

Package 'lidR'

March 3, 2019

Type Package

Title Airborne LiDAR Data Manipulation and Visualization for Forestry Applications

Version 2.0.2

Date 2019-02-15

Description Airborne LiDAR (Light Detection and Ranging) interface for data manipulation and visualization. Read/write 'las' and 'laz' files, computation of metrics in area based approach, point filtering, artificial point reduction, classification from geographic data, normalization, individual tree segmentation and other manipulations.

URL <https://github.com/Jean-Romain/lidR>

BugReports <https://github.com/Jean-Romain/lidR/issues>

License GPL-3

Depends R (>= 3.1.0), methods, raster, sp

Imports data.table, future (>= 1.10.0), gdalUtils, geometry, glue, grDevices, gstat, lazyeval, imager, Rcpp (>= 0.11.0), RCSF, rgeos, rgdal, rgl, rlas (>= 1.2.5), sf, stats, tools, utils

Suggests EBImage, concaveman, crayon, hexbin, mapview, progress, testthat, knitr, rmarkdown

LazyData true

RoxygenNote 6.1.1

LinkingTo BH, Rcpp

Encoding UTF-8

ByteCompile true

VignetteBuilder knitr

biocViews

Collate 'LASmethods-generic.r' 'Class-LASheader.r' 'Class-LAS.r' 'Class-LAScatalog.r' 'Class-LAScluster.r' 'LASmethods-LAS.r' 'LASmethods-LAScatalog.r' 'LASmethods-LAScluster.r'

'LASmethods-LASheader.r' 'RcppExports.R'
 'algo_digital_surface_model.r' 'algo_ground_segmentation.r'
 'algo_points_decimation.r' 'algo_snags_segmentation.r'
 'algo_spatial_interpolations.r' 'algo_tree_detection.R'
 'algo_tree_segmentation.r' 'catalog_apply.r'
 'catalog_fakerun.r' 'catalog_index.r' 'catalog_intersect.r'
 'catalog_laxindex.r' 'catalog_makecluster.r' 'catalog_retile.r'
 'catalog_select.r' 'clusters_apply.r' 'grid_canopy.r'
 'grid_density.r' 'grid_hexametrics.r' 'grid_metrics.r'
 'grid_metrics3d.r' 'grid_terrain.r' 'io_readLAS.r'
 'io_writeANY.r' 'io_writeLAS.r' 'lascheck.r' 'lasclip.r'
 'lasfilter.r' 'lasfilterdecimate.r' 'lasfilterduplicates.r'
 'lasfiltersurfacepoints.r' 'lasground.r' 'lasindentify.r'
 'lasmergelas.r' 'lasmergespatial.r' 'lasmetrics.r'
 'lasnormalize.r' 'lasroi.r' 'lassmooth.r' 'lassnags.r'
 'lastransform.r' 'lastrees.r' 'lasupdateheader.r'
 'lasvoxelize.r' 'plot.s3.r' 'tree_detection.r' 'tree_hulls.r'
 'tree_metrics.r' 'utils_assertive.r' 'utils_catalog_options.r'
 'utils_colors.r' 'utils_define_constant.R' 'utils_geometry.r'
 'utils_is.r' 'utils_match_chm_and_seeds.r' 'utils_metrics.r'
 'utils_misc.r' 'utils_typecast.r' 'zzz.r'

NeedsCompilation yes

Author Jean-Romain Roussel [aut, cre, cph],

David Auty [aut, ctb] (Reviews the documentation),

Florian De Boissieu [ctb] (Fixed bugs and improved catalog features),

Andrew Sánchez Meador [ctb] (Implemented lassnags)

Maintainer Jean-Romain Roussel <jean-romain.rousseau.1@ulaval.ca>

Repository CRAN

Date/Publication 2019-03-03 20:30:03 UTC

R topics documented:

area	4
as.list.LASheader	5
as.spatial	5
catalog	6
catalog_apply	7
catalog_intersect	11
catalog_options_tools	12
catalog_retile	13
csf	15
dalponte2016	16
dsmtin	18
entropy	19
epsg	20
extent,LAS-method	21

gap_fraction_profile	22
grid_canopy	23
grid_density	25
grid_hexametrics	26
grid_metrics	28
grid_metrics3d	30
grid_terrain	32
highest	34
homogenize	34
is.empty	35
knnidw	36
kriging	37
LAD	37
LAS-class	38
lasaddattribute	40
LAScatalog-class	42
lascheck	45
lasclip	46
lasfilter	49
lasfilterdecimate	50
lasfilterduplicates	51
lasfilters	52
lasfiltersurfacepoints	54
lasground	55
LASheader	57
LASheader-class	57
lasmergespatial	58
lasmetrics	59
lasnormalize	60
laspulse	63
lassmooth	64
lassnags	65
lastransform	67
lastrees	68
lasvoxelize	68
li2012	70
lidrpalettes	71
lmf	71
manual	73
p2r	74
pitfree	75
plot	76
plot.lasmetrics3d	78
plot_3d	79
pmf	81
print	82
random	82
rbind.LAS	83

readLAS	83
rumple_index	84
silva2016	86
stdmetrics	87
tin	90
tree_detection	91
tree_hulls	92
tree_metrics	94
util_makeZhangParam	96
VCI	97
watershed	98
wing2015	99
writeLAS	101
\$<- ,LAS-method	102

Index **104**

area *Surface covered by a LAS* object.*

Description

Surface covered by a LAS* object. For LAS point clouds it is computed based on the convex hull of the points. For a LAScatalog it is computed as the sum of the bounding boxes of the files. For overlapping tiles the value may be larger than the total covered area because some regions are sampled twice.

Usage

```
area(x, ...)

## S4 method for signature 'LAS'
area(x, ...)

## S4 method for signature 'LAScatalog'
area(x, ...)
```

Arguments

```
x          An object of the class LAS*
...        unused
```

Value

numeric. The area of the object computed in the same units as the coordinate reference system

as.list.LASheader	<i>Transform to a list</i>
-------------------	----------------------------

Description

Functions to construct, coerce and check for both kinds of R lists.

Usage

```
## S3 method for class 'LASheader'  
as.list(x, ...)
```

Arguments

x	A LASheader object
...	unused

as.spatial	<i>Transform a LAS* object into an sp object</i>
------------	--

Description

LAS and LAScatalog objects are transformed into SpatialPointsDataFrame and SpatialPolygons-DataFrame objects, respectively.

Usage

```
as.spatial(x)
```

Arguments

x	an object from the lidR package
---	---------------------------------

Value

An object from sp

`catalog`*Create an object of class `LAScatalog`*

Description

Create an object of class `LAScatalog` from a folder or a set of filenames. A `LAScatalog` is a representation of a set of las/laz files. A computer cannot load all the data at once. A `LAScatalog` is a simple way to manage all the files sequentially. Most functions from `lidR` can be used seamlessly with a `LAScatalog` using the internal `LAScatalog` processing engine. To take advantage of the `LAScatalog` processing engine the user must first adjust some processing options using the [appropriated functions](#). Careful reading of the [LAScatalog class documentation](#) is required to use the `LAScatalog` class correctly.

Usage

```
catalog(folder, ...)
```

Arguments

<code>folder</code>	string. The path of a folder containing a set of las/laz files. Can also be a vector of file paths.
<code>...</code>	Extra parameters to list.files . Typically <code>'recursive = TRUE'</code> .

Value

A `LAScatalog` object

Examples

```
# A single file LAScatalog using data provided with the package
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
ctg = catalog(LASfile)
plot(ctg)

## Not run:
ctg <- catalog("/path/to/a/folder/of/las/files")

# Internal engine will compute in parallel using two cores
opt_cores(ctg) <- 2L

# Internal engine will sequentially process regions of interest of size 500 x 500 m (clusters)
opt_chunk_size(ctg) <- 500

# Internal engine will align the 500 x 500 m clusters on x = 250 and y = 300
opt_alignment(ctg) <- c(250, 300)

# Internal engine will not display a progress estimation
opt_progress(ctg) <- FALSE
```

```

# Internal engine will not return results into R. Instead it will write results in files.
opt_output_files(ctg) <- "/path/to/folder/templated_filename_{XBOTTOM}_{ID}"

# More details in the documentation
help("LAScatalog-class", "lidR")
help("catalog_options_tools", "lidR")

## End(Not run)

```

catalog_apply

LAScatalog processing engine

Description

This function gives users access to the [LAScatalog](#) processing engine. It allows the application of a user-defined routine over an entire catalog. The LAScatalog processing engine tool is explained in the [LAScatalog class](#)

Warning: the LAScatalog processing engine has a mechanism to load buffered data to avoid edge artifacts, but no mechanism to remove the buffer after applying user-defined functions, since this task is specific to each process. In other lidR functions this task is performed specifically for each function. In catalog_apply the user's function can return any output, thus users must take care of this task themselves (See section "Edge artifacts")

Usage

```
catalog_apply(ctg, FUN, ..., .options = NULL)
```

Arguments

ctg	A LAScatalog object.
FUN	A user-defined function that respects a given template (see section function template)
...	Optional arguments to FUN.
.options	See dedicated section and example.

Edge artifacts

It is important to take precautions to avoid 'edge artifacts' when processing wall-to-wall tiles. If the points from neighboring tiles are not included during certain processes, this could create 'edge artifacts' at the tile boundaries. For example, empty or incomplete pixels in a rasterization process, or dummy elevations in a ground interpolation. The LAScatalog processing engine provides internal tools to load buffered data. However, there is no mechanism to remove the results computed in the buffered area since this task depends on the output of the user-defined function. The user must take care of this task (see examples).

Buffered data

The LAS objects read by the user function have a special attribute called 'buffer' that indicates, for each point, if it comes from a buffered area or not. Points from non-buffered areas have a 'buffer' value of 0, while points from buffered areas have a 'buffer' value of 1, 2, 3 or 4, where 1 is the bottom buffer and 2, 3 and 4 are the left, top and right buffers, respectively.

Function template

The parameter FUN expects a function with a first argument that will be supplied automatically by the LAScatalog processing engine. This first argument is a LAScluster. A LAScluster is an internal undocumented class but the user needs to know only three things about this class:

- It represents a chunk of the catalog
- The function `readLAS` can be used with a LAScluster
- The function `extent` or `bbox` can be used with a LAScluster and it returns the bounding box of the cluster without the buffer. It can be used to clip the output and remove the buffered region (see examples).

A user-defined function must be templated like this:

```
myfun = function(cluster, ...)
{
  las = readLAS(cluster)
  if (is.empty(las)) return(NULL)
  # do something
  # remove the buffer of the output
  return(something)
}
```

The line `if(is.empty(las)) return(NULL)` is important because some clusters (chunks) may contain 0 points (we can't know that before reading the file). In this case an empty point cloud with 0 points is returned by `readLAS` and this may fail in subsequent code. Thus, exiting early from the user-defined function by returning `NULL` indicates to the internal engine that the cluster was empty.

.options

User may have noticed that some lidR functions throw an errors when the processing options are inappropriate. For example, some functions need a buffer and thus `buffer = 0` is forbidden. User can add the same constrains to protect against inappropriate options. The `.options` argument can be a list of options.

- `need_buffer = TRUE` the function complains if the buffer is 0
- `need_output_file = TRUE` the function complains if no output file template is provided
- `drop_null = FALSE` the `NULL` outputs are not automatically removed (useful in very specific internal cases)
- `raster_alignment = ...` very important option, see below.

When the function FUN returns a raster it is important to ensure that the chunks are aligned with the raster to avoid edge artifacts. Indeed, if the edge of a chunk does not correspond to the edge of the pixels the output will not be strictly continuous and will have edge artifacts (that might not be visible). Users can check this with the options `raster_alignment`, that can take the resolution of the raster as input, as well as the starting point if needed. The following are accepted:

```
# check if the chunks are aligned with a raster of resolution 20
raster_alignment = 20
raster_alignment = list(res = 20)

# check if chunks are aligned with a raster of resolution 20
# that starts a (0,10)
raster_alignment = list(res = 20, start = c(0,10))
```

See also [grid_metrics](#) for more details.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk_size**: How much data is loaded at once.
- **chunk_buffer**: Load chunks with a buffer.
- **chunk_alignment**: Align the chunks.
- **cores**: How many chunks are loaded and processed at once.
- **progress**: Displays a progress estimate.
- **output_files**: The user-defined function outputs will be written to files instead of being returned into R.
- **laz_compression**: write las or laz files only if the user-defined function returns a LAS object.
- **select**: Select only the data of interest to save processing memory.
- **filter**: Read only the points of interest.

Examples

```
# More examples might be available in the official lidR vignettes or
# on the github wiki <http://jean-romain.github.io/lidR/wiki>

## =====
## Example 1: detect all the tree tops over an entire catalog
## (this is basically the reproduction of existing lidR function 'tree_detection')
## =====

# 1. Build the user-defined function that analyzes each chunk of the catalog.
# The function's first argument is a LAScluster object. The other arguments can be freely
# chosen by the user.
my_tree_detection_method <- function(cluster, ws)
```

```

{
  # The cluster argument is a LAScluster object. The user does not need to know how it works.
  # readLAS will load the region of interest (chunk) with a buffer around it, taking advantage of
  # point cloud indexation if possible. The filter and select options are propagated automatically
  las <- readLAS(cluster)
  if (is.empty(las)) return(NULL)

  # Find the tree tops using a user-developed method (here simply a LMF).
  ttops <- tree_detection(las, lmf(ws))

  # ttops is a SpatialPointsDataFrame that contains the tree tops in our region of interest
  # plus the trees tops in the buffered area. We need to remove the buffer otherwise we will get
  # some trees more than once.
  bbox <- raster::extent(cluster)
  ttops <- raster::crop(ttops, bbox)

  return(ttops)
}

# 2. Build a project (here, a single file catalog for the purposes of this dummy example).
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
project <- catalog(LASfile)
plot(project)

# 3. Set some catalog options.
# For this dummy example, the chunk size is 80 m and the buffer is 10 m using a single core.
opt_chunk_buffer(project) <- 10
opt_cores(project) <- 1L
opt_chunk_size(project) <- 80 # small because this is a dummy example.
opt_select(project) <- "xyz" # read only the coordinates.
opt_filter(project) <- "-keep_first" # read only first returns.

# 4. Apply a user-defined function to take advantage of the internal engine
opt <- list(need_buffer = TRUE) # catalog_apply will throw an error if buffer = 0
output <- catalog_apply(project, my_tree_detection_method, ws = 5, .options = opt)

# 5. Post-process the output to merge the results (depending on the output computed).
# Here, each value of the list is a SpatialPointsDataFrame, so rbind does the job:
output <- do.call(rbind, output)
splot(output)

## =====
## Example 2: compute a rumple index on surface points
## =====

rumple_index_surface = function(cluster, res)
{
  las = readLAS(cluster)
  if (is.empty(las)) return(NULL)

  las <- lasfiltersurfacepoints(las, 1)
  rumple <- grid_metrics(las, rumple_index(X,Y,Z), res)
  bbox <- raster::extent(cluster)

```

```

    rumple <- raster::crop(rumple, bbox)

    return(rumple)
}

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
project <- catalog(LASfile)

opt_chunk_buffer(project) <- 1
opt_cores(project) <- 1L
opt_chunk_size(project) <- 120 # small because this is a dummy example.
opt_select(project) <- "xyz" # read only the coordinates.

opt <- list(raster_alignment = 20) # catalog_apply will adjust the chunks if required
output <- catalog_apply(project, rumple_index_surface, res = 20, .options = opt)
output <- do.call(raster::merge, output)
plot(output, col = height.colors(50))

```

catalog_intersect *Subset a LAScatalog with a Spatial* object*

Description

Subset a LAScatalog with a Spatial* object to keep only the tiles of interest. Internally, it uses the function [intersect](#) from raster with a tweak to make it work with a LAScatalog. It can be used to select tiles of interest that encompass Spatial* objects such as SpatialPoints, SpatialPolygons or SpatialLines.

Usage

```
catalog_intersect(ctg, y)
```

Arguments

ctg	A LAScatalog object
y	Extent, Raster*, SpatialPolygons*, SpatialLines* or SpatialPoints* object

Value

A LAScatalog

catalog_options_tools *Get or set LAScatalog processing engine options*

Description

The names of the options and their roles are documented in [LAScatalog](#). The options are used by all the functions that support a LAScatalog as input.

Usage

```
opt_chunk_buffer(ctg)
opt_chunk_buffer(ctg) <- value
opt_chunk_size(ctg)
opt_chunk_size(ctg) <- value
opt_chunk_alignment(ctg)
opt_chunk_alignment(ctg) <- value
opt_cores(ctg)
opt_cores(ctg) <- value
opt_progress(ctg)
opt_progress(ctg) <- value
opt_stop_early(ctg)
opt_stop_early(ctg) <- value
opt_wall_to_wall(ctg)
opt_wall_to_wall(ctg) <- value
opt_output_files(ctg)
opt_output_files(ctg) <- value
opt_laz_compression(ctg)
opt_laz_compression(ctg) <- value
opt_select(ctg)
```

```

opt_select(ctg) <- value

opt_filter(ctg)

opt_filter(ctg) <- value

```

Arguments

ctg	An object of class LAScatalog
value	An appropriate value depending on the expected input.

Examples

```

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
ctg = catalog(LASfile)

plot(ctg, chunk_pattern = TRUE)

opt_chunk_size(ctg) <- 150
plot(ctg, chunk_pattern = TRUE)

opt_chunk_buffer(ctg) <- 10
plot(ctg, chunk_pattern = TRUE)

opt_chunk_alignment(ctg) <- c(270,250)
plot(ctg, chunk_pattern = TRUE)

summary(ctg)

opt_output_files(ctg) <- "/path/to/folder/templated_filename_{XBOTTOM}_{ID}"
summary(ctg)

```

catalog_retile

Retile a LAScatalog

Description

Splits or merges files to reshape the original catalog files (.las or .laz) into smaller or larger files. It also enables the addition or removal of a buffer around the tiles. The function first displays the layout of the new tiling pattern and then asks the user to validate the command.

Internally, the function reads and writes the clusters defined by the internal processing options of a [LAScatalog](#) processing engine. Thus, the function is flexible and enables the user to retile the dataset, retile while adding or removing a buffer (negative buffers are allowed), or optionally to compress the data by retiling without changing the pattern but by changing the format (las/laz).

Note that this function is not actually very useful since `lidR` manages everything (clipping, processing, buffering, ...) internally using the proper options. Thus, retiling may be useful for working in other software, for example, but not in `lidR`.

Usage

```
catalog_retile(ctg)
```

Arguments

ctg A [LAScatalog](#) object

Value

A new LAScatalog object

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk_size**: Size of the new tiles.
- **buffer**: Load new tiles with a buffer. The expected value is usually 0.
- **alignment**: Alignment of the new tiles.
- **cores**: The number of cores used. `catalog_retile` streams the data (nothing is loaded at the R level). The maximum number of cores can be safely used.
- **progress**: Displays a progress estimation.
- **output_files***: Mandatory. The new tiles will be written in new files.
- **laz_compression**: save las or laz files.
- **select**: `catalog_retile` preserve the file format anyway.
- **filter**: Retile and save only the points of interest.

Examples

```
## Not run:
ctg = catalog("path/to/catalog")

# Create a new set of .las files 500 x 500 wide in the folder
# path/to/new/catalog/ and iteratively named Forest_1.las, Forest_2.las
# Forest_3.las, and so on.

opt_chunk_buffer(ctg) <- 0
opt_chunk_size(ctg) <- 500
opt_output_files(ctg) <- "path/to/new/catalog/Forest_{ID}"
newctg = catalog_retile(ctg)

# Create a new set of .las files equivalent to the original,
# but extended with a 50 m buffer in the folder path/to/new/catalog/
# and iteratively named named after the original files.

opt_chunk_buffer(ctg) <- 50
opt_chunk_size(ctg) <- 0
opt_output_files(ctg) <- "path/to/new/catalog/{ORIGINALFILENAME}_buffered"
newctg = catalog_retile(ctg)

# Create a new set of compressed .laz file equivalent to the original, keeping only
# first returns above 2 m

opt_chunk_buffer(ctg) <- 0
opt_chunk_size(ctg) <- 0
opt_laz_compression(ctg) <- TRUE
opt_filter(ctg) <- "-keep_first -drop_z_below 2"
newctg = catalog_retile(ctg)

## End(Not run)
```

csf

Ground Segmentation Algorithm

Description

This function is made to be used in [lasground](#). It implements an algorithm for segmentation of ground points base on a Cloth Simulation Filter. This method is a strict implementation of the CSF algorithm made by Zhang et al. (2016) (see references) that relies on the authors' original source code written and exposed to R via the the RCSF package.

Usage

```
csf(sloop_smooth = FALSE, class_threshold = 0.5,
    cloth_resolution = 0.5, rigidity = 1L, iterations = 500L,
    time_step = 0.65)
```

Arguments

<code>sloop_smooth</code>	logical. When steep slopes exist, set this parameter to TRUE to reduce errors during post-processing.
<code>class_threshold</code>	scalar. The distance to the simulated cloth to classify a point cloud into ground and non-ground. The default is 0.5.
<code>cloth_resolution</code>	scalar. The distance between particles in the cloth. This is usually set to the average distance of the points in the point cloud. The default value is 0.5.
<code>rigidness</code>	integer. The rigidness of the cloth. 1 stands for very soft (to fit rugged terrain), 2 stands for medium, and 3 stands for hard cloth (for flat terrain). The default is 1.
<code>iterations</code>	integer. Maximum iterations for simulating cloth. The default value is 500. Usually, there is no need to change this value.
<code>time_step</code>	scalar. Time step when simulating the cloth under gravity. The default value is 0.65. Usually, there is no need to change this value. It is suitable for most cases.

References

W. Zhang, J. Qi*, P. Wan, H. Wang, D. Xie, X. Wang, and G. Yan, "An Easy-to-Use Airborne LiDAR Data Filtering Method Based on Cloth Simulation," *Remote Sens.*, vol. 8, no. 6, p. 501, 2016. (<http://www.mdpi.com/2072-4292/8/6/501/htm>)

See Also

Other ground segmentation algorithms: [pmf](#)

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn")

las <- lasground(las, csf())
plot(las, color = "Classification")
```

dalponte2016

Individual Tree Segmentation Algorithm

Description

This function is made to be used in [lastrees](#). It implements an algorithm for tree segmentation based on the Dalponte and Coomes (2016) algorithm (see reference). This is a seeds + growing region algorithm. This algorithm exists in the package `itcSegment`. This version has been written from the paper in C++. Consequently it is hundreds to millions times faster than the original version. Note that this algorithm strictly performs a segmentation, while the original method as implemented in `itcSegment` and described in the manuscript also performs pre- and post-processing tasks. Here these tasks are expected to be done by the user in separate functions.

Usage

```
dalponte2016(chm, treetops, th_tree = 2, th_seed = 0.45,
             th_cr = 0.55, max_cr = 10, ID = "treeID")
```

Arguments

chm	RasterLayer. Image of the canopy. Can be computed with grid_canopy or read from an external file.
treetops	SpatialPointsDataFrame. Can be computed with tree_detection or read from an external shapefile.
th_tree	numeric. Threshold below which a pixel cannot be a tree. Default is 2.
th_seed	numeric. Growing threshold 1. See reference in Dalponte et al. 2016. A pixel is added to a region if its height is greater than the tree height multiplied by this value. It should be between 0 and 1. Default is 0.45.
th_cr	numeric. Growing threshold 2. See reference in Dalponte et al. 2016. A pixel is added to a region if its height is greater than the current mean height of the region multiplied by this value. It should be between 0 and 1. Default is 0.55.
max_cr	numeric. Maximum value of the crown diameter of a detected tree (in pixels). Default is 10.
ID	character. If the SpatialPointsDataFrame contains an attribute with the ID for each tree, the name of this attribute. This way, original IDs will be preserved. If there is no such data trees will be numbered sequentially.

Details

Because this algorithm works on a CHM only there is no actual need for a point cloud. Sometimes the user does not even have the point cloud that generated the CHM. `lidR` is a point cloud-oriented library, which is why this algorithm must be used in [lastrees](#) to merge the result with the point cloud. However the user can use this as a stand-alone function like this:

```
chm = raster("file/to/a/chm/")
ttops = tree_detection(chm, lmf(3))
crowns = dalponte2016(chm, ttops)()
```

References

Dalponte, M. and Coomes, D. A. (2016), Tree-centric mapping of forest carbon density from airborne laser scanning and hyperspectral data. *Methods Ecol Evol*, 7: 1236–1245. doi:10.1111/2041-210X.12575.

See Also

Other individual tree segmentation algorithms: [li2012](#), [silva2016](#), [watershed](#)

Other raster based tree segmentation algorithms: [silva2016](#), [watershed](#)

Examples

```

LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")
col <- pastel.colors(200)

chm <- grid_canopy(las, 0.5, p2r(0.3))
ker <- matrix(1,3,3)
chm <- raster::focal(chm, w = ker, fun = mean, na.rm = TRUE)

ttops <- tree_detection(chm, lmf(4, 2))
las <- lastrees(las, dalponte2016(chm, ttops))
plot(las, color = "treeID", colorPalette = col)

```

dsmtin

Digital Surface Model Algorithm

Description

This function is made to be used in [grid_canopy](#). It implements an algorithm for digital surface model computation using a Delaunay triangulation of first returns with a linear interpolation within each triangle.

Usage

```
dsmtin(max_edge = 0)
```

Arguments

max_edge	numeric. Maximum edge length of a triangle in the Delaunay triangulation. If a triangle has an edge length greater than this value it will be removed to trim dummy interpolation on non-convex areas. If max_edge = 0 no trimming is done (see examples).
----------	--

See Also

Other digital surface model algorithms: [p2r](#), [pitfree](#)

Examples

```

LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
col <- height.colors(50)

# Basic triangulation and rasterization of first returns
chm <- grid_canopy(las, res = 1, dsmtin())
plot(chm, col = col)

## Not run:
# Potentially complex concave subset of point cloud

```

```

x = c(481340, 481340, 481280, 481300, 481280, 481340)
y = c(3812940, 3813000, 3813000, 3812960, 3812940, 3812940)
las2 = lasclipPolygon(las,x,y)
plot(las2)

# Since the TIN interpolation is done within the convex hull of the point cloud
# dummy pixels are interpolated that are strictly correct according to the interpolation method
# used, but meaningless in our CHM
chm <- grid_canopy(las2, res = 0.5, dsmtin())
plot(chm, col = col)

# Use 'max_edge' to trim dummy triangles
chm = grid_canopy(las2, res = 0.5, dsmtin(max_edge = 3))
plot(chm, col = col)

## End(Not run)

```

entropy

Normalized Shannon diversity index

Description

A normalized Shannon vertical complexity index. The Shannon diversity index is a measure for quantifying diversity and is based on the number and frequency of species present. This index, developed by Shannon and Weaver for use in information theory, was successfully transferred to the description of species diversity in biological systems (Shannon 1948). Here it is applied to quantify the diversity and the evenness of an elevational distribution of las points. It makes bins between 0 and the maximum elevation. If there are negative values the function returns NA.

Usage

```
entropy(z, by = 1, zmax = NULL)
```

Arguments

z	vector of positive z coordinates
by	numeric. The thickness of the layers used (height bin)
zmax	numeric. Used to turn the function entropy to the function VCI .

Value

A number between 0 and 1

References

Pretzsch, H. (2008). Description and Analysis of Stand Structures. Springer Berlin Heidelberg. <http://doi.org/10.1007/978-3-540-88307-4> (pages 279-280) Shannon, Claude E. (1948), "A mathematical theory of communication," Bell System Tech. Journal 27, 379-423, 623-656.

See Also[VCI](#)**Examples**

```
z = runif(10000, 0, 10)

# expected to be close to 1. The highest diversity is given for a uniform distribution
entropy(z, by = 1)

z = runif(10000, 9, 10)

# Must be 0. The lowest diversity is given for a unique possibility
entropy(z, by = 1)

z = abs(rnorm(10000, 10, 1))

# expected to be between 0 and 1.
entropy(z, by = 1)
```

epsg

Get or set epsg code of a LAS object*

Description

Currently, at least in R, the only way to store the CRS in a las file is to store the epsg code of the CRS. On the other hand, all spatial R packages use the proj4string to store the CRS. This is why the CRS is duplicated in a LAS object. The proj4string is accessible with the functions `projection` from `raster` or `proj4string` from `sp`. These functions do nothing special except they return or update the proj4string, which is not recognized in a las file. The function `epsg`, in turn, updates the header of the LAS object **and** updates the proj4string and thus should be preferred to ensure that the CRS will be written into a las file.

Usage

```
epsg(object, ...)
```

```
epsg(object) <- value
```

```
## S4 method for signature 'LASheader'
```

```
epsg(object, ...)
```

```
## S4 replacement method for signature 'LASheader'
```

```
epsg(object) <- value
```

```
## S4 method for signature 'LAS'
```

```
epsg(object)
```

```
## S4 replacement method for signature 'LAS'
epsg(object) <- value
```

Arguments

object	An object of class LAS or eventually LASheader (regular users don't need to manipulate LASheader objects)
...	Unused
value	integer. EPSG code.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)

projection(las)
epsg(las)

# Changing the CRS via projection updates the proj4string but not the header
# The proj4string will be propagated in all subsequent spatial objects but the las file
# itself will still have the epsg code 26917 if written on the disk
projection(las) = sp::CRS("+init=epsg:2567")
projection(las)
epsg(las)

# Changing the CRS via epsg updates the proj4string and the header. This way the good CRS will
# be written in the las file (if written).
epsg(las) = 2567
epsg(las)
projection(las)
```

extent,LAS-method	<i>Extent</i>
-------------------	---------------

Description

Returns an Extent object of a LAS*.

Usage

```
## S4 method for signature 'LAS'
extent(x, ...)
```

```
## S4 method for signature 'LAScatalog'
extent(x, ...)
```

Arguments

x An object of the class LAS or LAScatalog
... Unused

Value

Extent object from **raster**

See Also

[raster::extent](#)

gap_fraction_profile *Gap fraction profile*

Description

Computes the gap fraction profile using the method of Bouvier et al. (see reference)

Usage

```
gap_fraction_profile(z, dz = 1, z0 = 2)
```

Arguments

z vector of positive z coordinates
dz numeric. The thickness of the layers used (height bin)
z0 numeric. The bottom limit of the profile

Details

The function assesses the number of laser points that actually reached the layer $z+dz$ and those that passed through the layer $[z, z+dz]$. By definition the layer 0 will always return 0 because no returns pass through the ground. Therefore, the layer 0 is removed from the returned results.

Value

A data.frame containing the bin elevations (z) and the gap fraction for each bin (gf)

References

Bouvier, M., Durrieu, S., Fournier, R. a, & Renaud, J. (2015). Generalizing predictive models of forest inventory attributes using an area-based approach with airborne las data. Remote Sensing of Environment, 156, 322-334. <http://doi.org/10.1016/j.rse.2014.10.004>

See Also

[LAD](#)

Examples

```

z = c(rnorm(1e4, 25, 6), rgamma(1e3, 1, 8)*6, rgamma(5e2, 5,5)*10)
z = z[z<45 & z>0]

hist(z, n=50)

gapFraction = gap_fraction_profile(z)

plot(gapFraction, type="l", xlab="Elevation", ylab="Gap fraction")

```

grid_canopy

Digital Surface Model

Description

Creates a digital surface model using several possible algorithms.

Usage

```
grid_canopy(las, res, algorithm)
```

Arguments

las	An object of class LAS or LAScatalog .
res	numeric. The resolution of the output Raster. Can optionally be a RasterLayer. In that case the RasterLayer is used as the layout.
algorithm	function. A function that implements an algorithm to compute a digital surface model. lidR implements p2r , dsmtin , pitfree (see respective documentation and examples).

Value

A RasterLayer containing a numeric value in each cell. If the RasterLayers are written on disk when running the function with a LAScatalog, a virtual raster mosaic is returned (see [gdalbuildvrt](#))

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports `.lax` files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a `.lax` files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog in `grid_*` functions (in bold). For more details see the [LAScatalog engine documentation](#):

- **tiling_size**: How much data is loaded at once. The tiling size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when tiling size is equal to 0 (processing by file).
- **buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **alignment**: Align the processed clusters. The alignment may be slightly modified.
- **chunk size**: How much data is loaded at once. The chunk size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when chunk size is equal to 0 (processing by file).
- **chunk buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks. The alignment may be slightly modified internally to ensure a strict continuous wall-to-wall output.
- **cores**: How many cores are used.
- **progress**: Displays a progress estimate.
- **output_files**: Return the output in R or write each cluster's output in a file. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk size is equal to 0 (processing by file), ORIGINALFILENAME.
- **laz_compression**: is not supported because this function will never write las/laz files.
- **select**: The `grid_*` functions usually 'know' what should be loaded and this option is not considered. In [grid_metrics](#) this option is respected.
- **filter**: Read only the points of interest.

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
col <- height.colors(50)

# Points-to-raster algorithm with a resolution of 1 meter
chm <- grid_canopy(las, res = 1, p2r())
plot(chm, col = col)

# Points-to-raster algorithm with a resolution of 0.5 meters replacing each
# point by a 20-cm radius circle of 8 points
chm <- grid_canopy(las, res = 0.5, p2r(0.2))
plot(chm, col = col)

# Basic triangulation and rasterization of first returns
chm <- grid_canopy(las, res = 0.5, dsmtin())
plot(chm, col = col)

# Khosravipour et al. pitfree algorithm
```



```
chm <- grid_canopy(las, res = 0.5, pitfree(c(0,2,5,10,15), c(0, 1.5)))
plot(chm, col = col)
```

grid_density	<i>Map the pulse or point density</i>
--------------	---------------------------------------

Description

Creates a map of the point density. If a "pulseID" attribute is found, also returns a map of the pulse density.

Usage

```
grid_density(las, res = 4)
```

Arguments

las	An object of class LAS or LAScatalog .
res	numeric. The size of a grid cell in LiDAR data coordinates units. Default is 4 = 16 square meters.

Value

A RasterLayer or a RasterBrick containing a numeric value in each cell. If the RasterLayers are written on disk when running the function with a LAScatalog, a virtual raster mosaic is returned (see [gdalbuildvrt](#))

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog in `grid_*` functions (in bold). For more details see the [LAScatalog engine documentation](#):

- **tiling_size**: How much data is loaded at once. The tiling size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when tiling size is equal to 0 (processing by file).
- **buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **alignment**: Align the processed clusters. The alignment may be slightly modified.
- **chunk size**: How much data is loaded at once. The chunk size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when chunk size is equal to 0 (processing by file).
- **chunk buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks. The alignment may be slightly modified internally to ensure a strict continuous wall-to-wall output.
- **cores**: How many cores are used.
- **progress**: Displays a progress estimate.
- **output_files**: Return the output in R or write each cluster's output in a file. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk size is equal to 0 (processing by file), ORIGINALFILENAME.
- **laz_compression**: is not supported because this function will never write las/laz files.
- **select**: The `grid_*` functions usually 'know' what should be loaded and this option is not considered. In [grid_metrics](#) this option is respected.
- **filter**: Read only the points of interest.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
lidar = readLAS(LASfile)

d = grid_density(lidar, 5)
plot(d)
d = grid_density(lidar, 10)
plot(d)
```

grid_hexametrics

Area-Based Approach in hexagonal cells.

Description

Computes a series of descriptive statistics for a LiDAR dataset within hexagonal cells. This function is identical to [grid_metrics](#) but with hexagonal cells instead of square pixels. After all, we conduct circular plot inventories and we map models on pixel-based maps. `grid_hexametrics` provides the opportunity to test something else. Refer to [grid_metrics](#) for more information.

Usage

```
grid_hexametrics(las, func, res = 20)
```

Arguments

las	An object of class LAS.
func	the function to be applied to each hexagonal cell.
res	numeric. To be consistent with grid_metrics , the square of res give the area of the hexagonal cells, like in grid_metrics . The difference being the fact that for square pixels this is obvious. Here res = 20 gives 400-square-meter hexagonal cells.

Value

A [hexbin](#) object from package [hexbin](#) or a list of [hexbin](#) objects if several metrics are returned.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
lidar = readLAS(LASfile)

col = grDevices::colorRampPalette(c("blue", "cyan2", "yellow", "red"))

# Maximum elevation with a resolution of 8 m
hm = grid_hexametrics(lidar, max(Z), 8)
hexbin::plot(hm, colramp = col, main = "Max Z")

# Mean height with a resolution of 20 m
hm = grid_hexametrics(lidar, mean(Z), 20)
hexbin::plot(hm, colramp = col, main = "Mean Z")

# Define your own new metrics
myMetrics = function(z, i)
{
  metrics = list(
    zwimean = sum(z*i)/sum(i), # Mean elevation weighted by intensities
    zimean = mean(z*i),       # Mean products of z by intensity
    zsqmean = sqrt(mean(z^2)) # Quadratic mean
  )

  return(metrics)
}

metrics = grid_hexametrics(lidar, myMetrics(Z, Intensity), 10)

hexbin::plot(metrics$zwimean, colramp = col, main = "zwimean")
hexbin::plot(metrics$zimean, colramp = col, main = "zimean")
hexbin::plot(metrics$zsqmean, colramp = col, main = "zsqmean")
```

grid_metrics

*Area-Based Approach***Description**

Computes a series of user-defined descriptive statistics for a LiDAR dataset within each pixel of a raster (area-based approach). The grid cell coordinates are pre-determined for a given resolution, so the algorithm will always provide the same coordinates independently of the dataset. When `start = (0,0)` and `res = 20` `grid_metrics` will produce the following cell centers: (10,10), (10,30), (30,10) etc. aligning the corner of a cell on (0,0). When `start = (-10, -10)` and `res = 20` `grid_metrics` will produce the following cell centers: (0,0), (0,20), (20,0) etc. aligning the corner of a cell on (-10, -10).

Usage

```
grid_metrics(las, func, res = 20, start = c(0, 0))
```

Arguments

<code>las</code>	An object of class LAS or LAScatalog .
<code>func</code>	expression. The function to be applied to each cell (see section "Parameter func").
<code>res</code>	numeric. The resolution of the output Raster. Can optionally be a RasterLayer. In that case the RasterLayer is used as the layout.
<code>start</code>	vector x and y coordinates for the reference raster. Default is (0,0) meaning that the grid aligns on (0,0).

Value

A RasterLayer or a RasterBrick containing a numeric value in each cell. If the RasterLayers are written on disk when running the function with a [LAScatalog](#), a virtual raster mosaic is returned (see [gdalbuildvrt](#))

Parameter func

The function to be applied to each cell is a classical function (see examples) that returns a labeled list of metrics. For example, the following function `f` is correctly formed.

```
f = function(x) {list(mean = mean(x), max = max(x))}
```

And could be applied either on the Z coordinates or the intensities. These two statements are valid:

```
grid_metrics(las, f(Z), res = 20)
grid_metrics(las, f(Intensity), res = 20)
```

The following existing functions allow the user to compute some predefined metrics:

- [stdmetrics](#)
- [entropy](#)
- [VCI](#)
- [LAD](#)

But usually users must write their own functions to create metrics. `grid_metrics` will dispatch the point cloud in the user's function.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In `lidR` when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each `lidR` function should come with a section that documents the supported engine options.

The LAScatalog engine supports `.lax` files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a `.lax` files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog in `grid_*` functions (in bold). For more details see the [LAScatalog engine documentation](#):

- **tiling_size**: How much data is loaded at once. The tiling size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when tiling size is equal to 0 (processing by file).
- **buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **alignment**: Align the processed clusters. The alignment may be slightly modified.
- **chunk size**: How much data is loaded at once. The chunk size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when chunk size is equal to 0 (processing by file).
- **chunk buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks. The alignment may be slightly modified internally to ensure a strict continuous wall-to-wall output.
- **cores**: How many cores are used.
- **progress**: Displays a progress estimate.
- **output_files**: Return the output in R or write each cluster's output in a file. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk size is equal to 0 (processing by file), ORIGINALFILENAME.

- `laz_compression`: is not supported because this function will never write las/laz files.
- `select`: The `grid_*` functions usually 'know' what should be loaded and this option is not considered. In [grid_metrics](#) this option is respected.
- `filter`: Read only the points of interest.

Examples

```

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)
colors = height.colors(50)

# Canopy surface model with 4 m^2 cells
metrics = grid_metrics(las, max(Z), 2)
plot(metrics, col = colors)

# Mean height with 400-m^2 cells
metrics = grid_metrics(las, mean(Z), 20)
plot(metrics, col = colors)

# Define your own new metrics
myMetrics = function(z, i)
{
  metrics = list(
    zwimean = sum(z*i)/sum(i), # Mean elevation weighted by intensities
    zimean = mean(z*i), # Mean products of z by intensity
    zsqmean = sqrt(mean(z^2)) # Quadratic mean
  )

  return(metrics)
}

metrics = grid_metrics(las, myMetrics(Z, Intensity))

plot(metrics, col = colors)
plot(metrics, "zwimean", col = colors)
plot(metrics, "zimean", col = colors)
plot(metrics, "zsqmean", col = colors)

```

grid_metrics3d

Voxelize the space and compute metrics for each voxel

Description

This is a 3D version of [grid_metrics](#). It creates a 3D matrix of voxels with a given resolution. It creates a voxel from the cloud of points if there is at least one point in the voxel. For each voxel the function allows computation of one or several derived metrics in the same way as the [grid_metrics](#) functions. The function will dispatch the LiDAR data for each voxel in the user's function (see [grid_metrics](#)).

Usage

```
grid_metrics3d(las, func, res = 1)
```

Arguments

las	An object of class LAS.
func	expression. The function to be applied to each voxel (see also grid_metrics).
res	numeric. The resolution of the voxels. <code>res = 1</code> for a 1x1x1 cubic voxels. Optionally <code>res = c(1,2)</code> for non-cubic voxels (1x1x2 cuboid voxel).

Value

It returns a `data.table` containing the metrics for each voxel. The table has the class `lasmetrics3d` enabling easier plotting.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)

# Cloud of points is voxelized with a 3-meter resolution and in each voxel
# the number of points is computed.
grid_metrics3d(las, length(Z), 3)

# Cloud of points is voxelized with a 3-meter resolution and in each voxel
# the mean scan angle of points is computed.
grid_metrics3d(las, mean(Intensity), 3)

## Not run:
# Define your own metric function
myMetrics = function(i)
{
  ret = list(
    npoints = length(i),
    imean   = mean(i)
  )

  return(ret)
}

voxels = grid_metrics3d(las, myMetrics(Intensity), 3)

plot(voxels, color = "imean")
#etc.

## End(Not run)
```

 grid_terrain

Digital Terrain Model

Description

Interpolates the ground points and creates a rasterized digital terrain model. The algorithm uses the points classified as "ground" (Classification = 2 according to [LAS file format specifications](#)) to compute the interpolation.

How well the edges of the dataset are interpolated depends on the interpolation method used. Thus, a buffer around the region of interest is always recommended to avoid edge effects.

Usage

```
grid_terrain(las, res = 1, algorithm, keep_lowest = FALSE)
```

Arguments

las	An object of class LAS or LAScatalog .
res	numeric. The resolution of the output Raster. Can optionally be a RasterLayer. In that case the RasterLayer is used as the layout.
algorithm	function. A function that implements an algorithm to compute spatial interpolation. lidR implements knnidw , tin , and kriging (see respective documentation and examples).
keep_lowest	logical. This option forces the original lowest ground point of each cell (if it exists) to be chosen instead of the interpolated values.

Value

A RasterLayer containing a numeric value in each cell. If the RasterLayers are written on disk when running the function with a LAScatalog, a virtual raster mosaic is returned (see [gdalbuildvrt](#))

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports `.lax` files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a `.lax` files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog in grid_* functions (in bold). For more details see the [LAScatalog engine documentation](#):

- **tiling_size**: How much data is loaded at once. The tiling size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when tiling size is equal to 0 (processing by file).
- **buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **alignment**: Align the processed clusters. The alignment may be slightly modified.
- **chunk size**: How much data is loaded at once. The chunk size may be slightly modified internally to ensure a strict continuous wall-to-wall output even when chunk size is equal to 0 (processing by file).
- **chunk buffer**: This function guarantees a strict continuous wall-to-wall output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks. The alignment may be slightly modified internally to ensure a strict continuous wall-to-wall output.
- **cores**: How many cores are used.
- **progress**: Displays a progress estimate.
- **output_files**: Return the output in R or write each cluster's output in a file. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk size is equal to 0 (processing by file), ORIGINALFILENAME.
- **laz_compression**: is not supported because this function will never write las/laz files.
- **select**: The grid_* functions usually 'know' what should be loaded and this option is not considered. In [grid_metrics](#) this option is respected.
- **filter**: Read only the points of interest.

See Also

[lasnormalize](#)

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)
plot(las)

dtm1 = grid_terrain(las, algorithm = knnidw(k = 6L, p = 2))
dtm2 = grid_terrain(las, algorithm = tin())
dtm3 = grid_terrain(las, algorithm = kriging(k = 10L))

## Not run:
plot(dtm1)
plot(dtm2)
plot(dtm3)
```

```

plot_dtm3d(dtm1)
plot_dtm3d(dtm2)
plot_dtm3d(dtm3)

## End(Not run)

```

highest

Point Cloud Decimation Algorithm

Description

This function is made to be used in [lasfilterdecimate](#). It implements an algorithm that creates a grid with a given resolution and filters the point cloud by selecting the highest point within each cell.

Usage

```
highest(res = 1)
```

Arguments

res numeric. The resolution of the grid used to filter the point cloud

See Also

Other point cloud decimation algorithms: [homogenize](#), [random](#)

Examples

```

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile, select = "xyz")

# Select the highest point within each cell of an overlaid grid
thinned = lasfilterdecimate(las, highest(4))
plot(thinned)

```

homogenize

Point Cloud Decimation Algorithm

Description

This function is made to be used in [lasfilterdecimate](#). It implements an algorithm that creates a grid with a given resolution and filters the point cloud by randomly selecting some points in each cell. It is designed to produce point clouds that have uniform densities throughout the coverage area. For each cell, the proportion of points or pulses that will be retained is computed using the actual local density and the desired density. If the desired density is greater than the actual density it returns an unchanged set of points (it cannot increase the density). The cell size must be large enough to compute a coherent local density. For example in a 2 points/m² point cloud, 25 square meters would be feasible; however 1 square meter cells would not be feasible because density does not have meaning at this scale.

Usage

```
homogenize(density, res = 5, use_pulse = FALSE)
```

Arguments

density numeric. The desired output density.

res numeric. The resolution of the grid used to filter the point cloud

use_pulse logical. Decimate by removing random pulses instead of random points (requires running [laspulse](#) first)

See Also

Other point cloud decimation algorithms: [highest](#), [random](#)

Examples

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

# Select points randomly to reach an homogeneous density of 1
thinned = lasfilterdecimate(las, homogenize(1,5))
plot(grid_density(thinned))
```

<code>is.empty</code>	<i>Test if a LAS object is empty</i>
-----------------------	--------------------------------------

Description

An empty LAS object is a point cloud with 0 points

Usage

```
is.empty(object, ...)
```

```
## S4 method for signature 'LAS'
is.empty(object, ...)
```

Arguments

object A LAS object

... Unused

Value

TRUE or FALSE

Examples

```
LASfile <- system.file("extdata", "example.laz", package="rlas")
las = readLAS(LASfile)
is.empty(las)

las = new("LAS")
is.empty(las)
```

knnidw

Spatial Interpolation Algorithm

Description

This function is made to be used in [grid_terrain](#) or [lasnormalize](#). It implements an algorithm for spatial interpolation. Interpolation is done using a k-nearest neighbour (KNN) approach with an inverse-distance weighting (IDW).

Usage

```
knnidw(k = 10, p = 2)
```

Arguments

k	numeric. Number of k-nearest neighbours. Default 10.
p	numeric. Power for inverse-distance weighting. Default 2.

See Also

Other spatial interpolation algorithms: [kriging](#), [tin](#)

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)

# plot(las)

dtm = grid_terrain(las, algorithm = knnidw(k = 6L, p = 2))

plot(dtm, col = terrain.colors(50))
plot_dtm3d(dtm)
```

kriging	<i>Spatial Interpolation Algorithm</i>
---------	--

Description

This function is made to be used in [grid_terrain](#) or [lasground](#). It implements an algorithm for spatial interpolation. Spatial interpolation is based on universal kriging using the [krige](#) function from `gstat`. This method combines the KNN approach with the kriging approach. For each point of interest it kriges the terrain using the k-nearest neighbour ground points. This method is more difficult to manipulate but it is also the most advanced method for interpolating spatial data.

Usage

```
kriging(model = gstat::vgm(0.59, "Sph", 874), k = 10L)
```

Arguments

model	A variogram model computed with vgm . If NULL it performs an ordinary or weighted least squares prediction.
k	numeric. Number of k-nearest neighbours. Default 10.

See Also

Other spatial interpolation algorithms: [knnidw](#), [tin](#)

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)

# plot(las)

dtm = grid_terrain(las, algorithm = kriging())

plot(dtm, col = terrain.colors(50))
plot_dtm3d(dtm)
```

LAD	<i>Leaf area density</i>
-----	--------------------------

Description

Computes a leaf area density profile based on the method of Bouvier et al. (see reference)

Usage

```
LAD(z, dz = 1, k = 0.5, z0 = 2)
```

Arguments

<code>z</code>	vector of positive <code>z</code> coordinates
<code>dz</code>	numeric. The thickness of the layers used (height bin)
<code>k</code>	numeric. is the extinction coefficient
<code>z0</code>	numeric. The bottom limit of the profile

Details

The function assesses the number of laser points that actually reached the layer `z+dz` and those that passed through the layer `[z, z+dz]` (see [gap_fraction_profile](#)). Then it computes the log of this quantity and divides it by the extinction coefficient `k` as described in Bouvier et al. By definition the layer 0 will always return infinity because no returns pass through the ground. Therefore, the layer 0 is removed from the returned results.

Value

A data.frame containing the bin elevations (`z`) and leaf area density for each bin (`lad`)

References

Bouvier, M., Durrieu, S., Fournier, R. a, & Renaud, J. (2015). Generalizing predictive models of forest inventory attributes using an area-based approach with airborne las data. Remote Sensing of Environment, 156, 322-334. <http://doi.org/10.1016/j.rse.2014.10.004>

See Also

[gap_fraction_profile](#)

Examples

```
z = c(rnorm(1e4, 25, 6), rgamma(1e3, 1, 8)*6, rgamma(5e2, 5,5)*10)
z = z[z<45 & z>0]

lad = LAD(z)

plot(lad, type="l", xlab="Elevation", ylab="Leaf area density")
```

LAS-class

An S4 class to represent a .las or .laz file

Description

Class LAS is the representation of a las/laz file according to the [LAS file format specifications](#).

Usage

```
LAS(data, header = list(), proj4string = sp::CRS(), check = TRUE)
```

Arguments

data	a data.table containing the data of a las or laz file.
header	a list or a LASheader containing the header of a las or laz file.
proj4string	projection string of class CRS-class .
check	logical. Conformity tests while building the object.

Details

A LAS object inherits a [Spatial](#) object from `sp`. Thus it is a `Spatial` object plus a `data.table` with the data read from a las/laz file and a [LASheader](#) (see the ASPRS documentation for the [LAS file format](#) for more information). Because las files are standardized the table of attributes read from the las/laz file is also standardized. Columns are named:

- X (numeric)
- Y (numeric)
- Z (numeric)
- Intensity (integer)
- ReturnNumber (integer)
- NumberOfReturns (integer)
- ScanDirectionFlag (integer)
- EdgeOfFlightline (integer)
- Classification (integer)
- Synthetic_flag (logical)
- Keypoint_flag (logical)
- Withheld_flag (logical)
- ScanAngle (integer)
- UserData (integer)
- PointSourceID (integer)

Value

An object of class LAS

Functions

- LAS: Create objects of class LAS

Slots

bbox Object of class `matrix`, with bounding box

proj4string Object of class [CRS](#), projection string

data Object of class [data.table](#). Point cloud data according to the [LAS file format](#)

header Object of class [LASheader](#). las file header according to the [LAS file format](#)

Extends

Class [Spatial](#), directly.

See Also

[readLAS](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)
```

lasaddattribute	<i>Add attributes into a LAS object</i>
-----------------	---

Description

A [LAS](#) object represents a .las file in R. According to the [LAS specifications](#) a las file contains a core of defined attributes, such as XYZ coordinates, intensity, return number, and so on for each point. It is possible to add supplementary attributes. The functions `lasadd*` enable the user to add new attributes (see details).

Usage

```
lasadddata(las, x, name)
```

```
lasaddextrabytes(las, x, name, desc)
```

```
lasaddextrabytes_manual(las, x, name, desc, type, offset = NULL,
  scale = NULL, NA_value = NULL)
```

```
lasremoveextrabytes(las, name)
```

Arguments

<code>las</code>	An object of class LAS
<code>x</code>	a vector that needs to be added in the LAS object. For <code>lasaddextrabytes*</code> it can be missing (see details).
<code>name</code>	character. The name of the extra bytes attribute to add in the file.
<code>desc</code>	character. A short description of the extra bytes attribute to add in the file (32 characters).
<code>type</code>	character. The data type of the extra bytes attribute. Can be "uchar", "char", "ushort", "short", "u
<code>scale, offset</code>	numeric. The scale and offset of the data. NULL if not relevant.
<code>NA_value</code>	numeric or integer. NA is not a valid value in a las file. At time of writing it will be replaced by this value that will be considered as NA. NULL if not relevant.

Details

Users cannot assign names that are the same as the names of the core attributes. These functions are dedicated to adding data not part of the LAS specification. For example, `lasaddextrabytes(las, x, "R")` will fail because R is a name reserved for the red channel of las file that contains RGB attributes.

`lasadddata` simply adds a new column in the data but does not update the header. Thus the LAS object is not strictly valid. These data will be temporarily usable at the R level but will not be written in a las file with `writeLAS`.

`lasaddextrabytes` does the same as `lasadddata` but automatically updates the header of the LAS object. Thus, the LAS object is valid and the new data is considered as "extra bytes". This new data will be written in a las file with `writeLAS`.

`lasaddextrabytes_manual` allows the user to manually write all the extra bytes metadata. This function is reserved for experienced users with a good knowledge of the LAS specifications. The function does not perform tests to check the validity of the information.

When using `lasaddextrabytes` and `lasaddextrabytes_manual`, `x` can only be of type numeric (integer or double). It cannot be of types character or logical as these are not supported by the las specifications. The types that are supported in lidR are types 0 to 10 (table 24 page 25 of the specification). Types greater than 10 are not supported.

Value

An object of class `LAS`

Examples

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

print(las)
print(las@header)

x <- 1:30

las <- lasadddata(las, x, "mydata")
print(las)          # The las object has a new attribute called "mydata"
print(las@header)  # But the header has not been updated. This new data will not be written

las <- lasaddextrabytes(las, x, "mydata2", "A new data")
print(las)          # The las object has a new attribute called "mydata2"
print(las@header)  # The header has been updated. This new data will be written

# Optionally if the data is already in the LAS object you can update the header skipping the
# parameter x
las <- lasaddextrabytes(las, name = "mydata", desc = "Amplitude")
print(las@header)

# Remove an extra bytes attribute
```

```
las <- lasremoveextrabytes(las, "mydata2")
print(las)
print(las@header)

las <- lasremoveextrabytes(las, "mydata")
print(las)
print(las@header)
```

LAScatalog-class*An S4 class to represent a catalog of .las or .laz files*

Description

A LAScatalog object is a representation of a set of las/laz files. A LAScatalog is a way to manage and process an entire dataset. It allows the user to process a large area, or to selectively clip data from a large area without loading all the data into computer memory. A LAScatalog can be built with the function `catalog` and is formally an extension of a `SpatialPolygonsDataFrame` containing extra data to allow users greater control over how the dataset is processed (see details).

Details

A LAScatalog is formally a `SpatialPolygonsDataFrame` extended with new slots that contain processing options. In `lidR`, each function that supports a LAScatalog as input will respect these processing options. Internally, processing a catalog is almost always the same and relies on just a few steps:

1. Define chunks. A chunk is an arbitrarily-defined region of interest (ROI) of the catalog. Altogether, the chunks are a wall-to-wall set of ROIs that encompass the whole dataset.
2. Loop over each chunk (in parallel or not).
3. For each chunk, load the points inside the ROI into R, run some R functions, return the expected output.
4. Merge the outputs of the different chunks once they are all processed to build a continuous (wall-to-wall) output.

So basically, a LAScatalog is an object that allows for batch processing but with the specificity that `lidR` does not loop through las files, but loops seamlessly through chunks that do not necessarily match with the file pattern. This way `lidR` can sequentially process tiny ROIs even if each file may be individually too big to fit in memory. This is also why point cloud indexation with `laz` files may significantly speed-up the processing.

It is important to note that catalogs with files that overlap each other are not natively supported by `lidR`. When encountering such datasets the user should always filter any overlaps if possible. This is possible if the overlapping points are flagged, for example in the 'withheld' attribute. Otherwise `lidR` will not be able to process the dataset correctly.

Slots

`processing_options` list. A list that contains some settings describing how the catalog will be processed (see dedicated section).

`chunk_options` list. A list that contains some settings describing how the catalog will be subdivided into chunks to be processed (see dedicated section).

`output_options` list. A list that contains some settings describing how the catalog will return the outputs (see dedicated section).

`input_options` list. A list of parameters to pass to [readLAS](#) (see dedicated section).

Processing options

The slot `@processing_options` contains a list of options that determine how chunks (the sub-areas that are sequentially processed) are processed.

- **core**: integer. The number of cores used. Default is 1. In lidR algorithms are all single core algorithms. What is parallelized is the number of chunks processed together. Thus by using 4 cores, 4 chunks of point cloud are loaded at once using 4 times more memory. Thus it is not always pertinent to set `core > 1`.
- **progress**: boolean. Display a progress bar and a chart of progress. Default is TRUE. Progress estimation can be enhanced by installing the package `progress`.
- **stop_early**: boolean. Stop the processing if an error occurs in a chunk. If FALSE the process can run until the end removing chunks that failed. Default is TRUE and the user should have no reason to change this.
- **wall.to.wall** logical. The catalog processing engine always guarantees to return a continuous output without edge effects, assuming that the catalog is a wall-to-wall catalog. To do so, some options are checked internally to guard against bad settings, such as `buffer = 0` for an algorithm that requires a buffer. In rare cases it might be useful to disable these controls. If `wall.to.wall = FALSE` controls are disabled and wall-to-wall outputs cannot be guaranteed.

Chunk options

The slot `@clustering_options` contains a list of options that determine how chunks (the sub-areas that are sequentially processed) are made.

- **chunk_size**: numeric. The size of the chunks that will be sequentially processed. A small size allows small amounts of data to be loaded at once, saving computer memory, and vice versa. The computation is usually faster but uses much more memory. If `chunk_size = 0` the catalog is processed sequentially *by file* i.e. a chunk is a file. Default is 0 i.e. by default the processing engine respects the existing tiling pattern.
- **buffer**: numeric. Each chunk can be read with an extra buffer around it to ensure there is no edge effect between two independent chunks and that the output is continuous. This is mandatory for some algorithms. Default is 30.
- **alignment**: numeric. A vector of size 2 (x and y coordinates, respectively) to align the chunk pattern. By default the alignment is made along (0,0), meaning that the edge of the first chunk will belong on `x = 0` and `y = 0` and all the other chunks will be multiples of the chunk size. Not relevant if `chunk_size = 0`.

Output options

The slot @output_options contains a list of options that determine how clusters (the sub-areas that are sequentially processed) are written. By "written" we mean written to files or written in R memory.

- **output_files**: string. If output_files = "" outputs are returned in R. Otherwise, if output_files is a string the outputs will be written to files. This is useful if the output is too big to be returned in R. A path to a templated filename without extension (the engine guesses it for you) is expected. When several files are going to be written a single string is provided with a template that is automatically filled. For example, the following file names are possible:

```
"/home/user/als/normalized/file_{ID}_segmented"
"C:/user/document/als/zone52_{XLEFT}_{YBOTTOM}_confidential"
"C:/user/document/als/{ORIGINALFILENAME}_normalized"
```

This option will generate as many filenames as needed with custom names for each file. The list of allowed templates is described in the documentation for each function.

- **drivers**: list. This contains all the drivers required to seamlessly write Raster*, Spatial*, LAS objects. It is recommended that only advanced users change this option.

Input options

The slot @input_options contains a list of options that are passed to the function [readLAS](#). Indeed, the readLAS function is not called directly by the user but by the internal processing engine. Users can propagate these options through the LAScatalog settings.

- **select**: string. The select option. Usually this option is not respected because each function knows which data must be loaded or not. This is documented in each function.
- **filter**: string. The filter option.

Examples

```
## Not run:
# Build a catalog
ctg <- catalog("filder/to/las/files/")

# Set some options
opt_cores(ctg) <- 2
opt_filter(ctg) <- "-keep_first"

# Summary gives a summary of how the catalog will be processed
summary(ctg)

# We can seamlessly use lidR functions
hmean <- grid_metrics(ctg, mean(Z), 20)
ttops <- tree_detection(ctg, lmf(5))

# For low memory config it is probably advisable not to load entire files
# and process chunks instead
opt_cores(ctg) <- 1
opt_chunk_size(ctg) <- 500
```

```

# Outputs are expected to be strictly identical
hmean <- grid_metrics(ctg, mean(Z), 20)
ttops <- tree_detection(ctg, lmf(5))

# Sometimes the output is likely to be very large
dtm <- grid_terrain(ctg, 1, tin())

# In that case it is advisable to write the output(s) to files
opt_output_files(ctg) <- "path/to/folder/DTM_chunk_{XLEFT}_{YBOTTOM}"

# Raster will be written to disk. The list of written files is returned
# or, in this specific case, a virtual raster mosaic.
dtm <- grid_terrain(ctg, 1, tin())

# When chunks are files the original names of the las files can be preserved
opt_chunk_size(ctg) <- 0
opt_output_files(ctg) <- "path/to/folder/DTM_{ORIGINALFILENAME}"
dtm <- grid_terrain(ctg, 1, tin())

# For some functions, files MUST be written to disk. Indeed, it is certain that R cannot
# handle the entire output.
opt_chunk_size(ctg) <- 0
opt_output_files(ctg) <- "path/to/folder/{ORIGINALFILENAME}_norm"
opt_laz_compression(ctg) <- TRUE
new_ctg <- lasnormalize(ctg, tin())

# The user has access to the catalog engine through the function catalog_apply
output <- catalog_apply(ctg, FUN, ...)

## End(Not run)

```

lascheck

Inspect a LAS object

Description

Performs a deep inspection of a LAS or LAScatalog object and prints a report.

For a LAS object it checks:

- if the point cloud is valid according to las specification
- if the header is valid according to las specification
- if the point cloud is in accordance with the header
- if the point cloud has duplicated points and degenerated ground points
- if the coordinate reference system is correctly recorded
- if some pre-processing, such as normalization or ground filtering, is already done.

For a LAScatalog object it checks:

- if the headers are consistent across files
- if the files are overlapping
- if some pre-processing, such as normalization, is already done.

For the pre-processing tests the function only makes an estimation and may not be correct.

Usage

```
lascheck(las)
```

Arguments

las An object of class [LAS](#) or [LAScatalog](#).

lasclip *Clip LiDAR points*

Description

Clip LiDAR points within a given geometry from a point cloud (LAS object) or a catalog (LAScatalog object). With a LAS object, the user first reads and loads a point-cloud into memory and then can clip it to get a subset within a region of interest (ROI). With a LAScatalog object, the user can extract any arbitrary ROI for a set of las/laz files, loading only the points of interest. This is faster, easier and much more memory-efficient for extracting ROIs.

Usage

```
lasclip(las, geometry, ...)
```

```
lasclipRectangle(las, xleft, ybottom, xright, ytop)
```

```
lasclipPolygon(las, xpoly, ypoly)
```

```
lasclipCircle(las, xcenter, ycenter, radius)
```

Arguments

las An object of class [LAS](#) or [LAScatalog](#).

geometry a geometric object. Many types are supported, see section 'supported geometries'.

... optional supplementary options (see supported geometries)

xleft numeric. left x coordinates of rectangles.

ybottom numeric. bottom y coordinates of rectangles.

xright numeric. right x coordinates of rectangles.

ytop numeric. top y coordinates of rectangles.

xpoly	numeric. x coordinates of a polygon.
ypoly	numeric. y coordinates of a polygon.
xcenter	numeric. x coordinates of discs centers.
ycenter	numeric. y coordinates of discs centers.
radius	numeric. disc radius or radii.

Value

If the input is a LAS object: an object of class LAS, or a list of LAS objects if the query implies several regions of interest will be returned.

If the input is a LAScatalog object: an object of class LAS, or a list of LAS objects if the query implies several regions of interest will be returned, or a LAScatalog if the queries are immediately written into files without loading anything in R.

Supported geometries

- **WKT string**: describing a POINT, a POLYGON or a MULTIPOLYGON.
- [Polygon](#) or [Polygons](#)
- [SpatialPolygons](#) or [SpatialPolygonsDataFrame](#)
- [SpatialPoints](#) or [SpatialPointsDataFrame](#) in that case a parameter 'radius' must be passed in '...'
- [SimpleFeature](#) that consistently contains POINT or POLYGON/MULTIPOLYGON. In case of POINT a parameter 'radius' must be passed in '...'
- [Extent](#)
- [matrix](#) 2 x 2 describing a bounding box following this order:

```

min    max
x 684816 684943
y 5017823 5017957

```

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- `chunk_size`: Does not make sense here.
- `buffer`: Not supported yet.
- `alignment`: Does not makes sense here.
- **`cores`**: How many cores are used.
- **`progress`**: Displays a progress estimation.
- **`stop_early`**: Leave this 'as-is' unless you are an advanced user.
- **`output_files`**: If 'output_files' is set in the catalog, the ROIs will not be returned in R. They will be written immediately in files. See [LAScatalog-class](#) and examples. The allowed templates in lasclip are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {ID}, {XCENTER}, {YCENTER} or any names from the table of attributes of a spatial object given as input such as {PLOT_ID} or {YEAR}, for example, if these attributes exist. If empty everything is returned into R.
- **`laz_compression`**: write las or laz files
- **`drivers`**: Leave this 'as-is' unless you are an advanced user.
- `select`: The function will write files equivalent to the originals. This option is not respected.
- **`filter`**: Read only the points of interest.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")

# Load the file and clip the region of interest
las = readLAS(LASfile)
subset1 = lasclipRectangle(las, 684850, 5017850, 684900, 5017900)

# Do not load the file(s), extract only the region of interest from a bigger dataset
ctg = catalog(LASfile)
subset2 = lasclipRectangle(ctg, 684850, 5017850, 684900, 5017900)

# Extract all the polygons from a shapefile
shapefile_dir <- system.file("extdata", package = "lidR")
lakes = shapefile(paste0(shapefile_dir, "/lake_polygons_UTM17.shp"))
subset3 = lasclip(ctg, lakes)

# Extract the polygons, write them in files named after the lake names, do not load anything in R
opt_output_files(ctg) <- paste0(tempfile(), "_{LAKENAME_1}")
new_ctg = lasclip(ctg, lakes)

## Not run:
plot(subset1)
plot(subset2)
plot(subset3)

## End(Not run)
```

lasfilter	<i>Return points with matching conditions</i>
-----------	---

Description

Return points with matching conditions

Usage

```
lasfilter(las, ...)
```

Arguments

las	An object of class LAS
...	Logical predicates. Multiple conditions are combined with '&' or ','

Value

An object of class [LAS](#)

See Also

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilters](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
lidar = readLAS(LASfile)

# Select the first returns classified as ground
firstground = lasfilter(lidar, Classification == 1 & ReturnNumber == 1)

# Multiple arguments are equivalent to &
firstground = lasfilter(lidar, Classification == 1, ReturnNumber == 1)

# Multiple criteria
first_or_ground = lasfilter(lidar, Classification == 1 | ReturnNumber == 1)
```

lasfilterdecimate *Decimate a LAS object*

Description

Reduce the number of points using several possible algorithms.

Usage

```
lasfilterdecimate(las, algorithm)
```

Arguments

las	An object of class LAS or LAScatalog .
algorithm	function. An algorithm of point decimation. lidR have: random , homogenize and highest .

Value

If the input is a LAS object, returns a LAS object. If the input is a LAScatalog, returns a LAScatalog.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports `.lax` files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a `.lax` files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- chunk buffer: This function guarantee a strict wall-to-wall continuous output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.

- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk is size equal to 0 (processing by file), ORIGINALFILENAME.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*"` and cannot be changed.
- **filter**: Read only points of interest.

Examples

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

# Select points randomly to reach an overall density of 1
thinned1 = lasfilterdecimate(las, random(1))
plot(grid_density(las))
plot(grid_density(thinned1))

# Select points randomly to reach an homogeneous density of 1
thinned2 = lasfilterdecimate(las, homogenize(1,5))
plot(grid_density(thinned2))

# Select the highest point within each pixel of an overlaid grid
thinned3 = lasfilterdecimate(las, highest(5))
plot(thinned3)
```

lasfilterduplicates *Filter duplicated points*

Description

Filter points that appear more than once in the point cloud according to their X Y Z coordinates

Usage

```
lasfilterduplicates(las)
```

Arguments

las An object of class [LAS](#) or [LAScatalog](#).

Value

If the input is a LAS object, returns a LAS object. If the input is a LAScatalog, returns a LAScatalog.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer**: This function guarantee a strict wall-to-wall continuous output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.
- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk is size equal to 0 (processing by file), ORIGINALFILENAME.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*"` and cannot be changed.
- **filter**: Read only points of interest.

See Also

Other lasfilters: [lasfiltersurfacepoints](#), [lasfilters](#), [lasfilter](#)

lasfilters

Predefined filters

Description

Select only some returns

Usage

lasfilterfirst(las)

lasfilterfirstlast(las)

lasfilterfirstofmany(las)

lasfilterground(las)

lasfilterlast(las)

lasfilternth(las, n)

lasfiltersingle(las)

lasfilterfirstofmany(las)

Arguments

las	An object of class LAS
n	the position in the return sequence

Details

- lasfilterfirst Select only the first returns.
- lasfilterfirstlast Select only the first and last returns.
- lasfilterground Select only the returns classified as ground according to LAS specification.
- lasfilterlast Select only the last returns i.e. the last returns and the single returns.
- lasfilternth Select the returns from their position in the return sequence.
- lasfilterfirstofmany Select only the first returns from pulses which returned multiple points.
- lasfiltersingle Select only the returns that return only one point.

Value

An object of class [LAS](#)

See Also

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Other lasfilters: [lasfilterduplicates](#), [lasfiltersurfacepoints](#), [lasfilter](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
lidar = readLAS(LASfile)
```

```
firstReturns = lasfilterfirst(lidar)
groundReturns = lasfilterground(lidar)
```

lasfiltersurfacepoints

Filter the surface points

Description

This function is superseded by the algorithm [highest](#) usable in [lasfilterdecimate](#)

Usage

```
lasfiltersurfacepoints(las, res)
```

Arguments

las	An object of class LAS or LAScatalog .
res	numeric. The resolution of the grid used to filter the point cloud

Value

If the input is a LAS object, returns a LAS object. If the input is a LAScatalog, returns a LAScatalog.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- chunk buffer: This function guarantee a strict wall-to-wall continuous output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.
- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are XLEFT, XRIGHT, YBOTTOM, YTOP, XCENTER, YCENTER ID and, if chunk is size equal to 0 (processing by file), ORIGINALFILENAME.

- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*"` and cannot be changed.
- **filter**: Read only points of interest.

See Also

Other lasfilters: [lasfilterduplicates](#), [lasfilters](#), [lasfilter](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)
subset = lasfiltersurfacepoints(las, 2)
plot(subset)
```

lasground

Classify points as 'ground' or 'not ground'

Description

Classify points as 'ground' or 'not ground' with several possible algorithms. The function updates the attribute `Classification` of the LAS object. The points classified as 'ground' are assigned a value of 2 according to [las specifications](#).

Usage

```
lasground(las, algorithm, last_returns = TRUE)
```

Arguments

<code>las</code>	An object of class LAS or LAScatalog .
<code>algorithm</code>	a ground-segmentation function. <code>lidR</code> has: pmf and csf .
<code>last_returns</code>	logical. The algorithm will use only the last returns (including the first returns in cases of a single return) to run the algorithm. If <code>FALSE</code> all the returns are used. If the attribute 'ReturnNumber' or 'NumberOfReturns' are absent, 'last_returns' is turned to <code>FALSE</code> automatically.

Value

If the input is a LAS object, return a LAS object. If the input is a LAScatalog, returns a LAScatalog.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.
- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {XCENTER}, {YCENTER} {ID} and, if chunk size is equal to 0 (processing by file), {ORIGINALFILENAME}.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*"` and cannot be changed.
- **filter**: Read only points of interest.

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn")

# Using the Progressive Morphological Filter
# -----

ws <- seq(3,12, 3)
th <- seq(0.1, 1.5, length.out = length(ws))

las <- lasground(las, pmf(ws, th))
plot(las, color = "Classification")
```



```
# Using the Cloth Simulation Filter
# -----

las <- lasground(las, csf())
```

LASheader *Create a LASheader object*

Description

Create a LASheader object

Usage

```
LASheader(data = list())
```

Arguments

data a list containing the data from the header of a las file.

Value

An object of class LASheader

LASheader-class *An S4 class to represent the header of .las or .laz files*

Description

An S4 class to represent the header of .las or .laz files according to the [LAS file format specifications](#). A LASheader object contains a list in the slot @PHB with the data read from the Public Header Block and list in the slot @VLR with the data read from the Variable Length Records

Slots

PHB list. Represents the Public Header Block

VLR list. Represents the Variable Length Records

lasmergespatial *Merge a point cloud with a source of spatial data*

Description

Merge a point cloud with a source of spatial data. It adds an attribute along each point based on a value found in the spatial data. Sources of spatial data can be a `SpatialPolygonsDataFrame` or a `RasterLayer`.

- `SpatialPolygonsDataFrame`: it checks if the points belongs within each polygon. If the parameter `attribute` is the name of an attribute in the table of attributes of the shapefile, it assigns to the points the values of that attribute. Otherwise it classifies the points as boolean. TRUE if the points are in a polygon, FALSE otherwise.
- `RasterLayer`: it attributes to each point the value found in each pixel of the `RasterLayer`.
- `RasterStack` or `RasterBrick` must have 3 channels for RGB colors. It colorizes the point cloud with RGB values.

Usage

```
lasmergespatial(las, source, attribute = NULL)
```

Arguments

<code>las</code>	An object of class LAS
<code>source</code>	An object of class <code>SpatialPolygonsDataFrame</code> or <code>RasterLayer</code> or a <code>RasterStack</code> or <code>RasterBrick</code> with RGB colors.
<code>attribute</code>	character. The name of an attribute in the table of attributes of the shapefile or the name of a new column in the LAS object. Not relevant for RGB colorization.

Value

An object of the class LAS.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
shp      <- system.file("extdata", "lake_polygons_UTM17.shp", package = "lidR")

las  <- readLAS(LASfile)
lakes <- shapefile(shp)

# The attribute "inlake" does not exist in the shapefile.
# Points are classified as TRUE if in a polygon
las  <- lasmergespatial(las, lakes, "inlakes")    # New attribute 'inlakes' is added.
forest <- lasfilter(las, inlakes == FALSE)
plot(las)
```

```
plot(forest)

# The attribute "LAKENAME_1" exists in the shapefile.
# Points are classified with the values of the polygons
las <- lasmergespatial(las, lakes, "LAKENAME_1")    # New column 'LAKENAME_1' is added.
```

lasmetrics

Compute metrics for a cloud of points

Description

lasmetrics computes a series of user-defined descriptive statistics for a LiDAR dataset. See [grid_metrics](#) to compute metrics on a grid. Basically there are no predefined metrics. Users must write their own functions to create metrics (see example). The following existing functions can serve as a guide to help users compute their own metrics:

- [stdmetrics](#)
- [entropy](#)
- [VCI](#)
- [LAD](#)

Usage

```
lasmetrics(obj, func)
```

Arguments

obj	An object of class LAS
func	The function to be applied to a cloud of points. Function must return a list (see example)

Value

It returns a list containing the metrics

See Also

[grid_metrics](#) [stdmetrics](#) [entropy](#) [VCI](#) [LAD](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
lidar = readLAS(LASfile)

lasmetrics(lidar, max(Z))
lasmetrics(lidar, mean(Intensity))

# Define your own new metrics
```

```

myMetrics = function(z, i)
{
  metrics = list(
    zwimean = sum(z*i)/sum(i), # Mean elevation weighted by intensities
    zimean = mean(z*i),      # Mean products of z by intensity
    zsqmean = sqrt(mean(z^2)) # Quadratic mean
  )

  return(metrics)
}

metrics = lasmetrics(lidar, myMetrics(Z, Intensity))

# Predefined metrics
lasmetrics(lidar, .stdmetrics)

```

 lasnormalize

Remove the topography from a point cloud

Description

Subtract digital terrain model (DTM) from LiDAR point cloud to create a dataset normalized with the ground at 0. The DTM can originate from an external file or can be computed by the user. It can also be computed on-the-fly. In this case the algorithm does not use rasterized data and each point is interpolated. There is no inaccuracy due to the discretization of the terrain and the resolution of the terrain is virtually infinite.

How well the edges of the dataset are interpolated depends on the interpolation method used. Thus, a buffer around the region of interest is always recommended to avoid edge effects.

The attribute Z of the returned LAS object is the normalized elevation. A new attribute 'Zref' records the former elevation values, which enables the use of lasunnormalize to restore original point elevations.

Usage

```

lasnormalize(las, algorithm, na.rm = FALSE)

lasunnormalize(las)

## S4 method for signature 'LAS,RasterLayer'
e1 - e2

## S4 method for signature 'LAS,function`'
e1 - e2

```

Arguments

las	An object of class LAS or LAScatalog .
algorithm	a spatial interpolation function. <code>lidR</code> has tin , kriging , knnidw or a RasterLayer representing a digital terrain model (can be computed with grid_terrain)
na.rm	logical. When using a RasterLayer as DTM, by default the function fails if a point fall in an empty pixel because a Z elevation cannot be NA. If <code>na.rm = TRUE</code> points with an elvation of NA are filtered. Be careful this creates a copy of the point cloud.
e1	a LAS object
e2	RasterLayer representing a digital terrain model (can be computed with grid_terrain) or a spatial interpolation function. <code>lidR</code> has tin , kriging , and knnidw .

Value

If the input is a LAS object, return a LAS object. If the input is a [LAScatalog](#), returns a [LAScatalog](#).

Working with a LAScatalog

This section appears in each function that supports a [LAScatalog](#) as input.

In `lidR` when the input of a function is a [LAScatalog](#) the function uses the [LAScatalog](#) processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing [LAScatalogs](#). Each `lidR` function should come with a section that documents the supported engine options.

The [LAScatalog](#) engine supports `.lax` files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a `.lax` files, but this is not mandatory.

Supported processing options

Supported processing options for a [LAScatalog](#) (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.
- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are `{XLEFT}`, `{XRIGHT}`, `{YBOTTOM}`, `{YTOP}`, `{XCENTER}`, `{YCENTER}` `{ID}` and, if chunk size is equal to 0 (processing by file), `{ORIGINALFILENAME}`.

- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*" and cannot be changed.`
- **filter**: Read only points of interest.

See Also

[raster grid_terrain](#)

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile)

plot(las)

# First option: use a RasterLayer as DTM
# =====

dtm <- grid_terrain(las, 1, kriging(k = 10L))
las <- lasnormalize(las, dtm)

plot(dtm)
plot(las)

# restore original elevations
las <- lasunnormalize(las)
plot(las)

# operator - can be used. This is equivalent to the previous
las <- las - dtm
plot(las)

# restore original elevations
las <- lasunnormalize(las)

# Second option: interpolate each point (no discretization)
# =====

las <- lasnormalize(las, tin())
plot(las)

# operator - can be used. This is equivalent to the previous
las <- lasunnormalize(las)
las <- las - tin()

## Not run:
# All the following syntaxes are correct
las <- lasnormalize(las, knnidw())
las <- lasnormalize(las, knnidw(k = 8, p = 2))
las <- las - knnidw()
las <- las - knnidw(k = 8)
```

```
las <- lasnormalize(las, kriging())
las <- las - kriging(k = 8)

## End(Not run)
```

laspulse	<i>Retrieve individual pulses, flightlines or scanlines</i>
----------	---

Description

Retrieve each individual pulse, individual flightline or individual scanline and assigns a number to each point.

Usage

```
laspulse(las)

lasflightline(las, dt = 30)

lasscanline(las)
```

Arguments

las	A LAS object
dt	numeric. The threshold time-lag used to retrieve flightlines

Details

`laspulse` Retrieves each individual pulse. It uses GPS time. An attribute 'pulseID' is added in the LAS object

`lasscanline` Retrieves each individual scanline. When data are sampled according to a saw-tooth pattern (oscillating mirror), a scanline is one line, or row of data. The function relies on the GPS field time to order the data. Then, the 'ScanDirectionFlag' attribute (when available) is used to retrieve each scanline. An attribute 'scanlineID' is added in the LAS object

`lasflightline` Retrieves each individual flightline. It uses GPS time. In a continuous dataset, once points are ordered by GPS time, the time between two consecutive points does not exceed a few milliseconds. If the time between two consecutive points is too long it means that the second point is from a different flightline. The default threshold is 30 seconds. An attribute 'flightlineID' is added in the LAS object.

Value

An object of class LAS

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile)

las <- laspulse(las)
las

las <- lasflightline(las)
plot(las, color = "flightlineID")
```

lassmooth	<i>Smooth a point cloud</i>
-----------	-----------------------------

Description

Point cloud-based smoothing algorithm. Two methods are available: average within a window and Gaussian smooth within a window. The attribute Z of the returned LAS object is the smoothed Z. A new attribute Zraw is added to store the original values and can be used to restore the point cloud with `lasunsmooth`.

Usage

```
lassmooth(las, size, method = c("average", "gaussian"),
  shape = c("circle", "square"), sigma = size/6)

lasunsmooth(las)
```

Arguments

las	An object of class LAS
size	numeric. The size of the windows used to smooth.
method	character. Smoothing method. Can be 'average' or 'gaussian'.
shape	character. The shape of the windows. Can be circle or square.
sigma	numeric. The standard deviation of the gaussian if the method is gaussian.

Details

This method does not use raster-based methods to smooth the point cloud. This is a true point cloud smoothing. It is not really useful by itself but may be interesting in combination with filters such as [lasfiltersurfacepoints](#), for example to develop new algorithms.

Value

An object of the class LAS.

See Also[lasfiltersurfacepoints](#)**Examples**

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

las <- lasfiltersurfacepoints(las, 1)
plot(las)

las <- lassmooth(las, 5, "gaussian", "circle", sigma = 2)
plot(las)

las <- lasunsmooth(las)
plot(las)
```

lassnags

Snag classification

Description

Snag classification/segmentation using several possible algorithms (see details). The function attributes a number identifying a snag class (snagCls attribute) to each point of the point cloud. The classification/segmentation is done at the point cloud level and currently only one algorithm implemented, which uses LiDAR intensity thresholds and specified neighborhoods to differentiate bole and branch from foliage points (see details).

Usage

```
lassnags(las, algorithm, attribute = "snagCls")
```

Arguments

las	An object of class LAS or LAScatalog .
algorithm	function. An algorithm for snag segmentation. lidR has wing2015 .
attribute	character. The returned LAS object automatically has a new attribute (a new column). This parameter is the name of this new attribute.

Value

If the input is a LAS object, return a LAS object. If the input is a LAScatalog, returns a LAScatalog.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.
- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {XCENTER}, {YCENTER} {ID} and, if chunk size is equal to 0 (processing by file), {ORIGINALFILENAME}.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*"` and cannot be changed.
- **filter**: Read only points of interest.

Examples

```
## Not run:
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzi", filter="-keep_first") # Wing also included -keep_single

# For the Wing2015 method, supply a matrix of snag BranchBolePtRatio conditional
# assessment thresholds (see Wing et al. 2015, Table 2, pg. 172)
bbpr_thresholds <- matrix(c(0.80, 0.80, 0.70,
                           0.85, 0.85, 0.60,
                           0.80, 0.80, 0.60,
                           0.90, 0.90, 0.55),
                          nrow = 3, ncol = 4)
```

```

# Run snag classification and assign classes to each point
las <- lassnags(las, wing2015(neigh_radii = c(1.5, 1, 2), BBPRthrsh_mat = bbpr_thresholds))

# Plot it all, tree and snag points...
plot(las, color="snagCls", colorPalette = rainbow(5))

# Filter and plot snag points only
snags <- lasfilter(las, snagCls > 0)
plot(snags, color="snagCls", colorPalette = rainbow(5)[-1])

# Wing et al's (2015) methods ended with performing tree segmentation on the
# classified and filtered point cloud using the watershed method

## End(Not run)

```

lastransform	<i>Datum transformation for LAS objects</i>
--------------	---

Description

A version of [spTransform](#) for [LAS](#) objects. Returns transformed coordinates of a LAS object from the projection of the object to the the projection given by arguments.

Usage

```
lastransform(las, epsg, CRSobj = NULL)
```

Arguments

las	An object of class LAS
epsg	integer. Unlike spTransform the EPSG code of the CRS should be preferred for LAS objects. See also epsg to understand why. If missing, CRSobj will be used instead.
CRSobj	logical. Object of class CRS or of class character, in which case it is converted to CRS .

Value

An object of class [LAS](#) with coordinates XY transformed to the new coordinate reference system.

Examples

```

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn")

head(las@data)

las <- lastransform(las, 26918)

```

lastrees	<i>Individual tree segmentation</i>
----------	-------------------------------------

Description

Individual tree segmentation with several possible algorithms. The returned point cloud has a new extra byte attribute named after the parameter `attribute` independently of the algorithm used.

Usage

```
lastrees(las, algorithm, attribute = "treeID")
```

Arguments

<code>las</code>	An object of class LAS .
<code>algorithm</code>	function. An algorithm of individual tree segmentation. lidR has: dalponte2016 , watershed , mcwatershed , li2012 and silva2016 . More experimental algorithms may be found in the package lidRplugins
<code>attribute</code>	character. The returned LAS object as a new extra byte attribute (in a new column). This parameter controls the name of the new attribute. Default is "treeID".

Value

An object of the class LAS

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")
col <- pastel.colors(200)

# Using Li et al. (2012)
las <- lastrees(las, li2012(R = 3, speed_up = 5))
plot(las, color = "treeID", colorPalette = col)
```

lasvoxelize	<i>Voxelize a point cloud</i>
-------------	-------------------------------

Description

Reduce the number of points by voxelizing the point cloud. If the Intensity is part of the attributes it is preserved and aggregated as `mean(Intensity)`. Other attributes cannot be aggregated and are lost.

Usage

```
lasvoxelize(las, res)
```

Arguments

`las` An object of class [LAS](#) or [LAScatalog](#).

`res` numeric. The resolution of the voxels. `res = 1` for a 1x1x1 cubic voxels. Optionally `res = c(1, 2)` for non-cubic voxels (1x1x2 cuboid voxel).

Value

If the input is a LAS object, returns a LAS object. If the input is a LAScatalog, returns a LAScatalog.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports `.lax` files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a `.lax` files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used. More cores means more data is loaded at once.
- **progress**: Displays a progression estimation.
- **output_files***: Mandatory because the output is likely to be too big to be returned in R and needs to be written in las/laz files. Supported templates are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {XCENTER}, {YCENTER} {ID} and, if chunk size is equal to 0 (processing by file), {ORIGINALFILENAME}.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*" and cannot be changed.`
- **filter**: Read only points of interest.

Examples

```

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile, select = "xyz")

las2 = lasvoxelize(las, 2)
plot(las2)

```

li2012

*Individual Tree Segmentation Algorithm***Description**

This functions is made to be used in [lastrees](#). It implements an algorithm for tree segmentation based on the Li et al. (2012) article (see reference). This method is a growing region method working at the point cloud level. It is an implementation, as strict as possible, made by the lidR author but with the addition of a parameter `hmin` to prevent over-segmentation for objects that are too low.

Usage

```

li2012(dt1 = 1.5, dt2 = 2, R = 2, Zu = 15, hmin = 2,
       speed_up = 10)

```

Arguments

<code>dt1</code>	numeric. Threshold number 1. See reference page 79 in Li et al. (2012). Default is 1.5.
<code>dt2</code>	numeric. Threshold number 2. See reference page 79 in Li et al. (2012). Default is 2.
<code>R</code>	numeric. Search radius. See page 79 in Li et al. (2012). Default is 2. If <code>R = 0</code> all the points are automatically considered as local maxima and the search step is skipped (much faster).
<code>Zu</code>	numeric. If point elevation is greater than <code>Zu</code> , <code>dt2</code> is used, otherwise <code>dt1</code> is used. See page 79 in Li et al. (2012). Default is 15.
<code>hmin</code>	numeric. Minimum height of a detected tree. Default is 2.
<code>speed_up</code>	numeric. Maximum radius of a crown. Any value greater than a crown is good because this parameter does not affect the result. However, it greatly affects the computation speed. The lower the value, the faster the method. Default is 10.

References

Li, W., Guo, Q., Jakubowski, M. K., & Kelly, M. (2012). A new method for segmenting individual trees from the lidar point cloud. *Photogrammetric Engineering & Remote Sensing*, 78(1), 75-84.

See Also

Other individual tree segmentation algorithms: [dalponte2016](#), [silva2016](#), [watershed](#)

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")
col <- pastel.colors(200)

las <- lastrees(las, li2012(dt1 = 1.4))
plot(las, color = "treeID", colorPalette = col)
```

lidrpalettes

Palettes

Description

Create a vector of n contiguous (or not) colors

Usage

```
height.colors(n)

forest.colors(n)

random.colors(n)

pastel.colors(n)
```

Arguments

n The number of colors (> 1) to be in the palette

See Also

[grDevices::Palettes](#)

lmf

Individual Tree Detection Algorithm

Description

This function is made to be used in [tree_detection](#). It implements an algorithm for tree detection based on a local maximum filter. The windows size can be fixed or variable and its shape can be square or circular. The internal algorithm works either with a raster or a point cloud. It is deeply inspired from Popescu & Wynne (2004) (see references).

Usage

```
lmf(ws, hmin = 2, shape = c("circular", "square"))
```

Arguments

ws	numeric or function. Length or diameter of the moving window used to detect the local maxima in the unit of the input data (usually meters). If it is numeric a fixed window size is used. If it is a function, the function determines the size of the window at any given location on the canopy. The function should take the height of a given pixel or points as its only argument and return the desired size of the search window when centered on that pixel/point.
hmin	numeric. Minimum height of a tree. Threshold below which a pixel or a point cannot be a local maxima. Default is 2.
shape	character. Shape of the moving windows used to find the local maxima. Can be "square" or "circular".

References

Popescu, Sorin & Wynne, Randolph. (2004). Seeing the Trees in the Forest: Using Lidar and Multispectral Data Fusion with Local Filtering and Variable Window Size for Estimating Tree Height. *Photogrammetric Engineering and Remote Sensing*. 70. 589-604. 10.14358/PERS.70.5.589.

See Also

Other individual tree detection algorithms: [manual](#)

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las = readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")

# point-cloud-based
# =====

# 5x5 m fixed windows size
ttops = tree_detection(las, lmf(5))

x = plot(las)
add_treetops3d(x, ttops)

# variable windows size
f = function(x) { x * 0.07 + 3}
ttops = tree_detection(las, lmf(f))

x = plot(las)
add_treetops3d(x, ttops)

# raster-based
# =====

# 5x5 m fixed windows size
chm = grid_canopy(las, res = 1, p2r(0.15))
kernel = matrix(1,3,3)
chm = raster::focal(chm, w = kernel, fun = median, na.rm = TRUE)
```



```
ttops = tree_detection(chm, lmf(5))

raster::plot(chm, col = height.colors(30))
sp::plot(ttops, add = TRUE)

# variable windows size
f = function(x) { x * 0.07 + 3 }
ttops = tree_detection(chm, lmf(f))

raster::plot(chm, col = height.colors(30))
sp::plot(ttops, add = TRUE)
```

manual

Individual Tree Detection Algorithm

Description

This function is made to be used in [tree_detection](#). It implements an algorithm for manual tree detection. Users can pinpoint the tree top positions manually and interactively using the mouse. This is only suitable for small-sized plots. First the point cloud is displayed, then the user is invited to select a rectangular region of interest in the scene using the right mouse button. Within the selected points the highest one will be flagged as 'tree top' in the scene. Once all the trees are labeled the user can exit the tool by selecting an empty region. Points can also be unflagged. The goal of this tool is mainly for minor correction of automatically-detected tree outputs.

Usage

```
manual(detected = NULL, radius = 0.5, color = "red", ...)
```

Arguments

detected	SpatialPointsDataFrame of already found tree tops that need manual corrections.
radius	numeric. Radius of the spheres displayed on the point cloud (aesthetic purpose only).
color	character. Color of the spheres displayed on the point cloud (aesthetic purpose only).
...	supplementary parameters to be passed to plot .

See Also

Other individual tree detection algorithms: [lmf](#)

Examples

```
## Not run:
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las = readLAS(LASfile)

# Full manual tree finding
ttops = tree_detection(las, manual())

# Automatic finding with manual correction
ttops = tree_detection(las, lmf(5))
ttops = tree_detection(las, manual(ttops))

## End(Not run)
```

p2r

Digital Surface Model Algorithm

Description

This function is made to be used in [grid_canopy](#). It implements an algorithm for digital surface model computation based on a points-to-raster method: for each pixel of the output raster the function attributes the height of the highest point found. The `subcircle` tweak replaces each point with 8 points around the original one. This allows for virtual 'emulation' of the fact that a lidar point is not a point as such, but more realistically a disc. This tweak densifies the point cloud and the resulting canopy model is smoother and contains fewer 'pits' and empty pixels.

Usage

```
p2r(subcircle = 0, na.fill = NULL)
```

Arguments

<code>subcircle</code>	numeric. Radius of the circles. To obtain fewer empty pixels the algorithm can replace each return with a circle composed of 8 points (see details).
<code>na.fill</code>	function. A function that implements an algorithm to compute spatial interpolation to fill the empty pixel often left by points-to-raster methods. <code>lidR</code> has knnidw , tin , and kriging (see also grid_terrain for more details).

See Also

Other digital surface model algorithms: [dsmtin](#), [pitfree](#)

Examples

```

LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
col <- height.colors(50)

# Points-to-raster algorithm with a resolution of 1 meter
chm <- grid_canopy(las, res = 1, p2r())
plot(chm, col = col)

# Points-to-raster algorithm with a resolution of 0.5 meters replacing each
# point by a 20 cm radius circle of 8 points
chm <- grid_canopy(las, res = 0.5, p2r(0.2))
plot(chm, col = col)

## Not run:
chm <- grid_canopy(las, res = 0.5, p2r(0.2, na.fill = tin()))
plot(chm, col = col)

## End(Not run)

```

pitfree

Digital Surface Model Algorithm

Description

This function is made to be used in [grid_canopy](#). It implements the pit-free algorithm developed by Khosravipour et al. (2014), which is based on the computation of a set of classical triangulations at different heights (see references). The `subcircle` tweak replaces each point with 8 points around the original one. This allows for virtual 'emulation' of the fact that a lidar point is not a point as such, but more realistically a disc. This tweak densifies the point cloud and the resulting canopy model is smoother and contains fewer 'pits' and empty pixels.

Usage

```

pitfree(thresholds = c(0, 2, 5, 10, 15), max_edge = c(0, 1),
        subcircle = 0)

```

Arguments

<code>thresholds</code>	numeric. Set of height thresholds according to the Khosravipour et al. (2014) algorithm description (see references)
<code>max_edge</code>	numeric. Maximum edge length of a triangle in the Delaunay triangulation. If a triangle has an edge length greater than this value it will be removed. The first number is the value for the classical triangulation (threshold = 0, see also dsmtin), the second number is the value for the pit-free algorithm (for thresholds > 0). If <code>max_edge = 0</code> no trimming is done (see examples).
<code>subcircle</code>	numeric. radius of the circles. To obtain fewer empty pixels the algorithm can replace each return with a circle composed of 8 points (see details).

References

Khosravipour, A., Skidmore, A. K., Isenburg, M., Wang, T., & Hussin, Y. A. (2014). Generating pit-free canopy height models from airborne lidar. *Photogrammetric Engineering & Remote Sensing*, 80(9), 863-872.

See Also

Other digital surface model algorithms: [dsmtin](#), [p2r](#)

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
col <- height.colors(50)

# Basic triangulation and rasterization of first returns
chm <- grid_canopy(las, res = 0.5, dsmtin())
plot(chm, col = col)

# Khosravipour et al. pitfree algorithm
chm <- grid_canopy(las, res = 0.5, pitfree(c(0,2,5,10,15), c(0, 1.5)))
plot(chm, col = col)

## Not run:
# Potentially complex concave subset of point cloud
x = c(481340, 481340, 481280, 481300, 481280, 481340)
y = c(3812940, 3813000, 3813000, 3812960, 3812940, 3812940)
las2 = lasclipPolygon(las,x,y)
plot(las2)

# Since the TIN interpolation is done within the convex hull of the point cloud
# dummy pixels are interpolated that are strictly correct according to the interpolation method
# used, but meaningless in our CHM
chm <- grid_canopy(las2, res = 0.5, pitfree())
plot(chm, col = col)

chm = grid_canopy(las2, res = 0.5, pitfree(max_edge = c(3, 1.5)))
plot(chm, col = col)

## End(Not run)
```

plot

Plot a LAS object*

Description

Plot displays a 3D interactive windows-based on rgl for [LAS](#) objects

Plot displays an interactive view for [LAScatalog](#) objects with pan and zoom capabilities based on [mapview](#). If the coordinate reference system (CRS) of the LAScatalog is non empty, the plot can be displayed on top of base maps (satellite data, elevation, street, and so on).

Plot displays a [LASheader](#) object exactly like it displays a LAScatalog object.

Usage

```
plot(x, y, ...)

## S4 method for signature 'LAS,missing'
plot(x, y, color = "Z",
     colorPalette = height.colors(50), bg = "black", trim = Inf,
     backend = c("rgl", "pcv"), clear_artifacts = TRUE, nbits = 16, ...)

## S4 method for signature 'LAScatalog,missing'
plot(x, y, mapview = FALSE,
     chunk_pattern = FALSE, ...)

## S4 method for signature 'LASheader,missing'
plot(x, y, mapview = TRUE, ...)
```

Arguments

x	A LAS* object
y	Unused (inherited from R base)
...	Will be passed to points3d (LAS) or plot if mapview = FALSE or to mapview if mapview = TRUE (LAScatalog).
color	characters. The attribute used to color the point cloud. Default is Z coordinates. RGB is an allowed string even if it refers to three attributes simultaneously.
colorPalette	characters. A vector of colors such as that generated by <code>heat.colors</code> , <code>topo.colors</code> , <code>terrain.colors</code> or similar functions. Default is <code>height.colors(50)</code> provided by the package <code>lidR</code>
bg	The color for the background. Default is black.
trim	numeric. Enables trimming of values when outliers break the color palette range. Every point with a value higher than <code>trim</code> will be plotted with the highest color.
backend	character. Can be "rgl" or "pcv". If "rgl" is chosen the display relies on the <code>rgl</code> package. If "pcv" is chosen it relies on the <code>PointCloudViewer</code> package, which is much more efficient and can handle million of points using less memory. <code>PointCloudViewer</code> is not available on CRAN yet and should be installed from github (see. https://github.com/Jean-Romain/PointCloudViewer).
clear_artifacts	logical. It is a known and documented issue that the 3D visualisation with <code>rgl</code> displays artifacts. The points look aligned and/or regularly spaced in some view angles. This is because <code>rgl</code> computes with single precision float. To fix that the point cloud is shifted to (0,0) to reduce the number of digits needed to represent its coordinates. The drawback is that the point cloud is not plotted at its actual coordinates.

`nbits` integer. If `color = RGB` it assumes that RGB colors are coded on 16 bits as described in the LAS format specification. However, this is not always respected. If the colors are stored on 8 bits set this parameter to 8.

`mapview` logical. If `FALSE` the catalog is displayed in a regular plot from R base.

`chunk_pattern` logical. Display the current chunk pattern used to process the catalog.

Methods (by class)

- `x = LAScatalog, y = missing`: plot LAScatalog
- `x = LASheader, y = missing`: plot LASheader

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile)

plot(las, clear_artifact = TRUE)

# Outliers in intensity break the color range. Use the trim parameter.
col <- heat.colors(50)
plot(las, color = Intensity, colorPalette = col)
plot(las, color = Intensity, colorPalette = col, trim = 70)

## Not run:

# single file catalog using data provided in lidR
ctg = catalog(LASfile)
plot(ctg)

## End(Not run)

## Not run:

# single file catalog using data provided in lidR
ctg = catalog(LASfile)
plot(ctg)

## End(Not run)
```

plot.lasmetrics3d *Plot voxelized LiDAR data*

Description

This function implements a 3D plot method for 'lasmetrics3d' objects

Usage

```
## S3 method for class 'lasmetrics3d'
plot(x, y, color = "Z",
     colorPalette = height.colors(50), bg = "black", trim = 1, ...)
```

Arguments

x	An object of the class 'lasmetrics3d'
y	Unused (inherited from R base)
color	characters. The field used to color the points. Default is Z coordinates. Or a vector of colors.
colorPalette	characters. A color palette name. Default is height.colors provided by the package lidR
bg	The color for the background. Default is black.
trim	numeric. Enables trimming of values when outliers break the color palette range. Default is 1 meaning that the whole range of the values is used for the color palette. 0.9 means that 10% of the values higher than the 90th percentile are set to the highest color. They are not removed.
...	Supplementary parameters for points3d if the display method is "points".

See Also

[grid_metrics3d](#) [points3d](#) [height.colors](#) [forest.colors](#) [heat.colors](#) [colorRampPalette](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
lidar = readLAS(LASfile)

voxels = grid_metrics3d(lidar, list(Imean = mean(Intensity)))
plot(voxels, color = "Imean", colorPalette = heat.colors(50), trim=0.99)
```

plot_3d

Add a spatial object to a point cloud scene

Description

Add a RasterLayer object that represents a digital terrain model or a SpatialPointsDataFrame that represents tree tops to a point cloud scene. To add elements to a scene with a point cloud plotted with the function plot from lidR, the functions add_* take as first argument the output of the plot function (see examples), because the plot function does not plot the actual coordinates of the point cloud, but offsetted values. See function [plot](#) and its argument clear_artifacts for more details.

Usage

```
plot_dtm3d(dtm, bg = "black", clear_artifacts = TRUE, ...)

add_dtm3d(x, dtm, ...)

add_treetops3d(x, ttops, z = "Z", ...)
```

Arguments

dtm	An object of the class RasterLayer
bg	The color for the background. Default is black.
clear_artifacts	logical. It is a known and documented issue that 3D visualisation with rgl displays artifacts. The points and lines are inaccurately positioned in the space and thus the rendering may look false or weird. This is because rgl computes with single precision float. To fix this, the objects are shifted to (0,0) to reduce the number of digits needed to represent their coordinates. The drawback is that the objects are not plotted at their actual coordinates.
...	Supplementary parameters for surface3d or spheres3d .
x	The output of the function plot used with a LAS object.
ttops	A SpatialPointsDataFrame that contains tree tops coordinates.
z	character. The name of the attribute that contains the height of the tree tops.

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)

dtm = grid_terrain(las, algorithm = tin())
ttops <- tree_detection(las, lmf(ws = 5))

plot_dtm3d(dtm)

x = plot(las)
add_dtm3d(x, dtm)
add_treetops3d(x, ttops)

## Not run:
library(magrittr)
plot(las) %>% add_dtm3d(dtm) %>% add_treetops3d(ttops)

## End(Not run)
```


Description

This function is made to be used in `lasground`. It implements an algorithm for segmentation of ground points based on a progressive morphological filter. This method is an implementation of the Zhang et al. (2003) algorithm (see reference). Note that this is not a strict implementation of Zhang et al. This algorithm works at the point cloud level without any rasterization process. The morphological operator is applied on the point cloud, not on a raster. Also, Zhang et al. proposed some formulas (eq. 4, 5 and 7) to compute the sequence of windows sizes and thresholds. Here, these parameters are free and specified by the user. The function `util_makeZhangParam` enables computation of the parameters according to the original paper.

Usage

```
pmf(ws, th)
```

Arguments

<code>ws</code>	numeric. Sequence of windows sizes to be used in filtering ground returns. The values must be positive and in the same units as the point cloud (usually meters, occasionally feet).
<code>th</code>	numeric. Sequence of threshold heights above the parameterized ground surface to be considered a ground return. The values must be positive and in the same units as the point cloud.

References

Zhang, K., Chen, S. C., Whitman, D., Shyu, M. L., Yan, J., & Zhang, C. (2003). A progressive morphological filter for removing nonground measurements from airborne LIDAR data. *IEEE Transactions on Geoscience and Remote Sensing*, 41(4 PART I), 872–882. <http://doi.org/10.1109/TGRS.2003.810682>.

See Also

Other ground segmentation algorithms: `csf`

Examples

```
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn")

ws <- seq(3,12, 3)
th <- seq(0.1, 1.5, length.out = length(ws))

las <- lasground(las, pmf(ws, th))
plot(las, color = "Classification")
```

print	<i>Summary and Print for LAS* objects</i>
-------	---

Description

Summary and Print for LAS* objects

Usage

```
print(x, ...)

## S4 method for signature 'LAS'
summary(object, ...)

## S4 method for signature 'LAS'
print(x)

## S4 method for signature 'LAScatalog'
summary(object, ...)
```

Arguments

...	Unused
object, x	A LAS* object

Value

NULL, used for its side-effect of printing information

random	<i>Point Cloud Decimation Algorithm</i>
--------	---

Description

This function is made to be used in [lasfilterdecimate](#). It implements an algorithm that randomly removes points or pulses to reach the desired density over the whole area (see [area](#)).

Usage

```
random(density, use_pulse = FALSE)
```

Arguments

density	numeric. The desired output density.
use_pulse	logical. Decimate by removing random pulses instead of random points (requires running laspulse first)

See Also

Other point cloud decimation algorithms: [highest](#), [homogenize](#)

Examples

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

# Reach a pulse density of 1 on the overall dataset
thinned1 = lasfilterdecimate(las, random(1))
plot(grid_density(las))
plot(grid_density(thinned1))
```

rbind.LAS	<i>Merge LAS objects</i>
-----------	--------------------------

Description

Merge LAS objects

Usage

```
## S3 method for class 'LAS'
rbind(...)
```

Arguments

... LAS objects

readLAS	<i>Read .las or .laz files</i>
---------	--------------------------------

Description

Reads .las or .laz files in format 1 to 3 according to LAS specifications and returns an object of class [LAS](#). If several files are read at once the returned LAS object is considered as one LAS file. The optional parameters enable the user to save a substantial amount of memory by choosing to load only the attributes or points of interest. The LAS format 1.4 is currently only partially supported.

Usage

```
readLAS(files, select = "*", filter = "")
```

Arguments

files	characters. Path(s) to one or several a file(s). Can also be a LAScatalog object.
select	character. Read only attributes of interest to save memory (see details).
filter	character. Read only points of interest to save memory (see details).

Details

Select: the 'select' argument specifies which attribute will actually be loaded. For example, 'xyzia' means that the x, y, and z coordinates, the intensity and the scan angle will be loaded. The supported entries are t - gpstime, a - scan angle, i - intensity, n - number of returns, r - return number, c - classification, u - user data, p - point source ID, e - edge of flight line flag, d - direction of scan flag, R - red channel of RGB color, G - green channel of RGB color, B - blue channel of RGB color, N - near infrared channel. Also numbers from 1 to 9 are available for the extra bytes data 1 to 9. 0 enables loading of all extra bytes and '*' is the wildcard and enables everything to be loaded from the LAS file. Note that x, y, z are implicit and always loaded, thus 'xyzia' is equivalent to 'ia'.

Filter: the 'filter' argument allows filtering of the point cloud while reading files. This is much more efficient than [lasfilter](#) in many ways. If the desired filters are known before reading the file, the internal filters should always be preferred. The available filters are those from [LASlib](#) and can be found by running the following command: `rlas::lasfilterusage()`. (see also [rlas::read.las](#))

Value

A LAS object

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)
las = readLAS(LASfile, select = "xyz")
las = readLAS(LASfile, select = "xyzi", filter = "-keep_first")
las = readLAS(LASfile, select = "xyziar", filter = "-keep_first -drop_z_below 0")

# Negation of attribute is also possible (all except intensity and angle)
las = readLAS(LASfile, select = "* -i -a")
```

rumple_index

Rumple index of roughness

Description

Computes the roughness of a surface as the ratio between its area and its projected area on the ground. If the input is a gridded object (`lasmetric` or `raster`) the function computes the surfaces using Jenness's algorithm (see references). If the input is a point cloud the function uses a Delaunay triangulation of the points and computes the area of each triangle.

Usage

```
rumple_index(x, y = NULL, z = NULL, ...)
```

Arguments

x	A 'RasterLayer' or a vector of x point coordinates.
y	numeric. If x is a vector of coordinates: the associated y coordinates.
z	numeric. If x is a vector of coordinates: the associated z coordinates.
...	unused

Value

numeric. The computed Rumple index.

References

Jenness, J. S. (2004). Calculating landscape surface area from digital elevation models. *Wildlife Society Bulletin*, 32(3), 829–839.

Examples

```
x = runif(20, 0, 100)
y = runif(20, 0, 100)

# Perfectly flat surface, rumple_index = 1
z = rep(10, 20)
rumple_index(x, y, z)

# Rough surface, rumple_index > 1
z = runif(20, 0, 10)
rumple_index(x, y, z)

# Rougher surface, rumple_index increases
z = runif(20, 0, 50)
rumple_index(x, y, z)

# Measure of roughness is scale-dependent
rumple_index(x, y, z)
rumple_index(x/10, y/10, z)

# Use with a canopy height model
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)
chm = grid_canopy(las, 2, p2r())
rumple_index(chm)
```

Description

This function is made to be used in [lastrees](#). It implements an algorithm for tree segmentation based on the Silva et al. (2016) article (see reference). This is a simple method based on seed + voronoi tessellation (equivalent to nearest neighbour). This algorithm is implemented in the package rLiDAR. This version is *not* the version from rLiDAR. It is code written from the original article by the lidR authors and is considerably (between 250 and 1000 times) faster.

Usage

```
silva2016(chm, treetops, max_cr_factor = 0.6, exclusion = 0.3,
          ID = "treeID")
```

Arguments

chm	RasterLayer. Image of the canopy. Can be computed with grid_canopy or read from an external file.
treetops	SpatialPointsDataFrame. Can be computed with tree_detection or read from an external shapefile.
max_cr_factor	numeric. Maximum value of a crown diameter given as a proportion of the tree height. Default is 0.6, meaning 60% of the tree height.
exclusion	numeric. For each tree, pixels with an elevation lower than exclusion multiplied by the tree height will be removed. Thus, this number belongs between 0 and 1.
ID	character. If the SpatialPointsDataFrame contains an attribute with the ID for each tree, the name of this column. This way, original IDs will be preserved. If there is no such data trees will be numbered sequentially.

Details

Because this algorithm works on a CHM only there is no actual need for a point cloud. Sometimes the user does not even have the point cloud that generated the CHM. lidR is a point cloud-oriented library, which is why this algorithm must be used in [lastrees](#) to merge the result into the point cloud. However, the user can use this as a stand-alone function like this:

```
chm = raster("file/to/a/chm/")
ttops = tree_detection(chm, lmf(3))
crowns = silva2016(chm, ttops())
```

References

Silva, C. A., Hudak, A. T., Vierling, L. A., Loudermilk, E. L., O'Brien, J. J., Hiers, J. K., Khosravipour, A. (2016). Imputation of Individual Longleaf Pine (*Pinus palustris* Mill.) Tree Attributes from Field and LiDAR Data. *Canadian Journal of Remote Sensing*, 42(5), 554–573. <https://doi.org/10.1080/07038992.2016.1196582>.

See Also

Other individual tree segmentation algorithms: [dalponte2016](#), [li2012](#), [watershed](#)

Other raster based tree segmentation algorithms: [dalponte2016](#), [watershed](#)

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")
col <- pastel.colors(200)

chm <- grid_canopy(las, res = 0.5, p2r(0.3))
ker <- matrix(1,3,3)
chm <- raster::focal(chm, w = ker, fun = mean, na.rm = TRUE)

ttops <- tree_detection(chm, lmf(4, 2))
las <- lastrees(las, silva2016(chm, ttops))
plot(las, color = "treeID", colorPalette = col)
```

stdmetrics

Predefined standard metrics functions

Description

Predefined functions usable in [grid_metrics](#), [grid_hexametrics](#), [lasmetrics](#), [tree_metrics](#), and their convenient shortcuts. The philosophy of the lidR package is to provide an easy way to compute user-defined metrics rather than to provide them. However, for efficiency and to save time, a set of standard metrics has been predefined. To use these functions please read the details and examples sections.

Usage

```
stdmetrics(x, y, z, i, rn, class, dz = 1, th = 2)

.stdmetrics

stdmetrics_z(z, dz = 1, th = 2)

.stdmetrics_z

stdmetrics_i(i, z = NULL, class = NULL, rn = NULL)

.stdmetrics_i

stdmetrics_rn(rn, class = NULL)

.stdmetrics_rn
```

```
stdmetrics_pulse(pulseID, rn)
```

```
.stdmetrics_pulse
```

```
stdmetrics_ctrl(x, y, z)
```

```
.stdmetrics_ctrl
```

```
stdtreemetrics(x, y, z)
```

```
.stdtreemetrics
```

Arguments

x, y, z, i	Coordinates of the points, Intensity
rn, class	ReturnNumber, Classification
dz	numeric. Layer thickness metric entropy
th	numeric. Threshold for metrics pzabovex. Can be a vector to compute with several thresholds.
pulseID	The number referencing each pulse

Format

An object of class formula of length 2.

Details

The function names, their parameters and the output names of the metrics rely on a nomenclature chosen for brevity:

- z: refers to the elevation
- i: refers to the intensity
- rn: refers to the return number
- q: refers to quantile
- a: refers to the ScanAngleRank or ScanAngle
- n: refers to a number (a count)
- p: refers to a percentage

For example the metric named zq60 refers to the elevation, quantile, 60 i.e. the 60th percentile of elevations. The metric pground refers to a percentage. It is the percentage of points classified as ground. The function stdmetric_i refers to metrics of intensity. A description of each existing metric can be found on the [lidR wiki page](#).

Some functions have optional parameters. If these parameters are not provided the function computes only a subset of existing metrics. For example, stdmetrics_i requires the intensity values, but if the elevation values are also provided it can compute additional metrics such as cumulative intensity at a given percentile of height.

Each function has a convenient associated variable. It is the name of the function, with a dot before the name. This enables the function to be used without writing parameters. The cost of such a feature is inflexibility. It corresponds to a predefined behavior (see examples)

stdtreemetrics is a special function that works with [tree_metrics](#). Actually, it won't fail with other functions but the output makes more sense if computed at the individual tree level.

See Also

[grid_metrics](#) [lasmetrics](#) [grid_hexametrics](#) [grid_metrics3d](#) [tree_metrics](#)

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile, select = "*")

# All the predefined metrics
m1 = grid_metrics(las, stdmetrics(X,Y,Z,Intensity,ReturnNumber,Classification,dz=1))

# Convenient shortcut
m2 = grid_metrics(las, .stdmetrics)

# Basic metrics from intensities
m3 = grid_metrics(las, stdmetrics_i(Intensity))

# All the metrics from intensities
m4 = grid_metrics(las, stdmetrics_i(Intensity, Z, Classification, ReturnNumber))

# Convenient shortcut for the previous example
m5 = grid_metrics(las, .stdmetrics_i)

# Compute the metrics only on first return
first = lasfilterfirst(las)
m6 = grid_metrics(first, .stdmetrics_z)

# Compute the metrics with a threshold at 2 meters
over2 = lasfilter(las, Z > 2)
m7 = grid_metrics(over2, .stdmetrics_z)

# Works also with lasmetrics and grid_hexametrics
m8 = lasmetrics(las, .stdmetrics)
m9 = grid_hexametrics(las, .stdmetrics)

# Combine some predefined function with your own new metrics
# Here convenient shortcuts are no longer usable.
myMetrics = function(z, i, rn)
{
  first = rn == 1L
  zfirst = z[first]
  nfirst = length(zfirst)
  above2 = sum(z > 2)
```

```

x = above2/nfirst*100

# User's metrics
metrics = list(
  above2aboven1st = x,      # Num of returns above 2 divided by num of 1st returns
  zimean = mean(z*i),      # Mean products of z by intensity
  zsqmean = sqrt(mean(z^2)) # Quadratic mean of z
)

# Combined with standard metrics
return( c(metrics, stdmetrics_z(z)) )
}

m10 = grid_metrics(las, myMetrics(Z, Intensity, ReturnNumber))

# Users can write their own convenient shortcuts like this:
.myMetrics = ~myMetrics(Z, Intensity, ReturnNumber)

m11 = grid_metrics(las, .myMetrics)

```

tin

*Spatial Interpolation Algorithm***Description**

This function is made to be used in [grid_terrain](#) or [lasnormalize](#). It implements an algorithm for spatial interpolation. Spatial interpolation is based on a Delaunay triangulation, which performs a linear interpolation within each triangle. There are usually a few points outside the convex hull, determined by the ground points at the very edge of the dataset, that cannot be interpolated with a triangulation. Extrapolation is done using the nearest neighbour approach.

Usage

```
tin()
```

See Also

Other spatial interpolation algorithms: [knnidw](#), [kriging](#)

Examples

```

LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)

# plot(las)

dtm = grid_terrain(las, algorithm = tin())

plot(dtm, col = terrain.colors(50))
plot_dtm3d(dtm)

```

tree_detection	<i>Individual tree detection</i>
----------------	----------------------------------

Description

Individual tree detection function that find the position of the trees using several possible algorithms.

Usage

```
tree_detection(las, algorithm)
```

Arguments

las	An object of class LAS or LAScatalog. Can also be a RasterLayer representing a canopy height model, in which case it is processed like a regularly-spaced point cloud.
algorithm	An algorithm for individual tree detection. lidR has: lmf and manual . More experimental algorithms may be found in the package lidRplugins .

Value

A SpatialPointsDataFrame with an attribute Z for the tree tops and treeID with an individual ID for each tree.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used.
- **progress**: Displays a progression estimation.
- **output_files**: Supported templates are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {XCENTER}, {YCENTER} {ID} and, if chunk size is equal to 0 (processing by file), {ORIGINALFILENAME}.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus select = "*" and cannot be changed.
- **filter**: Read only points of interest.

Examples

```

LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")

ttops <- tree_detection(las, lmf(ws = 5))

x = plot(las)
add_treetops3d(x, ttops)

```

tree_hulls

Compute the hull of each tree.

Description

Compute the hull of each segmented tree. The hull can be convex, concave or a bounding box (see details and references).

Usage

```

tree_hulls(las, type = c("convex", "concave", "bbox"), concavity = 3,
  length_threshold = 0, attribute = "treeID")

```

Arguments

las	An object of class LAS or LAScatalog .
type	character. Hull type. Can be 'convex', 'concave' or 'bbox'.
concavity	numeric. If type = "concave", a relative measure of concavity. 1 results in a relatively detailed shape, Infinity results in a convex hull.
length_threshold	numeric. If type = "concave", when a segment length is below this threshold, no further detail is added. Higher values result in simpler shapes.
attribute	character. The attribute where the ID of each tree is stored. In lidR, the default is "treeID".

Details

The concave hull method under the hood is described in Park & Oh (2012). The function relies on the [concaveman](#) function which itself is a wrapper around [Vladimir Agafonking's implementation](#).

Value

A `SpatialPolygonsDataFrame`. If a tree has less than 4 points it is not considered.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used.
- **progress**: Displays a progression estimation.
- **output_files**: Supported templates are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {XCENTER}, {YCENTER} {ID} and, if chunk size is equal to 0 (processing by file), {ORIGINALFILENAME}.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*"` and cannot be changed.
- **filter**: Read only points of interest.

References

Park, J. S., & Oh, S. J. (2012). A new concave hull algorithm and concaveness measure for n-dimensional datasets. *Journal of Information science and engineering*, 28(3), 587-600.

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las = readLAS(LASfile, select = "xyz0", filter = "-drop_z_below 0")

convex_hulls = tree_hulls(las)
sp::plot(convex_hulls)

bbox_hulls = tree_hulls(las, "bbox")
sp::plot(bbox_hulls)
```

```
## Not run:
concave_hulls = tree_hulls(las, "concave")
sp::plot(concave_hulls)

## End(Not run)
```

tree_metrics	<i>Compute metrics for each tree</i>
--------------	--------------------------------------

Description

Once the trees are segmented, i.e. attributes exist in the point cloud that reference each tree, computes a set of user-defined descriptive statistics for each individual tree. This is the "tree version" of [grid_metrics](#).

Usage

```
tree_metrics(las, func, field = "treeID")
```

Arguments

las	An object of class LAS or LAScatalog .
func	The function to be applied to each tree.
field	character. The column name of the field containing tree IDs. Default is "treeID"

Details

By default the function computes the xyz-coordinates of the highest point of each tree and uses xy as tree coordinates in `SpatialPointsDataFrame`. z is stored in the table of attributes along with the id of each tree. All the other attributes are user-defined attributes:

The following existing functions contain a small set of pre-defined metrics:

- [stdmetrics_tree](#)

Users must write their own functions to create their own metrics. `tree_metrics` will dispatch the LiDAR data for each segmented tree in the user-defined function. Functions are defined without the need to consider each segmented tree i.e. only the point cloud (see examples).

Value

A `SpatialPointsDataFrame` that references the xy-position with a table of attributes that associates the z-elevation (highest points) of the trees and the id of the trees, plus the metrics defined by the user.

Working with a LAScatalog

This section appears in each function that supports a LAScatalog as input.

In lidR when the input of a function is a [LAScatalog](#) the function uses the LAScatalog processing engine. The user can modify the engine options using the [available options](#). A careful reading of the [engine documentation](#) is recommended before processing LAScatalogs. Each lidR function should come with a section that documents the supported engine options.

The LAScatalog engine supports .lax files that *significantly* improve the computation speed of spatial queries using a spatial index. Users should really take advantage a .lax files, but this is not mandatory.

Supported processing options

Supported processing options for a LAScatalog (in bold). For more details see the [LAScatalog engine documentation](#):

- **chunk size**: How much data is loaded at once.
- **chunk buffer***: Mandatory to get a continuous output without edge effects. The buffer is always removed once processed and will never be returned either in R or in files.
- **chunk alignment**: Align the processed chunks.
- **cores**: How many cores are used.
- **progress**: Displays a progression estimation.
- **output_files**: Supported templates are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {XCENTER}, {YCENTER} {ID} and, if chunk size is equal to 0 (processing by file), {ORIGINALFILENAME}.
- **laz_compression**: write las or laz files
- **select**: The function will write files equivalent to the original ones. Thus `select = "*" and cannot be changed.`
- **filter**: Read only points of interest.

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las = readLAS(LASfile, filter = "-drop_z_below 0")

# Mean height and mean intensity for each tree
metrics = tree_metrics(las, list(`Mean Z` = mean(Z), `Mean I` = mean(Intensity)))

# Define your own new metrics function
myMetrics = function(z, i)
{
  metrics = list(
    imean = mean(i),
    imax = max(i),
    npoint = length(z)
  )
}
```

```

    return(metrics)
}

metrics = tree_metrics(las, myMetrics(Z, Intensity))

# predefined metrics (see ?stdmetrics)
metrics = tree_metrics(las, .stdtreemetrics)

```

util_makeZhangParam *Parameters for progressive morphological filter*

Description

The function `lasground` with the progressive morphological filter allows for any sequence of parameters. This function enables computation of the sequences using equations (4), (5) and (7) from Zhang et al. (see reference and details).

Usage

```

util_makeZhangParam(b = 2, dh0 = 0.5, dhmax = 3, s = 1,
  max_ws = 20, exp = FALSE)

```

Arguments

<code>b</code>	numeric. This is the parameter b in Zhang et al. (2003) (eq. 4 and 5).
<code>dh0</code>	numeric. This is dh_0 in Zhang et al. (2003) (eq. 7).
<code>dhmax</code>	numeric. This is dh_{max} in Zhang et al. (2003) (eq. 7).
<code>s</code>	numeric. This is s in Zhang et al. (2003) (eq. 7).
<code>max_ws</code>	numeric. Maximum window size to be used in filtering ground returns. This limits the number of windows created.
<code>exp</code>	logical. The window size can be increased linearly or exponentially (eq. 4 or 5).

Details

In the original paper the windows size sequence is given by eq. 4 or 5:

$$w_k = 2kb + 1$$

or

$$w_k = 2b^k + 1$$

In the original paper the threshold sequence is given by eq. 7:

$$th_k = s * (w_k - w_{k-1}) * c + th_0$$

Because the function `lasground` applies the morphological operation at the point cloud level the parameter `c` is set to 1 and cannot be modified.

Value

A list with two components: the windows size sequence and the threshold sequence.

References

Zhang, K., Chen, S. C., Whitman, D., Shyu, M. L., Yan, J., & Zhang, C. (2003). A progressive morphological filter for removing nonground measurements from airborne LIDAR data. *IEEE Transactions on Geoscience and Remote Sensing*, 41(4 PART I), 872–882. <http://doi.org/10.1109/TGRS.2003.810682>.

Examples

```
p = util_makeZhangParam()
```

VCI

Vertical Complexity Index

Description

A fixed normalization of the entropy function (see references)

Usage

```
VCI(z, zmax, by = 1)
```

Arguments

<code>z</code>	vector of z coordinates
<code>zmax</code>	numeric. Used to turn the function entropy to the function vci.
<code>by</code>	numeric. The thickness of the layers used (height bin)

Value

A number between 0 and 1

References

van Ewijk, K. Y., Treitz, P. M., & Scott, N. A. (2011). Characterizing Forest Succession in Central Ontario using LAS-derived Indices. *Photogrammetric Engineering and Remote Sensing*, 77(3), 261-269. Retrieved from <Go to ISI>://WOS:000288052100009

See Also[entropy](#)**Examples**

```
z = runif(10000, 0, 10)

VCI(z, by = 1, zmax = 20)

z = abs(rnorm(10000, 10, 1))

# expected to be closer to 0.
VCI(z, by = 1, zmax = 20)
```

watershed*Individual Tree Segmentation Algorithm*

Description

This function is made to be used in [lastrees](#). It implements an algorithm for tree segmentation based on a watershed or a marker-controlled watershed.

- **Simple watershed** is based on the bioconductor package `EBIimage`. You need to install this package to run this method (see its [github page](#)). Internally, the function `EBIimage::watershed` is called.
- **Marker-controlled watershed** is based on the `imager` package. Internally, the function `imager::watershed` is called using the tree tops as a priority map.

Usage

```
watershed(chm, th_tree = 2, tol = 1, ext = 1)

mcwatershed(chm, treetops, th_tree = 2, ID = "treeID")
```

Arguments

chm	RasterLayer. Image of the canopy. Can be computed with grid_canopy or read from an external file.
th_tree	numeric. Threshold below which a pixel cannot be a tree. Default is 2.
tol	numeric. Tolerance see <code>?EBImage::watershed</code> .
ext	numeric. see <code>?EBImage::watershed</code> .
treetops	<code>SpatialPointsDataFrame</code> . Can be computed with tree_detection or read from an external shapefile.
ID	character. If the <code>SpatialPointsDataFrame</code> contains an attribute with the ID for each tree, the name of this column. This way, original IDs will be preserved. If there is no such data trees will be numbered sequentially.

Details

Because this algorithm works on a CHM only there is no actual need for a point cloud. Sometimes the user does not even have the point cloud that generated the CHM. `lidR` is a point cloud-oriented library, which is why this algorithm must be used in `lastrees` to merge the result into the point cloud. However, the user can use this as a stand-alone function like this:

```
chm = raster("file/to/a/chm/")
ttops = tree_detection(chm, lmf(3))
crowns = watershed(chm())
```

See Also

Other individual tree segmentation algorithms: [dalponte2016](#), [li2012](#), [silva2016](#)

Other raster based tree segmentation algorithms: [dalponte2016](#), [silva2016](#)

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-drop_z_below 0")
col <- pastel.colors(250)

chm <- grid_canopy(las, res = 0.5, p2r(0.3))
ker <- matrix(1,3,3)
chm <- raster::focal(chm, w = ker, fun = mean, na.rm = TRUE)

ttops <- tree_detection(chm, lmf(4, 2))
las <- lastrees(las, mcwatershed(chm, ttops))

x = plot(las, color = "treeID", colorPalette = col)
add_treetops3d(x, ttops)
```

Description

This function is made to be used in `lassnags`. It implements an algorithms for snags segmentation based on Wing et al (2015) (see references). This is an automated filtering algorithm that utilizes three dimensional neighborhood lidar point-based intensity and density statistics to remove lidar points associated with live trees and retain lidar points associated with snags.

Usage

```
wing2015(neigh_radii = c(1.5, 1, 2), low_int_thrsh = 50,
  uppr_int_thrsh = 170, pt_den_req = 3, BBPRthrsh_mat = NULL)
```

Arguments

<code>neigh_radii</code>	numeric. A vector of three radii used in quantifying local-area centered neighborhoods. See Wing et al. (2015) reference page 171 and Figure 4. Defaults are 1.5, 1, and 2 for the sphere, small cylinder and large cylinder neighborhoods, respectively.
<code>low_int_thrsh</code>	numeric. The lower intensity threshold filtering value. See Wing et al. (2015) page 171. Default is 50.
<code>uppr_int_thrsh</code>	numeric. The upper intensity threshold filtering value. See Wing et al. (2015) page 171. Default is 170.
<code>pt_den_req</code>	numeric. Point density requirement based on plot-level point density defined classes. See Wing et al. (2015) page 172. Default is 3.
<code>BBPRthrsh_mat</code>	matrix. A 3x4 matrix providing the four average BBPR (branch and bole point ratio) values for each of the three neighborhoods (sphere, small cylinder and large cylinder) to be used for conditional assessments and classification into the following four snag classes: 1) general snag 2) small snag 3) live crown edge snag 4) high canopy cover snag. See Wing et al. (2015) page 172 and Table 2. This matrix must be provided by the user.

Details

Note that this algorithm strictly performs a classification based on user input while the original publication's methods also included a segmentation step and some pre- (filtering for first and single returns only) and post-process (filtering for only the snag classified points prior to segmentation) tasks which are now expected to be performed by the user. Also, this implementation may have some differences compared with the original method due to potential mis-interpretation of the Wing et al. manuscript, specifically Table 2 where they present four groups of conditional assessments with their required neighborhood point density and average BBPR values (BBPR = branch and bole point ratio; PDR = point density requirement).

This algorithm attributes each point in the point cloud (`snagCls` column) into the following five snag classes:

- 0: live tree - not a snag
- 1: general snag - the broadest range of snag point situations
- 2: small snag - isolated snags with lower point densities
- 3: live crown edge snag - snags located directly adjacent or intermixing with live trees crowns
- 4: high canopy cover snag - snags protruding above the live canopy in dense conditions (e.g., canopy cover $\geq 55\%$).

The current implementation is known to use a large amount of memory for storing the $N \times k$ integer matrix returning the near neighbor indices for each point in the point cloud. Improvements are possible in future package versions.

Author(s)

Implementation by Andrew Sánchez Meador

References

Wing, Brian M.; Ritchie, Martin W.; Boston, Kevin; Cohen, Warren B.; Olsen, Michael J. 2015. Individual snag detection using neighborhood attribute filtered airborne lidar data. Remote Sensing of Environment. 163: 165-179 <https://doi.org/10.1016/j.rse.2015.03.013>

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzi", filter="-keep_first") # Wing also included -keep_single

# For the Wing2015 method, supply a matrix of snag BranchBolePtRatio conditional
# assessment thresholds (see Wing et al. 2015, Table 2, pg. 172)
bbpr_thresholds <- matrix(c(0.80, 0.80, 0.70,
                           0.85, 0.85, 0.60,
                           0.80, 0.80, 0.60,
                           0.90, 0.90, 0.55),
                          nrow = 3, ncol = 4)

# Run snag classification and assign classes to each point
las <- lassnags(las, wing2015(neigh_radii = c(1.5, 1, 2), BBPRthrsh_mat = bbpr_thresholds))

# Plot it all, tree and snag points...
plot(las, color="snagCls", colorPalette = rainbow(5))

# Filter and plot snag points only
snags <- lasfilter(las, snagCls > 0)
plot(snags, color="snagCls", colorPalette = rainbow(5)[-1])

# Wing et al's (2015) methods ended with performing tree segmentation on the
# classified and filtered point cloud using the watershed method
```

writeLAS

Write a .las or .laz file

Description

Write a [LAS](#) object into a binary .las or .laz file (compression specified in filename)

Usage

```
writeLAS(las, file)
```

Arguments

las an object of class LAS.
 file character. A character string naming an output file.

Value

Nothing. This function is used for its side-effect of writing a file.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile)
subset = lasclipRectangle(las, 684850, 5017850, 684900, 5017900)
writeLAS(subset, tempfile(fileext = ".laz"))
```

\$<-,LAS-method

Inherited but modified methods from sp

Description

LAS* objects are [Spatial](#) objects so they inherit several methods from sp. However, some have modified behaviors to prevent some irrelevant modifications. Indeed, a LAS* object cannot contain anything, as the content is restricted by the LAS specifications. If a user attempts to use one of these functions inappropriately an informative error will be thrown.

Usage

```
## S4 replacement method for signature 'LAS'
x$name <- value

## S4 replacement method for signature 'LAS,ANY,missing'
x[[i, j]] <- value

## S4 method for signature 'LAScatalog,ANY,ANY'
x[i, j, ..., drop = TRUE]

## S4 replacement method for signature 'LAScatalog,ANY,ANY'
x[[i, j]] <- value

## S4 replacement method for signature 'LAScatalog'
x$name <- value
```

Arguments

<code>x</code>	A LAS* object
<code>name</code>	A literal character string or a name (possibly backtick quoted).
<code>value</code>	typically an array-like R object of a similar class as <code>x</code> .
<code>i</code>	string, name of elements to extract or replace.
<code>j</code>	Unused.
<code>...</code>	Unused
<code>drop</code>	Unused

Examples

```
## Not run:  
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")  
las = readLAS(LASfile)  
  
las$Z = 2L  
las[["Z"]] = 1:10  
las$NewCol = 0  
las[["NewCol"]] = 0  
  
## End(Not run)
```

Index

*Topic **datasets**

- stdmetrics, [87](#)
- , LAS, RasterLayer-method (lasnormalize), [60](#)
- , LAS, function-method (lasnormalize), [60](#)
- .stdmetrics (stdmetrics), [87](#)
- .stdmetrics_ctrl (stdmetrics), [87](#)
- .stdmetrics_i (stdmetrics), [87](#)
- .stdmetrics_pulse (stdmetrics), [87](#)
- .stdmetrics_rn (stdmetrics), [87](#)
- .stdmetrics_z (stdmetrics), [87](#)
- .stdtreemetrics (stdmetrics), [87](#)
- [, LAScatalog, ANY, ANY-method (\$<-, LAS-method), [102](#)
- [[<-, LAS, ANY, missing-method (\$<-, LAS-method), [102](#)
- [[<-, LAScatalog, ANY, ANY-method (\$<-, LAS-method), [102](#)
- \$<-, LAS-method, [102](#)
- \$<-, LAScatalog-method (\$<-, LAS-method), [102](#)

- add_dtm3d (plot_3d), [79](#)
- add_treetops3d (plot_3d), [79](#)
- appropriated functions, [6](#)
- area, [4](#), [82](#)
- area, LAS-method (area), [4](#)
- area, LAScatalog-method (area), [4](#)
- as.list.LASheader, [5](#)
- as.spatial, [5](#)
- available options, [14](#), [23](#), [25](#), [29](#), [32](#), [47](#), [50](#), [52](#), [56](#), [61](#), [66](#), [69](#), [93](#), [95](#)

- bbox, [8](#)

- catalog, [6](#), [42](#)
- catalog_apply, [7](#)
- catalog_intersect, [11](#)
- catalog_options_tools, [12](#)
- catalog_retile, [13](#)

- colorRampPalette, [79](#)
- concaveman, [92](#)
- CRS, [39](#), [67](#)
- CRS-class, [39](#)
- csf, [15](#), [55](#), [81](#)

- dalponte2016, [16](#), [68](#), [70](#), [87](#), [99](#)
- data.table, [39](#)
- dsmtin, [18](#), [23](#), [74–76](#)

- engine documentation, [14](#), [23](#), [25](#), [29](#), [32](#), [47](#), [50](#), [52](#), [56](#), [61](#), [66](#), [69](#), [93](#), [95](#)
- entropy, [19](#), [29](#), [59](#), [88](#), [98](#)
- epsg, [20](#), [67](#)
- epsg, LAS-method (epsg), [20](#)
- epsg, LASheader-method (epsg), [20](#)
- epsg<- (epsg), [20](#)
- epsg<-, LAS-method (epsg), [20](#)
- epsg<-, LASheader-method (epsg), [20](#)
- Extent, [47](#)
- extent, [8](#)
- extent, LAS-method, [21](#)
- extent, LAScatalog-method (extent, LAS-method), [21](#)

- forest.colors, [79](#)
- forest.colors (lidrpalettes), [71](#)

- gap_fraction_profile, [22](#), [38](#)
- gdalbuildvrt, [23](#), [25](#), [28](#), [32](#)
- grDevices::Palettes, [71](#)
- grid_canopy, [17](#), [18](#), [23](#), [74](#), [75](#), [86](#), [98](#)
- grid_density, [25](#)
- grid_hexametrics, [26](#), [87](#), [89](#)
- grid_metrics, [9](#), [24](#), [26](#), [27](#), [28](#), [30](#), [31](#), [33](#), [59](#), [87](#), [89](#), [94](#)
- grid_metrics3d, [30](#), [79](#), [89](#)
- grid_terrain, [32](#), [36](#), [37](#), [61](#), [62](#), [74](#), [90](#)

- heat.colors, [79](#)
- height.colors, [79](#)

- height.colors (lidrpalettes), 71
- hexbin, 27
- highest, 34, 35, 50, 54, 83
- homogenize, 34, 34, 50, 83
- imager::watershed, 98
- intersect, 11
- is.empty, 35
- is.empty, LAS-method (is.empty), 35
- knnidw, 32, 36, 37, 61, 74, 90
- krige, 37
- kriging, 32, 36, 37, 61, 74, 90
- LAD, 22, 29, 37, 59
- LAS, 23, 25, 28, 32, 40, 41, 46, 49–51, 53–55, 61, 65, 67–69, 76, 83, 92, 94, 101
- LAS (LAS-class), 38
- LAS-class, 38
- lasaddattribute, 40
- lasadddata (lasaddattribute), 40
- lasaddextrabytes (lasaddattribute), 40
- lasaddextrabytes_manual (lasaddattribute), 40
- LAScatalog, 6, 7, 11–14, 23, 25, 28, 29, 32, 46, 47, 50–52, 54–56, 61, 65, 66, 69, 77, 84, 92–95
- LAScatalog class, 7
- LAScatalog class documentation, 6
- LAScatalog engine documentation, 9, 14, 24, 26, 29, 33, 48, 50, 52, 54, 56, 61, 66, 69, 91, 93, 95
- LAScatalog-class, 42, 48
- lascheck, 45
- lasclip, 46
- lasclipCircle (lasclip), 46
- lasclipPolygon (lasclip), 46
- lasclipRectangle (lasclip), 46
- lasfilter, 49, 52–55, 84
- lasfilterdecimate, 34, 50, 82
- lasfilterduplicates, 49, 51, 53–55
- lasfilterfirst (lasfilters), 52
- lasfilterfirstlast (lasfilters), 52
- lasfilterfirstofmany (lasfilters), 52
- lasfilterground (lasfilters), 52
- lasfilterlast (lasfilters), 52
- lasfilternth (lasfilters), 52
- lasfilters, 49, 52, 52, 55
- lasfiltersingle (lasfilters), 52
- lasfiltersurfacepoints, 49, 52–54, 54, 64, 65
- lasflightline (laspulse), 63
- lasground, 15, 37, 55, 81, 96, 97
- LASheader, 39, 57, 77
- LASheader-class, 57
- lasmergespatial, 58
- lasmetrics, 59, 87, 89
- lasnormalize, 33, 36, 60, 90
- laspulse, 35, 63, 82
- lasremoveextrabytes (lasaddattribute), 40
- lasscanline (laspulse), 63
- lassmooth, 64
- lassnags, 65, 99
- lastransform, 67
- lastrees, 16, 17, 68, 70, 86, 98, 99
- lasunnormalize (lasnormalize), 60
- lasunsmooth (lassmooth), 64
- lasvoxelize, 68
- li2012, 17, 68, 70, 87, 99
- lidrpalettes, 71
- list.files, 6
- lmf, 71, 73, 91
- manual, 72, 73, 91
- mapview, 77
- matrix, 47
- mcwatershed, 68
- mcwatershed (watershed), 98
- opt_chunk_alignment (catalog_options_tools), 12
- opt_chunk_alignment<- (catalog_options_tools), 12
- opt_chunk_buffer (catalog_options_tools), 12
- opt_chunk_buffer<- (catalog_options_tools), 12
- opt_chunk_size (catalog_options_tools), 12
- opt_chunk_size<- (catalog_options_tools), 12
- opt_cores (catalog_options_tools), 12
- opt_cores<- (catalog_options_tools), 12
- opt_filter (catalog_options_tools), 12
- opt_filter<- (catalog_options_tools), 12
- opt_laz_compression (catalog_options_tools), 12

- opt_laz_compression<-
 (catalog_options_tools), 12
- opt_output_files
 (catalog_options_tools), 12
- opt_output_files<-
 (catalog_options_tools), 12
- opt_progress (catalog_options_tools), 12
- opt_progress<- (catalog_options_tools),
 12
- opt_select (catalog_options_tools), 12
- opt_select<- (catalog_options_tools), 12
- opt_stop_early (catalog_options_tools),
 12
- opt_stop_early<-
 (catalog_options_tools), 12
- opt_wall_to_wall
 (catalog_options_tools), 12
- opt_wall_to_wall<-
 (catalog_options_tools), 12

- p2r, 18, 23, 74, 76
- pastel.colors (lidrpalettes), 71
- pitfree, 18, 23, 74, 75
- plot, 73, 76, 77, 79
- plot,LAS,missing-method (plot), 76
- plot,LAScatalog,missing-method (plot),
 76
- plot,LASheader,missing-method (plot), 76
- plot.lasmetrics3d, 78
- plot_3d, 79
- plot_dtm3d (plot_3d), 79
- pmf, 16, 55, 81
- points3d, 77, 79
- Polygon, 47
- Polygons, 47
- print, 82
- print,LAS-method (print), 82

- random, 34, 35, 50, 82
- random.colors (lidrpalettes), 71
- raster, 62
- raster::extent, 22
- RasterLayer, 61
- rbind.LAS, 83
- readLAS, 8, 40, 43, 44, 83
- rlas::read.las, 84
- rumple_index, 84

- silva2016, 17, 68, 70, 86, 99

- SimpleFeature, 47
- Spatial, 39, 40, 102
- SpatialPoints, 47
- SpatialPointsDataFrame, 47
- SpatialPolygons, 47
- SpatialPolygonsDataFrame, 47
- spheres3d, 80
- spTransform, 67
- stdmetrics, 29, 59, 87
- stdmetrics_ctrl (stdmetrics), 87
- stdmetrics_i (stdmetrics), 87
- stdmetrics_pulse (stdmetrics), 87
- stdmetrics_rn (stdmetrics), 87
- stdmetrics_tree, 94
- stdmetrics_z (stdmetrics), 87
- stdtreemetrics (stdmetrics), 87
- summary (print), 82
- summary,LAS-method (print), 82
- summary,LAScatalog-method (print), 82
- surface3d, 80

- tin, 32, 36, 37, 61, 74, 90
- tree_detection, 17, 71, 73, 86, 91, 98
- tree_hulls, 92
- tree_metrics, 87, 89, 94

- util_makeZhangParam, 81, 96

- VCI, 19, 20, 29, 59, 97
- vgm, 37

- watershed, 17, 68, 70, 87, 98
- wing2015, 65, 99
- writeLAS, 41, 101