Package ‘stplanr’

November 8, 2022

Type    Package
Title   Sustainable Transport Planning
Version 1.0.2
Maintainer Robin Lovelace <robo0x@gmail.com>

Description Tools for transport planning with an emphasis on spatial transport data and non-motorized modes. Create geographic "desire lines" from origin-destination (OD) data (building on the 'od' package); calculate routes on the transport network locally and via interfaces to routing services such as <https://cyclestreets.net/>; calculate route segment attributes such as bearing. The package implements the 'travel flow aggregation' method described in Morgan and Lovelace (2020) <doi:10.1177/2399808320942779>. Further information on the package's aim and scope can be found in the vignettes and in a paper in the R Journal (Lovelace and Ellison 2018) <doi:10.32614/RJ-2018-053>.

License MIT + file LICENSE

URL https://github.com/ropensci/stplanr,
     https://docs.ropensci.org/stplanr/

BugReports https://github.com/ropensci/stplanr/issues

Depends R (>= 3.5.0)
Imports curl (>= 3.2), data.table, dplyr (>= 0.7.6), httr (>= 1.3.1), jsonlite (>= 1.5), lwgeom (>= 0.1.4), magrittr, methods, nabor (>= 0.5.0), od, pbapply, Rcpp (>= 0.12.1), rlang (>= 0.2.2), sf (>= 0.6.3), sfheaders

Suggests cyclestreets, dodgr (>= 0.2.15), geodist, igraph (>= 1.2.2), knitr (>= 1.20), leaflet, mapsapi, opentripplanner, osrm, pct, markdown (>= 1.10), testthat (>= 2.0.0), tmap

VignetteBuilder knitr

Encoding UTF-8

LazyData yes

RoxygenNote 7.2.1
SystemRequirements GNU make

NeedsCompilation no

Author Robin Lovelace [aut, cre] (<https://orcid.org/0000-0001-5679-6536>),
Richard Ellison [aut],
Malcolm Morgan [aut] (<https://orcid.org/0000-0002-9488-9183>),
Barry Rowlingson [ctb],
Nick Bearman [ctb],
Nikolai Berkoff [ctb],
Scott Chamberlain [rev] (Scott reviewed the package for rOpenSci, see https://github.com/ropensci/onboarding/issues/10),
Mark Padgham [ctb],
Andrea Gilardi [ctb] (<https://orcid.org/0000-0002-9424-7439>)

Repository CRAN

Date/Publication 2022-11-08 12:40:02 UTC

R topics documented:

- stplanr-package .................................................. 4
- angle_diff .......................................................... 4
- bbox_scale .......................................................... 5
- cents_sf ............................................................. 6
- destinations_sf ..................................................... 7
- flow ................................................................. 7
- flowlines_sf ......................................................... 8
- flow_dests .......................................................... 8
- geo_bb ............................................................... 9
- geo_bb_matrix ....................................................... 10
- geo_buffer .......................................................... 10
- geo_code ............................................................ 11
- geo_length ......................................................... 12
- geo_projected ....................................................... 13
- geo_select_aeq .................................................... 13
- geo_toptail ......................................................... 14
- gsection ............................................................ 15
- Islines ............................................................... 16
- is_linepoint ......................................................... 17
- line2df ............................................................. 17
- line2points ......................................................... 18
- line_bearing ....................................................... 19
- line_breakup ....................................................... 20
- line_midpoint ..................................................... 21
- line_segment ....................................................... 21
- line_via ............................................................ 22
- mats2line ........................................................... 23
- n_vertices .......................................................... 24
- od2line .............................................................. 24
### R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>od2odf</td>
<td>26</td>
</tr>
<tr>
<td>odmatrix_to_od</td>
<td>27</td>
</tr>
<tr>
<td>od_aggregate_from</td>
<td>27</td>
</tr>
<tr>
<td>od_aggregate_to</td>
<td>28</td>
</tr>
<tr>
<td>od_coords</td>
<td>29</td>
</tr>
<tr>
<td>od_coords2line</td>
<td>30</td>
</tr>
<tr>
<td>od_data_lines</td>
<td>31</td>
</tr>
<tr>
<td>od_data_routes</td>
<td>31</td>
</tr>
<tr>
<td>od_data_sample</td>
<td>32</td>
</tr>
<tr>
<td>od_id</td>
<td>32</td>
</tr>
<tr>
<td>od_id_order</td>
<td>33</td>
</tr>
<tr>
<td>od_oneway</td>
<td>34</td>
</tr>
<tr>
<td>od_to_odmatrix</td>
<td>35</td>
</tr>
<tr>
<td>onewaygeo</td>
<td>36</td>
</tr>
<tr>
<td>osm_net_example</td>
<td>37</td>
</tr>
<tr>
<td>overline</td>
<td>37</td>
</tr>
<tr>
<td>overline_intersection</td>
<td>40</td>
</tr>
<tr>
<td>points2flow</td>
<td>41</td>
</tr>
<tr>
<td>points2line</td>
<td>41</td>
</tr>
<tr>
<td>points2odf</td>
<td>42</td>
</tr>
<tr>
<td>quadrant</td>
<td>42</td>
</tr>
<tr>
<td>read_table_builder</td>
<td>43</td>
</tr>
<tr>
<td>rnet_add_node</td>
<td>44</td>
</tr>
<tr>
<td>rnet_boundary_points</td>
<td>45</td>
</tr>
<tr>
<td>rnet_breakup_vertices</td>
<td>46</td>
</tr>
<tr>
<td>rnet_cycleway_intersection</td>
<td>48</td>
</tr>
<tr>
<td>rnet_get_nodes</td>
<td>48</td>
</tr>
<tr>
<td>rnet_group</td>
<td>49</td>
</tr>
<tr>
<td>rnet_overpass</td>
<td>50</td>
</tr>
<tr>
<td>rnet_roundabout</td>
<td>51</td>
</tr>
<tr>
<td>route</td>
<td>51</td>
</tr>
<tr>
<td>routes_fast_sf</td>
<td>52</td>
</tr>
<tr>
<td>routes_slow_sf</td>
<td>53</td>
</tr>
<tr>
<td>route_average_gradient</td>
<td>53</td>
</tr>
<tr>
<td>route_bikecitizens</td>
<td>54</td>
</tr>
<tr>
<td>route_dodgr</td>
<td>55</td>
</tr>
<tr>
<td>route_google</td>
<td>56</td>
</tr>
<tr>
<td>route_nearest_point</td>
<td>56</td>
</tr>
<tr>
<td>route_network_sf</td>
<td>57</td>
</tr>
<tr>
<td>route_osrm</td>
<td>57</td>
</tr>
<tr>
<td>route_rolling_average</td>
<td>58</td>
</tr>
<tr>
<td>route_rolling_diff</td>
<td>59</td>
</tr>
<tr>
<td>route_rolling_gradient</td>
<td>60</td>
</tr>
<tr>
<td>route_sequential_dist</td>
<td>61</td>
</tr>
<tr>
<td>route_slope_matrix</td>
<td>62</td>
</tr>
<tr>
<td>route_slope_vector</td>
<td>63</td>
</tr>
<tr>
<td>route_split</td>
<td>63</td>
</tr>
<tr>
<td>route_split_id</td>
<td>64</td>
</tr>
</tbody>
</table>
Description

The stplanr package provides functions to access and analyse data for transportation research, including origin-destination analysis, route allocation and modelling travel patterns.

Author(s)

Robin Lovelace <rob00x@gmail.com>

See Also

https://github.com/ropensci/stplanr

angle_diff

Description

This function was designed to find lines that are close to parallel and perpendicular to some predefined route. It can return results that are absolute (contain information on the direction of turn, i.e. + or - values for clockwise/anticlockwise), bidirectional (which mean values greater than +/- 90 are impossible).

Usage

angle_diff(l, angle, bidirectional = FALSE, absolute = TRUE)

Arguments

l A spatial lines object
angle an angle in degrees relative to North, with 90 being East and -90 being West. (direction of rotation is ignored).
bidirectional Should the result be returned in a bidirectional format? Default is FALSE. If TRUE, the same line in the opposite direction would have the same bearing
absolute If TRUE (the default) only positive values can be returned
bbox_scale

Details

Building on the convention used in the `bearing()` function from the geosphere package and in many applications, North is defined as 0, East as 90 and West as -90.

See Also

Other lines: `geo_toptail()`, `is_linepoint()`, `line2df()`, `line2points()`, `line_bearing()`, `line_breakup()`, `line_midpoint()`, `line_segment()`, `line_via()`, `mats2line()`, `n_vertices()`, `onewaygeo()`, `points2line()`, `toptail_buff()`

Examples

```r
lib_versions <- sf::sf_extSoftVersion()
lib_versions
# fails on some systems (with early versions of PROJ)
if (lib_versions[3] >= "6.3.1") {
  # Find all routes going North-South
  lines_sf <- od2line(od_data_sample, zones = zones_sf)
  angle_diff(lines_sf[2, ], angle = 0)
  angle_diff(lines_sf[2:3, ], angle = 0)
}
```

bbox_scale

Scale a bounding box

Description

Takes a bounding box as an input and outputs a bounding box of a different size, centred at the same point.

Usage

bbox_scale(bb, scale_factor)

Arguments

- **bb**: Bounding box object
- **scale_factor**: Numeric vector determining how much the bounding box will grow or shrink. Two numbers refer to extending the bounding box in x and y dimensions, respectively. If the value is 1, the output size will be the same as the input.

See Also

Other geo: `geo_bb_matrix()`, `geo_bb()`, `geo_buffer()`, `geo_length()`, `geo_projected()`, `geo_select_aeq()`, `quadrant()`
Examples

```r
bb <- matrix(c(-1.55, 53.80, -1.50, 53.83), nrow = 2)
bb1 <- bbox_scale(bb, scale_factor = 1.05)
bb2 <- bbox_scale(bb, scale_factor = c(2, 1.05))
bb3 <- bbox_scale(bb, 0.1)
plot(x = bb2[1, ], y = bb2[2, ])
points(bb1[1, ], bb1[2, ])
points(bb3[1, ], bb3[2, ])
points(bb[1, ], bb[2, ], col = "red")
```

cents_sf

| cents_sf | Spatial points representing home locations |

Description

These points represent population-weighted centroids of Medium Super Output Area (MSOA) zones within a 1 mile radius of of my home when I was writing this package.

Format

A spatial dataset with 8 rows and 5 columns

Details

- `geo_code` the official code of the zone
- `MSOA11NM` name zone name
- `percent_fem` the percent female
- `avslope` average gradient of the zone

Cents was generated from the data repository pct-data: https://github.com/npct/pct-data. This data was accessed from within the pct repo: https://github.com/npct/pct, using the following code:

See Also

Other data: destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

Examples

cents_sf
**destinations_sf**

**Example destinations data**

**Description**

This dataset represents trip destinations on a different geographic level than the origins stored in the object cents_sf.

**Format**

A spatial dataset with 87 features

**See Also**

Other data: cents_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

**Examples**

destinations_sf

---

**flow**

**Data frame of commuter flows**

**Description**

This dataset represents commuter flows (work travel) between origin and destination zones. The data is from the UK and is available as open data: [https://wicid.ukdataservice.ac.uk/](https://wicid.ukdataservice.ac.uk/).

**Format**

A data frame with 49 rows and 15 columns

**Details**

The variables are as follows:

- Area.of.residence. id of origin zone
- Area.of.workplace id of destination zone
- All. Travel to work flows by all modes
- [,4:15]. Flows for different modes
- id. unique id of flow

Although these variable names are unique to UK data, the data structure is generalisable and typical of flow data from any source. The key variables are the origin and destination ids, which link to the georeferenced spatial objects.
See Also

Other data: cents_sf, destinations_sf, flow_dests, flowlines_sf, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

flowlines_sf

Spatial lines dataset of commuter flows

Description

Flow data after conversion to a spatial format.

Format

A spatial lines dataset with 49 rows and 15 columns

See Also

Other data: cents_sf, destinations_sf, flow_dests, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

flow_dests

Data frame of invented commuter flows with destinations in a different layer than the origins

Description

Data frame of invented commuter flows with destinations in a different layer than the origins

Usage

data(flow_dests)

Format

A data frame with 49 rows and 15 columns

See Also

Other data: cents_sf, destinations_sf, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf
geo_bb

Examples

## Not run:
# This is how the dataset was constructed
flow_dests <- flow
flow_dests$Area.of.workplace <- sample(x = destinations$WZ11CD, size = nrow(flow))
flow_dests <- dplyr::rename(flow_dests, WZ11CD = Area.of.workplace)
develtools::use_data(flow_dests)

## End(Not run)

---

**geo_bb**

*Flexible function to generate bounding boxes*

**Description**

Takes a geographic object or bounding box as an input and outputs a bounding box, represented as a bounding box, corner points or rectangular polygon.

**Usage**

```r
geo_bb(
  shp, 
  scale_factor = 1, 
  distance = 0, 
  output = c("polygon", "points", "bb")
)
```

**Arguments**

- `shp`: Spatial object
- `scale_factor`: Numeric vector determining how much the bounding box will grow or shrink. Two numbers refer to extending the bounding box in x and y dimensions, respectively. If the value is 1, the output size will be the same as the input.
- `distance`: Distance in metres to extend the bounding box by
- `output`: Type of object returned (polygon by default)

**See Also**

- `bb_scale`
- Other geo: `bbox_scale()`, `geo_bb_matrix()`, `geo_buffer()`, `geo_length()`, `geo_projected()`, `geo_select_aeq()`,
Examples

```r
shp <- routes_fast_sf
shp_bb <- geo_bb(shp, distance = 100)
plot(shp_bb, col = "red", reset = FALSE)
plot(geo_bb(routes_fast_sf, scale_factor = 0.8), col = "green", add = TRUE)
plot(routes_fast_sf$geometry, add = TRUE)
geo_bb(shp, output = "point")
```

---

**geo_bb_matrix**

Create matrix representing the spatial bounds of an object

**Description**

Converts a range of spatial data formats into a matrix representing the bounding box

**Usage**

```r
geo_bb_matrix(shp)
```

**Arguments**

- `shp` Spatial object

**See Also**

Other geo: `bbox_scale()`, `geo_bb()`, `geo_buffer()`, `geo_length()`, `geo_projected()`, `geo_select_aeq()`, `quadrant()`

**Examples**

```r
geo_bb_matrix(routes_fast_sf)
geo_bb_matrix(cents_sf[1, ])
geo_bb_matrix(c(-2, 54))
geo_bb_matrix(sf::st_coordinates(cents_sf))
```

---

**geo_buffer**

Perform a buffer operation on a temporary projected CRS

**Description**

This function solves the problem that buffers will not be circular when used on non-projected data.

**Usage**

```r
geo_buffer(shp, dist = NULL, width = NULL, ...)
```
Arguments

shp A spatial object with a geographic CRS (e.g. WGS84) around which a buffer should be drawn

dist The distance (in metres) of the buffer (when buffering simple features)

width The distance (in metres) of the buffer (when buffering sp objects)

... Arguments passed to the buffer (see ?sf::st_buffer for details)

Details

Requires recent version of PROJ (>= 6.3.0). Buffers on sf objects with geographic (lon/lat) coordinates can also be done with the s2 package.

See Also

Other geo: bbox_scale(), geo_bb_matrix(), geo_bb(), geo_length(), geo_projected(), geo_select_aeq(), quadrant()

Examples

lib_versions <- sf::sf_extSoftVersion()
lib_versions
if (lib_versions[3] >= "6.3.1") {
  buff_sf <- geo_buffer(routes_fast_sf, dist = 50)
  plot(buff_sf$geometry)
  geo_buffer(routes_fast_sf$geometry, dist = 50)
}

geo_code

Convert text strings into points on the map

Description

Generate a lat/long pair from data using Google’s geolocation API.

Usage

geo_code(
  address,
  service = "nominatim",
  base_url = "https://maps.google.com/maps/api/geocode/json",
  return_all = FALSE,
  pat = NULL
)
Arguments

- **address**: Text string representing the address you want to geocode
- **service**: Which service to use? Nominatim by default
- **base_url**: The base url to query
- **return_all**: Should the request return all information returned by Google Maps? The default is FALSE: to return only two numbers: the longitude and latitude, in that order
- **pat**: Personal access token

Examples

```r
## Not run:
geo_code(address = "Hereford")
geo_code("LS7 3HB")
geo_code("hereford", return_all = TRUE)
# needs api key in .Renviron
geo_code("hereford", service = "google", pat = Sys.getenv("GOOGLE"), return_all = TRUE)
## End(Not run)
```

---

**geo_length**

*Calculate line length of line with geographic or projected CRS*

Description

Takes a line (represented in sf or sp classes) and returns a numeric value representing distance in meters.

Usage

```r
geo_length(shp)
```

Arguments

- **shp**: A spatial line object

See Also

Other geo: `bbox_scale()`, `geo_bb_matrix()`, `geo_bb()`, `geo_buffer()`, `geo_projected()`, `geo_select_aeq()`, `quadrant()`

Examples

```r
lib_versions <- sf::sf_extSoftVersion()
lib_versions
if (lib_versions[3] >= "6.3.1") {
  geo_length(routes_fast_sf)
}
```
**geo_projected**

Perform GIS functions on a temporary, projected version of a spatial object

**Description**

This function performs operations on projected data.

**Usage**

```
geo_projected(shp, fun, crs, silent, ...)
```

**Arguments**

- `shp`: A spatial object with a geographic (WGS84) coordinate system
- `fun`: A function to perform on the projected object (e.g. from the sf package)
- `crs`: An optional coordinate reference system (if not provided it is set automatically by `geo_select_aeq()`)
- `silent`: A binary value for printing the CRS details (default: TRUE)
- `...`: Arguments to pass to `fun`

**See Also**

Other geo: `bbox_scale()`, `geo_bb_matrix()`, `geo_bb()`, `geo_buffer()`, `geo_length()`, `geo_select_aeq()`, `quadrant()`

**Examples**

```r
lib_versions <- sf::sf_extSoftVersion()
lib_versions
# fails on some systems (with early versions of PROJ)
if (lib_versions[3] >= "6.3.1") {
  shp <- routes_fast_sf[2:4,]
  geo_projected(shp, sf::st_buffer, dist = 100)
}
```

**geo_select_aeq**

Select a custom projected CRS for the area of interest

**Description**

This function takes a spatial object with a geographic (WGS84) CRS and returns a custom projected CRS focussed on the centroid of the object. This function is especially useful for using units of metres in all directions for data collected anywhere in the world.
Usage

geo_select_aeq(shp)

Arguments

shp A spatial object with a geographic (WGS84) coordinate system

Details

The function is based on this stackexchange answer: https://gis.stackexchange.com/questions/121489

See Also

Other geo: bbox_scale(), geo_bb_matrix(), geo_bb(), geo_buffer(), geo_length(), geo_projected(), quadrant()

Examples

shp <- zones_sf
geo_select_aeq(shp)

Description

Clip the first and last n metres of SpatialLines

Usage

geo_toptail(l, toptail_dist, ...)

Arguments

1 An sf object representing lines

1 toptail_dist The distance (in metres) to top and tail the line by. Can either be a single value or a vector of the same length as the SpatialLines object.

1 ... Arguments passed to sf::st_buffer()

Details

Note: see the function toptailgs() in stplanr v0.8.5 for an implementation that uses the geosphere package.
Function to split overlapping SpatialLines into segments

Divides SpatialLinesDataFrame objects into separate Lines. Each new Lines object is the aggregate of a single number of aggregated lines.

Usage

gsection(sl, buff_dist = 0)

Arguments

sl SpatialLinesDataFrame with overlapping Lines to split by number of overlapping features.

buff_dist A number specifying the distance in meters of the buffer to be used to crop lines before running the operation. If the distance is zero (the default) touching but non-overlapping lines may be aggregated.

See Also

Other rnet: islins(), overline(), rnet_breakup_vertices(), rnet_group()
islines

Do the intersections between two geometries create lines?

Description

This is a function required in `overline()`. It identifies whether sets of lines overlap (beyond shared points) or not.

Usage

```r
islines(g1, g2)
```

Arguments

- `g1`: A spatial object
- `g2`: A spatial object

See Also

Other rnet: `gsection()`, `overline()`, `rnet_breakup_vertices()`, `rnet_group()`

Examples

```r
# Not run:
# sf implementation
islines(routes_fast_sf[2, ], routes_fast_sf[3, ])
islines(routes_fast_sf[2, ], routes_fast_sf[22, ])

# End(Not run)
```
is_linepoint

Identify lines that are points

Description

OD matrices often contain ‘intrazonal’ flows, where the origin is the same point as the destination. This function can help identify such intrazonal OD pairs, using 2 criteria: the total number of vertices (2 or fewer) and whether the origin and destination are the same.

Usage

is_linepoint(l)

Arguments

1 A spatial lines object

Details

Returns a boolean vector. TRUE means that the associated line is in fact a point (has no distance). This can be useful for removing data that will not be plotted.

See Also

Other lines: angle_diff(), geo_toptail(), line2df(), line2points(), line_bearing(), line_breakup(), line_midpoint(), line_segment(), line_via(), mats2line(), n_vertices(), onewaygeo(), points2line(), toptail_buff()

Examples

islp <- is_linepoint(flowlines_sf)
nrow(flowlines_sf)
sum(islp)
# Remove invisible 'linepoints'
nrow(flowlines_sf[!islp, ])

line2df

Convert geographic line objects to a data.frame with from and to coordinates

Description

This function returns a data frame with fx and fy and tx and ty variables representing the beginning and end points of spatial line features respectively.
Usage

`line2df(l)`

Arguments

1  A spatial lines object

See Also

Other lines: `angle_diff()`, `geo_toptail()`, `is_linepoint()`, `line2points()`, `line_bearing()`, `line_breakup()`, `line_midpoint()`, `line_segment()`, `line_via()`, `mats2line()`, `n_vertices()`, `onewaygeo()`, `points2line()`, `toptail_buff()`

Examples

```
line2df(routes_fast_sf[5:6, ]) # beginning and end of routes
```

---

## line2points

**Convert a spatial (linestring) object to points**

**Description**

The number of points will be double the number of lines with `line2points`. A closely related function, `line2pointsn` returns all the points that were line vertices. The points corresponding with a given line, i, will be 

\[(2i):(2i+1)\]

The last function, `line2vertices`, returns all the points that are vertices but not nodes. If the input `l` object is composed by only 1 LINestring with 2 POINTS, then it returns an empty `sf` object.

**Usage**

```
line2points(l, ids = rep(1:nrow(l)))
```

```
line2pointsn(l)
```

```
line2vertices(l)
```

**Arguments**

1  An `sf` object or a `SpatialLinesDataFrame` from the older `sp` package

ids  Vector of ids (by default `1:nrow(l)`)
line_bearing

Examples

```r
l <- routes_fast_sf[2, ]
lpoints <- line2points(l)
plot(l$geometry)
plot(lpoints, add = TRUE)
  # test all vertices:
plot(l$geometry)
lpoints2 <- line2pointsn(l)
plot(lpoints2$geometry, add = TRUE)

  # extract only internal vertices
l_internal_vertices <- line2vertices(l)
plot(sf::st_geometry(l), reset = FALSE)
plot(l_internal_vertices, add = TRUE)
  # The boundary points are missing
```

---

**line_bearing**  

*Find the bearing of straight lines*

**Description**

This function returns the bearing (in degrees relative to north) of lines.

**Usage**

```r
line_bearing(l, bidirectional = FALSE)
```

**Arguments**

- `l` A spatial lines object
- `bidirectional` Should the result be returned in a bidirectional format? Default is FALSE. If TRUE, the same line in the opposite direction would have the same bearing

**Details**

Returns a boolean vector. TRUE means that the associated line is in fact a point (has no distance). This can be useful for removing data that will not be plotted.

**See Also**

Other lines: `angle_diff`, `geo_toptail`, `is_linepoint`, `line2df`, `line2points`, `line_breakup`, `line_midpoint`, `line_segment`, `line_via`, `mats2line`, `n_vertices`, `onewaygeo`, `points2line`, `toptail_buff`
Examples

```r
lib_versions <- sf::sf_extSoftVersion()
lib_versions
# fails on some systems (with early versions of PROJ)
if (lib_versions[3] >= "6.3.1") {
  bearings_sf_1_9 <- line_bearing(flowlines_sf[1:5, ])
  line_bearing(flowlines_sf[1:5, ], bidirectional = TRUE)
}
```

---

### line_breakup

**Break up line objects into shorter segments**

#### Description

This function breaks up a LINESTRING geometries into smaller pieces.

#### Usage

```r
line_breakup(l, z)
```

#### Arguments

- `l`: An sf object with LINESTRING geometry
- `z`: An sf object with POLYGON geometry or a number representing the resolution of grid cells used to break up the linestring objects

#### Value

An sf object with LINESTRING geometry created after breaking up the input object.

#### See Also

Other lines: `angle_diff()`, `geo_toptail()`, `is_linepoint()`, `line2df()`, `line2points()`, `line_bearing()`, `line_midpoint()`, `line_segment()`, `line_via()`, `mats2line()`, `n_vertices()`, `onewaygeo()`, `points2line()`, `toptail_buff()`

#### Examples

```r
library(sf)
z <- zones_sf$geometry
l <- routes_fast_sf$geometry[2]
l_split <- line_breakup(l, z)
l
sf::st_length(l)
sum(sf::st_length(l_split))
plot(z)
plot(l, add = TRUE, lwd = 9, col = "grey")
plot(l_split, add = TRUE, col = 1:length(l_split))
```
line_midpoint  

Find the mid-point of lines

Description
Find the mid-point of lines

Usage
line_midpoint(l, tolerance = NULL)

Arguments
l  A spatial lines object
tolerance  The tolerance used to break lines at vertices. See `lwgeom::st_linesubstring()`.

See Also
Other lines: `angle_diff()`, `geo_toptail()`, `is_linepoint()`, `line2df()`, `line2points()`, `line_bearing()`, `line_breakup()`, `line_segment()`, `line_via()`, `mats2line()`, `n_vertices()`, `onewaygeo()`, `points2line()`, `toptail_buff()`

Examples
l = routes_fast_sf[2:5, ]
plot(l$geometry, col = 2:5)
midpoints = line_midpoint(l)
plot(midpoints, add = TRUE)

line_segment  

Divide sf LINESTRING objects into regular segments

Description
Divide sf LINESTRING objects into regular segments

Usage
line_segment(l, n_segments, segment_length = NA)

Arguments
l  A spatial lines object
n_segments  The number of segments to divide the line into
segment_length  The approximate length of segments in the output (overrides n_segments if set)
line_via

Add geometry columns representing a route via intermediary points

Description

Takes an origin (A) and destination (B), represented by the linestring l, and generates 3 extra geometries based on points p:

Usage

line_via(l, p)

Arguments

l  A spatial lines object
p  A spatial points object

Details

1. From A to P1 (P1 being the nearest point to A)
2. From P1 to P2 (P2 being the nearest point to B)
3. From P2 to B

See Also

Other lines: angle_diff(), geo_toptail(), is_linepoint(), line2df(), line2points(), line_bearing(), line_breakup(), line_midpoint(), line_via(), mats2line(), n_vertices(), onewaygeo(), points2line(), toptail_buff()
Examples

```r
library(sf)
l <- flowlines_sf[2:4,]
p <- destinations_sf
lv <- line_via(l, p)
lv
# library(mapview)
# mapview(lv) +
# mapview(lv$leg_orig, col = "red")
plot(lv[3], lwd = 9, reset = FALSE)
plot(lv$leg_orig, col = "red", lwd = 5, add = TRUE)
plot(lv$leg_via, col = "black", add = TRUE)
plot(lv$leg_dest, col = "green", lwd = 5, add = TRUE)
```

---

### mats2line

Convert 2 matrices to lines

#### Description

Convert 2 matrices to lines

#### Usage

```r
mats2line(mat1, mat2, crs = NA)
```

#### Arguments

- `mat1`: Matrix representing origins
- `mat2`: Matrix representing destinations
- `crs`: Number representing the coordinate system of the data, e.g. 4326

#### See Also

Other lines: `angle_diff()`, `geo_toptail()`, `is_linepoint()`, `line2df()`, `line2points()`, `line_bearing()`, `line_breakup()`, `line_midpoint()`, `line_segment()`, `line_via()`, `n_vertices()`, `onewaygeo()`, `points2line()`, `toptail_buff()`

#### Examples

```r
m1 <- matrix(c(1, 2, 1, 2), ncol = 2)
m2 <- matrix(c(9, 9, 9, 1), ncol = 2)
l <- mats2line(m1, m2)
class(l)
l
lsf <- sf::st_sf(l, crs = 4326)
class(lsf)
plot(lsf)
# mapview::mapview(lsf)
```
n_vertices

Retrieve the number of vertices in sf objects

Description

Returns a vector of the same length as the number of sf objects.

Usage

n_vertices(l)

Arguments

l  An sf object with LINestring geometry

See Also

Other lines: angle_diff(), geo_toptail(), is_linepoint(), line2df(), line2points(), line_bearing(), line_breakup(), line_midpoint(), line_segment(), line_via(), mats2line(), onewaygeo(), points2line(), toptail_buff()

Examples

l = routes_fast_sf
n_vertices(l)
n_vertices(zones_sf)

od2line

Convert origin-destination data to spatial lines

Description

Origin-destination (‘OD’) flow data is often provided in the form of 1 line per flow with zone codes of origin and destination centroids. This can be tricky to plot and link-up with geographical data. This function makes the task easier.

Usage

od2line(
  flow,
  zones,
  destinations = NULL,
  zone_code = names(zones)[1],
  origin_code = names(flow)[1],
  dest_code = names(flow)[2],
  zone_code_d = NA,
  silent = FALSE
)

Arguments

flow
A data frame representing origin-destination data. The first two columns of this data frame should correspond to the first column of the data in the zones. Thus in `cents_sf()`, the first column is geo_code. This corresponds to the first two columns of `flow()`.

zones
A spatial object representing origins (and destinations if no separate destinations object is provided) of travel.

destinations
A spatial object representing destinations of travel flows.

zone_code
Name of the variable in `zones` containing the ids of the zone. By default this is the first column names in the zones.

origin_code
Name of the variable in `flow` containing the ids of the zone of origin. By default this is the first column name in the flow input dataset.

dest_code
Name of the variable in `flow` containing the ids of the zone of destination. By default this is the second column name in the flow input dataset or the first column name in the destinations if that is set.

zone_code_d
Name of the variable in `destinations` containing the ids of the zone. By default this is the first column names in the destinations.

silent
TRUE by default, setting it to TRUE will show you the matching columns

Details

Origin-destination (OD) data is often provided in the form of 1 line per OD pair, with zone codes of the trip origin in the first column and the zone codes of the destination in the second column (see the vignette("stplanr-od")) for details. `od2line()` creates a spatial (linestring) object representing movement from the origin to the destination for each OD pair. It takes data frame containing origin and destination cones (flow) that match the first column in a a spatial (polygon or point) object (zones).

See Also

Other od: `od2odf()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_coords()`, `od_id_order()`, `od_id_odoneya()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()`, `points2odf()`

Examples

```r
od_data <- stplanr::flow[1:20, ]
1 <- od2line(flow = od_data, zones = cents_sf)
plot(sf::st_geometry(cents_sf))
plot(l, lwd = l$All / mean(l$All), add = TRUE)
```
od2odf

Extract coordinates from OD data

Description

Extract coordinates from OD data

Usage

od2odf(flow, zones)

Arguments

- **flow**: A data frame representing origin-destination data. The first two columns of this data frame should correspond to the first column of the data in the zones. Thus in `cents_sf()`, the first column is geo_code. This corresponds to the first two columns of `flow()`.
- **zones**: A spatial object representing origins (and destinations if no separate destinations object is provided) of travel.

Details

Origin-destination (OD) data is often provided in the form of 1 line per OD pair, with zone codes of the trip origin in the first column and the zone codes of the destination in the second column (see the vignette("stplanr-od")) for details. `od2odf()` creates an ’origin-destination data frame’, with columns containing origin and destination codes (`flow`) that match the first column in a a spatial (polygon or point `sf`) object (`zones`).

The function returns a data frame with coordinates for the origin and destination.

See Also

Other od: `od2line()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_coords()`, `od_id_order()`, `od_id`, `od_oneway()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()`, `points2odf()`

Examples

od2odf(flow[1:2,], zones_sf)
odmatrix_to_od

Convert origin-destination data from wide to long format

Description

This function takes a matrix representing travel between origins (with origin codes in the rownames of the matrix) and destinations (with destination codes in the colnames of the matrix) and returns a data frame representing origin-destination pairs.

Usage

odmatrix_to_od(odmatrix)

Arguments

odmatrix A matrix with row and columns representing origin and destination zone codes and cells representing the flow between these zones.

Details

The function returns a data frame with rows ordered by origin and then destination zone code values and with names orig, dest and flow.

See Also

Other od: od2line(), od2odf(), od_aggregate_from(), od_aggregate_to(), od.coords2line(), od_coords(), od_id_order(), od_id.od.oneway(), od_to_odmatrix(), points2flow(), points2odf()

Examples

odmatrix <- od_to_odmatrix(flow)
odmatrix_to_od(odmatrix)
flow[1:9, 1:3]
odmatrix_to_od(od_to_odmatrix(flow[1:9, 1:3]))

od_aggregate_from

Summary statistics of trips originating from zones in OD data

Description

This function takes a data frame of OD data and returns a data frame reporting summary statistics for each unique zone of origin.

Usage

od_aggregate_from(flow, attrib = NULL, FUN = sum, ..., col = 1)
Arguments

flow: A data frame representing origin-destination data. The first two columns of this data frame should correspond to the first column of the data in the zones. Thus in `cents_sf()`, the first column is geo_code. This corresponds to the first two columns of `flow()`.

attrib: character, column names in sl to be aggregated

FUN: A function to summarise OD data by

...: Additional arguments passed to FUN

col: The column that the OD dataset is grouped by (1 by default, the first column usually represents the origin)

Details

It has some default settings: the default summary statistic is `sum()` and the first column in the OD data is assumed to represent the zone of origin. By default, if attrib is not set, it summarises all numeric columns.

See Also

Other od: `od2line()`, `od2odf()`, `od_aggregate_to()`, `od_coords2line()`, `od_coords()`, `od_id_order()`, `od_id`, `od_oneway()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()``, `points2odf()`

Examples

`od_aggregate_from(flow)`

---

`od_aggregate_to`  
*Summary statistics of trips arriving at destination zones in OD data*

Description

This function takes a data frame of OD data and returns a data frame reporting summary statistics for each unique zone of destination.

Usage

`od_aggregate_to(flow, attrib = NULL, FUN = sum, ..., col = 2)`

Arguments

flow: A data frame representing origin-destination data. The first two columns of this data frame should correspond to the first column of the data in the zones. Thus in `cents_sf()`, the first column is geo_code. This corresponds to the first two columns of `flow()`.

attrib: character, column names in sl to be aggregated
**od_coords**

A function to summarise OD data by

**Details**

Additional arguments passed to FUN

The column that the OD dataset is grouped by (1 by default, the first column usually represents the origin)

**Details**

It has some default settings: it assumes the destination ID column is the 2nd and the default summary statistic is `sum()`. By default, if attrib is not set, it summarises all numeric columns.

**See Also**

Other od: `od2line()`, `od2odf()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_coords()`, `od_id_order()`, `od_id`, `od_oneway()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()`, `points2odf()`

**Examples**

```
od_aggregate_to(flow)
```

---

**od_coords**

Create matrices representing origin-destination coordinates

**Description**

This function takes a wide range of input data types (spatial lines, points or text strings) and returns a matrix of coordinates representing origin (fx, fy) and destination (tx, ty) points.

**Usage**

```
od_coords(from = NULL, to = NULL, l = NULL)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>An object representing origins (if lines are provided as the first argument, from is assigned to l)</td>
</tr>
<tr>
<td>to</td>
<td>An object representing destinations</td>
</tr>
<tr>
<td>l</td>
<td>Only needed if from and to are empty, in which case this should be a spatial object representing desire lines</td>
</tr>
</tbody>
</table>

**See Also**

Other od: `od2line()`, `od2odf()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_id_order()`, `od_id`, `od_oneway()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()`, `points2odf()`
Examples
   od_coords(from = c(0, 52), to = c(1, 53)) # lon/lat coordinates
   od_coords(cents_sf[1:3, ], cents_sf[2:4, ]) # sf points
   # od_coords("Hereford", "Leeds") # geocode locations
   od_coords(flowlines_sf[1:3, ])

---

od_coords2line  Convert origin-destination coordinates into desire lines

Description
Convert origin-destination coordinates into desire lines

Usage
   od_coords2line(odc, crs = 4326, remove_duplicates = TRUE)

Arguments
   odc  A data frame or matrix representing the coordinates of origin-destination data.
        The first two columns represent the coordinates of the origin (typically longitude
        and latitude) points; the third and fourth columns represent the coordinates of
        the destination (in the same CRS). Each row represents travel from origin to
        destination.
   crs  A number representing the coordinate reference system of the result, 4326 by
        default.
   remove_duplicates
        Should rows with duplicated rows be removed? TRUE by default.

See Also
Other od: od2line(), od2odf(), od_aggregate_from(), od_aggregate_to(), od_coords(),
od_id_order(), od_id, od_oneway(), od_to_odmatrix(), odmatrix_to_od(), points2flow(),
points2odf()

Examples
   odf <- od_coords(l = flowlines_sf)
   odlines <- od_coords2line(odf)
   odlines <- od_coords2line(odf, crs = 4326)
   plot(odlines)
   x_coords <- 1:3
   n <- 50
   d <- data.frame(lapply(1:4, function(x) sample(x_coords, n, replace = TRUE)))
   names(d) <- c("fx", "fy", "tx", "ty")
   l <- od_coords2line(d)
   plot(l)
   nrow(l)
od_data_lines

```r
l_with_duplicates <- od_coords2line(d, remove_duplicates = FALSE)
plot(l_with_duplicates)
nrow(l_with_duplicates)
```

---

**od_data_lines**

*Example of desire line representations of origin-destination data from UK Census*

**Description**

Derived from `od_data_sample` showing movement between points represented in `cents_sf`

**Format**

A data frame (tibble) object

**See Also**

Other data: `cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf`

**Examples**

`od_data_lines`

---

**od_data_routes**

*Example segment-level route data*

**Description**

See `data-raw/generate-data.Rmd` for details on how this was created. The dataset shows routes between origins and destinations represented in `od_data_lines`

**Format**

A data frame (tibble) object

**See Also**

Other data: `cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf`

**Examples**

`od_data_routes`
od_data_sample  Example of origin-destination data from UK Census

Description

See data-raw/generate-data.Rmd for details on how this was created.

Format

A data frame (tibble) object

See Also

Other data: cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

Examples

od_data_sample

od_id  Combine two ID values to create a single ID number

Description

Combine two ID values to create a single ID number

Usage

od_id_szudzik(x, y, ordermatters = FALSE)

od_id_max_min(x, y)

od_id_character(x, y)

Arguments

x a vector of numeric, character, or factor values

y a vector of numeric, character, or factor values

ordermatters logical, does the order of values matter to pairing, default = FALSE
Details
In OD data it is common to have many 'oneway' flows from "A to B" and "B to A". It can be useful to group these an have a single ID that represents pairs of IDs with or without directionality, so they contain 'twoway' or bi-directional values.

*od_id* functions take two vectors of equal length and return a vector of IDs, which are unique for each combination but the same for twoway flows.

- the Szudzik pairing function, on two vectors of equal length. It returns a vector of ID numbers. This function superseeds *od_id_order* as it is faster on large datasets.

See Also
Other od: *od2line(), od2odf(), od_aggregate_from(), od_aggregate_to(), od_coords2line(), od_coords(), od_id_order(), od_oneway(), od_to_odmatrix(), odmatrix_to_od(), points2flow(), points2odf()*

Examples
```
(d <- od_data_sample[2:9, 1:2])
(id <- od_id_character(d[[1]], d[[2]]))
 duplicated(id)
 od_id_szudzik(d[[1]], d[[2]])
 od_id_max_min(d[[1]], d[[2]])
```

---

**od_id_order**

Generate ordered ids of OD pairs so lowest is always first This function is slow on large datasets, see szudzik_pairing for faster alternative

Description
Generate ordered ids of OD pairs so lowest is always first This function is slow on large datasets, see szudzik_pairing for faster alternative

Usage
```
od_id_order(x, id1 = names(x)[1], id2 = names(x)[2])
```

Arguments
- **x** A data frame or SpatialLinesDataFrame, representing an OD matrix
- **id1** Optional (it is assumed to be the first column) text string referring to the name of the variable containing the unique id of the origin
- **id2** Optional (it is assumed to be the second column) text string referring to the name of the variable containing the unique id of the destination
See Also

Other od: od2line(), od2odf(), od_aggregate_from(), od_aggregate_to(), od_coords2line(), od_coords(), od_id, od_oneway(), od_to_odmatrix(), odmatrix_to_od(), points2flow(), points2odf()

Examples

x <- data.frame(id1 = c(1, 1, 2, 2, 3), id2 = c(1, 2, 3, 1, 4))
od_id_order(x) # 4th line switches id1 and id2 so stplanr.key is in order

---

**od_oneway**

*Aggregate od pairs they become non-directional*

Description

For example, sum total travel in both directions.

Usage

```r
od_oneway(
  x,
  attrib = names(x[-c(1:2)])[vapply(x[-c(1:2)], is.numeric, TRUE)],
  id1 = names(x)[1],
  id2 = names(x)[2],
  stplanr.key = NULL
)
```

Arguments

- `x`: A data frame or SpatialLinesDataFrame, representing an OD matrix
- `attrib`: A vector of column numbers or names, representing variables to be aggregated. By default, all numeric variables are selected. aggregate
- `id1`: Optional (it is assumed to be the first column) text string referring to the name of the variable containing the unique id of the origin
- `id2`: Optional (it is assumed to be the second column) text string referring to the name of the variable containing the unique id of the destination
- `stplanr.key`: Optional key of unique OD pairs regardless of the order, e.g., as generated by `od_id_max_min()` or `od_id_szudzik()`

Details

Flow data often contains movement in two directions: from point A to point B and then from B to A. This can be problematic for transport planning, because the magnitude of flow along a route can be masked by flows the other direction. If only the largest flow in either direction is captured in an analysis, for example, the true extent of travel will by heavily under-estimated for OD pairs which have similar amounts of travel in both directions. Flows in both direction are often represented by overlapping lines with identical geometries which can be confusing for users and are difficult to plot.
Value

`od_to_odmatrix` converts a data frame representing travel between origins and destinations into a wide format matrix with cell values representing the attribute data of interest. It is useful for summarizing origin-destination data.

Usage

```
od_to_odmatrix(flow, attrib = 3, name_orig = 1, name_dest = 2)
```

Arguments

- `flow`: A data frame representing flows between origin and destinations.
- `attrib`: A number or character string representing the column containing the attribute data of interest from the `flow` data frame.
- `name_orig`: A number or character string representing the zone of origin.
- `name_dest`: A number or character string representing the zone of destination.

Examples

```r
(od_min <- od_data_sample[c(1, 2, 9), 1:6])
(od_oneway <- od_to_odmatrix(od_min, attrib = 3))
```

Description

This function takes a data frame representing travel between origins (with origin codes in `name_orig`, typically the first column) and destinations (with destination codes in `name_dest`, typically the second column) and returns a matrix with cell values (from `attrib`, the third column by default) representing travel between origins and destinations.

See Also

Other `od` functions: `od2line()`, `od2odf()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_id_order()`, `od_id`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()`, `points2odf()`.
See Also

Other od: od2line(), od2odf(), od_aggregate_from(), od_aggregate_to(), od_coords2line(), od_coords(), od_id_order(), od_id, od_oneway(), odmatrix_to_od(), points2flow(), points2odf()

Examples

od_to_odmatrix(flow)
od_to_odmatrix(flow[1:9, ])
od_to_odmatrix(flow[1:9, ], attrib = "Bicycle")

onewaygeo Aggregate flows so they become non-directional (by geometry - the slow way)

Description

Flow data often contains movement in two directions: from point A to point B and then from B to A. This can be problematic for transport planning, because the magnitude of flow along a route can be masked by flows the other direction. If only the largest flow in either direction is captured in an analysis, for example, the true extent of travel will be heavily under-estimated for OD pairs which have similar amounts of travel in both directions.

Usage

onewaygeo(x, attrib)

Arguments

x A dataset containing linestring geometries
attrib A text string containing the name of the line’s attribute to aggregate or a numeric vector of the columns to be aggregated

Details

This function aggregates directional flows into non-directional flows, potentially halving the number of lines objects and reducing the number of overlapping lines to zero.

Value

onewaygeo outputs a SpatialLinesDataFrame with single lines and user-selected attribute values that have been aggregated. Only lines with a distance (i.e. not intra-zone flows) are included.

See Also

Other lines: angle_diff(), geo_toptail(), is_linepoint(), line2df(), line2points(), line_bearing(), line_breakup(), line_midpoint(), line_segment(), line_via(), mats2line(), n_vertices(), points2line(), toptail_buff()
Example of OpenStreetMap road network

Description

Example of OpenStreetMap road network

Format

An sf object

See Also

Other data: cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, read_table_builder(), route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

Examples

osm_net_example

overline

Convert series of overlapping lines into a route network

Description

This function takes a series of overlapping lines and converts them into a single route network.

This function is intended as a replacement for overline() and is significantly faster especially on large datasets. However, it also uses more memory.

Usage

overline(
  sl, 
  attrib, 
  ncores = 1, 
  simplify = TRUE, 
  regionalise = 1e+09, 
  quiet = ifelse(nrow(sl) < 1000, TRUE, FALSE), 
  fun = sum 
)

overline2(
  sl, 
  attrib,
overline

```r
ncore = 1,
simplify = TRUE,
regionalise = 1e+09,
quiet = ifelse(nrow(sl) < 1000, TRUE, FALSE),
fun = sum
)
```

**Arguments**

- **sl**: A spatial object representing routes on a transport network
- **attrib**: character, column names in sl to be aggregated
- **ncore**: integer, how many cores to use in parallel processing, default = 1
- **simplify**: logical, if TRUE group final segments back into lines, default = TRUE
- **regionalise**: integer, during simplification regionalisation is used if the number of segments exceeds this value
- **quiet**: Should the the function omit messages? NULL by default, which means the output will only be shown if sl has more than 1000 rows.
- **fun**: Named list of functions to summaries the attributes by? sum is the default. list(sum = sum, average = mean) will summarise all attributes by sum and mean.

**Details**

The function can be used to estimate the amount of transport 'flow' at the route segment level based on input datasets from routing services, for example linestring geometries created with the `route()` function.

The `overline()` function breaks each line into many straight segments and then looks for duplicated segments. Attributes are summed for all duplicated segments, and if simplify is TRUE the segments with identical attributes are recombined into linestrings.

The following arguments only apply to the sf implementation of `overline()`:

- **ncores**, the number of cores to use in parallel processing
- **simplify**, should the final segments be converted back into longer lines? The default setting is TRUE. `simplify = FALSE` results in straight line segments consisting of only 2 vertices (the start and end point), resulting in a data frame with many more rows than the simplified results (see examples).
- **regionalise** the threshold number of rows above which regionalisation is used (see details).

For sf objects Regionalisation breaks the dataset into a 10 x 10 grid and then performed the simplification across each grid. This significantly reduces computation time for large datasets, but slightly increases the final file size. For smaller datasets it increases computation time slightly but reduces memory usage and so may also be useful.

A known limitation of this method is that overlapping segments of different lengths are not aggregated. This can occur when lines stop halfway down a road. Typically these errors are small, but some artefacts may remain within the resulting data.
For very large datasets \(\text{nrow}(x) > 1000000\), memory usage can be significant. In these cases it is possible to overline subsets of the dataset, `rbind` the results together, and then overline again, to produce a final result.

Multicore support is only enabled for the regionalised simplification stage as it does not help with other stages.

**Value**

An `sf` object representing a route network

**Author(s)**

Barry Rowlingson
Malcolm Morgan

**References**


**See Also**

Other rnet: `gsection()`, `islines()`, `rnet_breakup_vertices()`, `rnet_group()`

Other rnet: `gsection()`, `islines()`, `rnet_breakup_vertices()`, `rnet_group()`

**Examples**

```r
sl <- routes_fast.sf[2:4, ]
sl$All <- flowlines_sf$All[2:4]
rnet <- overline(sl = sl, attrib = "All")
nrow(sl)
nrow(rnet)
plot(rnet)
rnet_mean <- overline(sl, c("All", "av_incline"), fun = list(mean = mean, sum = sum))
plot(rnet_mean, lwd = rnet_mean$All_sum / mean(rnet_mean$All_sum))
rnet_sf_raw <- overline(sl, attrib = "length", simplify = FALSE)
nrow(rnet_sf_raw)
summary(n_vertices(rnet_sf_raw))
plot(rnet_sf_raw)
rnet_sf_raw$n <- 1:nrow(rnet_sf_raw)
plot(rnet_sf_raw[10:25, ])
```
overline_intersection  

Convert series of overlapping lines into a route network

Description

This function takes overlapping LINESTRINGs stored in an sf object and returns a route network composed of non-overlapping geometries and aggregated values.

Usage

overline_intersection(sl, attrib, fun = sum)

Arguments

sl          An sf LINESTRING object with overlapping elements
attrib      character, column names in sl to be aggregated
fun         Named list of functions to summaries the attributes by?  sum is the default.  list(sum = sum, average = mean) will summarise all attributes by sum and mean.

Examples

routes_fast_sf$value <- 1
sl <- routes_fast_sf[4:6, ]
attrib <- c("value", "length")
rc <- overline_intersection(sl = sl, attrib)
plot(rc, lwd = rcs$value)
# A larger example
sl <- routes_fast_sf[4:7, ]
rc <- overline_intersection(sl = sl, attrib = c("value", "length"))
plot(rc, lwd = rcs$value)
rc_sf <- overline(routes_fast_sf[4:7, ], attrib = c("value", "length"))
plot(rc_sf, lwd = rc$vvalue)
# An even larger example (not shown, takes time to run)
# rc <- overline_intersection(routes_fast_sf, attrib = c("value", "length"))
# rc_sf <- overline(routes_fast_sf, attrib = c("value", "length"), buff_dist = 10)
# plot(rc$geometry, lwd = rc$vvalue * 2, col = "grey")
# plot(rc_sf$geometry, lwd = rc_sf$vvalue, add = TRUE)
**points2flow**

Convert a series of points into geographical flows

**Description**

Takes a series of geographical points and converts them into a spatial (linestring) object representing the potential flows, or ‘spatial interaction’, between every combination of points.

**Usage**

```r
points2flow(p)
```

**Arguments**

- `p`: A spatial (point) object

**See Also**

Other od: `od2line()`, `od2odf()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_id_order()`, `od_id`, `od_oneway()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2odf()`

**Examples**

```r
cents_sf <- points2flow(cents_sf[1:4, ])
plot(flow_sf)
```

---

**points2line**

Convert a series of points, or a matrix of coordinates, into a line

**Description**

This function makes the creation of `sf` objects with LINESTRING geometries easy.

**Usage**

```r
points2line(p)
```

**Arguments**

- `p`: A spatial (points) object or matrix representing the coordinates of points.

**See Also**

Other lines: `angle_diff()`, `geo_toptail()`, `is_linepoint()`., `line2df()`, `line2points()`, `line_bearing()`, `line_breakup()`, `line_midpoint()`, `line_segment()`, `line_via()`, `mats2line()`, `n_vertices()`, `onewaygeo()`, `toptail_buff()`
**Examples**

```r
l_sf <- points2line(cents_sf)
plot(l_sf)
```

---

**points2odf**  
*Convert a series of points into a dataframe of origins and destinations*

**Description**

Takes a series of geographical points and converts them into a data.frame representing the potential flows, or 'spatial interaction', between every combination of points.

**Usage**

```r
points2odf(p)
```

**Arguments**

- `p`  
  A spatial points object

**See Also**

Other od: `od2line()`, `od2odf()`, `od_aggregate_from()`, `od_aggregate_to()`, `od_coords2line()`, `od_coords()`, `od_id_order()`, `od_id`, `od_oneway()`, `od_to_odmatrix()`, `odmatrix_to_od()`, `points2flow()`

**Examples**

```r
points2odf(cents_sf)
```

---

**quadrant**  
*Split a spatial object into quadrants*

**Description**

Returns a character vector of NE, SE, SW, NW corresponding to north-east, south-east quadrants respectively. If `number_out` is TRUE, returns numbers from 1:4, respectively.

**Usage**

```r
quadrant(x, cent = NULL, number_out = FALSE)
```
**Arguments**

- `x`: Object of class `sf`
- `cent`: The centrepoint of the region of interest. Quadrants will be defined based on this point. By default this will be the geographic centroid of the zones.
- `number_out`: Should the result be returned as a number?

**See Also**

Other geo: `bbox_scale()`, `geo_bb_matrix()`, `geo_bb()`, `geo_buffer()`, `geo_length()`, `geo_projected()`, `geo_select_aeq()`

**Examples**

```r
x = zones_sf
(quads <- quadrant(x))
plot(x$geometry, col = factor(quads))
```

---

**Description**

Import and format Australian Bureau of Statistics (ABS) TableBuilder files

**Usage**

```r
read_table_builder(dataset, filetype = "csv", sheet = 1, removeTotal = TRUE)
```

**Arguments**

- `dataset`: Either a dataframe containing the original data from TableBuilder or a character string containing the path of the unzipped TableBuilder file.
- `filetype`: A character string containing the filetype. Valid values are 'csv', 'legacycsv' and 'xlsx' (default = 'csv'). Required even when dataset is a dataframe. Use 'legacycsv' for csv files derived from earlier versions of TableBuilder for which csv outputs were csv versions of the xlsx files. Current csv output from TableBuilder follow a more standard csv format.
- `sheet`: An integer value containing the index of the sheet in the xlsx file (default = 1).
- `removeTotal`: A boolean value. If TRUE removes the rows and columns with totals (default = TRUE).
Details

The Australian Bureau of Statistics (ABS) provides customised tables for census and other datasets in a format that is difficult to use in R because it contains rows with additional information. This function imports the original (unzipped) TableBuilder files in .csv or .xlsx format before creating an R dataframe with the data.

Note: we recommend using the readabs package for this purpose.

See Also

Other data: cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, route_network_sf, routes_fast_sf, routes_slow_sf, zones_sf

rnet_add_node

Add a node to route network

Description

Add a node to route network

Usage

rnet_add_node(rnet, p)

Arguments

rnet A route network of the type generated by overline()

p A point represented by an sf object the will split the route

Examples

sample_routes <- routes_fast_sf[2:6, NULL]
sample_routes$value <- rep(1:3, length.out = 5)
rnet <- overline2(sample_routes, attrib = "value")
p <- sf::st_sfc(sf::st_point(c(-1.540, 53.826)), crs = sf::st_crs(rnet))
r_split <- route_split(rnet, p)
plot(rnet$geometry, lwd = rnet$value * 5, col = "grey")
plot(p, cex = 9, add = TRUE)
plot(r_split, col = 1:nrow(r_split), add = TRUE, lwd = r_split$value)
rnet_boundary_points

Get points at the beginner and end of linestrings

Description

Get points at the beginner and end of linestrings

Usage

rnet_boundary_points(rnet)
rnet_boundary_df(rnet)
rnet_boundary_unique(rnet)
rnet_boundary_points_lwgeom(rnet)
rnet_duplicated_vertices(rnet, n = 2)

Arguments

rnet An sf or sfc object with LINESTRING geometry representing a route network.
n The minimum number of time a vertex must be duplicated to be returned

Examples

has_sfheaders <- requireNamespace("sfheaders", quietly = TRUE)
if(has_sfheaders) {
  rnet <- rnet_roundabout
  bp1 <- rnet_boundary_points(rnet)
  bp2 <- line2points(rnet) # slower version with lwgeom
  bp3 <- rnet_boundary_points_lwgeom(rnet) # slower version with lwgeom
  bp4 <- rnet_boundary_unique(rnet)
  nrow(bp1)
nrow(bp3)
identical(sort(sf::st_coordinates(bp1)), sort(sf::st_coordinates(bp2)))
identical(sort(sf::st_coordinates(bp3)), sort(sf::st_coordinates(bp4)))
plot(rnet$geometry)
plot(bp3, add = TRUE)
}
rnet_breakup_vertices  

Break up an sf object with LINESTRING geometry.

Description

This function breaks up a LINESTRING geometry into multiple LINESTRING(s). It is used mainly for preserving routability of an object that is created using Open Street Map data. See details, stplanr/issues/282, and stplanr/issues/416.

Usage

rnet_breakup_vertices(rnet, verbose = FALSE)

Arguments

rnet  
An sf or sfc object with LINESTRING geometry representing a route network.

verbose  
Boolean. If TRUE, the function prints additional messages.

Details

A LINESTRING geometry is broken-up when one of the two following conditions are met:

1. two or more LINESTRINGS share a POINT which is a boundary point for some LINESTRING(s), but not all of them (see the rnet_roundabout example);
2. two or more LINESTRINGS share a POINT which is not in the boundary of any LINESTRING (see the rnet_cycleway_intersection example).

The problem with the first example is that, according to algorithm behind SpatialLinesNetwork(), two LINESTRINGS are connected if and only if they share at least one point in their boundaries. The roads and the roundabout are clearly connected in the "real" world but the corresponding LINESTRING objects do not share two distinct boundary points. In fact, by Open Street Map standards, a roundabout is represented as a closed and circular LINESTRING, and this implies that the roundabout is not connected to the other roads according to SpatialLinesNetwork() definition. By the same reasoning, the roads in the second example are clearly connected in the "real" world, but they do not share any point in their boundaries. This function is used to solve this type of problem.

Value

An sf or sfc object with LINESTRING geometry created after breaking up the input object.

See Also

Other rnet: gsection(), islines(), overline(), rnet_group()
Examples

library(sf)
def_par <- par(no.readonly = TRUE)
par(mar = rep(0, 4))

# Check the geometry of the roundabout example. The dots represent the
# boundary points of the LINESTRINGS. The "isolated" red point in the
# top-left is the boundary point of the roundabout, and it is not shared
# with any other street.
plot(st_geometry(rnet_roundabout), lwd = 2, col = rainbow(nrow(rnet_roundabout)))
boundary_points <- st_geometry(line2points(rnet_roundabout))
points_cols <- rep(rainbow(nrow(rnet_roundabout)), each = 2)
plot(boundary_points, pch = 16, add = TRUE, col = points_cols, cex = 2)

# Clean the roundabout example.

rnet_roundabout_clean <- rnet_breakup_vertices(rnet_roundabout)
plot(st_geometry(rnet_roundabout_clean), lwd = 2, col = rainbow(nrow(rnet_roundabout_clean)))
boundary_points <- st_geometry(line2points(rnet_roundabout_clean))
points_cols <- rep(rainbow(nrow(rnet_roundabout_clean)), each = 2)
plot(boundary_points, pch = 16, add = TRUE, col = points_cols)
# The roundabout is now routable since it was divided into multiple pieces
# (one for each colour), which, according to SpatialLinesNetwork() function,
# are connected.

# Check the geometry of the overpasses example. This example is used to test
# that this function does not create any spurious intersection.
plot(st Geometry(rnet_overpass), lwd = 2, col = rainbow(nrow(rnet_overpass)))
boundary_points <- stGeometry(line2points(rnet_overpass))
points_cols <- rep(rainbow(nrow(rnet_overpass)), each = 2)
plot(boundary_points, pch = 16, add = TRUE, col = points_cols, cex = 2)
# At the moment the network is not routable since one of the underpasses is
# not connected to the other streets.

# Check interactively.
# mapview::mapview(rnet_overpass)

# Clean the network. It should not create any spurious intersection between
# roads located at different heights.

rnet_overpass_clean <- rnet_breakup_vertices(rnet_overpass)
plot(st Geometry(rnet_overpass_clean), lwd = 2, col = rainbow(nrow(rnet_overpass_clean)))
# Check interactively.
# mapview::mapview(rnet_overpass)

# Check the geometry of the cycleway_intersection example. The black dots
# represent the boundary points and we can see that the two roads are not
# connected according to SpatialLinesNetwork() function.
plot(
  rnet_cycleway_intersection$geometry,
  lwd = 2,
  col = rainbow(nrow(rnet_cycleway_intersection)),
  cex = 2
)
plot(st.geometry(line2points(rnet_cycleway_intersection)), pch = 16, add = TRUE)
# Check interactively
# mapview::mapview(rnet_overpass)

# Clean the rnet object and plot the result.
rnet_cycleway_intersection_clean <- rnet_breakup_vertices(rnet_cycleway_intersection)
plot(
  rnet_cycleway_intersection_clean$geometry,
  lwd = 2,
  col = rainbow(nrow(rnet_cycleway_intersection_clean)),
  cex = 2
)
plot(st.geometry(line2points(rnet_cycleway_intersection_clean)), pch = 16, add = TRUE)
par(def_par)

rnet_cycleway_intersection

Example of cycleway intersection data showing problems for SpatialLinesNetwork objects

Description
See data-raw/rnet_cycleway_intersection for details on how this was created.

Format
A sf object

Examples
rnet_cycleway_intersection

rnet_get_nodes Extract nodes from route network

Description
Extract nodes from route network

Usage
rnet_get_nodes(rnet, p = NULL)

Arguments

rnet A route network of the type generated by overline()
p A point represented by an sf object the will split the route
**Examples**

```r
rnet_get_nodes(route_network_sf)
```

---

**rnet_group**  
Assign segments in a route network to groups

**Description**

This function assigns linestring features, many of which in an `sf` object can form route networks, into groups. By default, the function `igraph::clusters()` is used to determine group membership, but any `igraph::cluster*()` function can be used. See examples and the web page `igraph.org/r/doc/communities.html` for more information. From that web page, the following clustering functions are available:

**Usage**

```r
rnet_group(rnet, ...)
```

```r
## Default S3 method:
rnet_group(rnet, ...)
```

```r
## S3 method for class 'sfc'
rnet_group(
  rnet,
  cluster_fun = igraph::clusters,  
  d = NULL,
  as.undirected = TRUE,
  ...
)
```

```r
## S3 method for class 'sf'
rnet_group(
  rnet,
  cluster_fun = igraph::clusters,  
  d = NULL,
  as.undirected = TRUE,
  ...
)
```

**Arguments**

- `rnet` An `sf`, `sfc`, or `sfNetwork` object representing a route network.
- `...` Arguments passed to other methods.
- `cluster_fun` The clustering function to use. Various clustering functions are available in the `igraph` package. Default: `igraph::clusters()`.
d

Optional distance variable used to classify segments that are close (within a certain distance specified by `d`) to each other but not necessarily touching.

`as.undirected`

Coerce the graph created internally into an undirected graph with `igraph::as.undirected()`? TRUE by default, which enables use of a wider range of clustering functions.

Details

`cluster_edge_betweenness, cluster_fast_greedy, cluster_label_prop, cluster_leading_eigen, cluster_louvain, cluster_optimal, cluster_spinglass, cluster_walktrap`

Value

If the input `rnet` is an sf/sfc object, it returns an integer vector reporting the groups of each network element. If the input is an sfNetwork object, it returns an sfNetwork object with an extra column called `rnet_group` representing the groups of each network element. In the latter case, the connectivity of the spatial object is derived from the sfNetwork object.

See Also

Other rnet: `gsection()`, `islines()`, `overline()`, `rnet_breakup_vertices()`

Examples

```r
rnet <- rnet_breakup_vertices(stplanr::osm_net_example)
rnet$group <- rnet_group(rnet)
plot(rnet["group"])
# mapview::mapview(rnet["group"])
rnet$group_25m <- rnet_group(rnet, d = 25)
plot(rnet["group_25m"])
rnet$group_walktrap <- rnet_group(rnet, igraph::cluster_walktrap)
plot(rnet["group_walktrap"])
rnet$group_louvain <- rnet_group(rnet, igraph::cluster_louvain)
plot(rnet["group_louvain"])
rnet$group_fast_greedy <- rnet_group(rnet, igraph::cluster_fast_greedy)
plot(rnet["group_fast_greedy"])
```

`rnet_overpass` Example of overpass data showing problems for SpatialLinesNetwork objects

Description

See data-raw/rnet_overpass.R for details on how this was created.

Format

A sf object

Examples

```r
rnet_overpass
```
Example of roundabout data showing problems for SpatialLinesNetwork objects

Description

See data-raw/rnet_roundabout.R for details on how this was created.

Format

A sf object

Examples

rnet_roundabout

Plan routes on the transport network

Description

Takes origins and destinations, finds the optimal routes between them and returns the result as a spatial (sf or sp) object. The definition of optimal depends on the routing function used

Usage

route(
  from = NULL,
  to = NULL,
  l = NULL,
  route_fun = cyclestreets::journey,
  wait = 0,
  n_print = 10,
  list_output = FALSE,
  cl = NULL,
  ...
)

Arguments

from An object representing origins (if lines are provided as the first argument, from is assigned to l)
to An object representing destinations
l A spatial (linestring) object
route_fun A routing function to be used for converting the lines to routes
routes_fast_sf

wait

How long to wait between routes? 0 seconds by default, can be useful when sending requests to rate limited APIs.

n_print

A number specifying how frequently progress updates should be shown

list_output

If FALSE (default) assumes spatial (linestring) object output. Set to TRUE to save output as a list.

c1

Cluster

... Arguments passed to the routing function

See Also

Other routes: `route_dodgr()`, `route_osrm()`

Examples

# Todo: add examples

---

<table>
<thead>
<tr>
<th>routes_fast_sf</th>
<th>Spatial lines dataset of commuter flows on the travel network</th>
</tr>
</thead>
</table>

Description

Simulated travel route allocated to the transport network representing the 'fastest' between cents_sf objects.

Usage

routes_fast_sf

Format

A spatial lines dataset with 49 rows and 15 columns

See Also

Other data: cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_slow_sf, zones_sf
**routes_slow_sf**

*Spatial lines dataset of commuter flows on the travel network*

**Description**

Simulated travel route allocated to the transport network representing the 'quietest' between cents_sf.

**Format**

A spatial lines dataset with 49 rows and 15 columns

**See Also**

Other data: cents_sf, destinations_sf, flow_dests, flowlines_sf, flow, od_data_lines, od_data_routes, od_data_sample, osm_net_example, read_table_builder(), route_network_sf, routes_fast_sf, zones_sf

---

**route_average_gradient**

*Return average gradient across a route*

**Description**

This function assumes that elevations and distances are in the same units.

**Usage**

```
route_average_gradient(elevations, distances)
```

**Arguments**

- `elevations`: Elevations, e.g. those provided by the cyclestreets package
- `distances`: Distances, e.g. those provided by the cyclestreets package

**See Also**

Other route_funs: `route_rolling_average()`, `route_rolling_diff()`, `route_rolling_gradient()`, `route_sequential_dist()`, `route_slope_matrix()`, `route_slope_vector()`

**Examples**

```r
r1 <- od_data_routes[od_data_routes$route_number == 2, ]
elevations <- r1$elevations
distances <- r1$distances
route_average_gradient(elevations, distances) # an average of a 4% gradient
```
route_bikecitizens

Get a route from the BikeCitizens web service

Description

See bikecitizens.net for an interactive version of the routing engine used by BikeCitizens.

Usage

```r
route_bikecitizens(
  from = NULL,
  to = NULL,
  base_url = "https://map.bikecitizens.net/api/v1/locations/route.json",
  ccode = "gb-leeds",
  routing_profile = "balanced",
  bike_profile = "citybike",
  from_lat = 53.8265,
  from_lon = -1.576195,
  to_lat = 53.80025,
  to_lon = -1.51577
)
```

Arguments

- `from` A numeric vector representing the start point
- `to` A numeric vector representing the end point
- `base_url` The base URL for the routes
- `cccode` The city code for the routes
- `routing_profile` What type of routing to use?
- `bike_profile` What type of bike?
- `from_lat` Latitude of origin
- `from_lon` Longitude of origin
- `to_lat` Latitude of destination
- `to_lon` Longitude of destination

Details

See the bikecitizens.R file in the data-raw directory of the package’s development repository for details on usage and examples.
route_dodgr

Route on local data using the dodgr package

Description

Route on local data using the dodgr package

Usage

route_dodgr(from = NULL, to = NULL, l = NULL, net = NULL)

Arguments

from An object representing origins (if lines are provided as the first argument, from is assigned to l)
to An object representing destinations
l A spatial (linestring) object
net sf object representing the route network

See Also

Other routes: route_osrm(), route()

Examples

if (requireNamespace("dodgr")) {
  from <- c(-1.5327, 53.8006) # from <- geo_code("pedallers arms leeds")
to <- c(-1.5279, 53.8044) # to <- geo_code("gzing")
# next 4 lines were used to generate `stplanr::osm_net_example`
# pts <- rbind(from, to)
# colnames(pts) <- c("X", "Y")
# net <- dodgr::dodgr_streetnet(pts = pts, expand = 0.1)
# osm_net_example <- net["highway", "name", "lanes", "maxspeed"]
r <- route_dodgr(from, to, net = osm_net_example)
plot(osm_net_example$geometry)
plot(r$geometry, add = TRUE, col = "red", lwd = 5)
}


route_google  
*Find shortest path using Google services*

**Description**

Find the shortest path using Google’s services. See the mapsapi package for details.

**Usage**

```r
route_google(from, to, mode = "walking", key = Sys.getenv("GOOGLE"), ...)
```

**Arguments**

- `from` An object representing origins (if lines are provided as the first argument, from is assigned to `l`)
- `to` An object representing destinations
- `mode` Mode of transport, walking (default), bicycling, transit, or driving
- `key` Google key. By default it is `Sys.getenv("GOOGLE")`. Set it with `usethis::edit_r_environ()`.
- `...` Arguments passed to the routing function

**Examples**

```r
## Not run:
from <- "university of leeds"
to <- "pedallers arms leeds"
r <- route(from, to, route_fun = cyclestreets::journey)
plot(r)
# r_google <- route(from, to, route_fun = mapsapi::mp_directions) # fails
r_google1 <- route_google(from, to)
plot(r_google1)
r_google <- route(from, to, route_fun = route_google)
## End(Not run)
```

route_nearest_point  
*Find nearest route to a given point*

**Description**

This function was written as a drop-in replacement for `sf::st_nearest_feature()`, which only works with recent versions of GEOS.

**Usage**

```r
route_nearest_point(r, p, id_out = FALSE)
```
**route_network_sf**

**spatial lines dataset representing a route network**

**Description**

The flow of commuters using different segments of the road network represented in the `flowlines_sf()` and `routes_fast_sf()` datasets

**Format**

A spatial lines dataset 80 rows and 1 column

**See Also**

Other data: `cents_sf`, `destinations_sf`, `flow_dests`, `flowlines_sf`, `flow`, `od_data_lines`, `od_data_routes`, `od_data_sample`, `osm_net_example`, `read_table_builder()`, `routes_fast_sf`, `routes_slow_sf`, `zones_sf`

---

**route_osrm**

*Plan routes on the transport network using the OSRM server*

**Description**

This function is a simplified and (because it uses GeoJSON not binary polyline format) slower R interface to OSRM routing services compared with the excellent `osrm::osrmRoute()` function (which can be used via the `route()` function.)

---

**Arguments**

- `r` The input route object from which the nearest route is to be found
- `p` The point whose nearest route will be found
- `id_out` Should the index of the matching feature be returned? FALSE by default

**Examples**

```r
r <- routes_fast_sf[2:6, NULL]
p <- sf::st_sfc(sf::st_point(c(-1.540, 53.826)), crs = sf::st_crs(r))
route_nearest_point(r, p, id_out = TRUE)
r_nearest <- route_nearest_point(r, p)
plot(r$geometry)
plot(p, add = TRUE)
plot(r_nearest, lwd = 5, add = TRUE)
```
route_rolling_average

Usage

route_osrm(
  from,
  to,
  osrm.server = "https://routing.openstreetmap.de/",
  osrm.profile = "foot"
)

Arguments

from An object representing origins (if lines are provided as the first argument, from is assigned to l)
to An object representing destinations
osrm.server The base URL of the routing server.getOption("osrm.server") by default.
osrm.profile The routing profile to use, e.g. "car", "bike" or "foot" (when using the routing.openstreetmap.de test server).getOption("osrm.profile") by default.
profile Which routing profile to use? One of "foot" (default) "bike" or "car" for the default open server.

See Also

Other routes: route_dodgr(), route()

Examples

# Examples no longer working due to API being down
# l1 = od_data_lines[49, ]
# l1m = od_coords(l1)
# from = l1m[, 1:2]
# to = l1m[, 3:4]
# if(curl::has_internet()) {
#  r_foot = route_osrm(from, to)
#  r_bike = route_osrm(from, to, osrm.profile = "bike")
#  r_car = route_osrm(from, to, osrm.profile = "car")
#  plot(r_foot$geometry, lwd = 9, col = "grey")
#  plot(r_bike, col = "blue", add = TRUE)
#  plot(r_car, col = "red", add = TRUE)
# }

route_rolling_average  Return smoothed averages of vector

Description

This function calculates a simple rolling mean in base R. It is useful for calculating route characteristics such as mean distances of segments and changes in gradient.
Usage

route_rolling_average(x, n = 3)

Arguments

x Numeric vector to smooth
n The window size of the smoothing function. The default, 3, will take the mean of values before, after and including each value.

See Also

Other route_funs: route_average_gradient(), route_rolling_diff(), route_rolling_gradient(), route_sequential_dist(), route_slope_matrix(), route_slope_vector()

Examples

y <- od_data_routes$elevations[od_data_routes$route_number == 2]
y route_rolling_average(y) route_rolling_average(y, n = 1) route_rolling_average(y, n = 2) route_rolling_average(y, n = 3)

route_rolling_diff Return smoothed differences between vector values

Description

This function calculates a simple rolling mean in base R. It is useful for calculating route characteristics such as mean distances of segments and changes in gradient.

Usage

route_rolling_diff(x, lag = 1, abs = TRUE)

Arguments

x Numeric vector to smooth
lag The window size of the smoothing function. The default, 3, will take the mean of values before, after and including each value.
abs Should the absolute (always positive) change be returned? True by default

See Also

Other route_funs: route_average_gradient(), route_rolling_average(), route_rolling_gradient(), route_sequential_dist(), route_slope_matrix(), route_slope_vector()
Examples

```r
r1 <- od_data_routes[od_data_routes$route_number == 2, ]
y <- r1$elevations
route_rolling_diff(y, lag = 1)
route_rolling_diff(y, lag = 2)
r1$elevations_diff_1 <- route_rolling_diff(y, lag = 1)
r1$elevations_diff_n <- route_rolling_diff(y, lag = 1, abs = FALSE)
d <- cumsum(r1$distances) - r1$distances / 2
diff_above_mean <- r1$elevations_diff_1 + mean(y)
diff_above_mean_n <- r1$elevations_diff_n + mean(y)
plot(c(0, cumsum(r1$distances)), c(y, y[length(y)]), ylim = c(80, 130))
lines(c(0, cumsum(r1$distances)), c(y, y[length(y)]))
points(d, diff_above_mean)
points(d, diff_above_mean_n, col = "blue")
abline(h = mean(y))
```

route_rolling.gradient

*Calculate rolling average gradient from elevation data at segment level*

Description

Calculate rolling average gradient from elevation data at segment level

Usage

```r
route_rolling.gradient(elevations, distances, lag = 1, n = 2, abs = TRUE)
```

Arguments

- `elevations`: Elevations, e.g. those provided by the cyclestreets package
- `distances`: Distances, e.g. those provided by the cyclestreets package
- `lag`: The window size of the smoothing function. The default, 3, will take the mean of values before, after and including each value.
- `n`: The window size of the smoothing function. The default, 3, will take the mean of values before, after and including each value.
- `abs`: Should the absolute (always positive) change be returned? True by default

See Also

Other route_funs: `route_average_gradient()`, `route_rolling_average()`, `route_rolling_diff()`, `route_sequential_dist()`, `route_slope_matrix()`, `route_slope_vector()`
Examples

```r
r1 <- od_data_routes[od_data_routes$route_number == 2, ]
y <- r1$elevations
distances <- r1$distances
route_rolling_gradient(y, distances)
route_rolling_gradient(y, distances, abs = FALSE)
route_rolling_gradient(y, distances, n = 3)
route_rolling_gradient(y, distances, n = 4)
r1$elevations_diff_1 <- route_rolling_diff(y, lag = 1)
r1$rolling_gradient <- route_rolling_gradient(y, distances, n = 2)
r1$rolling_gradient3 <- route_rolling_gradient(y, distances, n = 3)
r1$rolling_gradient4 <- route_rolling_gradient(y, distances, n = 4)
d <- cumsum(r1$distances) - r1$distances / 2
diff_above_mean <- r1$elevations_diff_1 + mean(y)
par(mfrow = c(2, 1))
plot(c(0, cumsum(r1$distances)), c(y, y[length(y)]), ylim = c(80, 130))
lines(c(0, cumsum(r1$distances)), c(y, y[length(y)]))
points(d, diff_above_mean)
abline(h = mean(y))
rg <- r1$rolling_gradient
rg[is.na(rg)] <- 0
plot(c(0, d), c(0, rg), ylim = c(0, 0.2))
points(c(0, d), c(0, r1$rolling_gradient3), col = "blue")
points(c(0, d), c(0, r1$rolling_gradient4), col = "grey")
par(mfrow = c(1, 1))
```

**route_sequential_dist**  
*Calculate the sequential distances between sequential coordinate pairs*

**Description**

Calculate the sequential distances between sequential coordinate pairs

**Usage**

```r
route_sequential_dist(m, lonlat = TRUE)
```

**Arguments**

- `m`  
  Matrix containing coordinates and elevations

- `lonlat`  
  Are the coordinates in lon/lat order? TRUE by default

**See Also**

Other route_funs: `route_average_gradient()`, `route_rolling_average()`, `route_rolling_diff()`, `route_rolling_gradient()`, `route_slope_matrix()`, `route_slope_vector()`
route_slope_matrix

Calculate the gradient of line segments from a matrix of coordinates

Examples

```r
x <- c(0, 2, 3, 4, 5, 9)
y <- c(0, 0, 0, 0, 0, 1)
m <- cbind(x, y)
route_sequential_dist(m)
```

```r
x <- c(0, 2, 3, 4, 5, 9)
y <- c(0, 0, 0, 0, 0, 9)
z <- c(1, 2, 2, 4, 3, 1) / 10
m <- cbind(x, y, z)
plot(x, z, ylim = c(-0.5, 0.5), type = "l")
(gx <- route_slope_vector(x, z))
(gxy <- route_slope_matrix(m, lonlat = FALSE))
abline(h = 0, lty = 2)
points(x[-length(x)], gx, col = "red")
points(x[-length(x)], gxy, col = "blue")
title("Distance (in x coordinates) elevation profile",
     sub = "Points show calculated gradients of subsequent lines")
```

Description

Calculate the gradient of line segments from a matrix of coordinates

Usage

```r
route_slope_matrix(m, e = m[, 3], lonlat = TRUE)
```

Arguments

- `m`: Matrix containing coordinates and elevations
- `e`: Elevations in same units as x (assumed to be metres)
- `lonlat`: Are the coordinates in lon/lat order? TRUE by default

See Also

Other route_funs: `route_average_gradient()`, `route_rolling_average()`, `route_rolling_diff()`, `route_rolling_gradient()`, `route_sequential_dist()`, `route_slope_vector()`
route_slope_vector

**Calculate the gradient of line segments from distance and elevation vectors**

**Description**
Calculate the gradient of line segments from distance and elevation vectors

**Usage**

\[
\text{route_slope_vector}(x, e)
\]

**Arguments**
- **x** Vector of locations
- **e** Elevations in same units as `x` (assumed to be metres)

**See Also**
Other route_funs: route_average_gradient(), route_rolling_average(), route_rolling_diff(), route_rolling_gradient(), route_sequential_dist(), route_slope_matrix()

**Examples**

```r
x <- c(0, 2, 3, 4, 5, 9)
e <- c(1, 2, 2, 4, 3, 1) / 10
route_slope_vector(x, e)
```

route_split

**Split route in two at point on or near network**

**Description**
Split route in two at point on or near network

**Usage**

\[
\text{route_split}(r, p)
\]

**Arguments**
- **r** An sf object with one feature containing a linestring geometry to be split
- **p** A point represented by an sf object the will split the route

**Value**
An sf object with 2 feature
Examples

```r
sample_routes <- routes_fast_sf[2:6, NULL]
r <- sample_routes[2, ]
p <- sf::st_sfc(sf::st_point(c(-1.540, 53.826)), crs = sf::st_crs(r))
plot(r$geometry, lwd = 9, col = "grey")
plot(p, add = TRUE)
r_split <- route_split(r, p)
plot(r_split, col = c("red", "blue"), add = TRUE)
```

---

**route_split_id**

Split route based on the id or coordinates of one of its vertices

Description

Split route based on the id or coordinates of one of its vertices

Usage

```r
route_split_id(r, id = NULL, p = NULL)
```

Arguments

- `r` An sf object with one feature containing a linestring geometry to be split
- `id` The index of the point on the number to be split
- `p` A point represented by an sf object the will split the route

Examples

```r
sample_routes <- routes_fast_sf[2:6, 3]
r <- sample_routes[2, ]
id <- round(n_vertices(r) / 2)
r_split <- route_split_id(r, id = id)
plot(r$geometry, lwd = 9, col = "grey")
plot(r_split, col = c("red", "blue"), add = TRUE)
```

---

**stplanr-deprecated**

Deprecated functions in stplanr

Description

These functions are deprecated and will be removed:
**toptail_buff**

*Clip the beginning and ends of sf LINestring objects*

**Description**

Takes lines and removes the start and end point, to a distance determined by the nearest buff polygon border.

**Usage**

```r
toptail_buff(l, buf, ...)
```

**Arguments**

- `l`: An sf object representing lines
- `buff`: An sf object with POLYGON geometry to buffer the linestring.
- `...`: Arguments passed to `sf::st_buffer()`

**See Also**

Other lines: `angle_diff()`, `geo_toptail()`, `is_linepoint()`, `line2df()`, `line2points()`, `line_bearing()`, `line_breakup()`, `line_midpoint()`, `line_segment()`, `line_via()`, `mats2line()`, `n_vertices()`, `onewaygeo()`, `points2line()`

**Examples**

```r
l <- routes_fast_sf
buff <- zones_sf
r_toptail <- toptail_buff(l, buff)
nrow(l)
nrow(r_toptail)
plot(zones_sf$geometry)
plot(l$geometry, add = TRUE)
plot(r_toptail$geometry, lwd = 5, add = TRUE)
```

---

**zones_sf**

*Spatial polygons of home locations for flow analysis.*

**Description**

These correspond to the `cents_sf` data.

**Details**

- `geo_code`: the official code of the zone
See Also

Other data: `cents_sf`, `destinations_sf`, `flow_dests`, `flowlines_sf`, `flow`, `od_data_lines`,
`od_data_routes`, `od_data_sample`, `osm_net_example`, `read_table_builder()`, `route_network_sf`,
`routes_fast_sf`, `routes_slow_sf`

Examples

```r
library(sf)
zones_sf
plot(zones_sf)
```
Index

* datasets
  cents_sf, 6
  destinations_sf, 7
  flow, 7
  flow_dests, 8
  flowlines_sf, 8
  od_data_lines, 31
  od_data_routes, 31
  od_data_sample, 32
  osm_net_example, 37
  rnet_cycleway_intersection, 48
  rnet_overpass, 50
  rnet_roundabout, 51
  route_network_sf, 57
  routes_fast_sf, 52
  routes_slow_sf, 53
  zones_sf, 65

* data
  cents_sf, 6
  destinations_sf, 7
  flow, 7
  flow_dests, 8
  flowlines_sf, 8
  od_data_lines, 31
  od_data_routes, 31
  od_data_sample, 32
  osm_net_example, 37
  read_table_builder, 43
  route_network_sf, 57
  routes_fast_sf, 52
  routes_slow_sf, 53
  zones_sf, 65

* geo
  bbox_scale, 5
  geo_bb, 9
  geo_bb_matrix, 10
  geo_buffer, 10
  geo_length, 12
  geo_projected, 13
  geo_select_aeq, 13
  quadrant, 42

* lines
  angle_diff, 4
  geo_toptail, 14
  is_linepoint, 17
  line2df, 17
  line2points, 18
  line_bearing, 19
  line_breakup, 20
  line_midpoint, 21
  line_segment, 21
  line_via, 22
  mats2line, 23
  n_vertices, 24
  onewaygeo, 36
  points2line, 41
  toptail_buff, 65

* nodes
  geo_code, 11

* od
  od2line, 24
  od2odf, 26
  od_aggregate_from, 27
  od_aggregate_to, 28
  od_coords, 29
  od_coords2line, 30
  od_id, 32
  od_id_order, 33
  od_oneway, 34
  od_to_odmatrix, 35
  odmatrix_to_od, 27
  points2flow, 41
  points2odf, 42

* package
  stplanr-package, 4

* rnet
  gsection, 15
  islines, 16
overline, 37
rnet_breakup_vertices, 46
rnet_group, 49
* route_funs
  route_average_gradient, 53
  route_rolling_average, 58
  route_rolling_diff, 59
  route_rolling_gradient, 60
  route_sequential_dist, 61
  route_slope_matrix, 62
  route_slope_vector, 63
* routes
  route, 51
  route_dodgr, 55
  route_osrm, 57
angle_diff, 4, 15, 17–24, 36, 41, 65
bb2poly (geo_bb), 9
bbox_scale, 5, 9–14, 43
cents_sf, 6, 7, 8, 31, 32, 37, 44, 52, 53, 57, 66
cents_sf(), 25, 26, 28
destinations_sf, 6, 7, 8, 31, 32, 37, 44, 52, 53, 57, 66
flow, 6, 7, 8, 31, 32, 37, 44, 52, 53, 57, 66
flow(), 25, 26, 28
flow_dests, 6–8, 8, 31, 32, 37, 44, 52, 53, 57, 66
flowlines_sf, 6–8, 8, 31, 32, 37, 44, 52, 53, 57, 66
flowlines_sf(), 57
geo_bb, 5, 9, 10–14, 43
geo_bb_matrix, 5, 9, 10, 11–14, 43
geo_buffer, 5, 9, 10, 12–14, 43
geo_code, 11
geo_length, 5, 9–11, 12, 13, 14, 43
g eo_projected, 5, 9–12, 13, 14, 43
g eo_select_aeq, 5, 9–13, 13, 43
g eo_select_aeq(), 13
g eo_toptail, 5, 14, 17–24, 36, 41, 65
gprojected (geo_projected), 13
gsection, 15, 16, 39, 46, 50
igraph::as.undirected(), 50
igraph::clusters(), 49
is_linepoint, 5, 15, 17, 18–24, 36, 41, 65
islines, 15, 16, 39, 46, 50
line2df, 5, 15, 17, 18–24, 36, 41, 65
line2points, 5, 15, 17, 18, 19–24, 36, 41, 65
line2pointsn (line2points), 18
line2vertices (line2points), 18
line_bearing, 5, 15, 17, 18, 19, 20–24, 36, 41, 65
line_breakup, 5, 15, 17–19, 20, 21–24, 36, 41, 65
line_midpoint, 5, 15, 17–20, 21, 22–24, 36, 41, 65
line_segment, 5, 15, 17–21, 21, 22–24, 36, 41, 65
line_via, 5, 15, 17–22, 22, 23, 24, 36, 41, 65
lwgeom::st_linesubstring(), 21
mats2line, 5, 15, 17–22, 23, 24, 36, 41, 65
n_vertices, 5, 15, 17–23, 24, 36, 41, 65
od2line, 24, 26–30, 33–36, 41, 42
od2df, 25, 26, 27–30, 33–36, 41, 42
od_aggregate_from, 25–27, 27, 29, 30, 33–36, 41, 42
od_aggregate_to, 25–28, 28, 29, 30, 33–36, 41, 42
od_coords, 25–29, 29, 30, 33–36, 41, 42
od_coords2line, 25–29, 30, 33–36, 41, 42
od_data_lines, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
od_data_routes, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
od_data_sample, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
od_id_character (od_id), 32
od_id_max_min (od_id), 32
od_id_max_min(), 34
od_id_order, 25–30, 33, 33, 35, 36, 41, 42
od_id_szudzik (od_id), 32
od_id_szudzik(), 34
od_oneway, 25–30, 33, 34, 36, 41, 42
odmatrix_to_od, 25, 26, 27, 28–30, 33–36, 41, 42
onewaygeo, 5, 15, 17–24, 36, 41, 65
osm_net_example, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
INDEX

osrm::osrmRoute(), 57
overline, 15, 16, 37, 46, 50
overline(), 16
overline2 (overline), 37
overline_intersection, 40
points2flow, 25–30, 33–36, 41, 42
points2line, 5, 15, 17–24, 36, 41, 65
points2odf, 25–30, 33–36, 41, 42
quadrant, 5, 9–14, 42
read_table_builder, 6–8, 31, 32, 37, 43, 52, 53, 57, 66
rnet_add_node, 44
rnet_boundary_df
  (rnet_boundary_points), 45
rnet_boundary_points, 45
rnet_boundary_points_lwgeom
  (rnet_boundary_points), 45
rnet_boundary_unique
  (rnet_boundary_points), 45
rnet_breakup_vertices, 15, 16, 39, 46, 50
rnet_cycleway_intersection, 48
rnet_duplicated_vertices
  (rnet_boundary_points), 45
rnet_get_nodes, 48
rnet_group, 15, 16, 39, 46, 49
rnet_overpass, 50
rnet_roundabout, 51
route, 51, 55, 58
route(), 57
route_average_gradient, 53, 59–63
route_bikecitizens, 54
route_dodgr, 52, 55, 58
route_google, 56
route_nearest_point, 56
route_network_sf, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
route_osrm, 52, 55, 57
route_rolling_average, 53, 58, 59–63
route_rolling_diff, 53, 59, 60–63
route_rolling_gradient, 53, 59, 60, 61–63
route_sequential_dist, 53, 59, 60, 61, 62, 63
route_slope_matrix, 53, 59–61, 62, 63
route_slope_vector, 53, 59–62, 63
route_split, 63
route_split_id, 64
routes_fast_sf, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
routes_fast_sf(), 57
routes_slow_sf, 6–8, 31, 32, 37, 44, 52, 53, 57, 66
stplanr (stplanr-package), 4
stplanr-deprecated, 64
stplanr-package, 4
toptyll (geo_toptyll), 14
toptyll_buff, 5, 15, 17–24, 36, 41, 65
zones_sf, 6–8, 31, 32, 37, 44, 52, 53, 57, 65