, miles, or kilometers

lint (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a data-FeatureCollection containing the dissolved polygons

See Also

Other grids: lawn_unkinkpolygon

Examples

```
x <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "",
        "value": 4,
        "id": 4
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          9.155731201171875,
          45.47216977418841
        ]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "",
        "value": 99,
        "id": 2
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          9.195213317871094,
          45.53689620055365
        ]
      }
```

```
    },
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "",
        "value": 10,
        "id": 1
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          9.175300598144531,
          45.49912810913339
        ]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "",
        "value": 6,
        "id": 3
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          9.231605529785156,
          45.49190839157102
        ]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "marker-color": "#7e7e7e",
        "marker-size": "medium",
        "marker-symbol": "",
        "value": 7,
        "id": 5
      },
      "geometry": {
        "type": "Point",
        "coordinates": [
          9.116249084472656,
          45.4391764115696
        ]
      }
    }
  ]
```

```
}'
lawn_idw(x, 'value', 0.5, 1)
```

`lawn_inside` *Does a point reside inside a polygon*

Description

Takes a data-Point and a data-Polygon or data-MultiPolygon and determines if the point resides inside the polygon

Usage

```
lawn_inside(point, polygon, lint = FALSE)
```

Arguments

`point`	Input point.
`polygon`	Input polygon or multipolygon.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

The polygon can be convex or concave. The function accounts for holes.

Value

`TRUE` if the Point IS inside the Polygon, `FALSE` if the Point IS NOT inside the Polygon.

See Also

Other joins: `lawn_tag`, `lawn_within`

Examples

```
point1 <- '{
  "type": "Feature",
  "properties": {
    "marker-color": "#f00"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [-111.467285, 40.75766]
  }
}'
point2 <- '{
```

```
  "type": "Feature",
  "properties": {
    "marker-color": "#0f0"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [-111.873779, 40.647303]
  }
}'
poly <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [-112.074279, 40.52215],
      [-112.074279, 40.853293],
      [-111.610107, 40.853293],
      [-111.610107, 40.52215],
      [-112.074279, 40.52215]
      ]]
  }
}'
lawn_inside(point1, poly)
lawn_inside(point2, poly)
```

---

lawn_intersect *Intersection*

Description

Finds the intersection of two data-Polygon's and returns just the intersection of the two

Usage

```
lawn_intersect(poly1, poly2, lint = FALSE)
```

Arguments

poly1	A data-Polygon.
poly2	A data-Polygon.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

Polygons with just a shared boundary will return the boundary. Polygons that do not intersect will return `NULL`.

Value

data-Polygon, data-MultiLineString, or undefined

Author(s)

Jeff Hollister <hollister.jeff@epa.gov>

See Also

Other transformations: `lawn_bezier`, `lawn_buffer`, `lawn_concave`, `lawn_convex`, `lawn_difference`, `lawn_merge`, `lawn_simplify`, `lawn_union`

Examples

```
## Not run:
poly1 <- '{
 "type": "Feature",
 "properties": {
   "fill": "#0f0"
 },
 "geometry": {
   "type": "Polygon",
   "coordinates": [[
     [-122.801742, 45.48565],
     [-122.801742, 45.60491],
     [-122.584762, 45.60491],
     [-122.584762, 45.48565],
     [-122.801742, 45.48565]
    ]]
 }
}'

poly2 <- '{
 "type": "Feature",
 "properties": {
   "fill": "#00f"
 },
 "geometry": {
   "type": "Polygon",
   "coordinates": [[
     [-122.520217, 45.535693],
     [-122.64038, 45.553967],
     [-122.720031, 45.526554],
     [-122.669906, 45.507309],
     [-122.723464, 45.446643],
     [-122.532577, 45.408574],
     [-122.487258, 45.477466],
     [-122.520217, 45.535693]
     ]]
 }
}'
lawn_intersect(poly1, poly2)
```

```
view(poly1)
view(poly2)
lawn_intersect(poly1, poly2) %>% view()

x1 <- lawn_buffer(lawn_point(c(-122.6375, 45.53)), 1500, "meters")
x2 <- lawn_buffer(lawn_point(c(-122.6475, 45.53)), 1500, "meters")
lawn_intersect(x1, x2)
structure(x1, class = "featurecollection") %>% view()
structure(x2, class = "featurecollection") %>% view()
lawn_intersect(x1, x2) %>% view()

# not overlapping
x3 <- lawn_buffer(lawn_point(c(-122.6375, 45.53)), 1500, "meters")
x4 <- lawn_buffer(lawn_point(c(-122.6975, 45.53)), 1500, "meters")
structure(x3, class = "featurecollection") %>% view()
structure(x4, class = "featurecollection") %>% view()
lawn_intersect(x3, x4)

## End(Not run)
```

`lawn_isolines` *Generate Isolines*

Description

Takes data-Point's with z-values and an array of value breaks and generates isolines

Usage

```
lawn_isolines(points, breaks, z, propertiesToAllIsolines = c(),
  propertiesPerIsoline = list(), resolution = NULL, lint = FALSE)
```

Arguments

- `points` Input points. a point grid, e.g., output of `lawn_point_grid()`
- `breaks` (numeric) Where to draw contours.
- `z` (character) The property name in points from which z-values will be pulled.
- `propertiesToAllIsolines` GeoJSON properties passed to ALL isolines
- `propertiesPerIsoline` GeoJSON properties passed, in order, to the correspondent isoline; the breaks array will define the order in which the isolines are created
- `resolution` (numeric) Resolution of the underlying grid. THIS PARAMETER IS DEFUNCT
- `lint` (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

Warning: this function seems to be broken, not sure why

Value

A data-FeatureCollection of isolines (data-LineString features).

See Also

Other interpolation: `lawn_hex_grid`, `lawn_planepoint`, `lawn_point_grid`, `lawn_square_grid`, `lawn_tin`, `lawn_triangle_grid`

Examples

```
## Not run:
# pts <- lawn_random(n = 100, bbox = c(0, 30, 20, 50))
pts <- lawn_point_grid(c(0, 30, 20, 50), 100, 'miles')
pts$features$properties <-
  data.frame(temperature = round(rnorm(NROW(pts$features), mean = 5)),
  stringsAsFactors = FALSE)
breaks <- c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
lawn_isolines(points = pts, breaks, z = 'temperature')

lawn_isolines(pts, breaks, 'temperature') %>% view

## End(Not run)
```

`lawn_kinks` *Get points at all self-intersections of a polygon*

Description

Get points at all self-intersections of a polygon

Usage

```
lawn_kinks(input, lint = FALSE)
```

Arguments

`input`	Feature of features.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Examples

```
poly <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Polygon",
   "coordinates": [[
      [-12.034835, 8.901183],
      [-12.060413, 8.899826],
      [-12.03638, 8.873199],
      [-12.059383, 8.871418],
      [-12.034835, 8.901183]
    ]]
  }
}'
lawn_kinks(poly)
# lint input object
# lawn_kinks(poly, TRUE)
## Not run:
poly %>% view
lawn_kinks(poly) %>% view

## End(Not run)
```

---

lawn_linestring *Create a linestring*

Description

Create a linestring

Usage

```
lawn_linestring(coordinates, properties = NULL)
```

Arguments

coordinates	A list of positions.
properties	A list of properties.

Value

A data-Feature<(data-LineString)>

See Also

Other data functions: lawn_featurecollection, lawn_feature, lawn_filter, lawn_geometrycollection, lawn_multilinestring, lawn_multipoint, lawn_multipolygon, lawn_point, lawn_polygon, lawn_random, lawn_remove, lawn_sample

Examples

```
linestring1 <- '[
   [-21.964416, 64.148203],
   [-21.956176, 64.141316],
   [-21.93901, 64.135924],
   [-21.927337, 64.136673]
]'
linestring2 <- '[
   [-21.929054, 64.127985],
   [-21.912918, 64.134726],
   [-21.916007, 64.141016],
   [-21.930084, 64.14446]
]'
lawn_linestring(linestring1)
lawn_linestring(linestring2)

pts <- list(
   c(-21.964416, 64.148203),
   c(-21.956176, 64.141316),
   c(-21.93901, 64.135924),
   c(-21.927337, 64.136673)
)
lawn_linestring(pts, properties = list(name = 'line1', distance = 145))

# completely non-sensical, but gets some data quickly
pts <- lawn_random()$features$geometry$coordinates
lawn_linestring(pts)
```

`lawn_line_distance` *Measure a linestring*

Description

Takes a data-LineString and measures its length in the specified units.

Usage

```
lawn_line_distance(line, units, lint = FALSE)
```

Arguments

`line`	Line to measure, a data-Feature<(data-LineString)>, or data-FeatureCollection<(data-LineString)>
`units`	Can be degrees, radians, miles, or kilometers.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

Length of the input line (numeric).

See Also

Other measurements: lawn_along, lawn_area, lawn_bbox_polygon, lawn_bbox, lawn_bearing, lawn_center_of_mass, lawn_center, lawn_centroid, lawn_destination, lawn_distance, lawn_envelope, lawn_extent, lawn_midpoint, lawn_point_on_surface, lawn_pt2line_distance, lawn_square

Examples

```
line <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "LineString",
    "coordinates": [
      [-77.031669, 38.878605],
      [-77.029609, 38.881946],
      [-77.020339, 38.884084],
      [-77.025661, 38.885821],
      [-77.021884, 38.889563],
      [-77.019824, 38.892368]
    ]
  }
}'
lawn_line_distance(line, 'kilometers')
lawn_line_distance(line, 'miles')
lawn_line_distance(line, 'radians')
lawn_line_distance(line, 'degrees')
```

---

lawn_line_offset *Offset a linestring*

Description

Takes a data-LineString and returns a data-LineString at offset by the specified distance.

Usage

```
lawn_line_offset(line, distance, units, lint = FALSE)
```

Arguments

line	Line to measure, a data-LineString.
distance	(integer/numeric) Distance along the line.
units	Can be degrees, radians, miles, kilometers, inches, yards, meters

lint (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-LineString

Examples

```
line <- '{
  "type": "Feature",
  "properties": {
    "stroke": "#F00"
  },
  "geometry": {
    "type": "LineString",
    "coordinates": [[-83, 30], [-84, 36], [-78, 41]]
  }
}'

lawn_line_offset(line, 2, 'miles')
lawn_line_offset(line, 200, 'miles')
lawn_line_offset(line, 0.5, 'radians')
lawn_line_offset(line, 4, 'yards')


line <- '{
    "type": "LineString",
    "coordinates": [[-83, 30], [-84, 36], [-78, 41]]
}'
lawn_line_offset(line, 4, 'yards')
```

---

lawn_line_slice *Slice a line given two points*

Description

Takes a line, a start Point, and a stop point and returns the line in between those points

Usage

```
lawn_line_slice(point1, point2, line, lint = FALSE)
```

Arguments

point1 Starting data-Feature<(data-Point)>

point2 Stopping data-Feature<(data-Point)>

line Line to slice, a data-Feature<(data-LineString)>

`lint` (logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-Feature<(data-LineString)>

Examples

```
start <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [-77.029609, 38.881946]
  }
}'
stop <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [-77.021884, 38.889563]
  }
}'
line <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "LineString",
   "coordinates": [
     [-77.031669, 38.878605],
     [-77.029609, 38.881946],
     [-77.020339, 38.884084],
     [-77.025661, 38.885821],
     [-77.021884, 38.889563],
     [-77.019824, 38.892368]
    ]
  }
}'
lawn_line_slice(start, stop, line)

# lint input objects
lawn_line_slice(start, stop, line, TRUE)
## Not run:
line %>% view
lawn_line_slice(point1 = start, point2 = stop, line) %>% view

## End(Not run)
```

lawn_line_slice_along *Slice a line given two points*

Description

Takes a line, a specified distance along the line to a start Point, and a specified distance along the line to a stop point and returns a subsection of the line in-between those points. This can be useful for extracting only the part of a route between two distances.

Usage

```
lawn_line_slice_along(startDist, stopDist, line, units = "kilometers",
  lint = FALSE)
```

Arguments

startDist	(numeric/integer) distance along the line to starting point
stopDist	(numeric/integer) distance along the line to ending point
line	Line to slice, a data-Feature<(data-LineString)>
units	can be degrees, radians, miles, or kilometers (default)
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

A data-LineString, the sliced line

Examples

```
line <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "LineString",
    "coordinates": [
      [ 7.66845703125, 45.058001435398296 ],
      [ 9.20654296875, 45.460130637921004 ],
      [ 11.348876953125, 44.48866833139467 ],
      [ 12.1728515625, 45.43700828867389 ],
      [ 12.535400390625, 43.98491011404692 ],
      [ 12.425537109375, 41.86956082699455 ],
      [ 14.2437744140625, 40.83874913796459 ],
      [ 14.765625, 40.681679458715635 ]
    ]
  }
}'
lawn_line_slice_along(12.5, 25, line)
```

```
## Not run:
line %>% view
lawn_line_slice_along(12.5, 25, line) %>% view

## End(Not run)
```

`lawn_max` *Maximum value of a field among points within polygons*

Description

Calculates the maximum value of a field for a set of data-Point's within a set of data-Polygon's.

Usage

```
lawn_max(polygons, points, in_field, out_field = "max", lint = FALSE)
```

Arguments

`polygons`	a `data-FeatureCollection` of `data-Polygon` features
`points`	a `data-FeatureCollection` of `data-Point` features
`in_field`	(character) the field in input data to analyze
`out_field`	(character) the field in which to store results
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A FeatureCollection of data-Polygon features with properties listed as `out_field`.

See Also

Other aggregations: `lawn_average`, `lawn_collect`, `lawn_count`, `lawn_deviation`, `lawn_median`, `lawn_min`, `lawn_sum`, `lawn_variance`

Examples

```
## Not run:
poly <- lawn_data$polygons_average
pt <- lawn_data$points_average
lawn_max(poly, pt, 'population')

## End(Not run)
```

lawn_median *Median value of a field among points within polygons*

Description

Calculates the **median** value of a field for a set of data-Point's within a set of data-Polygon's.

Usage

```
lawn_median(polygons, points, in_field, out_field = "median",
  lint = FALSE)
```

Arguments

`polygons`	a `data-FeatureCollection` of `data-Polygon` features
`points`	a `data-FeatureCollection` of `data-Point` features
`in_field`	(character) the field in input data to analyze
`out_field`	(character) the field in which to store results
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A FeatureCollection of data-Polygon features with properties listed as `out_field`.

See Also

Other aggregations: `lawn_average`, `lawn_collect`, `lawn_count`, `lawn_deviation`, `lawn_max`, `lawn_min`, `lawn_sum`, `lawn_variance`

Examples

```
## Not run:
poly <- lawn_data$polygons_average
pt <- lawn_data$points_average
lawn_median(polygons=poly, points=pt, in_field='population')

## End(Not run)
```

lawn_merge	*Merge polygons*

Description

Takes a set of data-Polygon's and returns a single merged polygon feature. If the input polygon features are not contiguous, returns a data-MultiPolygon feature.

Usage

```
lawn_merge(fc, lint = FALSE)
```

Arguments

`fc`	Input polygons, as data-FeatureCollection.
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

Merged data-Polygon or multipolygon data-MultiPolygon.

See Also

lawn_union

Other transformations: `lawn_bezier`, `lawn_buffer`, `lawn_concave`, `lawn_convex`, `lawn_difference`, `lawn_intersect`, `lawn_simplify`, `lawn_union`

Examples

```
polygons <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "fill": "#0f0"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[
          [9.994812, 53.549487],
          [10.046997, 53.598209],
          [10.117721, 53.531737],
          [9.994812, 53.549487]
        ]]
      }
```

```
    }, {
      "type": "Feature",
      "properties": {
        "fill": "#00f"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[
          [10.000991, 53.50418],
          [10.03807, 53.562539],
          [9.926834, 53.551731],
          [10.000991, 53.50418]
        ]]
      }
    }
  ]
}'
lawn_merge(polygons)
## Not run:
lawn_featurecollection(polygons) %>% view
lawn_merge(polygons) %>% view

## End(Not run)
```

`lawn_midpoint` *Get a point midway between two points*

Description

Takes two data-Point's and returns a point midway between them

Usage

```
lawn_midpoint(pt1, pt2, lint = FALSE)
```

Arguments

`pt1`	First data-Feature<(data-Point)>
`pt2`	Second data-Feature<(data-Point)>
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-Feature<(data-Point)> midway between pt1 and pt2

See Also

Other measurements: lawn_along, lawn_area, lawn_bbox_polygon, lawn_bbox, lawn_bearing, lawn_center_of_mass, lawn_center, lawn_centroid, lawn_destination, lawn_distance, lawn_envelope, lawn_extent, lawn_line_distance, lawn_point_on_surface, lawn_pt2line_distance, lawn_square

Examples

```
pt1 <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [144.834823, -37.771257]
  }
}'
pt2 <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [145.14244, -37.830937]
  }
}'
lawn_midpoint(pt1, pt2)
## Not run:
lawn_midpoint(pt1, pt2) %>% view
lawn_featurecollection(list(
  lawn_point(jsonlite::fromJSON(pt1)$geometry$coordinates),
  lawn_point(jsonlite::fromJSON(pt2)$geometry$coordinates),
  structure(lawn_midpoint(pt1, pt2), class = "point")
)) %>% view

## End(Not run)
```

---

lawn_min *Minimum value of a field among points within polygons*

Description

Calculates the minimum value of a field for a set of data-Point's within a set of data-Polygon's

Usage

```
lawn_min(polygons, points, in_field, out_field = "min", lint = FALSE)
```

Arguments

`polygons`	a `data-FeatureCollection` of `data-Polygon` features
`points`	a `data-FeatureCollection` of `data-Point` features
`in_field`	(character) the field in input data to analyze
`out_field`	(character) the field in which to store results
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A FeatureCollection of data-Polygon features with properties listed as `out_field`.

See Also

Other aggregations: `lawn_average`, `lawn_collect`, `lawn_count`, `lawn_deviation`, `lawn_max`, `lawn_median`, `lawn_sum`, `lawn_variance`

Examples

```
## Not run:
poly <- lawn_data$polygons_average
pt <- lawn_data$points_average
lawn_min(poly, pt, 'population')

## End(Not run)
```

`lawn_multilinestring` *Create a multilinestring*

Description

Create a multilinestring

Usage

```
lawn_multilinestring(coordinates, properties = NULL)
```

Arguments

`coordinates`	A list of positions.
`properties`	A list of properties.

Value

A data-Feature<(data-MultiLineString)>

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_random`, `lawn_remove`, `lawn_sample`

Examples

```
mlstr <- '[
  [
   [-21.964416, 64.148203],
   [-21.956176, 64.141316],
   [-21.93901, 64.135924],
   [-21.927337, 64.136673]
  ],
  [
   [-21.929054, 64.127985],
   [-21.912918, 64.134726],
   [-21.916007, 64.141016],
   [-21.930084, 64.14446]
  ]
]'
lawn_multilinestring(mlstr)

lawn_multilinestring(mlstr,
  properties = list(name = 'line1', distance = 145))

# Make a FeatureCollection
lawn_featurecollection(lawn_multilinestring(mlstr))

## Not run:
lawn_featurecollection(lawn_multilinestring(mlstr)) %>% view

## End(Not run)
```

`lawn_multipoint` *MultiPoint*

Description

Create a multipoint

Usage

```
lawn_multipoint(coordinates, properties = NULL)
```

Arguments

`coordinates`	A list of point pairs, either as a list or json, of the form e.g. `list(c(longitude, latitude), c(longitu` or as JSON e.g. `[[longitude, latitude], [longitude, latitude]]`.
`properties`	A list of properties. Default: `NULL`

Value

A data-Feature<(data-MultiPoint)>

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_random`, `lawn_remove`, `lawn_sample`

Examples

```
lawn_multipoint(list(c(-74.5, 40), c(-77.5, 45)))
lawn_multipoint("[[-74.5,40],[-77.5,45]]")
identical(
  lawn_multipoint(list(c(-74.5, 40), c(-77.5, 45))),
  lawn_multipoint("[[-74.5,40],[-77.5,45]]")
)
lawn_multipoint("[[-74.5,40],[-77.5,45]]",
  properties = list(city = 'Boston', population = 400))

# Make a FeatureCollection
lawn_featurecollection(
  lawn_multipoint(list(c(-74.5, 40), c(-77.5, 45)))
)
```

`lawn_multipolygon` *Create a multipolygon*

Description

Create a multipolygon

Usage

```
lawn_multipolygon(coordinates, properties = NULL)
```

Arguments

`coordinates`	A list of LinearRings, or in json.
`properties`	A list of properties.

Value

A data-Feature<(data-MultiPolygon)>

See Also

Other data functions: lawn_featurecollection, lawn_feature, lawn_filter, lawn_geometrycollection, lawn_linestring, lawn_multilinestring, lawn_multipoint, lawn_point, lawn_polygon, lawn_random, lawn_remove, lawn_sample

Examples

```
rings <- list(
  list(list(
   c(-2.27, 53.46),
   c(-2.27, 53.48),
   c(-2.21, 53.48),
   c(-2.21, 53.46),
   c(-2.27, 53.46)
  )),
  list(list(
   c(-4.27, 55.46),
   c(-4.27, 55.48),
   c(-4.21, 55.48),
   c(-4.21, 55.46),
   c(-4.27, 55.46)
  ))
)
lawn_multipolygon(rings)
lawn_multipolygon(rings, properties = list(name = 'poly1', population = 400))

x <- '[
  [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
  [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
  [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
]'
lawn_multipolygon(x)

lawn_multipolygon("[[[[0,0],[0,10],[10,10],[10,0],[0,0]]]]")

# Make a FeatureCollection
lawn_featurecollection(lawn_multipolygon(rings))

## Not run:
lawn_featurecollection(lawn_multipolygon(rings)) %>% view

## End(Not run)
```

---

lawn_nearest *Get nearest point*

Description

Takes a reference data-Point and a set of points to compare it against and returns the point from the set closest to the reference

Usage

```
lawn_nearest(point, against, lint = FALSE)
```

Arguments

`point`	The reference point, a data-Feature<(data-Point)>
`against`	Input point set, a data-FeatureCollection
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-Feature<(data-Point)>

Examples

```
point <- '{
  "type": "Feature",
  "properties": {
    "marker-color": "#0f0"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [28.965797, 41.010086]
  }
}'
against <- '{
 "type": "FeatureCollection",
 "features": [
   {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [28.973865, 41.011122]
     }
   }, {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [28.948459, 41.024204]
     }
   }, {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [28.938674, 41.013324]
     }
```

```
    }
  ]
}'
lawn_nearest(point, against)

## Not run:
lawn_nearest(point, against) %>% view

## End(Not run)
```

---

lawn_planepoint *Calculate a Planepoint*

Description

Takes a triangular plane as a data-Polygon and a data-Point within that triangle and returns the z-value at that point.

Usage

```
lawn_planepoint(pt, triangle, lint = FALSE)
```

Arguments

pt	The Point for which a z-value will be calculated.
triangle	A Polygon feature with three vertices.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

The Polygon needs to have properties a, b, and c that define the values at its three corners.

Value

The z-value for pt (numeric).

See Also

Other interpolation: `lawn_hex_grid`, `lawn_isolines`, `lawn_point_grid`, `lawn_square_grid`, `lawn_tin`, `lawn_triangle_grid`

Examples

```
pt <- lawn_point(c(-75.3221, 39.529))
triangle <- '{
  "type": "Feature",
  "properties": {
    "a": 11,
    "b": 122,
    "c": 44
  },
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [-75.1221, 39.57],
      [-75.58, 39.18],
      [-75.97, 39.86],
      [-75.1221, 39.57]
    ]]
  }
}'
lawn_planepoint(pt, triangle)
```

`lawn_point` *Create a point*

Description

Create a point

Usage

```
lawn_point(coordinates, properties = NULL)
```

Arguments

`coordinates`	A pair of points in a vector, list or json, of the form `e.g., c(longitude,latitude)`.
`properties`	A list of properties. Default: NULL

Value

A data-Feature<(data-Point)>

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_polygon`, `lawn_random`, `lawn_remove`, `lawn_sample`

Examples

```
lawn_point(c(-74.5, 40))
lawn_point(list(-74.5, 40))
lawn_point('[-74.5, 40]')
lawn_point(c(-74.5, 40), properties = list(name = 'poly1', population = 400))

# Make a FeatureCollection
lawn_featurecollection(lawn_point(c(-74.5, 40)))
```

`lawn_point_grid` *Create a PointGrid*

Description

Takes a bounding box and a cell depth and returns a set of data-Point's in a grid

Usage

```
lawn_point_grid(extent, cellSide, units = "kilometers",
  centered = TRUE, bboxIsMask = FALSE)
```

Arguments

`extent`	(numeric) Extent in `[minX, minY, maxX, maxY]` order.
`cellSide`	(integer) the distance between points
`units`	(character) Units to use for cellWidth, one of 'miles' or 'kilometers' (default).
`centered`	(logical) adjust points position to center the grid into bbox. This parameter is going to be removed in the next major release, having the output always centered into bbox. Default: `TRUE`
`bboxIsMask`	if `TRUE`, and bbox is a Polygon or MultiPolygon, the grid Point will be created only if inside the bbox Polygon(s). Default: `FALSE`

Value

data-FeatureCollection grid of points.

See Also

Other interpolation: `lawn_hex_grid`, `lawn_isolines`, `lawn_planepoint`, `lawn_square_grid`, `lawn_tin`, `lawn_triangle_grid`

Examples

```
lawn_point_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 30, 'miles')
lawn_point_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 10, 'miles')
lawn_point_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 3, 'miles')
```

lawn_point_on_line *Get closest point on linestring to reference point*

Description

Takes a line, a start data-Point, and a stop point and returns the line in between those points

Usage

```
lawn_point_on_line(line, point, lint = FALSE)
```

Arguments

`line`	data-Feature<(data-LineString)> to snap to
`point`	data-Feature<(data-Point)> to snap from
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-Feature<(data-Point)>

Examples

```
line <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "LineString",
    "coordinates": [
      [-77.031669, 38.878605],
      [-77.029609, 38.881946],
      [-77.020339, 38.884084],
      [-77.025661, 38.885821],
      [-77.021884, 38.889563],
      [-77.019824, 38.892368]
    ]
  }
}'
pt <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [-77.037076, 38.884017]
  }
}'
lawn_point_on_line(line, pt)
```

```
# lint input objects
lawn_point_on_line(line, pt, TRUE)
## Not run:
line %>% view
pt %>% view
lawn_point_on_line(line, pt) %>% view

## End(Not run)
```

`lawn_point_on_surface` *Get a point on the surface of a feature*

Description

Finds a data-Point guaranteed to be on the surface of data-GeoJSON object.

Usage

```
lawn_point_on_surface(x, lint = FALSE)
```

Arguments

`x`	Any data-GeoJSON object
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

What will be returned?

- Given a data-Polygon, the point will be in the area of the polygon
- Given a data-LineString, the point will be along the string
- Given a data-Point, the point will be the same as the input

Value

A data-Feature<(data-Point)> on the surface of `x`

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_pt2line_distance`, `lawn_square`

Examples

```
# polygon
x <- lawn_random("polygon")
lawn_point_on_surface(x)
# point
x <- lawn_random("point")
lawn_point_on_surface(x)
# linestring
linestring <- '[
   [-21.929054, 64.127985],
   [-21.912918, 64.134726],
   [-21.916007, 64.141016],
   [-21.930084, 64.14446]
]'
lawn_point_on_surface(lawn_linestring(linestring))
```

---

lawn_polygon *Create a polygon*

Description

Create a polygon

Usage

```
lawn_polygon(coordinates, properties = NULL)
```

Arguments

coordinates	A list of LinearRings, or in json.
properties	A list of properties.

Value

A data-Polygon feature.

See Also

Other data functions: lawn_featurecollection, lawn_feature, lawn_filter, lawn_geometrycollection, lawn_linestring, lawn_multilinestring, lawn_multipoint, lawn_multipolygon, lawn_point, lawn_random, lawn_remove, lawn_sample

Examples

```
rings <- list(list(
   c(-2.275543, 53.464547),
   c(-2.275543, 53.489271),
   c(-2.215118, 53.489271),
   c(-2.215118, 53.464547),
   c(-2.275543, 53.464547)
))
lawn_polygon(rings)
lawn_polygon(rings, properties = list(name = 'poly1', population = 400))

# Make a FeatureCollection
lawn_featurecollection(lawn_polygon(rings))

## Not run:
lawn_featurecollection(lawn_polygon(rings)) %>% view

## End(Not run)
```

---

lawn_propeach *Iterate over property objects in any GeoJSON object*

Description

Iterate over property objects in any GeoJSON object

Usage

```
lawn_propeach(x, fun = NULL, lint = FALSE)
```

Arguments

`x`	any data-GeoJSON object
`fun`	a Javascript function. if not given, returns self
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

matrix of coordinates, where each row in the matrix is a coordinate pair

Examples

```
x <- "{ type: 'Feature', geometry: null, properties: { foo: 1, bar: 3 } }"

# don't apply any function, identity essentially
lawn_propeach(x)

# appply a function callback
lawn_propeach(x, "z.foo === 1")

lawn_propeach(lawn_data$points_count)

z <- '{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "population": 200,
        "name": "things"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-112.0372, 46.608058]
      }
    }, {
      "type": "Feature",
      "properties": {
        "population": 600,
        "name": "stuff"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-112.045955, 46.596264]
      }
    }
    ]
}'
lawn_propeach(z)
lawn_propeach(z, "z.population === 200")
lawn_propeach(z, "z.name === 'stuff'")
```

`lawn_pt2line_distance` *Minimum distance between a point and a lineString*

Description

Returns the minimum distance between a data-Point and a data-LineString, being the distance from a line the minimum distance between the point and any segment of the LineString.

Usage

```
lawn_pt2line_distance(point, line, units = "kilometers",
  mercator = FALSE, lint = FALSE)
```

Arguments

`point`	(data-Feature<(data-Point)>) feature or geometry
`line`	Line to measure, a data-Feature<(data-LineString)>, or data-FeatureCollection<(data-LineString)>
`units`	(character) Can be degrees, radians, miles, or kilometers (default)
`mercator`	(logical) if distance should be on Mercator or WGS84 projection. Default: `FALSE`
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

distance between point and line (numeric)

See Also

Other measurements: `lawn_along`, `lawn_area`, `lawn_bbox_polygon`, `lawn_bbox`, `lawn_bearing`, `lawn_center_of_mass`, `lawn_center`, `lawn_centroid`, `lawn_destination`, `lawn_distance`, `lawn_envelope`, `lawn_extent`, `lawn_line_distance`, `lawn_midpoint`, `lawn_point_on_surface`, `lawn_square`

Examples

```
pt <- lawn_point("[0, 0]")
ln <- lawn_linestring("[[1, 1],[-1, 1]]")

lawn_pt2line_distance(pt, ln)
lawn_pt2line_distance(pt, ln, mercator = TRUE)

lawn_pt2line_distance(pt, ln, 'miles')
lawn_pt2line_distance(pt, ln, 'radians')
lawn_pt2line_distance(pt, ln, 'degrees')
lawn_pt2line_distance(pt, ln, mercator = TRUE)
```

`lawn_random` *Generate random data*

Description

Generates random data-GeoJSON data, including data-Point's and data-Polygon's, for testing and experimentation

Usage

```
lawn_random(type = "points", n = 10, bbox = NULL,
  num_vertices = NULL, max_radial_length = NULL)
```

Arguments

`type`	Type of features desired: 'points' or 'polygons'.
`n`	(integer) Number of features to generate.
`bbox`	A bounding box inside of which geometries are placed. In the case of Point features, they are guaranteed to be within this bounds, while Polygon features have their centroid within the bounds.
`num_vertices`	Number options.vertices the number of vertices added to polygon features.
`max_radial_length`	Number <optional> 10 The total number of decimal degrees longitude or latitude that a polygon can extent outwards to from its center.

Value

A data-FeatureCollection.

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_remove`, `lawn_sample`

Examples

```
## set of points
lawn_random(n = 2)
lawn_random(n = 10)
## set of polygons
lawn_random('polygons', 2)
lawn_random('polygons', 10)
# with options
lawn_random(bbox = c(-70, 40, -60, 60))
lawn_random(num_vertices = 5)
```

`lawn_remove` *Remove things from a FeatureCollection*

Description

Takes a data-FeatureCollection of any type, a property, and a value and returns a data-FeatureCollection with features matching that property-value pair removed.

Usage

```
lawn_remove(features, property, value, lint = FALSE)
```

Arguments

features	A set of input features.
property	Property to filter.
value	Value to filter.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-FeatureCollection.

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_random`, `lawn_sample`

Examples

```
cat(lawn_data$remove_features)
lawn_remove(lawn_data$remove_features, 'marker-color', '#00f')
lawn_remove(lawn_data$remove_features, 'marker-color', '#0f0')
```

`lawn_rewind` *Rewind*

Description

Rewind (Multi)LineString or (Multi)Polygon outer ring counterclockwise and inner rings clockwise (Uses Shoelace Formula (`https://en.wikipedia.org/wiki/Shoelace_formula`)).

Usage

```
lawn_rewind(x, reverse = FALSE, mutate = FALSE, lint = FALSE)
```

Arguments

x	A data-FeatureCollection or data-Feature with Polygon, MultiPolygon, LineString, or MultiLineString
reverse	(logical) enable reverse winding. Default: FALSE
mutate	(logical) allows GeoJSON input to be mutated (significant performance increase if true) Default: FALSE
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

A data-FeatureCollection

Examples

```
x <- '{
"type": "Feature",
"properties": {},
"geometry": {
  "type": "Polygon",
  "coordinates": [
    [[121, -29], [138, -29], [138, -18], [121, -18], [121, -29]]
   ]
  }
}'
lawn_rewind(x, TRUE)
lawn_rewind(x, mutate = TRUE)
lawn_rewind(x, lint = TRUE)
```

---

lawn_sample *Return features from FeatureCollection at random*

Description

Takes a data-FeatureCollection and returns a data-FeatureCollection with given number of features at random.

Usage

```
lawn_sample(features = NULL, n = 100, lint = FALSE)
```

Arguments

features	A data-FeatureCollection
n	(integer) Number of features to generate.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

A data-FeatureCollection

See Also

Other data functions: `lawn_featurecollection`, `lawn_feature`, `lawn_filter`, `lawn_geometrycollection`, `lawn_linestring`, `lawn_multilinestring`, `lawn_multipoint`, `lawn_multipolygon`, `lawn_point`, `lawn_polygon`, `lawn_random`, `lawn_remove`

Examples

```
lawn_sample(lawn_data$points_average, 1)
lawn_sample(lawn_data$points_average, 2)
lawn_sample(lawn_data$points_average, 3)
```

`lawn_simplify` *Simplify GeoJSON data*

Description

Takes a data-LineString or data-Polygon and returns a simplified version.

Usage

```
lawn_simplify(feature, tolerance = 0.01, high_quality = FALSE,
  lint = FALSE)
```

Arguments

`feature`	A data-Feature<(data-LineString, data-Polygon, data-MultiLineString, data-MultiPolygon)>, or data-FeatureCollection, or data-GeometryCollection
`tolerance`	(numeric) Simplification tolerance. Default value is 0.01.
`high_quality`	(boolean) Whether or not to spend more time to create a higher-quality simplification with a different algorithm. Default: `FALSE`
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

Internally uses simplify-js (`http://mourner.github.io/simplify-js/`) to perform simplification.

Value

A simplified feature.

A Feature of either data-Polygon or data-LineString.

See Also

Other transformations: lawn_bezier, lawn_buffer, lawn_concave, lawn_convex, lawn_difference, lawn_intersect, lawn_merge, lawn_union

Examples

```
feature <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [-70.603637, -33.399918],
      [-70.614624, -33.395332],
      [-70.639343, -33.392466],
      [-70.659942, -33.394759],
      [-70.683975, -33.404504],
      [-70.697021, -33.419406],
      [-70.701141, -33.434306],
      [-70.700454, -33.446339],
      [-70.694274, -33.458369],
      [-70.682601, -33.465816],
      [-70.668869, -33.472117],
      [-70.646209, -33.473835],
      [-70.624923, -33.472117],
      [-70.609817, -33.468107],
      [-70.595397, -33.458369],
      [-70.587158, -33.442901],
      [-70.587158, -33.426283],
      [-70.590591, -33.414248],
      [-70.594711, -33.406224],
      [-70.603637, -33.399918]
    ]]
  }
}'

lawn_simplify(feature, tolerance = 0.01)
## Not run:
lawn_simplify(feature, tolerance = 0.01) %>% view

## End(Not run)
```

---

lawn_square — *Calculate a square bounding box*

Description

Takes a bounding box and calculates the minimum square bounding box that would contain the input.

Usage

```
lawn_square(bbox)
```

Arguments

bbox	A bounding box.

Value

A square surrounding bbox, numeric vector of length four.

See Also

Other measurements: lawn_along, lawn_area, lawn_bbox_polygon, lawn_bbox, lawn_bearing, lawn_center_of_mass, lawn_center, lawn_centroid, lawn_destination, lawn_distance, lawn_envelope, lawn_extent, lawn_line_distance, lawn_midpoint, lawn_point_on_surface, lawn_pt2line_distance

Examples

```
bbox <- c(-20, -20, -15, 0)
lawn_square(bbox)
## Not run:
sq <- lawn_square(bbox)
lawn_featurecollection(list(lawn_bbox_polygon(bbox),
  lawn_bbox_polygon(sq))) %>% view

## End(Not run)
```

---

lawn_square_grid *Create a SquareGrid*

Description

Takes a bounding box and a cell depth and returns a set of square data-Polygon's in a grid.

Usage

```
lawn_square_grid(extent, cellWidth, units)
```

Arguments

extent	(numeric) Extent in [minX, minY, maxX, maxY] order.
cellWidth	(integer) Width of each cell.
units	(character) Units to use for cellWidth, one of 'miles' or 'kilometers'.

Value

data-FeatureCollection grid of polygons.

See Also

Other interpolation: lawn_hex_grid, lawn_isolines, lawn_planepoint, lawn_point_grid, lawn_tin, lawn_triangle_grid

Examples

```
lawn_square_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 30, 'miles')
lawn_square_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 10, 'miles')
lawn_square_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 3, 'miles')
```

---

lawn_sum — *Sum of a field among points within polygons*

Description

Calculates the sum of a field for a set of data-Point's within a set of data-Polygon's.

Usage

```
lawn_sum(polygons, points, in_field, out_field = "sum", lint = FALSE)
```

Arguments

polygons	a data-FeatureCollection of data-Polygon features
points	a data-FeatureCollection of data-Point features
in_field	(character) the field in input data to analyze
out_field	(character) the field in which to store results
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

A FeatureCollection of data-Polygon features with properties listed as out_field.

See Also

Other aggregations: lawn_average, lawn_collect, lawn_count, lawn_deviation, lawn_max, lawn_median, lawn_min, lawn_variance

Examples

```
## Not run:
poly <- lawn_data$polygons_average
pt <- lawn_data$points_average
lawn_sum(poly, pt, 'population')

## End(Not run)
```

---

lawn_tag *Spatial join of points and polygons*

Description

Takes a set of data-Point's and a set of data-Polygon's and performs a spatial join.

Usage

```
lawn_tag(points, polygons, field, out_field, lint = FALSE)
```

Arguments

points	Input data-FeatureCollection<(data-Point)>
polygons	Input data-FeatureCollection<(data-Polygon)> or data-FeatureCollection<(data-MultiPolygon)>
field	Property in polygons to add to joined Point features.
out_field	Property in points in which to store joined property from polygons.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

Points with containing_polyid property containing values from poly_id, as data-FeatureCollection<(data-Point)>

See Also

Other joins: `lawn_inside`, `lawn_within`

Examples

```
bbox <- c(0, 0, 10, 10)
pts <- lawn_random(n = 30, bbox = bbox)
polys <- lawn_triangle_grid(bbox, 50, 'miles')
polys$features$properties$fill <- "#f92"
polys$features$properties$stroke <- 0
polys$features$properties$`fill-opacity` <- 1
lawn_tag(pts, polys, 'fill', 'marker-color')
## Not run:
lawn_tag(pts, polys, 'fill', 'marker-color') %>% view

## End(Not run)
```

`lawn_tesselate` *Tesselate*

Description

Tesselates a data-Polygon into a data-FeatureCollection of triangles using earcut (https://github.com/mapbox/earcut)

Usage

```
lawn_tesselate(polygon, lint = FALSE)
```

Arguments

polygon	Input data-Feature<(data-Polygon)>
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

A data-FeatureCollection

See Also

Other assertions: `lawn_circle`, `lawn_dissolve`

Examples

```
poly <- '{
 "type": "Feature",
  "properties": {
    "fill": "#0f0"
  },
  "geometry": {
    "type": "Polygon",
    "coordinates": [[
      [-46.738586, -23.596711],
      [-46.738586, -23.458207],
      [-46.560058, -23.458207],
      [-46.560058, -23.596711],
      [-46.738586, -23.596711]
    ]]
  }
}'
lawn_tesselate(poly)

xx <- jsonlite::fromJSON(lawn_data$polygons_within, FALSE)
lawn_tesselate(xx$features[[1]])
```

```
## Not run:
lawn_tesselate(xx$features[[1]]) %>% view
lawn_tesselate(poly) %>% view

## End(Not run)
```

---

lawn_tin *Create a Triangulated Irregular Network*

Description

Takes a set of data-Point's and the name of a z-value property and creates a Triangulated Irregular Network (TIN).

Usage

```
lawn_tin(pt, propertyName = NULL, lint = FALSE)
```

Arguments

pt	Input points.
propertyName	(character) Name of the property from which to pull z values. This is optional: if not given, then there will be no extra data added to the derived triangles
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

Data returned as a collection of Polygons. These are often used for developing elevation contour maps or stepped heat visualizations.

This triangulates the points, as well as adds properties called a, b, and c representing the value of the given propertyName at each of the points that represent the corners of the triangle.

Value

TIN output, as a data-FeatureCollection.

See Also

Other interpolation: `lawn_hex_grid`, `lawn_isolines`, `lawn_planepoint`, `lawn_point_grid`, `lawn_square_grid`, `lawn_triangle_grid`

Examples

```
pts <- lawn_random(bbox = c(-70, 40, -60, 60))
lawn_tin(pts)
## Not run:
lawn_tin(pts) %>% view
lawn_tin(lawn_random(bbox = c(-70, 40, -60, 10))) %>% view

## End(Not run)
```

---

lawn_transform_rotate *Rotate a GeoJSON feature*

Description

Rotates any geojson Feature or Geometry of a specified angle, around its centroid or a given pivot point

Usage

```
lawn_transform_rotate(x, angle, pivot = c(0, 0), mutate = FALSE,
  lint = FALSE)
```

Arguments

`x`	a feature
`angle`	(integer/numeric) number of rotation (along the vertical axis), from North in decimal degrees, negative clockwise
`pivot`	(integer/numeric) point around which the rotation will be performed (optional, default `centroid`)
`mutate`	(logical) allows GeoJSON input to be mutated (significant performance increase if true) (optional). Default: `FALSE`
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a rotated data-Feature

Note

all rotations follow the right-hand rule: https://en.wikipedia.org/wiki/Right-hand_rule

Examples

```
x <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Polygon",
   "coordinates": [
    [
      [ 0, 29 ], [ 3.5, 29 ], [ 2.5, 32 ], [ 0, 29 ]
    ]
  ]
 }
}'
lawn_transform_rotate(x, angle = 100, pivot = c(15, 15))

lawn_transform_rotate(x, angle = 100)
lawn_transform_rotate(x, angle = 100, mutate = TRUE)

## Not run:
view(lawn_featurecollection(x))
view(lawn_featurecollection(lawn_transform_rotate(x, angle = 100)))
view(lawn_featurecollection(
  lawn_transform_rotate(x, angle = 100, pivot = c(15, 15))
))
view(lawn_featurecollection(
  lawn_transform_rotate(x, angle = 150, pivot = c(15, 15))
))
view(lawn_featurecollection(
  lawn_transform_rotate(x, angle = 300, pivot = c(0, 4))
))

## End(Not run)
```

`lawn_transform_scale` *Scale a GeoJSON feature*

Description

Scale a GeoJSON from a given point by a factor of scaling (ex: factor=2 would make the GeoJSON 200 the origin point will be calculated based on each individual Feature.

Usage

```
lawn_transform_scale(x, factor, origin = "centroid", mutate = FALSE,
  lint = FALSE)
```

Arguments

x	a feature
factor	(integer/numeric) of scaling, positive or negative values greater than 0
origin	(integer/numeric) Point from which the scaling will occur (string options: sw/se/nw/ne/center/centroid) (optional, default "centroid")
mutate	(logical) allows GeoJSON input to be mutated (significant performance increase if true) (optional). Default: `FALSE`
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a scaled data-Feature

Examples

```
x <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Polygon",
   "coordinates": [
    [
      [ 0, 29 ], [ 3.5, 29 ], [ 2.5, 32 ], [ 0, 29 ]
    ]
  ]
 }
}'
lawn_transform_scale(x, factor = 3)

lawn_transform_scale(x, factor = 100)
lawn_transform_scale(x, factor = 100, mutate = TRUE)

## Not run:
view(lawn_featurecollection(x))
view(lawn_featurecollection(
  lawn_transform_scale(x, factor = 2)
))
view(lawn_featurecollection(
  lawn_transform_scale(x, factor = 3)
))
view(lawn_featurecollection(
  lawn_transform_scale(x, factor = 2, origin = "sw")
))
view(lawn_featurecollection(
  lawn_transform_scale(x, factor = 2, origin = "ne")
))

## End(Not run)
```

lawn_transform_translate
Translate a GeoJSON feature

Description

Moves any geojson Feature or Geometry of a specified distance along a Rhumb Line on the provided direction angle.

Usage

```
lawn_transform_translate(x, distance, direction, units = "kilometers",
  zTranslation = 0, mutate = FALSE, lint = FALSE)
```

Arguments

`x`	a feature
`distance`	(integer/numeric) length of the motion; negative values determine motion in opposite direction
`direction`	(integer/numeric) of the motion; angle from North in decimal degrees, positive clockwise
`units`	(character) in which distance will be express; miles, kilometers, degrees, or radians (optional, default kilometers)
`zTranslation`	(integer/numeric) length of the vertical motion, same unit of distance (optional, default 0)
`mutate`	(logical) allows GeoJSON input to be mutated (significant performance increase if true) (optional). Default: `FALSE`
`lint`	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a changed data-Feature

Examples

```
x <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
   "type": "Polygon",
   "coordinates": [
    [
      [ 0, 29 ], [ 3.5, 29 ], [ 2.5, 32 ], [ 0, 29 ]
    ]
```

```
  ]
 }
}'
lawn_transform_translate(x, distance = 100, direction = 35)

lawn_transform_translate(x, distance = 100, direction = 24)
lawn_transform_translate(x, distance = 100, direction = 24, mutate = TRUE)

## Not run:
view(lawn_featurecollection(x))
view(lawn_featurecollection(
  lawn_transform_translate(x, distance = 130, direction = 35,
   units = "kilometers")
))
view(lawn_featurecollection(
  lawn_transform_translate(x, distance = 130, direction = -35,
   units = "kilometers")
))
view(lawn_featurecollection(
  lawn_transform_translate(x, distance = 130, direction = 35,
   units = "kilometers", zTranslation = 10)
))
view(lawn_featurecollection(
  lawn_transform_translate(x, distance = 130, direction = 35,
   units = "kilometers", mutate = TRUE)
))

## End(Not run)
```

`lawn_triangle_grid` *Create a TriangleGrid*

Description

Takes a bounding box and a cell depth and returns a set of triangular data-Polygon's in a grid.

Usage

```
lawn_triangle_grid(extent, cellWidth, units)
```

Arguments

`extent`	(numeric) Extent in `[minX, minY, maxX, maxY]` order.
`cellWidth`	(integer) Width of each cell.
`units`	(character) Units to use for cellWidth, one of 'miles' or 'kilometers'.

Value

data-FeatureCollection grid of data-Polygon's

See Also

Other interpolation: lawn_hex_grid, lawn_isolines, lawn_planepoint, lawn_point_grid, lawn_square_grid, lawn_tin

Examples

```
lawn_triangle_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 30, 'miles')
lawn_triangle_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 10, 'miles')
lawn_triangle_grid(c(-77.3876, 38.7198, -76.9482, 39.0277), 3, 'miles')
```

---

lawn_truncate *Truncate*

Description

Takes a GeoJSON Feature or FeatureCollection and truncates the precision of the geometry.

Usage

```
lawn_truncate(x, precision = 6, coordinates = 2, lint = FALSE)
```

Arguments

x	any data-Feature or data-FeatureCollection
precision	(integer) coordinate decimal precision. default: 6
coordinates	(integer) maximum number of coordinates (primarily used to remove z coordinates). default: 2
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

a data-Feature or data-FeatureCollection with truncated geometry

See Also

Other misc: lawn_flatten

Examples

```
cat(lawn_data$filter_features)
lawn_coordall(lawn_data$filter_features)
lawn_truncate(lawn_data$filter_features, 4) %>% lawn_coordall
lawn_truncate(lawn_data$filter_features, 2) %>% lawn_coordall
lawn_truncate(lawn_data$filter_features, 4, 1) %>% lawn_coordall
```

lawn_union *Merge polygons*

Description

Finds the intersection of two data-Polygon's and returns the union of the two

Usage

```
lawn_union(poly1, poly2, lint = FALSE)
```

Arguments

poly1	A data-Feature<(data-Polygon)>
poly2	A data-Feature<(data-Polygon)>
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Details

Contiguous polygons are combined, non-contiguous polygons are returned as MultiPolygon.

Value

data-Feature<(data-Polygon)> or data-Feature<(data-MultiPolygon)>

Author(s)

Jeff Hollister <hollister.jeff@epa.gov>

See Also

lawn_merge

Other transformations: `lawn_bezier`, `lawn_buffer`, `lawn_concave`, `lawn_convex`, `lawn_difference`, `lawn_intersect`, `lawn_merge`, `lawn_simplify`

Examples

```
## Not run:
poly1 <- '{
 "type": "Feature",
 "properties": {
   "fill": "#0f0"
 },
 "geometry": {
   "type": "Polygon",
   "coordinates": [[
```

```
      [-122.801742, 45.48565],
      [-122.801742, 45.60491],
      [-122.584762, 45.60491],
      [-122.584762, 45.48565],
      [-122.801742, 45.48565]
     ]]
  }
}'

poly2 <- '{
 "type": "Feature",
 "properties": {
   "fill": "#00f"
 },
 "geometry": {
   "type": "Polygon",
   "coordinates": [[
     [-122.520217, 45.535693],
     [-122.64038, 45.553967],
     [-122.720031, 45.526554],
     [-122.669906, 45.507309],
     [-122.723464, 45.446643],
     [-122.532577, 45.408574],
     [-122.487258, 45.477466],
     [-122.520217, 45.535693]
     ]]
 }
}'
lawn_union(poly1, poly2)

view(poly1)
view(poly2)
lawn_union(poly1, poly2) %>% view()

x1 <- lawn_buffer(lawn_point(c(-122.6375, 45.53)), 1500, "meters")
x2 <- lawn_buffer(lawn_point(c(-122.6475, 45.53)), 1500, "meters")
lawn_union(x1, x2)
view(x1)
view(x2)
lawn_union(x1, x2) %>% view()

## End(Not run)
```

`lawn_unkinkpolygon` *Unkink polygon*

Description

Takes a kinked polygon and returns a feature collection of polygons that have no kinks.

Usage

```
lawn_unkinkpolygon(x, lint = FALSE)
```

Arguments

x	A data-FeatureCollection<(data-Polygon)> or data-FeatureCollection<(data-MultiPolygon)>
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: `FALSE`

Value

a data-FeatureCollection<(data-Polygon)>

See Also

Other grids: `lawn_idw`

Examples

```
x <- '{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
      "coordinates": [[[0, 0], [2, 0], [0, 2], [2, 2], [0, 0]]]
   }
}'
lawn_unkinkpolygon(x)
view(x)
view(lawn_unkinkpolygon(x))
```

`lawn_variance` *Variance of a field among points within polygons*

Description

Calculates the variance value of a field for a set of data-Point's within a set of data-Polygon's.

Usage

```
lawn_variance(polygons, points, in_field, out_field = "variance",
  lint = FALSE)
```

Arguments

polygons	a data-FeatureCollection of data-Polygon features
points	a data-FeatureCollection of data-Point features
in_field	(character) the field in input data to analyze
out_field	(character) the field in which to store results
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

A FeatureCollection of data-Polygon features with properties listed as out_field.

A FeatureCollection of data-Polygon features with properties listed as out_field.

See Also

Other aggregations: lawn_average, lawn_collect, lawn_count, lawn_deviation, lawn_max, lawn_median, lawn_min, lawn_sum

Examples

```
## Not run:
poly <- lawn_data$polygons_average
pt <- lawn_data$points_average
lawn_variance(poly, pt, 'population')

## End(Not run)
```

---

lawn_within *Return points that fall within polygons*

Description

Takes a set of data-Point's and a set of data-Polygon's and returns points that fall within the polygons.

Usage

```
lawn_within(points, polygons, lint = FALSE)
```

Arguments

points	data-FeatureCollection of points.
polygons	data-FeatureCollection of polygons.
lint	(logical) Lint or not. Uses geojsonhint. Takes up increasing time as the object to get linted increases in size, so probably use by default for small objects, but not for large if you know they are good geojson objects. Default: FALSE

Value

Points that land within at least one polygon, as a data-FeatureCollection.

See Also

Other joins: `lawn_inside`, `lawn_tag`

Examples

```
## Not run:
cat(lawn_data$points_within)
cat(lawn_data$polygons_within)
lawn_within(lawn_data$points_within, lawn_data$polygons_within)

pt <- '{
 "type": "Feature",
 "properties": {},
 "geometry": {
    "type": "Point",
    "coordinates": [-90.548630, 14.616599]
  }
}'
poly <- lawn_featurecollection(lawn_buffer(pt, 5))
pts <- lawn_featurecollection(lawn_point(c(-90.55, 14.62)))

lawn_within(pts, poly)

## End(Not run)
```

`print-methods` *Lawn print methods to provide summary view*

Description

Lawn print methods to provide summary view

Arguments

`x`	Input.
`n`	(integer) Number of rows to print, when properties is large object.
`...`	Print options.

Examples

```
# point
lawn_point(c(-74.5, 40))

# polygon
```

```
rings <- list(list(
   c(-2.275543, 53.464547),
   c(-2.275543, 53.489271),
   c(-2.215118, 53.489271),
   c(-2.215118, 53.464547),
   c(-2.275543, 53.464547)
))
lawn_polygon(rings, properties = list(name = 'poly1', population = 400))

# linestring
linestring1 <- '[
   [-21.964416, 64.148203],
   [-21.956176, 64.141316],
   [-21.93901, 64.135924],
   [-21.927337, 64.136673]
]'
lawn_linestring(linestring1)
lawn_linestring(linestring1, properties = list(name = 'line1',
  distance = 145))

# featurecollection
lawn_featurecollection(lawn_data$featurecollection_eg1)

# feature
serbia <- '{
  "type": "Feature",
  "properties": {"color": "red"},
  "geometry": {
    "type": "Point",
    "coordinates": [20.566406, 43.421008]
   }
}'
lawn_flip(serbia)

# multipoint
mpt <- '{
 "type": "FeatureCollection",
 "features": [
   {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [19.026432, 47.49134]
     }
   }, {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "Point",
       "coordinates": [19.074497, 47.509548]
     }
   }
```

```
 ]
}'
x <- lawn_combine(mpt)
x$properties <- data.frame(color = c("red", "green"),
                           size = c("small", "large"),
                           popultion = c(5000, 10000L))
x

# multilinestring
mlstring <- '{
 "type": "FeatureCollection",
 "features": [
   {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "LineString",
       "coordinates": [
         [-21.964416, 64.148203],
         [-21.956176, 64.141316],
         [-21.93901, 64.135924],
         [-21.927337, 64.136673]
       ]
     }
   }, {
     "type": "Feature",
     "properties": {},
     "geometry": {
       "type": "LineString",
       "coordinates": [
         [-21.929054, 64.127985],
         [-21.912918, 64.134726],
         [-21.916007, 64.141016],
         [-21.930084, 64.14446]
       ]
     }
   }
 ]
}'
x <- lawn_combine(mlstring)
x$properties <- data.frame(color = c("red", "green"),
                           size = c("small", "large"),
                           popultion = c(5000, 10000L))
x
```

`view` *Visualize geojson*

Description

Visualize geojson

Usage

```
view(x)

view_(...)
```

Arguments

`x`	Input, a geojson character string or list.
`...`	Any geojson object, as list, json, or point, polygon, etc. class.

Details

`view_` is a special interface to `view` to accept arbitrary input via `...`.

Value

Opens a map with the geojson object(s).

Examples

```
## Not run:
# from character string
view(lawn_data$polygons_average)
view(lawn_data$filter_features)
view(lawn_data$polygons_within)
view(lawn_data$polygons_count)

# from json (a jsonlite class)
library(jsonlite)
x <- minify(lawn_data$points_count)
class(x)
view(x)

# from a list (a single object)
library("jsonlite")
x <- fromJSON(lawn_data$polygons_average, FALSE)
view(x)

# From a list of many objects
x <- list(
 lawn_point(c(-75.343, 39.984), properties = list(name = 'Location A')),
 lawn_point(c(-75.833, 39.284), properties = list(name = 'Location B')),
 lawn_point(c(-75.534, 39.123), properties = list(name = 'Location C'))
)
view(x)

# Use view_ to pass in arbitrary objects that will be combined
view_(
 lawn_point(c(-75.343, 39.984), properties = list(name = 'Location A')),
 lawn_point(c(-75.833, 39.284), properties = list(name = 'Location B')),
 lawn_point(c(-75.534, 39.123), properties = list(name = 'Location C'))
```

```
)

## another eg, smile :)
l1 <- list(
   c(-69.9609375, 35.460669951495305),
   c(-78.75, 39.095962936305504),
   c(-87.1875, 39.36827914916011),
   c(-92.46093749999999, 36.03133177633189)
)
l2 <- list(
   c(-46.0546875, 8.7547947),
   c(-33.0468750, -0.7031074),
   c(-14.0625000, 0.0000000),
   c(-0.3515625,  9.4490618)
)
l3 <- list(
   c(-1.40625, 38.81152),
   c(14.76562, 45.33670),
   c(23.20312, 45.58329),
   c(33.04688, 39.63954)
)
view_(lawn_point(c(-30, 20)),
   lawn_linestring(l1),
   lawn_linestring(l2),
   lawn_linestring(l3)
)

# From a geo_list object from geojsonio package
# library("geojsonio")
# vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0),
#  c(100.0,1.0), c(100.0,0.0))
# x <- geojson_list(vecs, geometry="polygon")
# view_(x)
# view_(x, lawn_point(c(101, 0)))

## End(Not run)
```

Index