Package 'smoothr'

June 25, 2025

Type Package

Title Smooth and Tidy Spatial Features

Version 1.1.0

Description Tools for smoothing and tidying spatial features (i.e. lines and polygons) to make them more aesthetically pleasing. Smooth curves, fill holes, and remove small fragments from lines and polygons.

License GPL-3

URL https://strimas.com/smoothr/, https://github.com/mstrimas/smoothr

BugReports https://github.com/mstrimas/smoothr/issues

Depends R (>= 3.1.2)

Imports sf, stats, terra, units

Suggests covr, knitr, lwgeom, methods, rmarkdown, sp, testthat

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

NeedsCompilation no

Author Matthew Strimas-Mackey [aut, cre] (ORCID: <https://orcid.org/0000-0001-8929-7776>)

Maintainer Matthew Strimas-Mackey <mstrimas@gmail.com>

Repository CRAN

Date/Publication 2025-06-25 19:50:02 UTC

Contents

`densify` *Densify spatial lines or polygons*

Description

A wrapper for `smooth(x, method = "densify")`. This function adds additional vertices to spatial feature via linear interpolation, always while keeping the original vertices. Each line segment will be split into equal length sub-segments. This densification algorithm treats all vertices as Euclidean points, i.e. new points will not fall on a great circle between existing vertices, rather they'll be along a straight line.

Usage

```
densify(x, n = 10L, max_distance)
```

Arguments

`x`	spatial features; lines or polygons from either the `sf`, `sp`, or `terra` packages.
`n`	integer; number of times to split each line segment. Ignored if `max_distance` is specified.
`max_distance`	numeric; the maximum distance between vertices in the resulting matrix. This is the Euclidean distance and not the great circle distance.

Value

A densified polygon or line in the same format as the input data.

Examples

```
library(sf)
l <- jagged_lines$geometry[[2]]
l_dense <- densify(l, n = 2)
plot(l, lwd = 5)
plot(l_dense, col = "red", lwd = 2, lty = 2, add = TRUE)
plot(l_dense %>% st_cast("MULTIPOINT"), col = "red", pch = 19,
     add = TRUE)
```

drop_crumbs *Remove small polygons or line segments*

Description

Remove polygons or line segments below a given area or length threshold.

Usage

```
drop_crumbs(x, threshold, drop_empty = TRUE)
```

Arguments

x	spatial features; lines or polygons from either the sf, sp, or terra packages.
threshold	an area or length threshold, below which features will be removed. Provided either as a units object (see units::set_units()), or a numeric threshold in the units of the coordinate reference system. If x is in unprojected coordinates, a numeric threshold is assumed to be in meters.
drop_empty	logical; whether features with sizes below the given threshold should be removed (the default) or kept as empty geometries. Note that sp objects cannot store empty geometries, so this argument will be ignored and empty geometries will always be removed.

Details

For multipart features, the removal threshold is applied to the individual components. This means that, in some cases, an entire feature may be removed, while in other cases, only parts of the multipart feature will be removed.

Value

A spatial feature, with small pieces removed, in the same format as the input data. If none of the features are larger than the threshold, sf inputs will return a geometry set with zero features, and sp inputs will return NULL.

Examples

```
# remove polygons smaller than 200km2
p <- jagged_polygons$geometry[7]
area_thresh <- units::set_units(200, km^2)
p_dropped <- drop_crumbs(p, threshold = area_thresh)
# plot
par(mar = c(0, 0, 1, 0), mfrow = c(1, 2))
plot(p, col = "black", main = "Original")
if (length(p_dropped) > 0) {
  plot(p_dropped, col = "black", main = "After drop_crumbs()")
}
```

```
# remove lines less than 25 miles
l <- jagged_lines$geometry[8]
# note that any units can be used
# conversion to units of projection happens automatically
length_thresh <- units::set_units(25, miles)
l_dropped <- drop_crumbs(l, threshold = length_thresh)
# plot
par(mar = c(0, 0, 1, 0), mfrow = c(1, 2))
plot(l, lwd = 5, main = "Original")
if (length(l_dropped)) {
  plot(l_dropped, lwd = 5, main = "After drop_crumbs()")
}
```

`fill_holes` *Fill small holes in polygons*

Description

Fill polygon holes that fall below a given area threshold.

Usage

```
fill_holes(x, threshold)
```

Arguments

`x`	spatial features; lines or polygons from either the `sf`, `sp`, or `terra` packages.
`threshold`	an area threshold, below which holes will be removed. Provided either as a `units` object (see `units::set_units()`), or a numeric threshold in the units of the coordinate reference system. If `x` is in unprojected coordinates, a numeric threshold is assumed to be in square meters. A threshold of 0 will return the input polygons unchanged.

Value

A spatial feature, with holes filled, in the same format as the input data.

Examples

```
# fill holes smaller than 1000km2
p <- jagged_polygons$geometry[5]
area_thresh <- units::set_units(1000, km^2)
p_dropped <- fill_holes(p, threshold = area_thresh)
# plot
par(mar = c(0, 0, 1, 0), mfrow = c(1, 2))
plot(p, col = "black", main = "Original")
plot(p_dropped, col = "black", main = "After fill_holes()")
```

jagged_lines *Jagged lines for smoothing*

Description

Spatial lines in sf format for smoothing. There are examples of lines forming a closed loop and multipart lines.

Usage

```
jagged_lines
```

Format

An sf object with 9 features and 3 attribute:

- `type`: character; the geometry, i.e. "polygon" or "line".
- `closed`: logical; whether the line forms a closed loop or not.
- `multipart`: logical; whether the feature is single or multipart.

jagged_lines_3d *3D jagged line with Z-dimension for smoothing*

Description

Spatial lines in sf format for smoothing in three dimensions. There are examples of open and closed loops

Usage

```
jagged_lines_3d
```

Format

An sf object with 9 features and 3 attribute:

- `type`: character; the geometry, i.e. "polygon" or "line".
- `closed`: logical; whether the line forms a closed loop or not.
- `multipart`: logical; whether the feature is single or multipart.

jagged_polygons *Jagged polygons for smoothing*

Description

Spatial polygons in sf format for smoothing. Most of these polygons have been created by converting rasters to polygons and therefore consist entirely of right angles. There are examples of polygons with holes and multipart polygons.

Usage

```
jagged_polygons
```

Format

An sf object with 9 features and 3 attribute:

- `type`: character; the geometry, i.e. "polygon" or "line".
- `hole`: logical; whether the polygon has holes or not.
- `multipart`: logical; whether the feature is single or multipart.

jagged_raster *Simulated raster for polygonizing and smoothing*

Description

One of the primary applications of this package is for smoothing polygons generated from rasters. This example raster dataset is meant to be a simulated occurrence probability for a species, consisting of a spatially auto-correlated Gaussian field with values between 0 and 1. This raster is a 25x25 grid of 100 square kilometer cells in a North American centered Albers Equal Area projection.

Usage

```
jagged_raster
```

Format

A wrapped SpatRaster object with one layer. Call `terra::rast()` to unwrap it.

smooth *Smooth a spatial feature*

Description

Smooth out the jagged or sharp corners of spatial lines or polygons to make them appear more aesthetically pleasing and natural.

Usage

```
smooth(x, method = c("chaikin", "ksmooth", "spline", "densify"), ...)
```

Arguments

x	spatial features; lines or polygons from either the sf, sp, or terra packages.
method	character; specifies the type of smoothing method to use. Possible methods are: "chaikin", "ksmooth", "spline", and "densify". Each method has one or more parameters specifying the amount of smoothing to perform. See Details for descriptions.
...	additional arguments specifying the amount of smoothing, passed on to the specific smoothing function, see Details below.

Details

Specifying a method calls one of the following underlying smoothing functions. Each smoothing method has one or more parameters that specify the extent of smoothing. Note that for multiple features, or multipart features, these parameters apply to each individual, singlepart feature.

- smooth_chaikin(): Chaikin's corner cutting algorithm smooths a curve by iteratively replacing every point by two new points: one 1/4 of the way to the next point and one 1/4 of the way to the previous point. Smoothing parameters:
  - refinements: number of corner cutting iterations to apply.
- smooth_ksmooth(): kernel smoothing via the stats::ksmooth() function. This method first calls smooth_densify() to densify the feature, then applies Gaussian kernel regression to smooth the resulting points. Smoothing parameters:
  - smoothness: a positive number controlling the smoothness and level of generalization. At the default value of 1, the bandwidth is chosen as the mean distance between adjacent vertices. Values greater than 1 increase the bandwidth, yielding more highly smoothed and generalized features, and values less than 1 decrease the bandwidth, yielding less smoothed and generalized features.
  - bandwidth: the bandwidth of the Guassian kernel. If this argument is supplied, then smoothness is ignored and an optimal bandwidth is not estimated.
  - n: number of times to split each line segment in the densification step. Ignored if max_distance is specified.
  - max_distance: the maximum distance between vertices in the resulting features for the densification step. This is the Euclidean distance and not the great circle distance.

- smooth_spline(): spline interpolation via the stats::spline() function. This method interpolates between existing vertices and can be used when the resulting smoothed feature should pass through the vertices of the input feature. Smoothing parameters:
  - vertex_factor: the proportional increase in the number of vertices in the smooth feature. For example, if the original feature has 100 vertices, a value of 2.5 will yield a new, smoothed feature with 250 vertices. Ignored if n is specified.
  - n: number of vertices in each smoothed feature.
- smooth_densify(): densification of vertices for lines and polygons. This is not a true smoothing algorithm, rather new vertices are added to each line segment via linear interpolation. Densification parameters:
  - n: number of times to split each line segment. Ignored if max_distance is specified.
  - max_distance: the maximum distance between vertices in the resulting feature. This is the Euclidean distance and not the great circle distance.

Value

A smoothed polygon or line in the same format as the input data.

References

See specific smoothing function help pages for references.

See Also

smooth_chaikin() smooth_ksmooth() smooth_spline() smooth_densify()

Examples

```
library(sf)
# compare different smoothing methods
# polygons
par(mar = c(0, 0, 0, 0), oma = c(4, 0, 0, 0), mfrow = c(3, 3))
p_smooth_chaikin <- smooth(jagged_polygons, method = "chaikin")
p_smooth_ksmooth <- smooth(jagged_polygons, method = "ksmooth")
p_smooth_spline <- smooth(jagged_polygons, method = "spline")
for (i in 1:nrow(jagged_polygons)) {
  plot(st_geometry(p_smooth_spline[i, ]), col = NA, border = NA)
  plot(st_geometry(jagged_polygons[i, ]), col = "grey40", border = NA, add = TRUE)
 plot(st_geometry(p_smooth_chaikin[i, ]), col = NA, border = "#E41A1C", lwd = 2, add = TRUE)
 plot(st_geometry(p_smooth_ksmooth[i, ]), col = NA, border = "#4DAF4A", lwd = 2, add = TRUE)
 plot(st_geometry(p_smooth_spline[i, ]), col = NA, border = "#377EB8", lwd = 2, add = TRUE)
}
par(fig = c(0, 1, 0, 1), oma = c(0, 0, 0, 0), new = TRUE)
plot(0, 0, type = "n", bty = "n", xaxt = "n", yaxt = "n", axes = FALSE)
legend("bottom", legend = c("chaikin", "ksmooth", "spline"),
       col = c("#E41A1C", "#4DAF4A", "#377EB8"),
       lwd = 2, cex = 2, box.lwd = 0, inset = 0, horiz = TRUE)

# lines
par(mar = c(0, 0, 0, 0), oma = c(4, 0, 0, 0), mfrow = c(3, 3))
```

```
l_smooth_chaikin <- smooth(jagged_lines, method = "chaikin")
l_smooth_ksmooth <- smooth(jagged_lines, method = "ksmooth")
l_smooth_spline <- smooth(jagged_lines, method = "spline")
for (i in 1:nrow(jagged_lines)) {
  plot(st_geometry(l_smooth_spline[i, ]), col = NA)
  plot(st_geometry(jagged_lines[i, ]), col = "grey20", lwd = 3, add = TRUE)
  plot(st_geometry(l_smooth_chaikin[i, ]), col = "#E41A1C", lwd = 2, lty = 2, add = TRUE)
  plot(st_geometry(l_smooth_ksmooth[i, ]), col = "#4DAF4A", lwd = 2, lty = 2, add = TRUE)
  plot(st_geometry(l_smooth_spline[i, ]), col = "#377EB8", lwd = 2, lty = 2, add = TRUE)
}
par(fig = c(0, 1, 0, 1), oma = c(0, 0, 0, 0), new = TRUE)
plot(0, 0, type = "n", bty = "n", xaxt = "n", yaxt = "n", axes = FALSE)
legend("bottom", legend = c("chaikin", "smooth", "spline"),
       col = c("#E41A1C", "#4DAF4A", "#377EB8"),
       lwd = 2, cex = 2, box.lwd = 0, inset = 0, horiz = TRUE)
```

`smooth_chaikin` *Chaikin's corner cutting algorithm*

Description

Chaikin's corner cutting algorithm smooths a curve by iteratively replacing every point by two new points: one 1/4 of the way to the next point and one 1/4 of the way to the previous point.

Usage

```
smooth_chaikin(x, wrap = FALSE, refinements = 3L)
```

Arguments

`x`	numeric matrix; 2-column matrix of coordinates.
`wrap`	logical; whether the coordinates should be wrapped at the ends, as for polygons and closed lines, to ensure a smooth edge.
`refinements`	integer; number of corner cutting iterations to apply.

Details

This function works on matrices of points and is generally not called directly. Instead, use `smooth()` with `method = "chaikin"` to apply this smoothing algorithm to spatial features.

Value

A matrix with the coordinates of the smoothed curve.

References

The original reference for Chaikin's corner cutting algorithm is:

- Chaikin, G. An algorithm for high speed curve generation. Computer Graphics and Image Processing 3 (

This implementation was inspired by the following StackOverflow answer:

- Where to find Python implementation of Chaikin's corner cutting algorithm?

See Also

smooth()

Examples

```
# smooth_chaikin works on matrices of coordinates
# use the matrix of coordinates defining a polygon as an example
m <- jagged_polygons$geometry[[2]][[1]]
m_smooth <- smooth_chaikin(m, wrap = TRUE)
class(m)
class(m_smooth)
plot(m, type = "l", axes = FALSE, xlab = NA, ylab = NA)
lines(m_smooth, col = "red")

# smooth is a wrapper for smooth_chaikin that works on spatial features
library(sf)
p <- jagged_polygons$geometry[[2]]
p_smooth <- smooth(p, method = "chaikin")
class(p)
class(p_smooth)
plot(p)
plot(p_smooth, border = "red", add = TRUE)
```

---

smooth_densify *Densify lines or polygons*

Description

This function adds additional vertices to lines or polygons via linear interpolation, always while keeping the original vertices. Each line segment will be split into equal length sub-segments. This densification algorithm treats all vertices as Euclidean points, i.e. new points will not fall on a great circle between existing vertices, rather they'll be along a straight line.

Usage

```
smooth_densify(x, wrap = FALSE, n = 10L, max_distance)
```

Arguments

x	numeric matrix; matrix of coordinates.
wrap	logical; whether the coordinates should be wrapped at the ends, as for polygons and closed lines, to ensure a smooth edge.
n	integer; number of times to split each line segment. Ignored if max_distance is specified.
max_distance	numeric; the maximum distance between vertices in the resulting matrix. This is the Euclidean distance and not the great circle distance.

Details

This function works on matrices of points and is generally not called directly. Instead, use smooth() with method = "densify" to apply this smoothing algorithm to spatial features.

Value

A matrix with the coordinates of the densified curve.

Examples

```
# smooth_densify works on matrices of coordinates
# use the matrix of coordinates defining a line as an example
m <- jagged_lines$geometry[[2]][]
m_dense <- smooth_densify(m, n = 5)
class(m)
class(m_dense)
plot(m, type = "b", pch = 19, cex = 1.5, axes = FALSE, xlab = NA, ylab = NA)
points(m_dense, col = "red", pch = 19, cex = 0.5)

# max_distance can be used to ensure vertices are at most a given dist apart
m_md <- smooth_densify(m, max_distance = 0.05)
plot(m, type = "b", pch = 19, cex = 1.5, axes = FALSE, xlab = NA, ylab = NA)
points(m_md, col = "red", pch = 19, cex = 0.5)

# smooth is a wrapper for smooth_densify that works on spatial features
library(sf)
l <- jagged_lines$geometry[[2]]
l_dense <- smooth(l, method = "densify", n = 2)
class(l)
class(l_dense)
plot(l, lwd = 5)
plot(l_dense, col = "red", lwd = 2, lty = 2, add = TRUE)
plot(l_dense %>% st_cast("MULTIPOINT"), col = "red", pch = 19,
     add = TRUE)
```

smooth_ksmooth *Kernel smooth*

Description

Kernel smoothing uses stats::ksmooth() to smooth out existing vertices using Gaussian kernel regression. Kernel smoothing is applied to the x and y coordinates are independently. Prior to smoothing, smooth_densify() is called to generate additional vertices, and the smoothing is applied to this densified set of vertices.

Usage

```
smooth_ksmooth(
  x,
  wrap = FALSE,
  smoothness = 1,
  bandwidth,
  n = 10L,
  max_distance
)
```

Arguments

x	numeric matrix; 2-column matrix of coordinates.
wrap	logical; whether the coordinates should be wrapped at the ends, as for polygons and closed lines, to ensure a smooth edge.
smoothness	numeric; a parameter controlling the bandwidth of the Gaussian kernel, and therefore the smoothness and level of generalization. By default, the bandwidth is chosen as the mean distance between adjacent points. The smoothness parameter is a multiplier of this chosen bandwidth, with values greater than 1 yielding more highly smoothed and generalized features and values less than 1 yielding less smoothed and generalized features.
bandwidth	numeric; the bandwidth of the Guassian kernel. If this argument is supplied, then smoothness is ignored and an optimal bandwidth is not estimated.
n	integer; number of times to split each line segment for smooth_densify(). Ignored if max_distance is specified.
max_distance	numeric; the maximum distance between vertices for smooth_densify(). This is the Euclidean distance and not the great circle distance.

Details

Kernel smoothing both smooths and generalizes curves, and the extent of these effects is dependent on the bandwidth of the smoothing kernel. Therefore, choosing a sensible bandwidth is critical when using this method. The choice of bandwidth will be dependent on the projection, scale, and desired amount of smoothing and generalization. The are two methods of adjusting the bandwidth. By default, the bandwidth will be set to the average distances between adjacent vertices. The

`smoothness` factor can then be used to adjust this calculated bandwidth, values greater than 1 will lead to more smoothing, values less than 1 will lead to less smoothing. Alternatively, the bandwidth can be chosen manually with the `bandwidth` argument. Typically, users will need to explore a range of bandwidths to determine which yields the best results for their situation.

This function works on matrices of points and is generally not called directly. Instead, use `smooth()` with `method = "ksmooth"` to apply this smoothing algorithm to spatial features.

Value

A matrix with the coordinates of the smoothed curve.

References

The kernel smoothing method was inspired by the following StackExchange answers:

- Nadaraya-Watson Optimal Bandwidth
- Smoothing polygons in contour map?

See Also

`smooth()`

Examples

```
# smooth_ksmooth works on matrices of coordinates
# use the matrix of coordinates defining a polygon as an example
m <- jagged_polygons$geometry[[2]][[1]]
m_smooth <- smooth_ksmooth(m, wrap = TRUE)
class(m)
class(m_smooth)
plot(m, type = "l", col = "black", lwd = 3, axes = FALSE, xlab = NA,
     ylab = NA)
lines(m_smooth, lwd = 3, col = "red")

# lines can also be smoothed
l <- jagged_lines$geometry[[2]][]
l_smooth <- smooth_ksmooth(l, wrap = FALSE, max_distance = 0.05)
plot(l, type = "l", col = "black", lwd = 3, axes = FALSE, xlab = NA,
     ylab = NA)
lines(l_smooth, lwd = 3, col = "red")

# explore different levels of smoothness
p <- jagged_polygons$geometry[[2]][[1]]
ps1 <- smooth_ksmooth(p, wrap = TRUE, max_distance = 0.01, smoothness = 0.5)
ps2 <- smooth_ksmooth(p, wrap = TRUE, max_distance = 0.01, smoothness = 1)
ps3 <- smooth_ksmooth(p, wrap = TRUE, max_distance = 0.01, smoothness = 2)
# plot
par(mar = c(0, 0, 0, 0), oma = c(10, 0, 0, 0))
plot(p, type = "l", col = "black", lwd = 3, axes = FALSE, xlab = NA,
     ylab = NA)
lines(ps1, lwd = 3, col = "#E41A1C")
```

```
lines(ps2, lwd = 3, col = "#4DAF4A")
lines(ps3, lwd = 3, col = "#377EB8")
par(fig = c(0, 1, 0, 1), oma = c(0, 0, 0, 0), new = TRUE)
plot(0, 0, type = "n", bty = "n", xaxt = "n", yaxt = "n", axes = FALSE)
legend("bottom", legend = c("0.5", "1", "2"),
       col = c("#E41A1C", "#4DAF4A", "#377EB8"),
       lwd = 3, cex = 2, box.lwd = 0, inset = 0, horiz = TRUE)

library(sf)
p <- jagged_polygons$geometry[[2]]
p_smooth <- smooth(p, method = "ksmooth")
class(p)
class(p_smooth)
plot(p_smooth, border = "red")
plot(p, add = TRUE)
```

`smooth_spline` *Spline interpolation*

Description

Spline interpolation uses `stats::spline()` to interpolate between existing vertices using piecewise cubic polynomials. The coordinates are interpolated independently. The curve will always pass through the vertices of the original feature.

Usage

```
smooth_spline(x, wrap = FALSE, vertex_factor = 5, n)
```

Arguments

`x`	numeric matrix; matrix of coordinates.
`wrap`	logical; whether the coordinates should be wrapped at the ends, as for polygons and closed lines, to ensure a smooth edge.
`vertex_factor`	double; the proportional increase in the number of vertices in the smooth curve. For example, if the original curve has 100 points, a value of `2.5` will yield a new smoothed curve with 250 points. Ignored if `n` is specified.
`n`	integer; number of vertices in the smoothed curve.

Details

This function works on matrices of points and is generally not called directly. Instead, use `smooth()` with `method = "spline"` to apply this smoothing algorithm to spatial features.

Value

A matrix with the coordinates of the smoothed curve.

References

The spline method was inspired by the following StackExchange answers:

- Create polygon from set of points distributed
- Smoothing polygons in contour map?

See Also

smooth()

Examples

```
# smooth_spline works on matrices of coordinates
# use the matrix of coordinates defining a polygon as an example
m <- jagged_polygons$geometry[[2]][[1]]
m_smooth <- smooth_spline(m, wrap = TRUE)
class(m)
class(m_smooth)
plot(m_smooth, type = "l", col = "red", axes = FALSE, xlab = NA, ylab = NA)
lines(m, col = "black")

# smooth is a wrapper for smooth_spline that works on spatial features
library(sf)
p <- jagged_polygons$geometry[[2]]
p_smooth <- smooth(p, method = "spline")
class(p)
class(p_smooth)
plot(p_smooth, border = "red")
plot(p, add = TRUE)
```

Index