Package: rTLS (via r-universe)

May 20, 2026

Type Package

Title Tools to Process Point Clouds Derived from Terrestrial Laser Scanning

Version 0.2.6.1

Maintainer J. Antonio Guzmán Q. <antguz06@gmail.com>

Description A set of tools to process and calculate metrics on point clouds derived from terrestrial LiDAR (Light Detection and Ranging; TLS). Its creation is based on key aspects of the TLS application in forestry and ecology. Currently, the main routines are based on filtering, neighboring features of points, voxelization, canopy structure, and the creation of artificial stands. It is written using data.table and C++ language and in most of the functions it is possible to use parallel processing to speed-up the routines.

License GPL (>= 3)

URL https://github.com/Antguz/rTLS

BugReports https://github.com/Antguz/rTLS/issues

Depends data.table (>= 1.17.8), R (>= 4.5.0)

Imports alphashape3d, boot, doSNOW (>= 1.0.16), foreach (>= 1.4.4), parallel, RcppHNSW (>= 0.3.0), rgl, sf

Suggests knitr, rmarkdown, covr, testthat (>= 3.0.1)

LinkingTo Rcpp, RcppArmadillo, RcppProgress

VignetteBuilder knitr

Encoding UTF-8

LazyData true

LazyDataCompression bzip2

RoxygenNote 7.3.2

SystemRequirements GNU make

ByteCompile true

Config/testthat/edition 3

Config/pak/sysreqs libabsl-dev cmake libfreetype6-dev libgdal-dev gdal-bin libgeos-dev libglu1-mesa-dev make texlive libpng-dev libuv1-dev libgl1-mesa-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev zlib1g-dev

Repository https://antguz.r-universe.dev

Date/Publication 2026-01-19 04:16:25 UTC

RemoteUrl https://github.com/antguz/rtls

RemoteRef HEAD

RemoteSha 1c8133f79615b8de96912dc08927d93810ce77e4

Contents

rTLS-package *rTLS: Tools to Process Point Clouds Derived From Terrestrial Laser Scanning*

Description

rTLS is a package that compiles a set of tools to process and calculate metrics on point clouds derived from terrestrial LiDAR (Light Detection and Ranging). Its creation is based on key aspects of the TLS application in forestry. Currently, the main routines are based on filtering, neighboring features of points, voxelization, optimal sphere or voxel size, and the creation of artificial stands. rTLS is written using data.table and C++ language and in most of the functions it is posible to use parallel processing to speed-up the routines.

Author(s)

Maintainer: J. Antonio Guzmán Q. <antguz06@gmail.com> (ORCID) [thesis advisor, copyright holder]

Authors:

- Ronny Hernandez <ronny.hernandezm@gmail.com> (ORCID)

Other contributors:

- Arturo Sanchez-Azofeifa <arturo.sanchez@ualberta.ca> (ORCID) [degree supervisor, scientific advisor]

See Also

Useful links:

- https://github.com/Antguz/rTLS
- Report bugs at https://github.com/Antguz/rTLS/issues

artificial_stand *Artificial Forest Stand*

Description

Create an artificial forest stand of a given area using tree point clouds.

Usage

```
artificial_stand(
  files,
  n.trees,
  dimension,
  coordinates = NULL,
  sample = TRUE,
  replace = TRUE,
  overlap = NULL,
  rotation = TRUE,
  degrees = NULL,
  n_attempts = 100,
  progress = TRUE,
  plot = TRUE,
  ...
)
```

Arguments

files	A character vector describing the file name or path of the tree point cloud to use. Those files most contain three columns representing the *XYZ* coordinates of a given point cloud.
n.trees	A positive numeric vector describing the number of point clouds to use.
dimension	A positive numeric vector of length two describing the width and length of the future forest stand.
coordinates	A data.table of two columns and with nrows equal to n.trees describing the basal *XYZ* coordinates of the point clouds in the future stand. If NULL, it uses random basal coordinates based on stand dimension. NULL as default.
sample	Logical. If TRUE, it performs a sample of the files to determine the order to build the artificial stand. If FALSE, it use the file order described in files. TRUE as default.
replace	Logical. If TRUE, it performs a sample selection with a replacement if sample = TRUE to determine the order to build the artificial stand. Useful if the n.trees is lower than length(files). TRUE as default.
overlap	A positive numeric vector between 0 and 100 describing the overlap percentage of a given the tree crowns in the future forest stand. If NULL, the degree of overlap is not controlled.
rotation	Logical. If TRUE, it performs a rotation in yaw axis of the point cloud. TRUE as default.
degrees	A positive numeric vector describing the degrees of rotation of the point clouds in the future stand. The length(degree) should be the same as n.trees. If NULL, it creates random degrees of rotation for each n.trees.
n_attempts	A positive numeric vector of length one describing the number of attempts to provide random coordinates until a tree met the overlap criteria. This needs to be used if coordinate = NULL and overlap != NULL. n_attempts = 100 as default.

progress	Logical, if TRUE displays a graphical progress bar. TRUE as default.
plot	Logical. If TRUE, it provides visual tracking of the distribution of each tree in the artificial stand. This can not be exported as a return object.
...	Parameters passed to fread for the reading of files.

Details

When coordinates = NULL, artifical_stand adds, in sequence, random coordinates to each files in the future stand based on the crown area overlap. That is, first a tree from files is randomly located within the stand dimention, then a second tree from files will be located in the future stand based on the crown area overlap from the previous tree, and so on. If during the random location a given tree does not meet the requirements of overlap, new random coordinates will be provided until the requirements are met.

Since artificial_stand will try to add tree to the stand until the requirements are met, this could lead to an infinite loop if the stand dimention is small or if the trees on files are large or many n.trees. Therefore, the use of n_attempts is recommended to avoid this scenario.

Value

A list which contain a data.table (Trees) with the information of the point clouds used and their current coordinates in the stand, and another data.table with that compile all the point clouds used.

Author(s)

J. Antonio Guzmán Q.

See Also

voxels_counting

Examples

```
#' #Import an example point cloud
path <- system.file("extdata", "pc_tree.txt", package = "rTLS")

#Creates a stand of 4 trees with 10% of overlap
files <- rep(path, 4)
artificial_stand(files, n.trees = 4, dimension = c(15, 15), overlap = 10)

#Creates a stand of 4 trees with their locations
location <- data.table(X = c(5, 10, 10, 5), Y = c(5, 5, 10, 10))
artificial_stand(files, n.trees = 4, dimension = c(15, 15), coordinates = location)
```

canopy_structure *Canopy Structure*

Description

Estimates the canopy structure from a discrete returns scan from different TLS.

Usage

```
canopy_structure(
  TLS.type,
  scan,
  zenith.range,
  zenith.rings,
  azimuth.range,
  vertical.resolution,
  TLS.pulse.counts,
  TLS.resolution = NULL,
  TLS.coordinates = c(0, 0, 0),
  TLS.frame = NULL,
  TLS.angles = NULL,
  threads = 1
)
```

Arguments

TLS.type	A character describing is the TLS used. It most be one of "single" return, "multiple" return, or "fixed.angle" scanner.
scan	If TLS.type is equal to "single" or "fixed.angle", a data.table with three columns describing *XYZ* coordinates of the discrete return. If TLS.type is equal to "multiple", a data.table with four columns describing *XYZ* coordinates and the target count pulses. Currently, "fixed.angle" present errors, use with discretion.
zenith.range	If TLS.type is equal to "single" or "multiple", a numeric vector of length two describing the min and max range of the zenith angle to use. Theoretically, the max range should be lower than 90 degrees.
zenith.rings	If TLS.type is equal to "single" or "multiple", a numeric vector of length one describing the number of zenith rings to use between zenith.range. This is used to estimate the frequency of laser shots from the scanner and returns in scan. If TLS.type = "fixed.angle", zenith.rings = 1 be default.
azimuth.range	A numeric vector of length two describing the range of the azimuth angle to use. Theoretically, it should be between 0 and 360 degrees.
vertical.resolution	A numeric vector of length one describing the vertical resolution to extract the vertical profiles. Low values lead to more variable profiles. The scale used needs to be in congruence with the scale of scan.

TLS.pulse.counts
: If TLS.type is equal to "single" or "multiple", a numeric vector of length two describing the horizontal and vertical pulse counts of the scanner. If TLS.type is equal to "fixed.angle", a numeric vector of length one describing the horizontal pulse counts resolution. Preferred parameter over TLS.resolution to estimate the number of pulses.

TLS.resolution
: If TLS.pulse.counts = NULL, the code use the angles resolution to estimate the pulse counts in a given TLS.frame. If TLS.type is equal to "single" or "multiple", a numeric vector of length two describing the horizontal and vertical angle resolution of the scanner. If TLS.type is equal to "fixed.angle", a numeric vector of length one describing the horizontal angle resolution.

TLS.coordinates
: A numeric vector of length three describing the scanner coordinates within scan. It assumes that the coordinates are c(X = 0, Y = 0, Z = 0) for default.

TLS.frame
: If TLS.type is equal to "single" or "multiple", a numeric vector of length four describing the min and max of the zenith and azimuth angle of the scanner frame. If TLS.type = "fixed.angle", a numeric vector of length three describing the fixed zenith angle and the min and max of the azimuth angle of the scanner frame. If NULL, it assumes that a complete hemisphere (c(zenith.min = 0, zenith.max = 90, azimuth.min = 0, azimuth.max = 360)), or a cone projection (c(zenith = 57.5, azimuth.min = 0, azimuth.max = 360)) depending on TLS.type.

TLS.angles
: A numeric vector of length three describing the roll (*X*), pitch (*Y*), and yaw (*Z*) angles of the scanner during the scan. If NULL, it assumes that there is no need to to correction of angles. This needs to be used if TLS.type is equal to "single" or "multiple", since it assumes that "fixed.angle" scanner is previously balanced. NULL as default.

threads
: An integer specifying the number of threads to use. Experiment to see what works best for your data on your hardware.

Details

Since scan describes discrete returns measured by the TLS, canopy_structre first simulates the number of pulses emitted based on Danson et al. (2007). The simulated pulses are created based on the TLS properties (TLS.pulse.counts, TLS.resolution, TLS.frame) assuming that the scanner is perfectly balance. Then these pulses are rotated (rotate3D) based on the TLS.angles roll, pitch, and yaw, and move to TLS.coordintates to simulate the positioning of the scanner during the scan. Rotated simulated-pulses of interest and scan returns are then extracted based on the zenith.range and azimuth.range for a given number of zenith.rings, azimuth.rings and vertical profiles. The probability of gap (Pgap) is then estimated using the frequency of pulses and returns. For TLS.type = "multiple", the frequency of returns is estimated using the sum of 1/target count following Lovell et al. (2011).

Using the Pgap estimated per each zenith ring and vertical profile, canopy_structure then estimates the accumulative L(z) profiles based on the closest zenith ring to 57.5 (hinge region) and, if TLS.type = "fixed.angle", the f(z) or commonly named PAVD based on the ratio of the derivative of L(z) and height (z) following Jupp et al. 2009 (Equation 18). If TLS.type is equal to "single" or "multiple", canopy_structure also estimates the normalized average weighted L/LAI, and then

PAVD based on the L (hinge angle) at the highest height (LAI) and the ratio between the derivative of L/LAI (average weighted) and the derivative of z (Jupp et al. 2009; Equation 21).

Jupp et al. 2009 excludes the zero zenith or fist ring to conduct the average weighted L/LAI estimations, `canopy_structre` does not excludes this sections since it depends on the regions of interest of the user. Therefore, user should consider this difference since it may introduce more variability to profile estimations.

Value

For any `TLS.type`, it returns a `data.table` with the height profiles defined by `vertical.resolution`, the gap probability based on the `zenith.range` and `zenith.rings`, and the accumulative L(z) profiles based on the closest zenith ring to 57.5 degrees (hinge angle). If `TLS.type` is equal to `"fixed.angle"`, it returns f(z) or commonly named PAVD based on on the ratio of the derivative of L(z) and the derivative of height (z). If `TLS.type` is equal to `"single"` or `"multiple"`, it returns the normalized average weighting L/LAI, and PAVD: based on the L (hinge angle) at the highest height and the ratio between the derivative of L/LAI average weighted and the derivative of z.

Author(s)

J. Antonio Guzmán Q.

References

Danson F.M., Hetherington D., Morsdorf F., Koetz B., Allgower B. 2007. Forest canopy gap fraction from terrestrial laser scanning. IEEE Geosci. Remote Sensing Letters 4:157-160. doi: 10.1109/LGRS.2006.887064

Lovell J.L., Jupp D.L.B., van Gorsel E., Jimenez-Berni J., Hopkinson C., Chasmer L. 2011. Foliage profiles from ground based waveform and discrete point LiDAR. In: SilviLaser 2011, Hobart, Australia, 16–20 October 2011.

Jupp D.L.B., Culvenor D.S., Lovell J.L., Newnham G.J., Strahler A.H., Woodcock C.E. 2009. Estimating forest LAI profiles and structural parameters using a ground-based laser called “Echidna®”. Tree Physiology 29(2): 171-181. doi: 10.1093/treephys/tpn022

Examples

```
data(TLS_scan)
#Using a multiple return file
#Select the four columns required
TLS_scan <- TLS_scan[, 1:4]

#This will take a while#
canopy_structure(TLS.type = "multiple",
                 scan = TLS_scan,
                 zenith.range = c(50, 70),
                 zenith.rings = 4,
                 azimuth.range = c(0, 360),
                 vertical.resolution = 0.25,
                 TLS.pulse.counts = c(2082, 580),
                 TLS.frame = c(30, 130.024, 0, 359.90),
```

```
                TLS.angles =  c(1.026, 0.760, -110.019))

#Using a single return file

data(TLS_scan)
#Subset to first return observations
TLS_scan <- TLS_scan[Target_index == 1, 1:3]

#This will take a while#
canopy_structure(TLS.type = "single",
                scan = TLS_scan,
                zenith.range = c(50, 70),
                zenith.rings = 4,
                azimuth.range = c(0, 360),
                vertical.resolution = 0.25,
                TLS.pulse.counts = c(2082, 580),
                TLS.frame = c(30, 130.024, 0, 359.90),
                TLS.angles =  c(1.026, 0.760, -110.019))
```

`cartesian_to_polar` *Cartesian to Polar Coordinates*

Description

Convert from East-North-Up cartesian coordinates to polar coordinates.

Usage

```
cartesian_to_polar(cartesian, anchor = c(0, 0, 0), digits = NULL)
```

Arguments

`cartesian`	A `data.table` with three columns describing the *XYZ* coordinates of a point cloud.
`anchor`	A `numeric` vector of length three which describe the *XYZ* anchor coordinate for reference to get the polar coordinates. It assumes that the reference coordinates are `c(X = 0, Y = 0, Z = 0)` as default.
`digits`	A `numeric` vector of length 1 describing the decimal numbers to `round` the zenith and azimuth angles. If `NULL`, `round` does not apply. `NULL` as default.

Details

It assumes that the positive *Z* axis is the reference vector for the zenith angle. Likewise, it assumes that the *Y* axis is the north-south direction (positive to negative) for the azimuth angle. If a point from `cartesian` presents the same *XY* coordinates than `anchor`, `angles` returns `NA`.

Value

A `data.table` with the zenith and azimuth angles (degrees), and the distance to the anchor coordinate.

Author(s)

J. Antonio Guzmán Q.

See Also

`polar_to_cartesian`

Examples

```
data(pc_tree)
cartesian_to_polar(pc_tree)
anchor <- c(1, 1, 1)
cartesian_to_polar(pc_tree, anchor)
```

`circleRANSAC` *Adaptive RANSAC Circle Fitting*

Description

Adaptive random sample consensus for cicle fitting.

Usage

```
circleRANSAC(
  cloud,
  fpoints,
  pconf,
  poutlier,
  max_iterations,
  threads = 1L,
  plot = TRUE
)
```

Arguments

`cloud`	A `data.table` with *XY* coordinates in the first two columns.
`fpoints`	A `numeric` vector between 0 and 1 representing the fraction of point samples that will be used during each iteration.
`pconf`	A `numeric` vector between 0 and 1 describing the confidence threshold to consider a point in a given fitted circle outlier or inlier.

poutlier	A numeric vector of length two describing the proportion of outliers to consider inside or outsite of the pconf threshold.
max_iterations	An integer specifying the number of iterations. If NULL, the number of iterations are automaticaly estimated using pconf, 1 - poutlier, and 1 - fpoints; see details.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
plot	Logical. If TRUE, it provides visual representation of the fitted circle.

Value

A data.table with the *XY* coordinate information of the circle center, the radius, the error based on the least squares fit, and the proportion of inliers.

Author(s)

J. Antonio Guzmán Q.

See Also

tree_metrics, trunk_volume

Examples

```
#Point cloud
data("pc_tree")

#Subset region at at breast height
sub <- pc_tree[between(Z, 1.25, 1.35),]

#Fit circle
circleRANSAC(sub, fpoints = 0.2, pconf = 0.95, poutlier = c(0.5, 0.5), max_iterations = 100)
```

---

euclidean_distance *Euclidean Distance Between 3D points*

Description

Estimate the distance between a point and a group of point.

Usage

```
euclidean_distance(point, cloud, threads = 1L)
```

Arguments

point	A numeric vector of length three describing the *XYZ* coordinates.
cloud	A data.table with *XYZ* coordinates in the first three columns representing a point cloud.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.

Value

A numeric vector describing of point to each row of cloud.

Author(s)

J. Antonio Guzmán Q.

Examples

```
data("pc_tree")

euclidean_distance(point = c(0, 0, 0), pc_tree)
```

---

filter *Filtering of Point Clouds*

Description

Filtering of point clouds using different methods

Usage

```
filter(
  cloud,
  method,
  radius,
  min_neighbours,
  k,
  nSigma,
  edge_length,
  distance = "euclidean",
  threads = 1L,
  verbose = FALSE,
  progress = FALSE,
  ...
)
```

Arguments

cloud	A data.table contain three columns representing the *XYZ* coordinates.
method	A filtering method to use. It most be "SOR", "min_neighbors", or "min_neighbors".
radius	A numeric vector representing the radius of the sphere to consider. This needs to be used if method = "voxel_center".
min_neighbours	An integer representing the minimum number of neighbors to keep a given point. This needs to be used if method = "min_n".
k	An integer vector representing the number of neighbors to consider. This needs be used if method = "SOR".
nSigma	A numeric vector representing the standard deviation multiplier. This needs to be used if method = "SOR".
edge_length	A positive numeric vector with the voxel-edge length for the x, y, and z coordinates. This needs to be used if method = "voxel_center".
distance	Type of distance to calculate. "euclidean" as default. Look hnsw_knn for more options.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
verbose	If TRUE, log messages to the console.
progress	If TRUE, log a progress bar when verbose = TRUE. Tracking progress could cause a small overhead.
...	Arguments passed to hnsw_build and hnsw_search.

Value

A data.table with the filtered points

Author(s)

J. Antonio Guzmán Q.

Examples

```
#Load data
data("pc_tree")

#Move pc_tree for comparison
pc_compare <- pc_tree
pc_compare$X <- pc_compare$X - 7

#SOR filter
r1 <- filter(pc_tree, method = "SOR", k = 30, nSigma = 1)
rgl::plot3d(r1, col = "red") #Filter
rgl::points3d(pc_compare, col = "black") #Original

#min_neighbours filter
```

```
r2 <- filter(pc_tree, "min_neighbors", radius = 0.02, min_neighbours = 20)
rgl::plot3d(r2, col = "red") #Filter
rgl::points3d(pc_compare, col = "black") #Original

#voxel_center filter
r3 <- filter(pc_tree, method = "voxel_center", edge_length = 0.1)
rgl::plot3d(r3, col = "red") #Filter
rgl::points3d(pc_compare, col = "black") #Original
```

`geometry_features` *Geometry features of Neighboring Points.*

Description

Estimate geometry features of neighboring points in a cloud.

Usage

```
geometry_features(
  cloud,
  method,
  radius,
  k,
  max_neighbour,
  distance = "euclidean",
  target = FALSE,
  threads = 1L,
  verbose = FALSE,
  progress = TRUE,
  ...
)
```

Arguments

`cloud`	A `data.table` with *XYZ* coordinates in the first three columns.
`method`	A character string specifying the method to estimate the neighbors. It most be one of `"radius_search"` or `"knn"`.
`radius`	A `numeric` vector representing the radius for search to consider. This needs be used if `method = "radius_search"`.
`k`	An `integer` vector representing the number of neighbors to consider. This needs be used if `method = "knn"`.
`max_neighbour`	An `integer` specifying the maximum number of points to look around each query point for a given radius. This needs be used if `method = "radius_search"`.
`distance`	Type of distance to calculate. `"euclidean"` as default. Look `hnsw_knn` for more options.

target	Logic. If TRUE, it consider the each target point for the calculations of geometry features.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
verbose	If TRUE, log messages to the console.
progress	If TRUE, log a progress bar when verbose = TRUE. Tracking progress could cause a small overhead.
...	Arguments passed to hnsw_build and hnsw_search.

Details

The function returns the geometry features of the neighboring points of a given point in cloud. Geometry features are represented by the relative values of the eigenvalues derived from a covariance matrix of the neighboring points. Geometry features are not estimated on target points with less than 3 neighboring points.

Value

A array describing the point of the cloud in rows, the relative eigenvalues in columns, and the radius or k per slide. If method = "radius_search", it add in the first column the number of neighboring points.

Author(s)

J. Antonio Guzmán Q.

Examples

```
#Create cloud
example <- data.table(X = runif(200, min=0, max=10),
                      Y = runif(200, min=0, max=10),
                      Z = runif(200, min=0, max=10))


#Using knn method with two different k
k_test <- c(5, 10)
geometry_features(example, method = "knn", k = k_test)

#Using radius search method with two different radius
radius_test <- c(3, 4)
geometry_features(example, method = "radius_search", radius = radius_test, max_neighbour = 200)
```

knn *K Nearest Neighbors*

Description

Adapted K nearest neighbors based on RcppHNSW

Usage

```
knn(
  query,
  ref,
  k,
  distance = "euclidean",
  same = FALSE,
  threads = 1L,
  verbose = FALSE,
  progress = FALSE,
  ...
)
```

Arguments

query	A data.table containing the set of query points where each row represent a point and each column a given coordinate.
ref	A numeric containing the set of reference points where each row represent a point and each column a given coordinate.
k	An integer describing the number of nearest neighbors to search for.
distance	Type of distance to calculate. "euclidean" as default. Look hnsw_knn for more options.
same	Logic. If TRUE, it delete neighbors with distance of 0, useful when the k search is based on the same query.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
verbose	If TRUE, log messages to the console.
progress	If TRUE, log a progress bar when verbose = TRUE. Tracking progress could cause a small overhead.
...	Arguments passed to hnsw_build and hnsw_search.

Details

This function is based on hnswlib C++ library (Malkov & Yashunin 2016) and its bindings for R (RcppHNSW; Melville 2020) for a fast estimation of neighbors points. It is adapted to simplify the workflow within rTLS. If you use this function, please consider cite the C++ library and RcppHNSW package.

Value

A data.table with three columns describing the indices of the query, ref, and k neighbors and the distances.

Author(s)

J. Antonio Guzmán Q.

References

Malkov, Y. A., & Yashunin, D. A. (2016). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv preprint arXiv:1603.09320.

See Also

radius_search

Examples

```
#Point cloud
data("pc_tree")

#knn search using k = 3
knn(pc_tree, pc_tree, k = 3, same = TRUE)
```

---

line_AABB *Line-AABB*

Description

Intersection of a line by an Axis-Aligned Bounding Box.

Usage

```
line_AABB(orig, end, AABB_min, AABB_max)
```

Arguments

orig	A data.table with the describing *XYZ* coordinates of the the start path of a line.
end	A data.table with the describing *XYZ* coordinates of the the end path of a line.
AABB_min	A numeric vector with the minimum *XYZ* coordinates of the AABB
AABB_max	A numeric vector with the maximum *XYZ* coordinates of the AABB.

Details

The interaction of a line with a AABB may result in five scenarios: i) the line is not intercepted by a AAABB (0), ii) the origin and end of the line falls within the AABB (1), iii) the origin point of the line falls within the AABB both not the end point (2), iv) the end point of the line falls within the AABB both not the origin point (3), and v) the line is intercepted by the AABB (4).

Value

An numeric vector of length two, describing if the line was intercepted or not, and the length of the intercepted line within in the AABB. See details.

Author(s)

J. Antonio Guzmán Q.

See Also

lines_interception, voxels,

Examples

```
#Create origins and end paths
orig <- data.table(X = c(0, 0, 0, 0, 0),
                   Y = c(-0.45, -0.25, 0, 0.25, 0.45),
                   Z = c(-1, -0.25, 0, -1, -1))

end <- data.table(X = c(0, 0, 0, 0, 0),
                  Y = c(-0.45, -0.25, 0, 0.25, 0.45),
                  Z = c(-0.75, 0.25, 1, 0, 1))

#Create the AABB
AABB <- matrix(c(0, 0, 0), ncol = 3)
edge_length <- c(1, 1, 1)

AABB_min <- c(AABB[1, 1] - edge_length[1]/2,
              AABB[1, 2] - edge_length[2]/2,
              AABB[1, 3] - edge_length[3]/2)

AABB_max <- c(AABB[1, 1] + edge_length[1]/2,
              AABB[1, 2] + edge_length[2]/2,
              AABB[1, 3] + edge_length[3]/2)

#Plot
cube <- rgl::cube3d()
cube <- rgl::scale3d(cube, edge_length[1]/2,
                           edge_length[2]/2,
                           edge_length[3]/2)
box <- rgl::translate3d(cube, AABB[1, 1], AABB[1, 2], AABB[1, 3])
rgl::shade3d(box, col= "green", alpha = 0.6)
rgl::points3d(orig, size = 4, col = "black")
rgl::points3d(end, size = 4, col = "red")
```

```
#Line no intercepted
rgl::lines3d(c(orig[1, 1], end[1, 1]),
             c(orig[1, 2], end[1, 2]),
             c(orig[1, 3], end[1, 3]), col = "grey")

line_AABB(orig[1,], end[1,], AABB_min, AABB_max)

#Both ends falls inside
rgl::lines3d(c(orig[2, 1], end[2, 1]),
             c(orig[2, 2], end[2, 2]),
             c(orig[2, 3], end[2, 3]), col = "red")

line_AABB(orig[2,], end[2,], AABB_min, AABB_max)

#Oring falls inside, but not the end.
rgl::lines3d(c(orig[3, 1], end[3, 1]),
             c(orig[3, 2], end[3, 2]),
             c(orig[3, 3], end[3, 3]), col = "blue")

line_AABB(orig[3,], end[3,], AABB_min, AABB_max)

#End falls inside, but not the orig
rgl::lines3d(c(orig[4, 1], end[4, 1]),
             c(orig[4, 2], end[4, 2]),
             c(orig[4, 3], end[4, 3]), col = "green")

line_AABB(orig[4,], end[4,], AABB_min, AABB_max)

#Some segments of the line are intercepted
rgl::lines3d(c(orig[5, 1], end[5, 1]),
             c(orig[5, 2], end[5, 2]),
             c(orig[5, 3], end[5, 3]), col = "black")

line_AABB(orig[5,], end[5,], AABB_min, AABB_max)
```

`lines_interception` *Intersection of Lines by AABBs*

Description

Intersection of lines by several Axis-Aligned Bounding Boxs.

Usage

```
lines_interception(orig, end, AABBs, edge_length, threads = 1, progress = TRUE)
```

Arguments

orig	A data.table with the describing *XYZ* coordinates of the the start path of the rays.
end	A data.table with the describing *XYZ* coordinates of the the end path of the rays.
AABBs	A data.table with *XYZ* coordinates of the center of AABBs.
edge_length	A positive numeric vector with the AABB length edge for the X, Y, and Z coordinates.
threads	An integer >= 0 describing the number of threads to use. This need to be used if parallel = TRUE.
progress	Logical, if TRUE displays a graphical progress bar. TRUE as default.

Value

It returns a data.table with nine columns: 1-5 columns with the counts for the code of intersection (see line_AABB), and 6-9 columns with sum the path length of intersection. The number of rows match with nrow(AABBs).

Author(s)

J. Antonio Guzmán Q.

See Also

line_AABB, voxels

Examples

```
#Create points with paths
n <- 20
orig <- data.table(X = runif(n, min = -5, max = 5),
                   Y = runif(n, min = -5, max = 5),
                   Z = runif(n, min = -5, max = 5))

end <- data.table(X = runif(n, min = -5, max = 5),
                  Y = runif(n, min = -5, max = 5),
                  Z = runif(n, min = -5, max = 5))

#Create a potential AABB
AABBs <- data.table(X = 0, Y = 0, Z = 0)
edge_length <- c(2, 2, 2)

#Plot

cube <- rgl::cube3d()
cube <- rgl::scale3d(cube,
                     edge_length[1]/2,
                     edge_length[2]/2,
                     edge_length[3]/2)
```

```
box <- rgl::translate3d(cube, AABBs[[1]], AABBs[[2]], AABBs[[3]])
rgl::shade3d(box, col= "green", alpha = 0.6)
rgl::points3d(orig, size = 5, col = "black")
rgl::points3d(end, size = 5, col = "red")

for(i in 1:nrow(orig)) {
rgl::lines3d(c(orig[[1]][i], end[[1]][i]),
             c(orig[[2]][i], end[[2]][i]),
             c(orig[[3]][i], end[[3]][i]), col = "grey")
}


#Estimation
lines_interception(orig, end, AABBs, edge_length, progress = FALSE)
```

`min_distance` *Minimum Distance Between Points*

Description

Estimate the minimum distance between points in a point cloud.

Usage

```
min_distance(
  cloud,
  distance = "euclidean",
  threads = 1L,
  verbose = FALSE,
  progress = FALSE,
  ...
)
```

Arguments

`cloud`	A `data.table` with *XYZ* coordinates in the first three columns representing a point cloud.
`distance`	Type of distance to calculate. `"euclidean"` as default. Look `hnsw_knn` for more options.
`threads`	An `integer` specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
`verbose`	If TRUE, log messages to the console.
`progress`	If TRUE, log a progress bar when `verbose = TRUE`. Tracking progress could cause a small overhead.
`...`	Arguments passed to `hnsw_build` and `hnsw_search`.

Value

A numeric vector describing the minimum distance between points.

Author(s)

J. Antonio Guzmán Q.

Examples

```
data("pc_tree")

#Estimate the minimum distance of a sample o 100 points
min_distance(pc_tree)
```

---

pc_tree *A Tree Point Cloud*

Description

A data.table from a point cloud of a tree with a spatial point resolution of 0.05 mm.

Usage

```
data(pc_tree)
```

Format

A data.table with three columns, which are:

X the "X" coordinate

Y the "Y coordinate

Z the "Z" coordinate

A data.table where the rows represent the points and the three columns represent the *XYZ* coordinates.

References

Guzman, Sharp, Alencastro, Sanchez-Azofeifa. 2018. To be published.

Examples

```
data(pc_tree)
head(pc_tree)
```

plot_voxels *Plot Method for Voxels*

Description

The plot method for objects of class "voxels" created using the voxels function.

Usage

```
plot_voxels(
  voxels,
  add.points = TRUE,
  add.voxels = TRUE,
  border = TRUE,
  points.size = 1,
  points.col = "black",
  fill.col = "forestgreen",
  line.lwd = 0.5,
  line.col = "black",
  alpha = 0.1,
  ...
)
```

Arguments

voxels	Object of class "voxels" from voxels.
add.points	Logical, if TRUE it adds the original points used to perform the voxelization. TRUE as default.
add.voxels	Logical, if TRUE it adds the voxels created. TRUE as default.
border	Logical, if TRUE it adds a line on the borders of each voxel. TRUE as default.
points.size	The points size, a positive number to use if plot add.points = TRUE.
points.col	A character defining the color of the points to use.
fill.col	A character vector defining the color to fill the voxels, it could be a range of colors or a solid color.
line.lwd	The line width, a positive number, defaulting to 0.5.
line.col	A character defining the color of the border lines to use.
alpha	A positive numeric vector describing the transparency of the voxels to fill. This value most be between 0.0 (fully transparent) .. 1.0 (opaque).
...	General arguments passed to plot3d.

Value

A 3D plot of a point cloud and voxels.

Author(s)

J. Antonio Guzmán Q.

See Also

voxels, voxels_counting, summary_voxels

Examples

```
data("pc_tree")

###Create cubes of a size of 7x7x3.5.
vox <- voxels(pc_tree, edge_length = c(7, 7, 3.5))
plot_voxels(vox)
```

---

polar_to_cartesian *Polar to Cartesian Coordinates*

Description

Convert from polar to cartesian coordinates.

Usage

```
polar_to_cartesian(polar, threads = 1, digits = NULL)
```

Arguments

polar	A data.table with three columns describing the zenith, azimuth, and distance of a point to the center.
threads	An integer vector describing the number of threads for parallel processing. Default 1.
digits	A numeric vector of length 1 describing the decimal numbers to round the cartesian coordinates. If NULL, round does not apply. NULL as default.

Value

A data.table with three columns describing the *XYZ* of the cartesian coordinates.

Author(s)

J. Antonio Guzmán Q.

See Also

cartesian_to_polar

Examples

```
#Creates a hemisphere of points each 2 degrees

zenith <- seq(0, 90, 2)
azimuth <- seq(0, 360, 2)
hemi <- CJ(zenith, azimuth)
hemi$distance <- 1
hemicloud <- polar_to_cartesian(hemi)
rgl::plot3d(hemicloud)
```

`radius_search` *Radius Search of Points*

Description

Adapted radius searching of points based on RcppHNSW

Usage

```
radius_search(
  query,
  ref,
  radius,
  max_neighbour,
  distance = "euclidean",
  same = FALSE,
  threads = 1L,
  verbose = FALSE,
  progress = FALSE,
  ...
)
```

Arguments

`query`	A `data.table` containing the set of query points where each row represent a point and each column a given coordinate.
`ref`	A `numeric` containing the set of reference points where each row represent a point and each column a given coordinate.
`radius`	A `numeric` describing maximum euclidean distance form the each query points in which a point can be consider a neighbor.
`max_neighbour`	An `integer` specifying the maximum number of ref points to look around to consider for a given radius.
`distance`	Type of distance to calculate. `"euclidean"` as default. Look `hnsw_knn` for more options.

same	Logic. If TRUE, it delete neighbors with distance of 0, useful when the k search is based on the same query.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
verbose	If TRUE, log messages to the console.
progress	If TRUE, log a progress bar when verbose = TRUE. Tracking progress could cause a small overhead.
...	Arguments passed to hnsw_build and hnsw_search.

Details

This function is based on hnswlib C++ library (Malkov & Yashunin 2016) and its bindings for R (RcppHNSW; Melville 2020) for a fast estimation of neighbors points. It is adapted to simplify the workflow within rTLS. If you use this function, please consider cite the C++ library and RcppHNSW package.

Value

A data.table with three columns describing the indices of the query and ref points and the distances.

Author(s)

J. Antonio Guzmán Q.

References

Malkov, Y. A., & Yashunin, D. A. (2016). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv preprint arXiv:1603.09320.

See Also

radius_search

Examples

```
#Point cloud
data("pc_tree")


#Radius search of 1
radius_search(pc_tree, pc_tree, radius = 1, max_neighbour = 100)
```

rotate2D *Rotate a plane of coordinates*

Description

Rotate a plane of coordinates to a given angle.

Usage

```
rotate2D(plane, angle, threads = 1)
```

Arguments

plane	A data.table with two columns describing the plane of coordinates.
angle	A numeric vector describing the degrees of rotation.
threads	An integer specifying the number of threads to use. Experiment to see what works best for your data on your hardware.

Value

A data.table with the rotation applied to plane.

Author(s)

J. Antonio Guzmán Q.

Examples

```
data(pc_tree)

plot(pc_tree[,1:2])

#Rotate in 45 degrees using Z axis of the cloud
plot(rotate2D(pc_tree[,1:2], angle = 45))
```

rotate3D *Rotate a Point Cloud*

Description

Rotate point cloud based on the roll, pitch, and yaw angles.

Usage

```
rotate3D(cloud, roll = 0, pitch = 0, yaw = 0, threads = 1)
```

Arguments

cloud	A data.table with three columns describing the *XYZ* coordinates of a point cloud.
roll	A numeric vector describing the degrees of rotation angles for roll (*X*).
pitch	A numeric vector describing the degrees of rotation angles for pitch (*Y*).
yaw	A numeric vector describing the degrees of rotation angles for yaw (*Z*). for the roll, pitch, and yaw.
threads	An integer specifying the number of threads to use. Experiment to see what works best for your data on your hardware.

Details

The *XYZ* coordinates are transformed to E-N-U coordinates (ENU system, East-North-Up).

Value

A data.table with the rotation applied to cloud.

Author(s)

J. Antonio Guzmán Q.

Examples

```
data(pc_tree)
rgl::plot3d(pc_tree)
rgl::plot3d(rotate3D(pc_tree, roll = 45, pitch = 45, yaw = 0))
```

---

stand_counting *Stand Counting*

Description

Applies the voxels_counting function on a grid base point cloud.

Usage

```
stand_counting(
  cloud,
  xy.res,
  z.res = NULL,
  points.min = NULL,
  min_size,
  edge_sizes = NULL,
  length_out = 10,
```

```
  bootstrap = FALSE,
  R = NULL,
  progress = TRUE,
  parallel = FALSE,
  threads = NULL
)
```

Arguments

cloud	A data.table of a point cloud with xyz coordinates in the first three columns.
xy.res	A positive numeric vector describing the grid resolution of the xy coordinates to perform.
z.res	A positive numeric vector of length 1 describing the vertical resolution. If z.res = NULL vertical profiles are not used.
points.min	A positive numeric vector of length 1 minimum number of points to retain a sub-grid.
min_size	A positive numeric vector of length 1 describing the minimum cube edge length to perform. This is required if edge_sizes = NULL.
edge_sizes	A positive numeric vector describing the edge length of the different cubes to perform within each subgrid when z.res = NULL. If edge_sizes = NULL, it uses the maximum range of values for the xyz coordinates.
length_out	A positive interger of length 1 indicating the number of different edge lengths to use for each subgrid. This is required if edge_sizes = NULL.
bootstrap	Logical. If TRUE, it computes a bootstrap on the H index calculations. FALSE as default.
R	A positive integer of length 1 indicating the number of bootstrap replicates. This need to be used if bootstrap = TRUE.
progress	Logical, if TRUE displays a graphical progress bar. TRUE as default.
parallel	Logical, if TRUE it uses a parallel processing for the voxelization. FALSE as default.
threads	An integer >= 0 describing the number of threads to use. This need to be used if parallel = TRUE.

Value

A data.table with the summary of the voxels per grid created with their features.

Author(s)

J. Antonio Guzmán Q.

See Also

voxels_counting, voxels, summary_voxels

Examples

```
data(pc_tree)

#Applying stand_counting.

stand_counting(pc_tree, xy.res = c(4, 4), min_size = 3)


#Applying stand_counting using bootstrap in the H index.

stand_counting(pc_tree,
               xy.res = c(4, 4),
               min_size = 3,
               bootstrap = TRUE,
               R = 10)
```

`summary_voxels` *Voxels Summary*

Description

Create a summary objects of class "voxels" created using the voxels.

Usage

```
summary_voxels(voxels, edge_length = NULL, bootstrap = FALSE, R = NULL)
```

Arguments

voxels	An object of class voxels created using the voxels() function or a data.table describing the voxels coordinates and their number of points produced using voxels().
edge_length	A positive numeric vector with the voxel-edge length for the x, y, and z coordinates. This need to be used if class(voxels) != "voxels". It use the same dimensional scale of the point cloud.
bootstrap	Logical, if TRUE it computes a bootstrap on the H index calculations. FALSE as default.
R	A positive integer of length 1 indicating the number of bootstrap replicates. This need to be used if bootstrap = TRUE.

Details

The function provides 12 main statistics of the voxels. Specifically, the first three columns represent the edge length of the voxels, the following three columns (ei. `N_voxels`, `Volume`, `Surface`) describe the number of voxels created, the total volume that they represent, and the surface area that they cover. Following columns represent the mean (`Density_mean`) and sd (`Density_sd`) of the density of points per voxel (e.g. points/m2). Columns 9:12 provide metrics calculated using the Shannon Index. Specifically, `H` describe the entropy, `H_max` the maximum entropy, `Equitavility` the ratio between `H` and `Hmax`, and `Negentropy` describe the product of `Hmax - H`. If `bootstrap = TRUE` four more columns are created (13:16). These represent the `mean` and `sd` of the H index estimated using bootstrap (`H_boot_mean` and `H_boot_sd`), the `Equtavility_boot` as the ratio of the ratio between `H_boot_sd` and `Hmax`, and `Negentropy_boot` as the product `Hmax - H_boot_mean`.

Value

A `data.table` with with the summary of `voxels`.

Author(s)

J. Antonio Guzmán Q.

See Also

`voxels`, `voxels_counting`, `plot_voxels`

Examples

```
data("pc_tree")

#Apply a summary on a object of class "voxels" using bootstrap with 1000 replicates.
vox <- voxels(pc_tree, edge_length = c(0.5, 0.5, 0.5))
summary_voxels(vox, bootstrap = TRUE, R = 1000)

#Apply a summary on a product from 'voxels' using bootstrap with 1000 replicates.
vox <- voxels(pc_tree, edge_length = c(0.5, 0.5, 0.5), obj.voxels = FALSE)
summary_voxels(vox, edge_length = c(0.5, 0.5, 0.5), bootstrap = TRUE, R = 1000)
```

TLS_scan *A TLS scan*

Description

A `data.table` from a TLS scan.

Usage

```
data(TLS_scan)
```

Format

A `data.table` with five columns, which are:

X the "X" coordinate

Y the "Y coordinate

Z the "Z" coordinate

Target_count The number of received by the same laser shot

Target_index The rank of the returned pulse in the target count of received by the same laser shot

A `data.table` where the rows represent the pulse returns and the three columns represent the *XYZ* coordinates, and the target count and index.

Details

A TLS scan conducted using a Reigel VZ400i with a vertical and horizontal resolution of 0.048 and 0.622 degrees (2082 and 580 lines, respectively). The scanner has frame of scanning between 30 and 130.024 degrees zenith and 0 and 359.90 degrees azimuth. At the moment of the scan the roll, pitch, and yaw of the scanner were 1.026, 0.746, -110.019, respectively. The scanner coordinates in this scan are `x = 0, y = 0, z = 0`.

Examples

```
data(TLS_scan)
head(TLS_scan)
```

---

tree_metrics *Tree Metrics*

Description

Estimate the tree height, crown area, and the diameter at breast height of a tree point cloud

Usage

```
tree_metrics(cloud, region.diameter = NULL, relocateZ = TRUE)
```

Arguments

`cloud`	A `data.table` of the target point with three columns of the *XYZ* coordinates.
`region.diameter`	A `numeric` vector of length 2 indicating the lower and higher region to subset the point cloud and get the diameter. If `region.diameter = NULL`, it use `c(1.25, 1.35)`. `NULL` as default.
`relocateZ`	Logical, if `TRUE` it relocates the *Z* coordinates to a minimum coordinate of zero based on the current `min(cloud[,3])`. Useful if the base value (*Z*) of a tree point cloud is not topography corrected.

Details

The tree height is estimated based on the maximum value of *Z*, the crown area is calculated applying a convex hull on the point cloud, while the DBH is calculated extracting the area of the convex hull on the subset of points between `region.diameter`, and then estimating the diameter of a circle. For another estimation of DBH try `circleRANSAC` or for irregular trucks try `trunk_volume`.

Value

A `data.table` with the tree height, crown area, and diameter

Author(s)

J. Antonio Guzman Q. and Ronny Hernandez

See Also

`circleRANSAC`, `trunk_volume`

Examples

```
data("pc_tree")
tree_metrics(pc_tree)
```

---

trunk_volume *Tree Trunk Volume*

Description

Estimates the tree trunk volume of a point cloud using the `ashape3d` package.

Usage

```
trunk_volume(cloud, max.height = NULL, alpha = 0.2, plot = TRUE, ...)
```

Arguments

`cloud`	A `data.table` with three columns representing the *XYZ* coordinates of a point cloud.
`max.height`	A `numeric` vector to contemplate points in the cloud lower than a specific height. If `NULL`, it performs the alpha-shape on the entire point cloud.
`alpha`	A `numeric` vector of length one passed to `ashape3d` to describes alpha. `alpha = 0.20` as default since it seems to provide better estimations of the trunk volume. However, the `alpha` value may depends on the resolution of the point cloud.
`plot`	Logical. If `TRUE`, it uses `plot.ashape3d` to represent the alpha-shape.
`...`	General arguments passed to `ashape3d`.

Details

This is an adaptation of the code develop by Lafarge & Pateiro-Lopez (2017) based on Edelsbrunner & Mucke (1994) for the quick extraction of the tree trunk volume. Therefore, if you use this code we kindly suggest to cite these documents in your research.

Value

A `numeric` vector with the estimated trunk volume.

Author(s)

J. Antonio Guzmán Q.

References

Lafarge, T., Pateiro-Lopez, B. (2017). Implementation of the 3D Alpha-Shape for the Reconstruction of 3D Sets from a Point Cloud. Available at https://CRAN.R-project.org/package=alphashape3d.

Edelsbrunner, H., Mucke, E. P. (1994). Three-Dimensional Alpha Shapes. ACM Transactions on Graphics, 13(1), pp.43-72.

See Also

`tree_metrics`, `circleRANSAC`

Examples

```
data("pc_tree")

#Estimates the trunk volume of a height lower than 1.75.
trunk_volume(pc_tree, max.height = 1.75)
```

---

voxels *Voxelization of a Point Cloud*

Description

Create cubes of a given distance in a point cloud though their voxelization. It use a modify version of the code used in Greaves et al. 2015.

Usage

```
voxels(cloud, edge_length, threads = 1L, obj.voxels = TRUE)
```

Arguments

cloud	A data.table with *XYZ* coordinates in the first three columns.
edge_length	A positive numeric vector with the voxel-edge length for the x, y, and z coordinates. It use the same dimensional scale of the point cloud.
threads	An integer specifying the number of threads to use for parallel processing. Experiment to see what works best for your data on your hardware.
obj.voxels	Logical. If obj.voxel = TRUE, it returns an object of class "voxels", If obj.voxel = FALSE, it returns a data.table with the coordinates of the voxels created and the number of points in each voxel. TRUE as default.

Details

Voxels are created from the negative to the positive *XYZ* coordinates.

Value

If obj.voxels == TRUE, it return an object of class "voxels" which contain a list with the points used to create the voxels, the parameter edge_length, and the voxels created. If FALSE, it returns a data.table with the coordinates of the voxels created and the number of points in each voxel.

Author(s)

J. Antonio Guzmán Q.

References

Greaves, H. E., Vierling, L. A., Eitel, J. U., Boelman, N. T., Magney, T. S., Prager, C. M., & Griffin, K. L. (2015). Estimating aboveground biomass and leaf area of low-stature Arctic shrubs with terrestrial LiDAR. Remote Sensing of Environment, 164, 26-35.

See Also

voxels_counting, plot_voxels, summary_voxels

Examples

```
data("pc_tree")

###Create cube of a size of 0.5.
voxels(pc_tree, edge_length = c(0.5, 0.5, 0.5))
```

voxels_counting *Voxels Counting*

Description

Creates cube like voxels of different size on a point cloud using the voxels function, and then return a summary_voxels of their features.

Usage

```
voxels_counting(
  cloud,
  edge_sizes = NULL,
  min_size,
  length_out = 10,
  bootstrap = FALSE,
  R = NULL,
  progress = TRUE,
  parallel = FALSE,
  threads = NULL
)
```

Arguments

cloud	A data.table with xyz coordinates of the point clouds in the first three columns.
edge_sizes	A positive numeric vector describing the edge length of the different cubes to perform. If NULL, it use edge sizes by default based on the largest range of XYZ and min_size.
min_size	A positive numeric vector of length 1 describing the minimum cube edge length to perform. This is required if edge_sizes = NULL.
length_out	A positive interger of length 1 indicating the number of different edge lengths to use. This is required if edge_sizes = NULL.
bootstrap	Logical. If TRUE, it computes a bootstrap on the H index calculations. FALSE as default.
R	A positive integer of length 1 indicating the number of bootstrap replicates. This need to be used if bootstrap = TRUE.
progress	Logical, if TRUE displays a graphical progress bar. TRUE as default.
parallel	Logical, if TRUE it uses a parallel processing for the voxelization. FALSE as default.
threads	An integer >= 0 describing the number of threads to use. This need to be used if parallel = TRUE.

Value

A data.table with the summary of the voxels created with their features.

Author(s)

J. Antonio Guzmán Q.

See Also

voxels, summary_voxels, plot_voxels

Examples

```
data(pc_tree)

#Applying voxels counting.
voxels_counting(pc_tree, min_size = 2)

#Voxels counting using bootstrap on the H indexes with 1000 repetitions.
voxels_counting(pc_tree, min_size = 2, bootstrap = TRUE, R = 1000)
```

Index