Package 'spanner'

May 9, 2026

Type Package

Title Utilities to Support Lidar Applications at the Landscape, Forest, and Tree Scale

Version 1.0.4

Date 2026-03-31

Description
Implements algorithms for terrestrial, mobile, and airborne lidar processing, tree detection, segmentation, and attribute estimation (Donager et al., 2021) <doi:10.3390/rs13122297>, and a hierarchical patch delineation algorithm 'PatchMorph' (Girvetz & Greco, 2007) <doi:10.1007/s10980-007-9104-8>. Tree detection uses rasterized point cloud metrics (relative neighborhood density and verticality) combined with RANSAC cylinder fitting to locate tree boles and estimate diameter at breast height. Tree segmentation applies graph-theory approaches inspired by Tao et al. (2015) <doi:10.1016/j.isprsjprs.2015.08.007> with cylinder fitting methods from de Conto et al. (2017) <doi:10.1016/j.compag.2017.07.019>. PatchMorph delineates habitat patches across spatial scales using organism-specific thresholds. Built on 'lidR' (Roussel et al., 2020) <doi:10.1016/j.rse.2020.112061>.

URL https://github.com/bi0m3trics/spanner

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.3

RdMacros mathjaxr

LinkingTo lidR, RcppArmadillo, Rcpp (>= 1.0.13), RcppEigen, BH,

Imports Rcpp (>= 1.0.13), conicfit, FNN, RANN, cppRouting, sf, terra, sfheaders, Rfast, geometry, dplyr, mathjaxr, data.table

Depends R (>= 4.0.0), lidR (>= 4.2.0),

Suggests testthat (>= 3.0.0), magick, rgl, rstac

Config/testthat/edition 3

NeedsCompilation yes

Author Andrew Sanchez Meador [aut, cre, ctb] (ORCID:
<https://orcid.org/0000-0003-4238-8587>),
Jonathon Donager [aut, ctb] (ORCID:
<https://orcid.org/0000-0001-9448-1703>),
Blackburn Ryan [aut, ctb] (ORCID:
<https://orcid.org/0000-0002-2952-0865>),
Cannon Jeffery [ctb] (ORCID: <https://orcid.org/0000-0002-8436-8712>),
Tiago de Conto [ctb, cph] (Author of included TreeLS code),
Keith O'Hara [ctb, cph] (Author of included OptimLib code)

Maintainer Andrew Sanchez Meador <Andrew.SanchezMeador@nau.edu>

Repository CRAN

Date/Publication 2026-04-01 05:10:02 UTC

Contents

`colorize_las` *Colorize a LAS object using multiple methods*

Description

Colors a LAS object using one of three methods: attribute-based coloring, raster-based RGB coloring, or PCV (Portion de Ciel Visible) ambient occlusion.

Usage

```
colorize_las(
  las,
  method = "attr",
  attribute_name = NULL,
```

```
  palette = c("black", "white"),
  raster_path = NULL,
  radius = 1,
  num_directions = 60,
  kernel_size = 5,
  pixel_size = 0.1,
  num_samples = 16,
  ncpu = 4
)
```

Arguments

`las`	LAS object to colorize
`method`	Character string specifying the coloring method: "attr" for attribute-based, "rgb" for raster-based, or "pcv" for ambient occlusion. Default is "attr".
`attribute_name`	Character string specifying the attribute name (required for method="attr"). The attribute must exist in the LAS data.
`palette`	Character vector of at least two colors for the color ramp (used with method="attr", "pcv", or "ssao"). Colors can be hex codes (e.g., "#FF0000") or named colors (e.g., "red"). Default is grayscale.
`raster_path`	Character string or vector of paths to raster files (required for method="rgb"). Can be a single RGB raster or three separate rasters for R, G, and B channels.
`radius`	Numeric radius for neighborhood search in PCV calculation (method="pcv"). Default is 1.0.
`num_directions`	Integer number of directional rays for PCV calculation (method="pcv"). Default is 60.
`kernel_size`	Integer kernel size in pixels for SSAO sampling (method="ssao"). Default is 5.
`pixel_size`	Numeric resolution of the depth map in spatial units (method="ssao"). Default is 0.1.
`num_samples`	Integer number of samples per point for SSAO (method="ssao"). Default is 16.
`ncpu`	Integer number of CPUs to use for parallel processing (method="pcv" or "ssao"). Default is 4.

Details

The function supports four coloring methods:

attr Attribute-based coloring: normalizes attribute values and maps them to colors using the palette.

rgb Raster-based coloring: extracts RGB values from georeferenced raster(s) that align with the point cloud. Requires matching CRS between LAS and raster. Can use a single 3-band RGB raster or three separate rasters.

pcv PCV (Portion de Ciel Visible): computes 3D ambient occlusion by calculating sky visibility for each point. Based on the algorithm from Duguet & Girardeau-Montaut (2004). More accurate but slower than SSAO.

ssao SSAO (Screen Space Ambient Occlusion): fast ambient occlusion using 2D depth map techniques. Projects points to a depth buffer and calculates occlusion based on depth differences. Much faster than PCV.

Value

A LAS object with updated R, G, and B fields based on the selected method.

Examples

```
LASfile <- system.file("extdata", "ALS_Clip.laz", package="spanner")
las <- readLAS(LASfile, select = "xyz")

# Attribute-based coloring
las_colored <- colorize_las(las, method="attr", attribute_name="Z",
                            palette=c("blue", "green", "yellow", "red"))

# Raster-based coloring with RGB file
rgb_file <- system.file("extdata", "UAS_Clip_RGB.tif", package="spanner")
las_colored <- colorize_las(las, method="rgb", raster_path=rgb_file)

# PCV ambient occlusion (slow, high quality)
las_colored <- colorize_las(las, method="pcv", radius=1.0,
                            num_directions=30, palette=c("black", "white"))

# SSAO ambient occlusion (faster alternative to PCV)
las_colored <- colorize_las(las, method="ssao", pixel_size=0.1,
                     kernel_size=5, num_samples=16, palette=c("black", "white"), ncpu=8)
```

---

create_rotation_gif *Create animated GIF of rotating 3D point cloud*

Description

Generates a 360-degree rotating animation of a LAS point cloud using rgl and saves it as an animated GIF.

Usage

```
create_rotation_gif(
  las,
  output_path = "pointcloud_rotation.gif",
  duration = 12,
  rpm = 5,
  background = "white",
  axis = "z",
  show_axis = TRUE,
  show_legend = TRUE,
  screen_size = c(800, 600),
  overwrite = FALSE
)
```

Arguments

las	LAS object to visualize. Should have R, G, B fields for color.
output_path	Character string specifying output GIF file path. Default is "pointcloud_rotation.gif".
duration	Numeric duration of the animation in seconds. Default is 12.
rpm	Numeric rotations per minute for the spin. Default is 5.
background	Character string specifying background color. Default is "white".
axis	Character specifying rotation axis: "z" for vertical rotation (default), "x" for horizontal rotation, "y" for front-to-back rotation.
show_axis	Logical whether to show axes. Default is TRUE.
show_legend	Logical whether to show legend. Default is TRUE.
screen_size	Numeric vector of length 2 specifying window dimensions as c(width, height). Default is c(800, 600).
overwrite	Logical whether to overwrite existing output file. Default is FALSE.

Details

This function creates a smooth 360-degree rotation animation by:

- Plotting the point cloud using lidR's plot function with RGB colors
- Using rgl's movie3d and spin3d to create smooth rotation
- Saving the result as an animated GIF

The rotation speed is controlled by the rpm (rotations per minute) parameter. The total duration determines how long the animation will be.

Requires the `rgl` package and lidR for plotting.

Value

Character string of the output file path (invisible)

Examples

```
# create_rotation_gif requires a live display (rgl). Only run interactively.
if (interactive()) {
  LASfile <- system.file("extdata", "ALS_Clip.laz", package="spanner")
  las <- readLAS(LASfile)
  las_colored <- colorize_las(las, method="attr", attribute_name="Z")

  create_rotation_gif(las_colored, output_path=tempfile(fileext = ".gif"))

  # High quality with specific settings
  create_rotation_gif(las_colored,
                      output_path=tempfile(fileext = ".gif"),
                      duration=15,
                      rpm=10,
                      background="black",
                      show_axis=FALSE,
```

```
                    show_legend=FALSE)

  # Rotate around X axis for side-to-side view
  create_rotation_gif(las_colored, output_path=tempfile(fileext = ".gif"), axis="x")
}
```

`cylinderFit` *Point cloud cylinder fitting as per de Conto et al. 2017 as implemented here: https://github.com/tiagodc/TreeLS*

Description

Fits a cylinder on a set of 3D points.

Usage

```
cylinderFit(
  las,
  method = "ransac",
  n = 5,
  inliers = 0.9,
  conf = 0.95,
  max_angle = 30,
  n_best = 20
)
```

Arguments

`las`	LAS normalized and segmented las object.
`method`	method for estimating the cylinder parameters. Currently available: `"nm"`, `"irls"`, `"ransac"` and `"bf"`.
`n`	number of points selected on every RANSAC iteration.
`inliers`	expected proportion of inliers among stem segments' point cloud chunks.
`conf`	confidence level.
`max_angle`	used when `method == "bf"`. The maximum tolerated deviation, in degrees, from an absolute vertical line (Z = c(0,0,1)).
`n_best`	estimate optimal RANSAC parameters as the median of the `n_best` estimations with lowest error.

Value

vector of parameters

Examples

```
# Define the cylinder attributes
npts = 500
cyl_length = 0.5
radius = 0.2718

# Generate the X,Y,Z values
Z=runif(n = npts, min = 0, max = cyl_length)
angs = runif(npts, 0, 2*pi)
X = sin(angs)*radius
Y = cos(angs)*radius

# Creation of a LAS object out of external data
cloud <- LAS(data.frame(X,Y,Z))

# Fit a cylinder and retrun the information
cyl_par = spanner::cylinderFit(cloud, method = 'ransac', n=5, inliers=.9,
                               conf=.95, max_angle=30, n_best=20)
```

---

download_naip_for_las *Download NAIP Imagery for LiDAR Extent*

Description

Downloads NAIP (National Agriculture Imagery Program) imagery from Microsoft Planetary Computer STAC API for the extent of a LAS/LAZ point cloud.

Usage

```
download_naip_for_las(
  las,
  output_path = NULL,
  year_range = c("2018-01-01", "2023-12-31"),
  buffer = 0,
  overwrite = FALSE
)
```

Arguments

las	A LAS object or path to a LAS/LAZ file
output_path	Character string specifying output file path for the downloaded imagery. If NULL, creates a file named based on the input LAS file.
year_range	Character vector of length 2 specifying date range for NAIP imagery in format c("YYYY-MM-DD", "YYYY-MM-DD"). Default is c("2018-01-01", "2023-12-31").
buffer	Numeric value to buffer the extent in meters. Default is 0.
overwrite	Logical, whether to overwrite existing output file. Default is FALSE.

Details

This function queries the Microsoft Planetary Computer STAC API to find and download NAIP imagery that overlaps with the extent of the input LAS file. The imagery is automatically cropped to match the LiDAR extent and saved as a GeoTIFF.

NAIP imagery is typically 4-band (RGB + NIR) with 0.6m or 1m resolution, collected annually or biannually across the continental United States.

Requires the `rstac` package for STAC API access.

Value

Character string of the output file path, or NULL if download failed.

Examples

```
# Load example LAS file
LASfile <- system.file("extdata", "ALS_Clip.laz", package="spanner")
las <- readLAS(LASfile)

# Download NAIP for a LAS file
naip_path <- download_naip_for_las(las, output_path = tempfile(fileext = ".tif"))

# Download with buffer and specific year range
naip_path2 <- download_naip_for_las(las, buffer = 10,
                                    output_path = tempfile(fileext = ".tif"),
                                    year_range = c("2020-01-01", "2023-12-31"))

# Then use with colorize_las
las_colored <- colorize_las(las, method = "rgb", raster_path = naip_path)
```

`eigen_metrics` *Calculates eigen decomposition metrics for fixed neighborhood point cloud data*

Description

This function calculates twelve (plus the first and second PCA) for several point geometry-related metrics (listed below) in parallel using C++ for a user-specified radius.

Usage

```
eigen_metrics(las = las, radius = 0.1, ncpu = 8)
```

Arguments

`las`	LAS Normalized las object.
`radius`	numeric the radius of the neighborhood
`ncpu`	integer the number of cpu's to be used in parallelfor the calculation

Value

A labeled data.table of point metrics for each point in the LAS object

List of available point metrics

- `eLargest`: first eigenvalue, λ_1
- `eMedium`: second eigenvalue, λ_2
- `eSmallest`: third eigenvalue, λ_3
- `eSum`: sum of eigenvalues, $\sum_{i=1}^{n=3} \lambda_i$
- `Curvature`: surface variation, $\lambda_3 / \sum_{i=1}^{n=3} \lambda_i$
- `Omnivariance`: high values correspond to spherical features and low values to planes or linear features, $(\lambda_1 * \lambda_2 * \lambda_3)^{1/3}$
- `Anisotropy`: relationships between the directions of the point distribution, $(\lambda_1 - \lambda_3)/\lambda_1$
- `Eigentropy`: entropy in the eigenvalues, $-\sum_{i=1}^{n=3} \lambda_i * ln(\lambda_i)$
- `Linearity`: linear saliency, $(\lambda_1 - \lambda_2)/\lambda_1$
- `Verticality`: vertical saliency, $1 - abs(\langle (0,0,1), e_3 \rangle)$
- `Planarity`: planar saliency, $(\lambda_2 - \lambda_3)/\lambda_1$
- `Sphericity`: spherical saliency, λ_3/λ_1
- `Nx,Ny,Nz`: 3 components of the normal vector (smallest eigenvector)
- `SurfaceVariation`: surface variation (change of curvature), same as Curvature
- `ChangeCurvature`: alternative name for surface variation
- `SurfaceDensity`: 2D point density using circle area, $k/(\pi R^2)$
- `VolumeDensity`: 3D point density using sphere volume, $k/(\frac{4}{3}\pi R^3)$
- `MomentOrder1`: 1st order moment from CloudCompare, projection onto 2nd eigenvector, m_1^2/m_2
- `NormalChangeRate`: normal change rate, same as Curvature, $\lambda_3 / \sum_{i=1}^{n=3} \lambda_i$
- `Roughness`: distance from query point to fitted plane, $|\vec{d} \cdot \vec{n}|$
- `MeanCurvature`: mean curvature from quadric surface fitting, $H = \frac{(1+f_y^2)f_{xx} - 2f_x f_y f_{xy} + (1+f_x^2)f_{yy}}{2(1+f_x^2+f_y^2)^{3/2}}$
- `GaussianCurvature`: Gaussian curvature from quadric surface fitting, $K = \frac{f_{xx}f_{yy} - f_{xy}^2}{(1+f_x^2+f_y^2)^2}$
- `PCA1`: eigenvector projection variance normalized by eigensum, $\sigma_{PC1}^2 / \sum_{i=1}^{n=3} \lambda_i$
- `PCA2`: eigenvector projection variance normalized by eigensum, $\sigma_{PC2}^2 / \sum_{i=1}^{n=3} \lambda_i$
- `NumNeighbors`: number of points in the spherical neighborhood, k

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
eigen = eigen_metrics(las, radius=2, ncpu=4)
```

get_raster_eigen_treelocs

Obtain tree information by rasterizing point cloud values of relative neighborhood density and verticality within a slice of a normalized point cloud

Description

get_raster_eigen_treelocs returns a data.frame containing TreeID, X, Y, Z, Radius and Error in the same units as the .las

Usage

```
get_raster_eigen_treelocs(
  las = las,
  res = 0.05,
  pt_spacing = 0.0254,
  dens_threshold = 0.2,
  neigh_sizes = c(0.333, 0.166, 0.5),
  eigen_threshold = 0.6666,
  grid_slice_min = 0.6666,
  grid_slice_max = 2,
  minimum_polygon_area = 0.025,
  cylinder_fit_type = "ransac",
  max_dia = 0.5,
  SDvert = 0.25,
  n_best = 25,
  n_pts = 20,
  inliers = 0.9,
  conf = 0.99,
  max_angle = 20
)
```

Arguments

las	LAS Normalized las object.
res	numeric Pixel width of rasterized point cloud metrics.
pt_spacing	numeric Subsample spacing for graph connections.
dens_threshold	numeric Minimum point density in raster cell to be considered as potential tree bole.
neigh_sizes	numeric Vector for verticality and relative density (small and large neighborhoods) calculations
eigen_threshold	numeric Minimum average verticality in raster cell to be considered as potential tree bole.

grid_slice_min	numeric Lower bound of point cloud slice in normalized point cloud.
grid_slice_max	numeric Upper bound of point cloud slice in normalized point cloud.
minimum_polygon_area	numeric Smallest allowable polygon area of potential tree boles.
cylinder_fit_type	character Choose "ransac" or "irls" cylinder fitting.
max_dia	numeric The max diameter (in m) of a resulting tree (use to eliminate commission errors).
SDvert	numeric The standard deviation threshold below which polygons will be considered as tree boles.
n_best	integer number of "best" ransac fits to keep when evaluating the best fit.
n_pts	integer number of point to be selected per ransac iteraiton for fitting.
inliers	integer expected proportion of inliers among cylinder points
conf	numeric confidence level
max_angle	numeric maximum tolerated deviation, in degrees, from vertical.

Details

For terrestrial and mobile lidar datasets, tree locations and estimates of DBH are provided by rasterizing individual point cloud values of relative neighborhood density (at 0.3 and 1 m radius) and verticality within a slice of the normalized point cloud around breast height (1.34 m). The algorithim then uses defined threshold values to classify the resulting rasters and create unique polygons from the resulting classified raster. These point-density and verticality polygons were selected by their intersection with one another, resulting in a final set of polygons which were used to clip out regions of the point cloud that were most likely to represent tree boles. A RANSAC cylinder fitting algorithm was then used to estimate the fit of a cylinder to individual bole points. Cylinder centers and radius were used as inputs to an individual tree segmentation

Value

sf A sf object containing the following tree seed information: `TreeID`, `Radius`, and `Error` in the same units as the .las, as well as the point geometry

Examples

```
# Set the number of threads to use in lidR
set_lidr_threads(8)

LASfile = system.file("extdata", "TLS_Clip.laz", package="spanner")
las = readTLSLAS(LASfile, select = "xyzcr", "-filter_with_voxel 0.01")
# Don't forget to make sure the las object has a projection
sf::st_crs(las) <- 26912

# Pre-process the example lidar dataset by classifying the ground  and noise points
# using lidR::csf(), normalizing it, and removing outlier points
# using lidR::ivf()
```

```
# las = classify_ground(las, csf(sloop_smooth = FALSE,
#                                class_threshold = 0.5,
#                               cloth_resolution = 0.5, rigidness = 1L,
#                                iterations = 500L, time_step = 0.65))
# las = normalize_height(las, tin())
# las = classify_noise(las, ivf(0.25, 3))
# las = filter_poi(las, Classification != LASNOISE)

# Plot the non-ground points, colored by height
# plot(filter_poi(las, Classification != 2), color = "Z")

# find tree locations and attribute data
myTreeLocs = get_raster_eigen_treelocs(las = las, res = 0.025, pt_spacing = 0.0254,
                                       dens_threshold = 0.25,
                                       neigh_sizes = c(0.25, 0.15, 0.66),
                                       eigen_threshold = 0.75,
                                       grid_slice_min = 1,
                                       grid_slice_max = 2,
                                       minimum_polygon_area = 0.005,
                                       cylinder_fit_type = "ransac",
                                       max_dia = 1,
                                       SDvert = 0.33,
                                       n_pts = 20,
                                       n_best = 25,
                                       inliers = 0.9,
                                       conf = 0.99,
                                       max_angle = 20)

# Plot results if trees were found
if (!is.null(myTreeLocs) && nrow(myTreeLocs) > 0) {
  plot(lidR::rasterize_canopy(las, res = 0.2, p2r()))
  symbols(sf::st_coordinates(myTreeLocs)[,1], sf::st_coordinates(myTreeLocs)[,2],
          circles = myTreeLocs$Radius^2*3.14, inches = FALSE, add = TRUE, bg = 'black')
} else {
  message("No tree locations were found. Try adjusting the parameters.")
}
```

`las2xyz` *Convert LAS object to XYZ matrix*

Description

Extracts the X, Y, and Z coordinates from a LAS object and returns them as a matrix.

Usage

```
las2xyz(las)
```

Arguments

las	LAS object to convert

Value

A numeric matrix with three columns (X, Y, Z) containing the point coordinates

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
xyz_matrix <- las2xyz(las)
head(xyz_matrix)
```

---

merge_las_colors *Merge RGB colors from two colorized LAS objects*

Description

Blends the RGB values from two LAS objects to create a new composite coloring. Useful for combining different coloring methods (e.g., ambient occlusion with raster RGB).

Usage

```
merge_las_colors(las1, las2, alpha = 0.5, method = "alpha")
```

Arguments

las1	First LAS object with R, G, B fields
las2	Second LAS object with R, G, B fields (must have same number of points as las1)
alpha	Numeric value between 0 and 1 controlling the blend ratio. 0 = all las1 colors, 1 = all las2 colors, 0.5 = equal blend. Default is 0.5.
method	Character string specifying blend method: "alpha" for alpha blending, "multiply" for multiplicative blending, "screen" for screen blending, "overlay" for overlay blending. Default is "alpha".

Details

Blending methods:

alpha Simple linear interpolation: (1-alpha)*las1* + *alpha*las2

multiply Multiplicative blend (darkens): (las1 * las2) / 255

screen Screen blend (lightens): 255 - ((255-las1) * (255-las2)) / 255

overlay Overlay blend: combines multiply and screen based on base color

Common use cases:

- Combine ambient occlusion (PCV/SSAO) with aerial RGB for realistic shading
- Blend attribute coloring with terrain colors
- Overlay multiple visualization layers

Value

A LAS object (copy of las1) with merged R, G, and B fields

Examples

```
# Load example LAS file
LASfile <- system.file("extdata", "ALS_Clip.laz", package="spanner")
las <- readLAS(LASfile)

# Combine SSAO ambient occlusion with aerial RGB
las_ao <- colorize_las(las, method="ssao", palette=c("black", "white"))
rgb_file <- system.file("extdata", "UAS_Clip_RGB.tif", package="spanner")
las_rgb <- colorize_las(las, method="rgb", raster_path=rgb_file)
las_merged <- merge_las_colors(las_ao, las_rgb, alpha=0.3, method="multiply")

# Blend attribute coloring with RGB at 50/50
las_height <- colorize_las(las, method="attr", attribute_name="Z",
                           palette=c("blue", "red"))
las_merged <- merge_las_colors(las_height, las_rgb, alpha=0.5)
```

---

plot_raster_by_name *Plot a raster by its name*

Description

plot_raster_by_name plots a raster from a list of rasters based on the provided raster name.

Usage

```
plot_raster_by_name(rasters, raster_name)
```

Arguments

rasters	list A list of rasters.
raster_name	character The name of the raster to be plotted.

Value

NULL This function does not return a value. It plots the raster if found.

Examples

```
# Define input parameters
las <- lidR::readLAS(system.file("extdata", "MixedConifer.laz", package="lidR"))
input_raster <- lidR::rasterize_canopy(las, res = 1, lidR::pitfree(c(0,2,5,10,15), c(0, 2)))
suitList <- c(0, 2, 32)
gapList <- seq(1, 8, by = 1)
spurList <- seq(1, 8, by = 1)

# Process the rasters
processed_rasters <- process_rasters_patchmorph(input_raster, suitList, gapList, spurList)

# Plot a raster by its name
plot_raster_by_name(processed_rasters, "suit_2_gap_2_spur_6")
```

`process_rasters_patchmorph` *Process rasters based on suitability, gap, and spur parameters*

Description

`process_rasters_patchmorph` processes an input raster by reclassifying it based on suitability levels, and then applying gap and spur distance transformations to generate a list of processed rasters.

Usage

```
process_rasters_patchmorph(input_raster, suitList, gapList, spurList)
```

Arguments

`input_raster`	RasterLayer The input raster to be processed.
`suitList`	numeric A vector of suitability levels for reclassification.
`gapList`	numeric A vector of gap distances for processing.
`spurList`	numeric A vector of spur distances for processing.

Value

list A list of processed rasters with names indicating the suitability, gap, and spur parameters used.

Examples

```
# Define input parameters
las <- lidR::readLAS(system.file("extdata", "MixedConifer.laz", package="lidR"))
input_raster <- lidR::rasterize_canopy(las, res = 1, lidR::pitfree(c(0,2,5,10,15), c(0, 2)))
suitList <- c(0, 2, 32)
gapList <- seq(1, 8, by = 1)
```

```
spurList <- seq(1, 8, by = 1)

# Process the rasters
processed_rasters <- process_rasters_patchmorph(input_raster, suitList, gapList, spurList)

# Plot the first processed raster
plot(processed_rasters[[1]])
```

---

process_tree_data *Obtain tree information by processing point cloud data*

Description

process_tree_data processes the output of get_raster_eigen_treelocs and segment_graph to add information about the height, crown area, and diameter for each unique TreeID. It also has an optional parameter to return an sf object representing the convex hulls for each tree.

Usage

```
process_tree_data(treeData, segmentedLAS, return_sf = FALSE)
```

Arguments

treeData	An sf object containing the following tree information: TreeID, X, Y, Z, Radius, and Error, output from the get_raster_eigen_treelocs function.
segmentedLAS	A LAS object that is the output from segment_graph.
return_sf	logical: If TRUE, returns an sf object representing the convex hulls for each tree.

Details

For terrestrial and mobile lidar datasets, tree locations and estimates of DBH are provided by rasterizing individual point cloud values of relative neighborhood density (at 0.3 and 1 m radius) and verticality within a slice of the normalized point cloud around breast height (1.34 m). The algorithm then uses defined threshold values to classify the resulting rasters and create unique polygons from the resulting classified raster. These point-density and verticality polygons were selected by their intersection with one another, resulting in a final set of polygons which were used to clip out regions of the point cloud that were most likely to represent tree boles. A RANSAC cylinder fitting algorithm was then used to estimate the fit of a cylinder to individual bole points. Cylinder centers and radius were used as inputs to an individual tree segmentation.

Value

sf object An updated `sf` object with the original columns plus:

height numeric: Height of the highest point for each TreeID.

crown_area numeric: Area of the convex hull for each TreeID.

crown_base_height numeric: Height to the base of the live crown for each TreeID.

crown_volume numeric: Volume of the convex hull for the crown of each TreeID.

diameter numeric: Diameter of the tree, calculated as twice the Radius.

If `return_sf` is TRUE, returns an `sf` object where the geometry is the convex hulls for each tree. If `return_sf` is FALSE, returns an `sf` object with point geometries using treeData.

Examples

```
# Set the number of threads to use in lidR
set_lidr_threads(8)

LASfile = system.file("extdata", "TLS_Clip.laz", package="spanner")
las = readTLSLAS(LASfile, select = "xyzcr", "-filter_with_voxel 0.01")
# Don't forget to make sure the las object has a projection
sf::st_crs(las) <- 26912

# Pre-process the example lidar dataset by classifying the ground  and noise points
# using lidR::csf(), normalizing it, and removing outlier points
# using lidR::ivf()
# las = classify_ground(las, csf(sloop_smooth = FALSE,
#                                class_threshold = 0.5,
#                               cloth_resolution = 0.5, rigidness = 1L,
#                                iterations = 500L, time_step = 0.65))
# las = normalize_height(las, tin())
# las = classify_noise(las, ivf(0.25, 3))
# las = filter_poi(las, Classification != LASNOISE)

# Plot the non-ground points, colored by height
# plot(filter_poi(las, Classification != 2), color = "Z")

# Find individual tree locations and attribute data
# find tree locations and attribute data
myTreeLocs = get_raster_eigen_treelocs(las = las, res = 0.025, pt_spacing = 0.0254,
                                       dens_threshold = 0.25,
                                       neigh_sizes = c(0.25, 0.15, 0.66),
                                       eigen_threshold = 0.75,
                                       grid_slice_min = 1,
                                       grid_slice_max = 2,
                                       minimum_polygon_area = 0.005,
                                       cylinder_fit_type = "ransac",
                                       max_dia = 1,
                                       SDvert = 0.33,
                                       n_pts = 20,
                                       n_best = 25,
```

```
                                      inliers = 0.9,
                                      conf = 0.99,
                                      max_angle = 20)

# Plot results if trees were found
if (!is.null(myTreeLocs) && nrow(myTreeLocs) > 0) {
  plot(lidR::rasterize_canopy(las, res = 0.2, p2r()))
  symbols(sf::st_coordinates(myTreeLocs)[,1], sf::st_coordinates(myTreeLocs)[,2],
          circles = myTreeLocs$Radius^2*3.14, inches = FALSE, add = TRUE, bg = 'black')
} else {
  message("No tree locations were found. Try adjusting the parameters.")
}

# Segment the point cloud
# For areas with interlocking crowns and trees of different sizes,
# enable metabolic scaling to prevent height overestimation
myTreeGraph = segment_graph(las = las, tree.locations = myTreeLocs, k = 50,
                            distance.threshold = 0.5,
                            use.metabolic.scale = FALSE,
                            ptcloud_slice_min = 1,
                            ptcloud_slice_max = 2,
                            subsample.graph = 0.1,
                            return.dense = TRUE)

# Plot it in 3D colored by treeID
plot(myTreeGraph, color = "treeID", pal=spanner_pal())

# Process the data
processed_data <- process_tree_data(myTreeLocs, myTreeGraph, return_sf = TRUE)

# Print the processed data
print(processed_data$data)
# Print the sf object if return_sf is TRUE
if (!is.null(processed_data$sf)) {
  print(processed_data$sf)
}
```

`segment_graph` *Segment a terrestrial point cloud using graph theory.*

Description

`segment_graph` returns a .las object with a new column "treeID".

Usage

```
segment_graph(
  las,
```

```
  tree.locations,
  k = 50,
  distance.threshold = 0.33,
  use.metabolic.scale = FALSE,
  ptcloud_slice_min = 0.5,
  ptcloud_slice_max = 2,
  metabolic.scale.function = NULL,
  subsample.graph = 0.1,
  return.dense = FALSE
)
```

Arguments

las	LAS normalized las object.
tree.locations	sf object sf object containing the following tree information: TreeID, X, Y, Z, Radius, and Error, output from the get_raster_eigen_treelocs function.
k	integer Number of nearest neighbors to be used in processing (k >= 50 suggested)
distance.threshold	numeric Maximum distance (in the same units as the .las) under which two points are connected in the graph object (greater than point spacing). Two points with a greater distance than this threshold are not connected in the graph for processing.
use.metabolic.scale	bool Use of weights in the assignment of points to a given treeID. Useful when interlocking crowns are present and trees are of different sizes.
ptcloud_slice_min	numeric Lower bound of point cloud slice in normalized point cloud used for treeID matching.
ptcloud_slice_max	numeric Upper bound of point cloud slice in normalized point cloud used for treeID matching.
metabolic.scale.function	string Supply your own function for defining segmentation weights based on a function of estimated tree diameter (e.g. metabolic.scale.function = 'x/2'). use.metabolic.scale must be set to TRUE. If not supplied, defaults to metabolic scale function from Tao et al., 2015.
subsample.graph	numeric The subsampled point spacing to use during processing. Note: processing time increases quickly with smaller point spacing with negligible returns in accuracy.
return.dense	bool Decision to return the subsampled point cloud or assign treeIDs back to points in the input dense point cloud.

Details

Preforms Individual tree segmentation following ecological principles for "growing" trees based on these input locations in a graph-theory approach inspired by work of Tao and others (2015).

Point coordinates are linked together based on proximity and turned into a connected graph object, using the estimated tree bole locations as origin points, connecting individual points back to those tree bole origins based on shortest paths within the graph network, and finally assigning those points a unique tree identification based on the bole coordinate for which they are connected. Input point cloud is subsampled to a lower resolution before processing to increase processing efficiency. However, graph objects can still get large quite rapidly. Take this into consideration when choosing the extent of the input las object.

Value

a sparse/dense normalized .las with the column treeID added.

References

Tao, S., Wu, F., Guo, Q., Wang, Y., Li, W., Xue, B., ... & Fang, J. (2015). Segmenting tree crowns from terrestrial and mobile LiDAR data by exploring ecological theories. ISPRS Journal of Photogrammetry and Remote Sensing, 110, 66-76.

Examples

```
# Set the number of threads to use in lidR
set_lidr_threads(8)

LASfile = system.file("extdata", "TLS_Clip.laz", package="spanner")
las = readTLSLAS(LASfile, select = "xyzcr", "-filter_with_voxel 0.01")
# Don't forget to make sure the las object has a projection
sf::st_crs(las) <- 26912

# Pre-process the example lidar dataset by classifying the ground  and noise points
# using lidR::csf(), normalizing it, and removing outlier points
# using lidR::ivf()
# las = classify_ground(las, csf(sloop_smooth = FALSE,
#                                class_threshold = 0.5,
#                               cloth_resolution = 0.5, rigidness = 1L,
#                                iterations = 500L, time_step = 0.65))
# las = normalize_height(las, tin())
# las = classify_noise(las, ivf(0.25, 3))
# las = filter_poi(las, Classification != LASNOISE)

# Plot the non-ground points, colored by height
# plot(filter_poi(las, Classification != 2), color = "Z")

# Perform a deep inspection of the las object. If you see any
# red text, you may have issues!
las_check(las)

# Find individual tree locations and attribute data
# find tree locations and attribute data
myTreeLocs = get_raster_eigen_treelocs(las = las, res = 0.025, pt_spacing = 0.0254,
                                       dens_threshold = 0.25,
                                       neigh_sizes = c(0.25, 0.15, 0.66),
```

```
                                        eigen_threshold = 0.75,
                                        grid_slice_min = 1,
                                        grid_slice_max = 2,
                                        minimum_polygon_area = 0.005,
                                        cylinder_fit_type = "ransac",
                                        max_dia = 1,
                                        SDvert = 0.33,
                                        n_pts = 20,
                                        n_best = 25,
                                        inliers = 0.9,
                                        conf = 0.99,
                                        max_angle = 20)

# Plot results if trees were found
if (!is.null(myTreeLocs) && nrow(myTreeLocs) > 0) {
  plot(lidR::rasterize_canopy(las, res = 0.2, p2r()))
  symbols(sf::st_coordinates(myTreeLocs)[,1], sf::st_coordinates(myTreeLocs)[,2],
          circles = myTreeLocs$Radius^2*3.14, inches = FALSE, add = TRUE, bg = 'black')
} else {
  message("No tree locations were found. Try adjusting the parameters.")
}

# Segment the point cloud
# For areas with interlocking crowns and trees of different sizes,
# enable metabolic scaling to prevent height overestimation
myTreeGraph = segment_graph(las = las, tree.locations = myTreeLocs, k = 50,
                            distance.threshold = 0.5,
                            use.metabolic.scale = FALSE,
                            ptcloud_slice_min = 1,
                            ptcloud_slice_max = 2,
                            subsample.graph = 0.1,
                            return.dense = TRUE)

# Plot it in 3D colored by treeID
plot(myTreeGraph, color = "treeID", pal=spanner_pal())

# Optional: Use a custom metabolic scaling function
# myTreeGraph = segment_graph(las = las, tree.locations = myTreeLocs, k = 50,
#                             distance.threshold = 0.5,
#                             use.metabolic.scale = TRUE,
#                             metabolic.scale.function = '1/((2*x)^(1/8))',
#                             ptcloud_slice_min = 1,
#                             ptcloud_slice_max = 2,
#                             subsample.graph = 0.1,
#                             return.dense = TRUE)
```

---

spanner_pal *Spanner color palette*

Description

Returns a named vector of colors for use in spanner visualizations. The palette includes 10 distinct colors suitable for categorical data visualization.

Usage

```
spanner_pal()
```

Value

A named character vector of hex color codes

Examples

```
# Get the palette
colors <- spanner_pal()

# Use in a plot
barplot(1:10, col = spanner_pal(), names.arg = names(spanner_pal()), las = 2)
```

---

sum_rasters_by_suitability

Sum rasters by suitability level

Description

`sum_rasters_by_suitability` sums rasters from a list based on their suitability levels.

Usage

```
sum_rasters_by_suitability(rasters, suitList)
```

Arguments

`rasters`	list A list of rasters.
`suitList`	numeric A vector of suitability levels.

Value

list A list of summed rasters for each suitability level.

Examples

```
# Define input parameters
las <- lidR::readLAS(system.file("extdata", "MixedConifer.laz", package="lidR"))
input_raster <- lidR::rasterize_canopy(las, res = 1, lidR::pitfree(c(0,2,5,10,15), c(0, 2)))
suitList <- c(0, 2, 32)
gapList <- seq(1, 8, by = 1)
spurList <- seq(1, 8, by = 1)

# Process the rasters
processed_rasters <- process_rasters_patchmorph(input_raster, suitList, gapList, spurList)

# Sum rasters by suitability level
summed_rasters <- sum_rasters_by_suitability(processed_rasters, suitList)

# Call the plot_raster_by_name function to plot the raster named "suit_2_sum"
plot_raster_by_name(summed_rasters, "suit_2_sum")
```

Index