Package ‘lidR’

November 13, 2021

Type Package

Title Airborne LiDAR Data Manipulation and Visualization for Forestry Applications

Version 3.2.3

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 (>= 3.4.0), methods, raster, sp (>= 1.4.2)

Imports data.table (>= 1.12.0), future, geometry, glue, grDevices, lazyeval, Rcpp (>= 1.0.3), rgl, rlas (>= 1.5.0), sf, stats, tools, utils

Suggests EBImage, gstat, RCSF, rgeos, mapview, mapedit, progress, rgdal (>= 1.5-8), testthat (>= 2.1.0), knitr, rmarkdown

RoxygenNote 7.1.2

LinkingTo BH (>= 1.72.0), Rcpp, RcppArmadillo

Encoding UTF-8

ByteCompile true

VignetteBuilder knitr

biocViews

R topics documented:

'catalog_boundaries.R' 'catalog_fakerun.R' 'catalog_index.R'
'catalog_intersect.R' 'catalog_laxindex.R'
'catalog_makecluster.R' 'catalog_merge_results.R'
'catalog_overlaps.R' 'catalog_retile.R' 'catalog_select.R'
'classify_ground.R' 'classify_noise.R' 'classify_poi.R'
'concaveman.R' 'decimate_points.R' 'delineate_crowns.R'
'deprecated.R' 'doc-drivers.R' 'doc-lidR.R' 'doc-parallelism.R'
'filter_duplicates.R' 'filter_poi.R' 'filter_roi.R'
'filter_surfacepoints.R' 'find_localmaxima.R' 'find_trees.R'
'fullwaveform.R' 'generate_las.R' 'grid_canopy.R'
'grid_density.R' 'grid_metrics.R' 'grid_terrain.R'
'io_readLAS.R' 'io_readLAScatalog.R' 'io_readXLAS.R'
'io_writeANY.R' 'io_writeLAS.R' 'las_check.R' 'las_tools.R'
'merge_las.R' 'merge_spatial.R' 'methods-LAS.R'
'methods-LAScatalog.R' 'methods-LAScluster.R'
'methods-LASheader.R' 'normalize_height.R'
'normalize_intensity.R' 'plot.R' 'plot.s3.R' 'plot_metrics.R'
'point_metrics.R' 'print.R' 'projection.R' 'retrieve_info.R'
'segment_shapes.R' 'segment_snags.R' 'segment_trees.R'
'sensor_tracking.R' 'smooth_height.R' 'tree_metrics.R'
'utilis_asseritive.R' 'utilis_catalog_options.R' 'utilis_colors.R'
'utilis_define_constant.R' 'utilis_delaunay.R' 'utilis_geometry.R'
'utilis_is.R' 'utilis_metrics.R' 'utilis_misc.R' 'utilis_raster.R'
'utilis_spatial_index.R' 'utilis_threads.R' 'utilis_typecast.R'
'voxel_metrics.R' 'voxelize_points.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 wing2015() for
   segment_snags()),
   Bourdon Jean-François [ctb] (Contributed to Roussel2020() for
   track_sensor()),
   Gatziolis Demetrios [ctb] (Implemented Gatziolis2019() for
   track_sensor()),
   Leon Steinmeier [ctb] (Contributed to parallelization management),
   Stanislaw Adaszewski [cph] (Author of the C++ concaveman code)

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

Repository CRAN

Date/Publication 2021-11-13 14:30:02 UTC

R topics documented:

lidR-package ............................................. 5
add_attribute ............................................ 6
<table>
<thead>
<tr>
<th>Topics Documented:</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
</tr>
<tr>
<td>as.list.LASheader</td>
</tr>
<tr>
<td>as.spatial</td>
</tr>
<tr>
<td>asprs</td>
</tr>
<tr>
<td>catalog_apply</td>
</tr>
<tr>
<td>catalog_boundaries</td>
</tr>
<tr>
<td>catalog_intersect</td>
</tr>
<tr>
<td>catalog_makechunks</td>
</tr>
<tr>
<td>catalog_options_tools</td>
</tr>
<tr>
<td>catalog_retile</td>
</tr>
<tr>
<td>catalog_select</td>
</tr>
<tr>
<td>classify_ground</td>
</tr>
<tr>
<td>classify_noise</td>
</tr>
<tr>
<td>classify.poi</td>
</tr>
<tr>
<td>clip</td>
</tr>
<tr>
<td>cloud_metrics</td>
</tr>
<tr>
<td>concaveman</td>
</tr>
<tr>
<td>csf</td>
</tr>
<tr>
<td>dalponte2016</td>
</tr>
<tr>
<td>decimate_points</td>
</tr>
<tr>
<td>delineate_crowns</td>
</tr>
<tr>
<td>deprecated</td>
</tr>
<tr>
<td>dsmitin</td>
</tr>
<tr>
<td>entropy</td>
</tr>
<tr>
<td>extent.LAS-method</td>
</tr>
<tr>
<td>filters</td>
</tr>
<tr>
<td>filter_duplicates</td>
</tr>
<tr>
<td>filter_poi</td>
</tr>
<tr>
<td>filter_surfacepoints</td>
</tr>
<tr>
<td>find_localmaxima</td>
</tr>
<tr>
<td>find_trees</td>
</tr>
<tr>
<td>gap_fraction_profile</td>
</tr>
<tr>
<td>Gatziolis2019</td>
</tr>
<tr>
<td>grid_canopy</td>
</tr>
<tr>
<td>grid_density</td>
</tr>
<tr>
<td>grid_metrics</td>
</tr>
<tr>
<td>grid_terrain</td>
</tr>
<tr>
<td>homogenize</td>
</tr>
<tr>
<td>interpret_waveform</td>
</tr>
<tr>
<td>is</td>
</tr>
<tr>
<td>ivf</td>
</tr>
<tr>
<td>knnidw</td>
</tr>
<tr>
<td>kriging</td>
</tr>
<tr>
<td>LAD</td>
</tr>
<tr>
<td>LAS-class</td>
</tr>
<tr>
<td>LAScatalog-class</td>
</tr>
<tr>
<td>LASheader</td>
</tr>
<tr>
<td>LASHeader-class</td>
</tr>
</tbody>
</table>
R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>las_check</td>
<td>80</td>
</tr>
<tr>
<td>las_utilities</td>
<td>81</td>
</tr>
<tr>
<td>li2012</td>
<td>83</td>
</tr>
<tr>
<td>lidR-LAScatalog-drivers</td>
<td>84</td>
</tr>
<tr>
<td>lidR-parallelism</td>
<td>86</td>
</tr>
<tr>
<td>lidR-spatial-index</td>
<td>89</td>
</tr>
<tr>
<td>lidrpalettes</td>
<td>92</td>
</tr>
<tr>
<td>lmf</td>
<td>93</td>
</tr>
<tr>
<td>manual</td>
<td>94</td>
</tr>
<tr>
<td>maxima</td>
<td>95</td>
</tr>
<tr>
<td>merge Spatial</td>
<td>96</td>
</tr>
<tr>
<td>normalize_height</td>
<td>97</td>
</tr>
<tr>
<td>normalize_intensity</td>
<td>100</td>
</tr>
<tr>
<td>p2r</td>
<td>102</td>
</tr>
<tr>
<td>pitfree</td>
<td>104</td>
</tr>
<tr>
<td>plot</td>
<td>103</td>
</tr>
<tr>
<td>plot.lasmetrics3d</td>
<td>106</td>
</tr>
<tr>
<td>plot_3d</td>
<td>107</td>
</tr>
<tr>
<td>plot_metrics</td>
<td>108</td>
</tr>
<tr>
<td>pmf</td>
<td>110</td>
</tr>
<tr>
<td>point_metrics</td>
<td>111</td>
</tr>
<tr>
<td>print</td>
<td>114</td>
</tr>
<tr>
<td>projection</td>
<td>115</td>
</tr>
<tr>
<td>random</td>
<td>117</td>
</tr>
<tr>
<td>random_per_voxel</td>
<td>118</td>
</tr>
<tr>
<td>range_correction</td>
<td>119</td>
</tr>
<tr>
<td>rbind</td>
<td>120</td>
</tr>
<tr>
<td>readLAS</td>
<td>120</td>
</tr>
<tr>
<td>readLAScatalog</td>
<td>122</td>
</tr>
<tr>
<td>readLASheader</td>
<td>125</td>
</tr>
<tr>
<td>retrieve_pulses</td>
<td>126</td>
</tr>
<tr>
<td>Roussel2020</td>
<td>127</td>
</tr>
<tr>
<td>rumple_index</td>
<td>128</td>
</tr>
<tr>
<td>segment_shapes</td>
<td>129</td>
</tr>
<tr>
<td>segment_snags</td>
<td>131</td>
</tr>
<tr>
<td>segment_trees</td>
<td>132</td>
</tr>
<tr>
<td>set_lidr_threads</td>
<td>135</td>
</tr>
<tr>
<td>shape_detection</td>
<td>135</td>
</tr>
<tr>
<td>silva2016</td>
<td>137</td>
</tr>
<tr>
<td>smooth_height</td>
<td>138</td>
</tr>
<tr>
<td>sor</td>
<td>139</td>
</tr>
<tr>
<td>stdmetrics</td>
<td>140</td>
</tr>
<tr>
<td>tin</td>
<td>143</td>
</tr>
<tr>
<td>track_sensor</td>
<td>144</td>
</tr>
<tr>
<td>tree_metrics</td>
<td>147</td>
</tr>
<tr>
<td>util_makeZhangParam</td>
<td>149</td>
</tr>
<tr>
<td>VCI</td>
<td>151</td>
</tr>
<tr>
<td>voxelize_points</td>
<td>152</td>
</tr>
</tbody>
</table>
**Description**

lidR provides a set of tools to manipulate airborne LiDAR data in forestry contexts. The package works essentially with .las or .laz files. The toolbox includes algorithms for DSM, CHM, DTM, ABA, normalisation, tree detection, tree segmentation and other tools, as well as an engine to process wide LiDAR coverages split into many files.

**Details**

To learn more about lidR, start with the vignettes: `browseVignettes(package = "lidR")`. Users can also find unofficial supplementary documentation in the lidR book. To ask "how to" questions please ask on gis.stackexchange.com with the tag `lidr`.

**Package options**

- `lidR.progress` Several functions have a progress bar for long operations (but not all). Should lengthy operations show a progress bar? Default: TRUE
- `lidR.progress.delay` The progress bar appears only for long operations. After how many seconds of computation does the progress bar appear? Default: 2
- `lidR.verbose` Make the package verbose. Default: FALSE
- `lidR.buildVRT` The functions `grid_*` can write the rasters sequentially on the disk and load back a virtual raster mosaic (VRT) instead of the list of written files. Should a VRT be built? Default: TRUE
- `lidR.check.nested.parallelism` The catalog processing engine (`catalog_apply`) checks the parallel strategy chosen by the user and verify if C++ parallelization with OpenMP should be disabled to avoid nested parallel loops. Default: TRUE. If FALSE the catalog processing engine will not check for nested parallelism and will respect the settings of `set_lidr_threads`.

**Author(s)**

**Maintainer**: Jean-Romain Roussel <jean-romain.roussel.1@ulaval.ca> [copyright holder]

Authors:

- David Auty (Reviews the documentation) [contributor]

Other contributors:
add_attribute

Add attributes into a LAS object

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.

Usage

add_attribute(las, x, name)

add_lasattribute(las, x, name, desc)

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

add_lasrgb(las, R, G, B)

remove_lasattribute(las, name)
Arguments

- **las**: An object of class `LAS`
- **x**: a vector that needs to be added in the LAS object. For `add_lasattribute*` it can be missing (see details).
- **name**: character. The name of the extra bytes attribute to add in the file.
- **desc**: character. A short description of the extra bytes attribute to add in the file (32 characters).
- **type**: character. The data type of the extra bytes attribute. Can be "uchar", "char", "ushort", "short", "uint", "int", "uint64", "int64", "float", "double".
- **scale, offset**: numeric. The scale and offset of the data. NULL if not relevant.
- **NA_value**: 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.
- **R, G, B**: integer. RGB values

Details

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

- `add_attribute` 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`.
- `add_lasattribute` Does the same as `add_attribute` 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`.
- `add_lasattribute_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 `add_lasattribute` and `add_lasattribute_manual`, x can only be of type numeric, (integer or double). It cannot be of type 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 on page 25 of the specification). Types greater than 10 are not supported.
- `add_lasrgb` Adds 3 columns named RGB and updates the point format of the LAS object for a format that supports RGB attributes. If the RGB values are ranging from 0 to 255 they are automatically scaled on 16 bits.

Value

An object of class `LAS`

Examples

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

x <- 1:30

las <- add_attribute(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 <- add_lasattribute(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 <- add_lasattribute(las, name = "mydata", desc = "Amplitude")
print(las@header)

# Remove an extra bytes attribute
las <- remove_lasattribute(las, "mydata2")
print(las)
print(las@header)

las <- remove_lasattribute(las, "mydata")
print(las)
print(las@header)

---

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. For a LASheader it is computed with the bounding box. The function npoints does what the user may expect it to do and the function density is equivalent to npoints(x)/area(x). As a consequence for the same file area applied on a LASheader or on a LAS can return slightly different values.

Usage

area(x, ...)

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

## S4 method for signature 'LASheader'
area(x, ...)
Functions to construct, coerce and check for both kinds of R lists.

### Arguments

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

### Value

numeric. A number. Notice that for area the measure is in the same units as the coordinate reference system.

### Description

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

### Usage

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

x A LASheader object
...

unused

as.spatial Transform a LAS* object into an sp object

Description

LAS and LAScatalog objects are transformed into SpatialPointsDataFrame and SpatialPolygonsDataFrame objects, respectively.

Usage

as.spatial(x)

Arguments

x an object from the lidR package

Value

An object from sp

asprs ASPRS LAS Classification

Description

A set of global variables corresponding to the point classification defined by the ASPRS for the LAS format. Instead of remembering the classification table of the specification it is possible to use one of these global variables.

Usage

LASNONCLASSIFIED
LASUNCLASSIFIED
LASGROUND
LASLOWVEGETATION
LASMEDIUMVEGETATION
LASHIGHVEGETATION
LASBUILDING
LASLOWPOINT
LASKEYPOINT
LASWATER
LASRAIL
LASROADSURFACELASWIREGUARD
LASWIRECONDUCTOR
LA TRANSMISSIONTOWER
LASBRIDGE
LASNOISE

Format
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
An object of class integer of length 1.
Examples

```r
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)
las2 = filter_poi(las, Classification %in% c(LASGROUND, LASWATER))
print(LASGROUND)
```

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.

catalog_apply() is the core of the lidR package. It drives every single function that can process a LAScatalog. It is flexible and powerful but also complex. catalog_sapply is the same with the option `automerge = TRUE` enforced to simplify the output.

Warning: the LAScatalog processing engine has a mechanism to load buffered data 'on-the-fly' 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

```r
catalog_apply(ctg, FUN, ..., .options = NULL)
catalog_sapply(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 examples.

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 ‘on-the-fly’. 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) to prevent unexpected output with duplicated entries or conflict between values computed twice.

**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. This allows for filtering of buffer points if required.

**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:

```r
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 this 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.

From v3.0.0 if autoread = TRUE the following template is accepted because the engine takes care of the above mentionned steps:

```r
myfun <- function(las, bbox ...) {
  # do something
}
```
Users may have noticed that some lidR functions throw an error when the processing options are inappropriate. For example, some functions need a buffer and thus \texttt{buffer = 0} is forbidden. Users can add the same constraints to protect against inappropriate options. The \texttt{.options} argument is a list that allows users to tune the behavior of the processing engine.

- \texttt{drop\_null = FALSE} Not intended to be used by regular users. The engine does not remove NULL outputs.
- \texttt{need\_buffer = TRUE} the function complains if the buffer is 0.
- \texttt{need\_output\_file = TRUE} the function complains if no output file template is provided.
- \texttt{raster\_alignment = \ldots} the function checks the alignment of the chunks. This option is important if the output is a raster. See below for more details.
- \texttt{automerge = TRUE} by default the engine returns a list with one item per chunk. If \texttt{automerge = TRUE}, it tries to merge the outputs into a single object: a Raster*, a Spatial*, a LAS* similar to other functions of the package. This is a fail-safe option so in the worst case, if the merge fails, the list is returned.
- \texttt{autoread = TRUE}. Introduced in v3.0.0 this option enables to get rid of the first steps of the function i.e \texttt{readLAS()} and if \texttt{(is.empty())}. In this case the function must take two objects as input, first a LAS object and second a Extent from raster.

When the function \texttt{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 \texttt{raster\_alignment}, that can take the resolution of the raster as input, as well as the starting point if needed. The following are accepted:

```r
# check if 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 at (0,10)
raster_alignment = list(res = 20, start = c(0,10))
```

See also \texttt{grid\_metrics} for more details.

**Supported processing options**

Supported processing options for a LAScatalog (in bold). For more details see the \texttt{LAScatalog engine documentation}:

- \texttt{chunk\_size}: How much data is loaded at once.
- \texttt{chunk\_buffer}: Load chunks with a buffer.
- \texttt{chunk\_alignment}: Align the chunks.
- \texttt{progress}: Displays a progress estimate.
- \texttt{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**

```r
# More examples might be available in the official lidR vignettes or

# Example 1: detect all the tree tops over an entire catalog
# (this is basically a reproduction of the 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 <- find_trees(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")
prj <- readLAScatalog(LASfile)
plot(prj)

# 3. Set some processing options.
# For this dummy example, the chunk size is 100 m and the buffer is 10 m
opt_chunk_buffer(prj) <- 10
opt_chunk_size(prj) <- 100 # small because this is a dummy example.
opt_chunk_alignment(prj) <- c(-50, -35) # Align such as it creates 2 chunks only.
opt_select(prj) <- "xyz" # Read only the coordinates.
opt_filter(prj) <- "-keep_first" # Read only first returns.

# 4. Apply a user-defined function to take advantage of the internal engine
catalog_boundaries

Computes the polygon that encloses the points

Description

Computes the polygon that encloses the points. It reads all the files one by one and computes a concave hull using the concaveman function. When all the hulls are computed it updates the LAScatalog to set the true polygons instead of the bounding boxes.
catalog_boundaries

Usage

catalog_boundaries(ctg, concavity = 5, length_threshold = 5)

Arguments

ctg A LAScatalog
concavity numeric. A relative measure of concavity. 1 results in a relatively detailed shape, Infinity results in a convex hull. You can use values lower than 1, but they can produce pretty crazy shapes.
length_threshold numeric. When a segment length is under this threshold, it stops being considered for further detailed processing. Higher values result in simpler shapes.

Value

A LAScatalog with true boundaries

Supported processing options

Supported processing options for more details see the LAScatalog engine documentation:

• chunk size: Not supported, it processes by file.
• chunk buffer: Not supported, it processes by file with no buffer.
• chunk alignment: Not supported, it processes by file.
• progress: Displays a progress estimate.
• output files: Not supported, it returns an R object.
• select: Not supported, it loads XYZ only.
• filter: Read only the points of interest.

Examples

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
ctg <- readLAScatalog(LASfile, filter = "-drop_z_below 0.5")
ctg2 <- catalog_boundaries(ctg, 1, 15)
plot(ctg)
plot(ctg2, add = TRUE)
catalog_makechunks

Subdivide a LAScatalog into chunks

Description

Virtually subdivide a LAScatalog into chunks. This function is an internal function exported to users in version 3.0.0 because it might be useful for some debugging purposes. It might also be useful for some advanced developers. Regular users are not expected to use this function. The chunks are made according to the catalog processing options.

Usage

catalog_makechunks(ctg, realignment = FALSE, plot = opt_progress(ctg))

Arguments

c tg an object of class LAScatalog
realignment FALSE or list(res = x, start = c(y, z)). Sometimes the chunk must be aligned with a raster, for example to ensure the continuity of the output. If the chunk size is 800 and the expected product is a raster with a resolution of 35, 800 and 35 are not compatible and will create 2 different partial pixels on the edges. The realignment option forces the chunk to fit the grid alignment.
plot logical. Displays the chunk pattern.

catalog_intersect

Subset a LAScatalog with a spatial object

Description

Subset a LAScatalog with a spatial object to keep only the tiles of interest. It can be used to select tiles of interest that encompass spatial objects such as Spatial* objects, Raster* objects or sf and sfc objects

Usage

catalog_intersect(ctg, y)

Arguments

c tg A LAScatalog object
y Extent, Raster*, Spatial*, sf, sfc, Extent or bbox objects

Value

A LAScatalog
Value

A list containing objects of class `LAScluster`.

---

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. Most options are easy to understand and to link to the documentation of `LAScatalog` but some need more details. See section 'Details'.

Usage

```r
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_restart(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_independent_files(ctg)
opt_independent_files(ctg) <- value
opt_output_files(ctg)
```
opt_output_files(ctg) <- value
opt_laz_compression(ctg)

opt_laz_compression(ctg) <- value
opt_merge(ctg)

opt_merge(ctg) <- value
opt_select(ctg)

opt_select(ctg) <- value
opt_filter(ctg)

opt_filter(ctg) <- value

Arguments

c tg An object of class LAScatalog

value An appropriate value depending on the expected input.

Details

• opt_restart() automatically sets the chunk option named "drop" in such a way that the engine will restart at a given chunk and skip all previous chunks. Useful for restarting after a crash.

• opt_independent_file() automatically sets the chunk size, chunk buffer and wall-to-wall options to process files that are not spatially related to each other, such as plot inventories.

• opt_laz_compression() automatically modifies the drivers to write LAZ files instead of LAS files.

Examples

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

```r
## Not run:
ctg = readLAScatalog("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 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)
```
catalog_select

Select LAS files manually from a LAScatalog

Description

Select a set of LAS tiles from a LAScatalog interactively using the mouse. This function allows users to subset a LAScatalog by clicking on a map of the file.

Usage

catalog_select(
  ctg,
  mapview = TRUE,
  method = c("subset", "flag_unprocessed", "flag_processed")
)

Arguments

ctg  
A LAScatalog object

mapview  
logical. If FALSE, use R base plot instead of mapview (no pan, no zoom, see also plot)

method  
character. By default selecting tiles that are a subset of the catalog. It is also possible to flag the files to maintain the catalog as a whole but process only a subset of its content. flag_unprocessed enables users to point and click on files that will not be processed. flag_processed enables users to point and click on files that will be processed.

Value

A LAScatalog object

Examples

## Not run:
ctg = readLAScatalog("<Path to a folder containing a set of .las files>")
new_ctg = catalog_select(ctg)

## End(Not run)
classify_ground  

Classify points as 'ground'

Description

Classify points as '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

classify_ground(las, algorithm, last_returns = TRUE)

Arguments

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

- **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.

- **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}.

- **select**: The function will write files equivalent to the original ones. Thus select = "*" and cannot be changed.

- **filter**: Read only points of interest.

**Examples**

```r
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn", filter = "-inside 273450 5274350 273550 5274450")

# Using the Progressive Morphological Filter
# --------------------------------------
# (Parameters chosen mainly for speed)
ws <- seq(3,12, 4)
th <- seq(0.1, 1.5, length.out = length(ws))
las <- classify_ground(las, pmf(ws, th))
#plot(las, color = "Classification")

#' # Using the Cloth Simulation Filter
# --------------------------------------
# (Parameters chosen mainly for speed)
mycsf <- csf(TRUE, 1, 1, time_step = 1)
las <- classify_ground(las, mycsf)
#plot(las, color = "Classification")
```

classify_noise **Classify points as 'noise'**

**Description**

Classify points as 'noise' (outliers) with several possible algorithms. The function updates the attribute Classification of the LAS object. The points classified as 'noise' are assigned a value of 18 according to las specifications.

**Usage**

```r
classify_noise(las, algorithm)
```
classify_noise

Arguments

las An object of class LAS or LAScatalog.
algorithm a noise-segmentation function. lidR has: sor, ivf.

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.
- **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}.
- **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, filter = "-inside 273450 5274350 273550 5274450")

# Add 20 artificial outliers
set.seed(314)
classify_poi

Classify points of interest

Description

Classify points that meet some logical criterion and/or that belong in a region of interest. The function updates the attribute Classification of the LAS object.

Usage

classify_poi(las, class, poi = NULL, roi = NULL, inverse_roi = FALSE, by_reference = FALSE, ...)

Arguments

las
An object of class LAS or LAScatalog.

class
The ASPRS class to attribute to the points that meet the criterion.

poi
a formula of logical predicates. The points that are TRUE will be classified class.

roi
A SpatialPolygons, SpatialPolygonDataFrame from sp or a POLYGON from sf. The points that are in the region of interest delimited by the polygon(s) are classified class.

inverse_roi
bool. Inverses the roi. The points that are outside the polygon(s) are classified class.

by_reference
bool. Updates the classification in place (LAS only).

...Unused
classify_poi

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: A buffer is not need for this function to works thus buffer = 0 and cannot be changed.
- chunk alignment: Align the processed chunks.
- 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\}.
- select: The function will write files equivalent to the original ones. Thus select = "*" and cannot be changed.
- filter: Read only points of interest.

Examples

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

las <- readLAS(LASfile, filter = "-keep_random_fraction 0.1")
lake <- sf::st_read(shp, quiet = TRUE)

# Classifies the points that are NOT in the lake and that are NOT ground points as class 5
poi <- ~Classification != LASGROUND
las <- classify_poi(las, LASHIGHVEGETATION, poi = poi, roi = lake, inverse = TRUE)
```
# Classifies the points that are in the lake as class 9
las <- classify_poi(las, LASWATER, roi = lake, inverse = FALSE)

#plot(las, color = "Classification")

---

clip 

Clip points in regions of interest

Description

Clip points within a given region of interest (ROI) 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. 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

clip_roi(las, geometry, ...)
clip_rectangle(las, xleft, ybottom, xright, ytop, ...)
clip_polygon(las, xpoly, ypoly, ...)
clip_circle(las, xcenter, ycenter, radius, ...)
clip_transect(las, p1, p2, width, xz = FALSE, ...)

Arguments

las An object of class LAS or LAScatalog.
geometry a geometric object. Many types are supported, see section ‘supported geometries’.
... in clip_roi: optional supplementary options (see supported geometries). Unused in other functions
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 disc centers.
ycenter numeric. y coordinates of disc centers.
radius numeric. disc radius or radii.
p1, p2  numeric vectors of length 2 that gives the coordinates of two points that define a transect
width  numeric. width of the transect.
xz    bool. If TRUE the point cloud is reoriented to fit on XZ coordinates

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. If points, a parameter 'radius' must be passed in . . .
- **Polygon, SpatialPolygons, SpatialPolygonsDataFrame, SpatialPoints or SpatialPointsDataFrame** in that case a parameter 'radius' must be passed in . . .
- **SimpleFeature** from sf that consistently contains POINT or POLYGON/MULTIPOLYGON. In case of POINT a parameter 'radius' must be passed in . . .
- **Extent** from package raster
- **bbox** from package sf
- **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 make sense here.
- progress: Displays a progress estimation.
- 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 clip_* are {XLEFT}, {XRIGHT}, {YBOTTOM}, {YTOP}, {ID}, {XCENTER}, {YCENTER}. In addition clip_roi supports any names from the table of attributes of a spatial object given as input such as {PLOTID}, {YEAR}, {SPECIES}, for examples, if these attributes exist. If empty everything is returned into R.
- laz_compression: write las or laz files
- select: The function will write files equivalent to the originals. This option is not respected.
- filter: Read only the points of interest.

Examples

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

# Load the file and clip the region of interest
las = readLAS(LASfile, select = "xyz", filter = "-keep_first")
subset1 = clip_rectangle(las, 684850, 5017850, 684900, 5017900)

# Do not load the file(s), extract only the region of interest
# from a bigger dataset
ctg = readLAScatalog(LASfile, progress = FALSE, filter = "-keep_first")
subset2 = clip_rectangle(ctg, 684850, 5017850, 684900, 5017900)

# Extract all the polygons from a shapefile
f <- system.file("extdata", "lake_polygons_UTM17.shp", package = "lidR")
lakes <- sf::st_read(f, quiet = TRUE)
subset3 <- clip_roi(las, lakes)

# Extract the polygons for a catalog, 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 = clip_roi(ctg, lakes)
plot(new_ctg)

# Extract a transect
p1 <- c(684800, y = 5017800)
p2 <- c(684900, y = 5017900)
tr1 <- clip_transect(las, p1, p2, width = 4)
```

```r
## Not run:
plot(subset1)
```
cloud_metrics

Compute metrics for a cloud of points

description

cloud_metrics 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

cloud_metrics(las, func)

arguments

las An object of class LAS
func formula. An expression to be applied to the point cloud (see example)

value

It returns a list containing the metrics

see also

grid_metrics stdmetrics entropy VCI LAD

Other metrics: grid_metrics(), plot_metrics(), point_metrics(), tree_metrics(), voxel_metrics()
Examples

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

cloud_metrics(lidar, ~max(Z))
cloud_metrics(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 = cloud_metrics(lidar, ~myMetrics(Z, Intensity))

# Predefined metrics
cloud_metrics(lidar, .stdmetrics)
```

concaveman

A very fast 2D concave hull algorithm

Description

A very fast 2D concave hull algorithm for a set of points

Usage

```r
concaveman(x, y = NULL, concavity = 2, length_threshold = 0)
```

Arguments

- `x, y` coordinate vectors of points. This can be specified as two vectors `x` and `y`, a 2-column matrix `x`, a list with two components, etc.
- `concavity` numeric a relative measure of concavity. 1 results in a relatively detailed shape, Infinity results in a convex hull. You can use values lower than 1, but they can produce pretty crazy shapes.
- `length_threshold` numeric. When a segment length is under this threshold, it stops being considered for further detailed processing. Higher values result in simpler shapes.
Details

The algorithm is based on ideas from Park and Oh (2012). A first implementation in JavaScript was proposed by Vladimir Agafonkin in mapbox. This implementation dramatically improved performance over the one stated in the paper using a spatial index. The algorithm was then ported to R by Joël Gombin in the R package concaveman that runs the JavaScript implementation proposed by Vladimir Agafonkin. Later a C++ version of Vladimir Agafonkin’s JavaScript implementation was proposed by Stanislaw Adaszewski in concaveman-cpp. This concaveman function uses Stanislaw Adaszewski’s C++ code making the concaveman algorithm an order of magnitude (up to 50 times) faster than the Javascript version.

References


Examples

```r
x = runif(35)
y = runif(35)
hull <- concaveman(x,y)
plot(x,y, asp = 1)
lines(hull, lwd = 3, col = "red")
```

---

**csf**

*Ground Segmentation Algorithm*

Description

This function is made to be used in classify_ground. 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

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

sloop_smooth logical. When steep slopes exist, set this parameter to TRUE to reduce errors during post-processing.

class_threshold scalar. The distance to the simulated cloth to classify a point cloud into ground and non-ground. The default is 0.5.

cloth_resolution 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.

rigidness 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.

iterations integer. Maximum iterations for simulating cloth. The default value is 500. Usually, there is no need to change this value.

time_step 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


See Also

Other ground segmentation algorithms: pmf()

Examples

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

mycsf <- csf(TRUE, 1, 1, time_step = 1)
las <- classify_ground(las, mycsf)
#plot(las, color = "Classification")

dalponte2016 Individual Tree Segmentation Algorithm

Description

This function is made to be used in segment_trees. 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 find_trees 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 segment_trees to merge the result with the point cloud. However the user can use this as a stand-alone function like this:

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

References

**decimate_points**

Reduce the number of points using several possible algorithms.

**Usage**

decimate_points(las, algorithm)

**Arguments**

- las: An object of class LAS or LAScatalog.
- algorithm: function. An algorithm of point decimation. lidR have: random, homogenize, highest, lowest and random_per_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**: This function guarantee a strict wall-to-wall continuous output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks.
- **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\}.
- **select**: The function will write files equivalent to the original ones. Thus select = "*" and cannot be changed.
- **filter**: Read only points of interest.

**Examples**

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

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

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

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

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

delineate_crowns(  
  las,  
  type = c("convex", "concave", "bbox"),  
  concavity = 3,  
  length_threshold = 0,  
  func = NULL,  
  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.
- **func**: formula. An expression to be applied to each tree. It works like in grid_metrics voxel_metrics or tree_metrics and computes, in addition to the hulls a set of metrics for each tree.
- **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.

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.
- **progress**: Displays a progression estimation.
- **output files**: Supported templates are \{XLEFT\}, \{XRIGHT\}, \{YBOTTOM\}, \{TOP\}, \{XCENTER\}, \{YCENTER\} \{ID\} and, if chunk size is equal to 0 (processing by file), \{ORIGINALFILENAME\}.
- **select**: Load only attributes of interest.
- **filter**: Read only points of interest.

References


Examples

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
poi = "-drop_z_below 0 -inside 481280 3812940 481320 3812980"
las = readLAS(LASfile, select = "xyz0", filter = poi)

# NOTE: This dataset is already segmented

#plot(las, color = "treeID", colorPalette = pastel.colors(200))

# Only the hulls
covex_hulls = delineate_crowns(las)
plot(covex_hulls)
```
# The hulls + some user-defined metrics
convex_hulls = delineate_crowns(las, func = ~list(Zmean = mean(Z)))
convex_hulls

# The bounding box
bbox_hulls = delineate_crowns(las, "bbox")
plot(bbox_hulls)

# With concave hull
concave_hulls = delineate_crowns(las, "concave")
plot(concave_hulls)

spplot(convex_hulls, "ZTOP")
spplot(convex_hulls, "Zmean")

deprecated

## Deprecated functions in lidR

### Description

These functions are provided for compatibility with older versions of lidR but are deprecated since lidR version 3. They will progressively print a message, throw a warning and eventually be removed. The links below point to the documentation of the new names

- lasadd
- lascheck
- lasclip
- lasdetectshape
- lasfiltersurfacepoints
- lasflightline
- lasground
- lasmergespatial
- lasnormalize
- laspulse
- lasrangecorrection
- lasflightline
- lasreoffset
- lasrescale
- lasscanlines
- lassmooth
- lassnags
- lastrees
- lasvoxelize
- sensor_tracking
- tree_detection
- tree_hull

### Usage

- `lascheck(las)`
- `lasclip(las, geometry, ...)`
- `lasclipRectangle(las, xleft, ybottom, xright, ytop, ...)`
- `lasclipPolygon(las, xpoly, ypoly, ...)`
- `lasclipCircle(las, xcenter, ycenter, radius, ...)`
- `lasdetectshape(las, algorithm, attribute = "Shape", filter = NULL)`
- `lasfilter(las, ...)`
- `lasfilterfirst(las)`
- `lasfilterfirstlast(las)`
lasfilterfirstofmany(las)
lasfilterground(las)
lasfilterlast(las)
lasfilternth(las, n)
lasfiltersingle(las)
lasfilterdecimate(las, algorithm)
lasfilterduplicates(las)
lasfiltersurfacepoints(las, res)
lasground(las, algorithm, last_returns = TRUE)
laspulse(las)
lasflightline(las, dt = 30)
lasscanline(las)
lasmergespatial(las, source, attribute = NULL)
lasnormalize(
    las,
    algorithm,
    na.rm = FALSE,
    use_class = c(2L, 9L),
    ...,
    add_lasattribute = FALSE
)
lasunnormalize(las)
lasrange correction(
    las,
    sensor,
    Rs,
    f = 2.3,
    gpstime = "gpstime",
    elevation = "Z"
)
lasrescale(las, xscale, yscale, zscale)
lasreoffset(las, xoffset, yoffset, zoffset)

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

lasunsmooth(las)

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

lastransform(las, CRSobj)

lastrees(las, algorithm, attribute = "treeID", uniqueness = "incremental")

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)

lasvoxelize(las, res)

sensor_tracking(  
    las,
    interval = 0.5,
    pmin = 50,
    extra_check = TRUE,
    thin_pulse_with_time = 0.001
)

tree_detection(las, algorithm)

tree_hulls(  

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

hexbin_metrics(...)

Arguments

las See the new functions that replace the old ones
group See the new functions that replace the old ones
... See the new functions that replace the old ones
xleft, ybottom, xright, ytop See the new functions that replace the old ones
xpoly, ypoly See the new functions that replace the old ones
xcenter, ycenter, radius See the new functions that replace the old ones
algorithm See the new functions that replace the old ones
attribute See the new functions that replace the old ones
filter See the new functions that replace the old ones
n, res, dt See the new functions that replace the old ones
last_returns See the new functions that replace the old ones
source See the new functions that replace the old ones
na.rm, use_class, add_lasattribute See the new functions that replace the old ones
sensor, Rs, f, gpstime, elevation See the new functions that replace the old ones
xscale, yscale, zscale, xshear, yshear, zshear See the new functions that replace the old ones
size, method, shape, sigma See the new functions that replace the old ones
CRSobj See the new functions that replace the old ones
uniqueness See the new functions that replace the old ones
x, name, desc, type, offset, scale, NA_value See the new functions that replace the old ones
interval, pmin, extra_check, thin_pulse_with_time See the new functions that replace the old ones
concavity, length_threshold, func See the new functions that replace the old ones
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**

```r
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 = clip_polygon(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))
```
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


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)

---

## extent,LAS-method

### Extent

**Description**

Returns an Extent object of a LAS.

**Usage**

```r
## 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`
Predefined point of interest filters

Description
Select only some returns

Usage
- filter_first(las)
- filter_firstlast(las)
- filter_firstofmany(las)
- filter_ground(las)
- filter_last(las)
- filter_nth(las, n)
- filter_single(las)

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

Details
- filter_first Select only the first returns.
- filter_firstlast Select only the first and last returns.
- filter_ground Select only the returns classified as ground according to LAS specification.
- filter_last Select only the last returns i.e. the last returns and the single returns.
- filter_nth Select the returns from their position in the return sequence.
- filter_firstofmany Select only the first returns from pulses which returned multiple points.
- filter_single Select only the returns that return only one point.

Value
An object of class LAS
See Also
Other filters: `filter_duplicates()`, `filter_poi()`, `filter_surfacepoints()`

Examples

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

firstReturns = filter_first(lidar)
groundReturns = filter_ground(lidar)
```

filter_duplicates

Filter duplicated points

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

Usage

```r
filter_duplicates(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 guarantees a strict wall-to-wall continuous output. The buffer option is not considered.
- **chunk alignment**: Align the processed chunks.
- **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}.
- **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 filters: filter_poi(), filter_surfacepoints(), filters

filter_poi                     Filter points of interest with matching conditions

Description

Filter points of interest (POI) from a LAS object where conditions are true.

Usage

filter_poi(las, ...)

Arguments

- **las**: An object of class LAS
- **...**: Logical predicates. Multiple conditions are combined with ’&’ or ‘,’
filter_surfacepoints

Value
An object of class LAS

See Also
Other filters: filter_duplicates(), filter_surfacepoints(), filters

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

# Select the first returns classified as ground
firstground = filter_poi(lidar, Classification == 2L & ReturnNumber == 1L)

# Multiple arguments are equivalent to &
firstground = filter_poi(lidar, Classification == 2L, ReturnNumber == 1L)

# Multiple criteria
first_or_ground = filter_poi(lidar, Classification == 2L | ReturnNumber == 1L)

filter_surfacepoints Filter the surface points

Description
This function is superseded by the algorithm highest usable in decimate_points

Usage
filter_surfacepoints(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 guarantees a strict wall-to-wall continuous output. The buffer option is not considered.
• chunk alignment: Align the processed chunks.
• 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\}.
• 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 filters: `filter_duplicates()`, `filter_poi()`, `filters`

---

find_localmaxima  
**Local Maximum Filter**

Description
Generic local maximum filter. For individual tree detection use `find_trees` with the lmf algorithm that is more adequate for ITD. This function is a more generic method for multiple purposes other than tree segmentation. This function is natively parallelized with OpenMP.

Usage

```r
find_localmaxima(las, w, filter = NULL)
```

Arguments

- `las`: An object of class LAS
- `w`: numeric. Window shape. 1 number for the diameter of a disc, 2 numbers for a rectangle (width, height), 3 numbers for an oriented rectangle (width, height, angle). The angle must be in radians.
- `filter`: formula. Memory efficient way to work only with a subset of the data without creating a copy of the data.

Value

`SpatialPointsDataFrame` with attributes from the corresponding point in the LAS object.
**find_trees**

**Examples**

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzi", filter = "-drop_z_below 0 -keep_random_fraction 0.5")

# Using a 20x5 rectangle with a 45 degrees angle.
# This won't find the tree properly in the general case
# but may find some oriented structure.
lm = find_localmaxima(las, c(20, 5, pi/4))
plot(lm)
```

---

**find_trees**  
*Individual tree detection*

**Description**

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

**Usage**

```r
find_trees(las, algorithm, uniqueness = "incremental")
```

**Arguments**

- `las`: An object of class LAS or LAStoolkitCatalog. 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`.
- `uniqueness`: character. A method to compute a unique ID. Can be 'incremental', 'gpstime' or 'bitmerge'. See section 'Uniqueness'. This feature must be considered as 'experimental'.

**Value**

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

**Uniqueness**

By default the tree IDs are numbered from 1 to n, n being the number of trees found. The problem with such incremental numbering is that, while it ensures a unique ID is assigned for each tree in a given point-cloud, it also guarantees duplication of tree IDs in different tiles or chunks when processing a `LAStoolkitCatalog`. This is because each chunk/file is processed independently of the others and potentially in parallel on different computers. Thus, the index always restarts at 1 on each chunk/file or chunk. Worse, in a tree segmentation process, a tree that is located exactly between 2 chunks/files will have two different IDs for its two halves.
This is why we introduced some uniqueness strategies that are all imperfect and that should be seen as experimental. Please report any troubleshooting. Using a uniqueness-safe strategy ensures that trees from different files will not share the same IDs. Moreover, it also means that two halves of a tree on the edge of a processing chunk will be assigned the same ID.

**incremental** Number from 0 to n. This method does not ensure uniqueness of the IDs. This is the legacy method.

**gpstime** This method uses the gpstime of the highest point of a tree (apex) to create a unique ID. This ID is not an integer but a 64-bit decimal number, which is suboptimal but at least it is expected to be unique if the gpstime attribute is consistent across files. If inconsistencies with gpstime are reported (for example gpstime records the week time and was reset to 0 in a coverage that takes more than a week to complete), there is a (low) probability of getting ID attribution errors.

**bitmerge** This method uses the XY coordinates of the highest point (apex) of a tree to create a single 64-bit number with a bitwise operation. First, XY coordinates are converted to 32-bit integers using the scales and offsets of the point cloud. For example, if the apex is at (10.32, 25.64) with a scale factor of 0.01 and an offset of 0, the 32-bit integer coordinates are X = 1032 and Y = 2564. Their binary representations are, respectively, (here displayed as 16 bits) 0000010000001000 and 0000101000000100. X is shifted by 32 bits and becomes a 64-bit integer. Y is kept as-is and the binary representations are unionized into a 64-bit integer like (here displayed as 32 bit) 00000100000010000000101000000100 that is guaranteed to be unique. However R does not support 64-bit integers. The previous steps are done at C++ level and the 64-bit binary representation is reinterpreted into a 64-bit decimal number to be returned in R. The IDs thus generated are somewhat weird. For example, the tree ID 00000100000010000000101000000100 which is 67635716 if interpreted as an integer becomes 3.34164837074751323479078607289E-316 if interpreted as a decimal number. This is far from optimal but at least it is guaranteed to be unique if all files have the same offsets and scale factors.

All the proposed options are suboptimal because they either do not guarantee uniqueness in all cases (inconsistencies in the collection of files), or they imply that IDs are based on non-integers or meaningless numbers. But at least it works and deals with some of the limitations of R.

**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.
- **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}.
- **select**: Load only attributes of interest.
- **filter**: Read only points of interest.
Examples

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-inside 481250 3812980 481300 3813030")
ttops <- find_trees(las, lmf(ws = 5))
#x = plot(las)
#add_treetops3d(x, ttops)
```

---

**gap_fraction_profile**  
*Gap fraction profile*

**Description**

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

**Usage**

```r
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**


**See Also**

LAD
Examples

```r
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")
```

Description

This function is made to be used in `track_sensor`. It implements an algorithm from Gatziolis and McGaughey 2019 (see reference) for sensor tracking using multiple returns to estimate the positioning of the sensor by computing the intersection in space of the lines passing through the first and last returns.

Usage

```r
Gatziolis2019(SEGLENFactor = 1.0059, AngleFactor = 0.8824, deltaT = 0.5)
```

Arguments

- `SEGLENFactor`: scalar. Weighting factor for the distance b/w 1st and last pulse returns
- `AngleFactor`: scalar. Weighting factor for view angle of mother pulse of a return
- `deltaT`: scalar. TimeBlock duration (in seconds)

Details

In the original paper, two steps are described: (1) closest point approach (CPA) and (2) cubic spline fitting. Technically, the cubic spline fitting step is a post-processing step and is not included in this algorithm.

The source code of the algorithm is a slight modification of the original source code provided with the paper to fit with the lidR package.

Author(s)

Demetrios Gatziolis and Jean-Romain Roussel

References

Examples

```
# A valid file properly populated
Lasfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)
flightlines <- track_sensor(las, Gatzios2019())

plot(las@header)
plot(flightlines, add = TRUE)
```

---

**grid_canopy** *Digital Surface Model*

Description

Creates a digital surface model (DSM) using several possible algorithms. If the user provides a normalised point cloud, the output is indeed a canopy height model (CHM).

Usage

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>las</td>
<td>An object of class LAS or LAScatalog.</td>
</tr>
<tr>
<td>res</td>
<td>numeric. The resolution of the output Raster. Can optionally be a RasterLayer. In that case the RasterLayer is used as the layout.</td>
</tr>
<tr>
<td>algorithm</td>
<td>function. A function that implements an algorithm to compute a digital surface model. lidR implements p2r, dsmin, pitfree (see respective documentation and examples).</td>
</tr>
</tbody>
</table>

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 of .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:

- **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.
- **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}.
- **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

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, filter = "-inside 481280 3812940 481330 3812990")
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**

*Map the pulse or point density*
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:

- **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.
- **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\}.

• **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**

```r
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile, filter = "-inside 684800 5017800 684900 5017900")

d <- grid_density(las, 5)
plot(d)

las <- retrieve_pulses(las)
d <- grid_density(las)
plot(d)
```

**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**

```r
grid_metrics(
  las,
  func,
  res = 20,
  start = c(0, 0),
  filter = NULL,
  by_echo = "all"
)
```

**Arguments**

- `las` An object of class LAS or LASCatalog.
- `func` formula. An expression to be applied to each cell (see section "Parameter func").
grid_metrics numeric. The resolution of the output Raster. Can optionally be a RasterLayer. In that case the RasterLayer is used as the layout.

start vector of x and y coordinates for the reference raster. Default is (0,0) meaning that the grid aligns on (0,0).

filter formula of logical predicates. Enables the function to run only on points of interest in an optimized way. See examples.

by_echo characters. The metrics are computed multiples times for different echo types. Can be one or more of "all", "first", "intermediate", "lastofmany", "single", "multiple". See examples. Default is "all" meaning that it computes metrics with all points provided.

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.

\[
f = \text{function}(x) \{\text{list}(\text{mean} = \text{mean}(x), \text{max} = \text{max}(x))\}
\]

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

\[
\text{grid\_metrics}(\text{las}, \sim f(\text{Z}), \text{res} = 20)
\]
\[
\text{grid\_metrics}(\text{las}, \sim f(\text{Intensity}), \text{res} = 20)
\]

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:

- **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.
- **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}.
- **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**

Other metrics: `cloud_metrics()`, `plot_metrics()`, `point_metrics()`, `tree_metrics()`, `voxel_metrics()`

**Examples**

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

# === Using all points ===

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

# 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 = col)
#plot(metrics, "zwimean", col = col)
#plot(metrics, "zimean", col = col)

# === With point filters ===

# Compute using only some points: basic
first = filter_poi(las, ReturnNumber == 1)
metrics = grid_metrics(first, ~mean(Z), 20)

# Compute using only some points: optimized
# faster and uses less memory. No intermediate object
metrics = grid_metrics(las, ~mean(Z), 20, filter = ~ReturnNumber == 1)

# Compute using only some points: best
# ~50% faster and uses ~10x less memory
las = readLAS(LASfile, filter = "-keep_first")
metrics = grid_metrics(las, ~mean(Z), 20)

# === Split by echo type ===

func = ~list(avgI = mean(Intensity))
echo = c("all", "first","multiple")
metrics <- grid_metrics(las, func , 20, by_echo = echo)
plot(metrics, col = heat.colors(25))

---

**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" and "water" (Classification = 2 and 9, respectively, 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. A buffer around the region of interest is always recommended to avoid edge effects.

**Usage**

grid_terrain(
  las,
  res = 1,
  algorithm,
..., 
  keep_lowest = FALSE, 
  full_raster = FALSE, 
  use_class = c(2L, 9L), 
  Wdegenerated = TRUE, 
  is_concave = 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).

**...**
Unused

**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.

**full_raster**
logical. By default the interpolation is made only within the convex hull of the point cloud. This prevents meaningless interpolations where there is no data. If TRUE, each pixel of the raster is interpolated.

**use_class**
integer vector. By default the terrain is computed by using ground points (class 2) and water points (class 9).

**Wdegenerated**
logical. The function always checks and removes degenerated ground points for computing the DTM to avoid unexpected behaviours, such as infinite elevation. If TRUE, a warning is thrown to alert users to the presence of degenerated ground points.

**is_concave**
boolean. By default the function tries to compute a DTM that has the same shape as the point cloud by interpolating only in the convex hull of the points. If the point cloud is concave this may lead to weird values where there are no points. Use is_concave = TRUE to use a concave hull. This is more computationally demanding. It uses concaveman internally.

**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:

- **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.
- **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\}.
- **select**: The function ‘knows’ what should be loaded and this option is not considered.
- **filter**: Read only the points of interest.

**Examples**

```r
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile, filter = "-inside 273450 5274350 273550 5274450")
#plot(las)

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

## Not run:
dtm3 = grid_terrain(las, algorithm = kriging(k = 10L))

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

## End(Not run)
```
homogenize

Point Cloud Decimation Algorithm

Description

This function is made to be used in `decimate_points`. 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/m$^2$ 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

```r
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 `retrieve_pulses` first)

See Also

Other point cloud decimation algorithms: `maxima`, `random_per_voxel()`, `random()`

Examples

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

# Select points randomly to reach an homogeneous density of 1
thinned <- decimate_points(las, homogenize(1,5))
plot(grid_density(thinned, 10))
```
**interpret_waveform**  
*Convert full waveform data into a regular point cloud*

### Description

Full waveform can be difficult to manipulate and visualize in R. This function converts a LAS object with full waveform data into a regular point cloud. Each waveform record becomes a point with XYZ coordinates and an amplitude (units: volts) and an ID that records each original pulse. Notice that this has the effect of drastically inflating the size of the object in memory, which is likely already very large.

### Usage

```r
interpret_waveform(las)
```

### Arguments

- `las`
  - An object of class LAS with full waveform data

### Value

An object of class LAS 1.2 format 0 with one point per records

### Full waveform

With most recent versions of the ‘rlas’ package, full waveform (FWF) can be read and ‘lidR’ provides some compatible functions. However, the support of FWF is still a work-in-progress in the ‘rlas’ package. How it is read, interpreted and represented in R may change. Consequently, tools provided by ‘lidR’ may also change until the support of FWF becomes mature and stable in ‘rlas’. See also `rlas::read.las`.

Remember that FWF represents an insanely huge amount of data. In terms of memory it is like having between 10 to 100 times more points. Consequently, loading FWF data in R should be restricted to relatively small point clouds.

### Examples

```r
## Not run:
LASfile <- system.file("extdata", "fwf.laz", package="rlas")
fwf <- readLAS(LASfile)
las <- interpret_waveform(fwf)
x <- plot(fwf, size = 3, colorPalette = "red")
plot(las, color = "Amplitude", bg = "white", add = x, size = 2)
## End(Not run)
```
is empty tests if a LAS object is a point cloud with 0 points.
is overlapping tests if a LAScatalog has overlapping tiles.
is indexed tests if the points of a LAScatalog are indexed with .lax files.
is algorithm tests if an object is an algorithm of the lidR package.
is parallelised tests if an algorithm of the lidR package is natively parallelised with OpenMP. Returns TRUE if the algorithm is at least partially parallelised i.e. if some portion of the code is computed in parallel.

Usage

is.empty(las)
is.overlapping(catalog)
is.indexed(catalog)
is.algorithm(x)
is.parallelised(algorithm)

Arguments

las A LAS object.
catalog A LAScatalog object.
x Any R object.
algorithm An algorithm object.

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)

f <- lmf(2)
is.parallelised(f)
ivf

---

### Noise Segmentation Algorithm

This function is made to be used in `classify_noise`. It implements an algorithm for outliers (noise) segmentation based on isolated voxels filter (IVF). It is similar to `lasnoise` from `lastools`. The algorithm finds points that have only a few other points in their surrounding $3 \times 3 \times 3 = 27$ voxels.

#### Usage

```r
ivf(res = 5, n = 6)
```

#### Arguments

- `res` numeric. Resolution of the voxels
- `n` integer. The maximal number of 'other points' in the 27 voxels

#### See Also

Other noise segmentation algorithms: `sor()`

#### Examples

```r
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, filter = "-inside 273450 5274350 273550 5274450")

# Add some artificial outliers
set.seed(314)
id = round(runif(20, 0, npoints(las)))
set.seed(42)
err = runif(20, -50, 50)
las$Z[id] = las$Z[id] + err

las <- classify_noise(las, ivf(5,2))
```
knnidw  

Spatial Interpolation Algorithm

Description

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

Usage

```r
knnidw(k = 10, p = 2, rmax = 50)
```

Arguments

- `k`  
- `p`  
- `rmax`  
  numeric. Maximum radius where to search for knn. Default 50.

See Also

Other spatial interpolation algorithms: `kriging()`, `tin()`

Examples

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

#plot(sas)
dtm = grid_terrain(sas, algorithm = knnidw(k = 6L, p = 2))

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

---

kriging  

Spatial Interpolation Algorithm

Description

This function is made to be used in `grid_terrain` or `classify_ground`. 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 `gstat`. 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

```r
## Not run:
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)
## End(Not run)
```

---

**LAD**

*Leaf area density*

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

- **z**: vector of positive z coordinates
- **dz**: numeric. The thickness of the layers used (height bin)
- **k**: numeric. is the extinction coefficient
- **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 \([z, z+dz]\) (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


See Also
gap_fraction_profile

Examples

```r
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, index = NULL)`

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.
- `index`: list with two elements list(sensor = 0L,index = 0L). See spatial indexing.
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)
- gpstime (numeric)
- Intensity (integer)
- ReturnNumber (integer)
- NumberOfReturns (integer)
- ScanDirectionFlag (integer)
- EdgeOfFlightline (integer)
- Classification (integer)
- Synthetic_flag (logical)
- Keypoint_flag (logical)
- Withheld_flag (logical)
- ScanAngleRank (integer)
- ScanAngle (numeric)
- UserData (integer)
- PointSourceID (integer)
- R, G, B (integer)
- NIR (integer)

Value

An object of class LAS

Functions

- LAS: creates objects of class LAS. The original data is updated by reference to quantize the coordinates according to the scale factor of the header if no header is provided. In this case the scale factor is set to 0.001

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
- index list. See spatial indexing.
Extends

Class Spatial, directly.

See Also

readLAS

Examples

# Read a las/laz file
LASfile <- system.file("extdata", "example.laz", package="rlas")
las <- readLAS(LASfile)
las

# Creation of a LAS object out of external data
data <- data.frame(X = runif(100, 0, 100),
                   Y = runif(100, 0, 100),
                   Z = runif(100, 0, 20))

# 'data' has many decimal digits
data

# Create a default header and quantize *by reference*
# the coordinates to fit with offset and scale factors
cloud <- LAS(data)

# 'data' has been updated and coordinates were quantized
data
cloud

# Be careful when providing a header the function assumes that
# it corresponds to the data and won't quantize the coordinates
data <- data.frame(X = runif(100, 0, 100),
                   Y = runif(100, 0, 100),
                   Z = runif(100, 0, 20))

header <- las@header

# This works but triggers warnings and creates an invalid LAS object
cloud <- LAS(data, header)

las_check(cloud)

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 `readLAScatalog` 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 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 or LAZ 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 lax 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).
- `index` list. See `spatial indexing`.

**Processing options**

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

- `progress`: boolean. Display a progress bar and a chart of progress. Default is TRUE. Progress estimation can be enhanced by installing the package `progress`. See `opt_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. See opt_stop_early.

• **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. See opt_wall_to_wall

**Chunk options**

The slot @chunk_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. With big chunks 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. See opt_chunk_size.

• **buffer**: numeric. Each chunk can be read with an extra buffer around it to ensure there are no edge effects between two independent chunks and that the output is continuous. This is mandatory for some algorithms. Default is 30. See opt_chunk_buffer.

• **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. See opt_chunk_alignment.

• **drop**: integers. A vector of integers that specify the IDs of the chunks that should not be created. This is designed to enable to restart a computation that failed without reprocessing everything. See opt_restart. Technically this options may be used for partial processing of a collection but should not. Partial processing is already a feature of the engine. See this vignette

**Output options**

The slot @output_options contains a list of options that determine how chunks (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 filename template 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. See `opt_output_files`.

- **drivers**: list. This contains all the drivers required to seamlessly write Raster*, Spatial*, sf, and LAS objects. It is recommended that only advanced users change this option. A dedicated page describes the drivers in lidR-LAScatalog-drivers.

- **merge**: boolean. Multiple objects are merged into a single object at the end of the processing.

**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 option `select`. Usually this option is not respected because each function knows which data must be loaded or not. This is documented in each function. See `opt_select`.

- **filter**: string. The option `filter`. See `opt_filter`.

**Examples**

```r
## Not run:
# Build a catalog
c tg <- readLAScatalog("folder/to/las/files/")

# Set some options
opt_filter(ctg) <- 

# 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_chunk_size(ctg) <- 500

# Sometimes the output is likely to be very large
# e.g. large coverage and small resolution
dtm <- grid_terrain(ctg, 1, tin())

# In that case it is advisable to write the output(s) to files
opt_output_files(ctg) <- 

# 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) <- 
```


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 <- normalize_height(ctg, tin())

# The user has access to the catalog engine through the function catalog_apply
output <- catalog_apply(ctg, FUN, ...)
## End(Not run)

LASHeader

Create a LASHeader object

Description

Creates a LASHeader object either from a raw list containing all the elements named according to the rlas package or creates a header from a data.frame or data.table containing a point cloud. In the latter case it will generate a header according to the data using rlas::header_create(). It will guess the LAS file format, the point data format, and initialize the scale factors and offsets, but these may not suit a user’s needs. Users are advised to manually modify the results to fit their specific needs.

Usage

LASHeader(data = list())

Arguments

data a list containing the data from the header of a LAS file. Can also be a data.frame or data.table

Value

An object of class LASHeader

Examples

data = data.frame(X = c(339002.889, 339002.983, 339002.918),
                 Y = c(5248000.515, 5248000.478, 5248000.318),
                 Z = c(975.589, 974.778, 974.471),
                 gpstime = c(269347.28141, 269347.28142, 269347.28143),
                 Intensity = c(82L, 54L, 27L),
                 ReturnNumber = c(1L, 1L, 2L),
                 NumberOfReturns = c(1L, 1L, 2L),
                 ScanDirectionFlag = c(1L, 1L, 1L),
                 lidarClass = c(1L, 1L, 1L),
                 scanLine = c(1L, 1L, 1L),
                 rangeValue = c(0L, 0L, 0L),
                 azimuth = c(0L, 0L, 0L),
                 rangeBin = c(200, 200, 200),
                 azimuthBin = c(200, 200, 200),
                 intensityPerClass = c(1L, 1L, 1L),
                 returnNumberPerClass = c(1L, 1L, 1L),
                 scanDirectionFlagPerClass = c(1L, 1L, 1L),
                 rangeValuePerClass = c(0L, 0L, 0L),
                 azimuthPerClass = c(0L, 0L, 0L),
                 rangeBinPerClass = c(200, 200, 200),
                 azimuthBinPerClass = c(200, 200, 200))
header = LASheader(data)
header

# XYZ values are given with 3 decimals. This was not inferred by the
# function so we changed it manually
# (Note: from package rlas 1.4.1 this is now inferred properly in most cases)
header@PHB[\"X scale factor\"] <- 0.001
header@PHB[\"Y scale factor\"] <- 0.001
header@PHB[\"Z scale factor\"] <- 0.001

# Record an EPSG code
epsg(header) <- 32618
header

las <- LAS(data, header)
las

# The function inferred a LAS 1.2 format 1 which is correct
# Upgrade to LAS 1.4 for the example
header@VLR <- list() # Erase VLR previously written
header@PHB[\"Global Encoding\"][[\"WKT\"]]<- TRUE
header@PHB[\"Version Minor\"] <- 4L
header@PHB[\"Header Size\"] <- 375L
header@PHB[\"Offset to point data\"] <- 375L
wkt(header) <- sf::st_crs("EPSG:32618")$wkt
header
las1.4 <- LAS(data, header)
las1.4

---

**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
- **EVLR** list. Represents the Extended Variable Length Records
las_check

Inspect a LAS object

Description

Performs a deep inspection of a LAS or LAStocatalog 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 LAStocatalog 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

las_check(las, print = TRUE, ...)

Arguments

las An object of class LAS or LAStocatalog.
print logical. By default, prints a report and returns a list invisibly. If print = FALSE the functions returns a list visibly and do not print the report.
... Use deep = TRUE on a LAStocatalog only. Instead of a shallow inspection it reads all the files and performs a deep inspection.

Value

A list with three elements named message, warnings and errors. This list is returned invisibly if print = TRUE. If deep = TRUE a nested list is returned with one element per file.

See Also

Other las utilities: las Utilities
Examples

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

Description

Tools to manipulate LAS objects maintaining compliance with ASPRS specification

Usage

las_rescale(las, xscale, yscale, zscale)
las_reoffset(las, xoffset, yoffset, zoffset)
las_quantize(las, by_reference = TRUE)
las_update(las)

quantize(x, scale, offset, by_reference = TRUE, ...)

is.quantized(x, scale, offset, ...)

count_not_quantized(x, scale, offset)

storable_coordinate_range(scale, offset)

Arguments

las An object of class LAS

xscale, yscale, zscale

scalar. Can be missing if not relevant.

xoffset, yoffset, zoffset

scalar. Can be missing if not relevant.

by_reference bool. Update the data in place without allocating new memory.

x numeric. Coordinates vector

scale, offset

scalar, scale and offset

... Unused.
Details

In the specification of the LAS format the coordinates are expected to be given with a certain precision e.g. 0.01 for a millimeter precision (or millifeet), meaning that a file records e.g. 123.46 not 123.45678. Also, coordinates are stored as integers. This is made possible with a scale and offset factor. For example, 123.46 with an offset of 100 and a scale factor of 0.01 is actually stored as \((123.46 - 100)/0.01 = 2346\). Storing 123.45678 with a scale factor of 0.01 and an offset of 100 is invalid because it does not convert to an integer: \((123.45678-100)/0.01 = 2345.678\). Having an invalid LAS object may be critical in some lidR applications. When writing into a LAS file, users will loose the extra precision without warning and some algorithms in lidR use the integer conversion to make integer-based computation and thus speed-up some algorithms and use less memory. Creation of an invalid LAS object may cause problems and incorrect outputs.

See Also

Other las utilities: \texttt{las\_check()}

Examples

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

# Manual modification of the coordinates (e.g. rotation, re-alignment, ...)
las@data\$X <- las@data\$X + 2/3
las@data\$Y <- las@data\$Y - 5/3

# The point cloud is no longer valid
las_check(las)

# It is important to fix that
las_quantize(las)

# Now the file is almost valid
las_check(las)

# Update the object to set up-to-date header data
las <- las_update(las)
las_check(las)

# In practice the above code is not useful for regular users because the operators
# $\leftarrow$ already perform such operations on-the-fly. Thus the following
# syntax must be preferred and returns valid objects. Previous tools
# were only intended to be used in very specific cases.
las$X <- las$X + 2/3
las$Y <- las$Y - 5/3

# Rescale and reoffset recompute the coordinates with
# new scales and offsets according to LAS specification
las <- las_rescale(las, xscale = 0.01, yscale = 0.01)
las <- las_reoffset(las, xoffset = 300000, yoffset = 5248000)
```
**Description**

This function is made to be used in `segment_trees`. 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**

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

**Arguments**

- `dt1`: numeric. Threshold number 1. See reference page 79 in Li et al. (2012). Default is 1.5.
- `R`: numeric. Search radius. See page 79 in Li et al. (2012). Default is 2. If `R = 0` all the points are automatically considered as local maxima and the search step is skipped (much faster).
- `Zu`: numeric. If point elevation is greater than `Zu`, `dt2` is used, otherwise `dt1` is used. See page 79 in Li et al. (2012). Default is 15.
- `speed_up`: 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**


**See Also**

Other individual tree segmentation algorithms: `dalponte2016()`, `silva2016()`, `watershed()`

**Examples**

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

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

This document explains how objects are written on disk when processing a LAScatalog. As mentioned in LAScatalog-class, users can set a templated filename to store the outputs on disk instead of in R memory. By default LAS objects are stored in .las files with writeLAS, Raster* objects are stored in .tif files with writeRaster, Spatial* objects are stored in .shp files with st_write, data.frame objects are stored in .csv files with fwrite, and other objects are not supported. However, users can modify all these default settings and even add new drivers. This manual page explain how. One may also refer to some unofficial documentation here or here.

Generic form of a driver

A driver is stored in the @output_options slot of a LAScatalog. It is a list that contains:

- **write** A function that receives an object and a path, and writes the object into a file using the path. The function can also have extra options.
- **extension** A string that gives the file extension.
- **object** A string that gives the name of the argument used to pass the object to write in the function used to write the object.
- **path** A string that gives the name of the argument used to pass the path of the file to write in the function used to write the object.
- **param** A labelled list of extra parameters for the function used to write the object.

For example, the driver to write a Raster* is

```r
list(
  write = raster::writeRaster,
  extension = "tif",
  object = "x",
  path = "filename",
  param = list(format = "GTiff"))
```

And the driver to write a LAS is

```r
list(
  write = lidR::writeLAS,
  extension = "las",
  object = "las",
  path = "file",
  param = list())
```
Modify a driver (1/2)

Users can modify the drivers to write different file types than the default. For example, to write in GeoPackage instead of shapefile, one must change the Spatial driver:

```
ctg@output_options$drivers$Spatial$extension <- "gpkg"
```

To write in .grd files instead of .tif files one must change the Raster driver:

```
ctg@output_options$drivers$Raster$extension <- "grd"
ctg@output_options$drivers$Raster$param$format <- "raster"
```

To write in .laz files instead of .las files one must change the LAS driver:

```
ctg@output_options$drivers$LAS$extension <- "laz"
```

Add a new driver

The drivers allow LAS, Spatial*, Raster* and data.frame objects to be written. When using the engine (catalog_apply) to build new tools, users may need to be able to write other objects such as a list. To do that users need to add a list element into the output_options:

```
ctg@output_options$drivers$list = list(
  write = base::saveRDS,
  object = "object",
  path = "file",
  extension = "rds",
  param = list(compress = TRUE))
```

The LAScatalog now has a new driver capable of writing a list.

Modify a driver (2/2)

It is also possible to completely overwrite an existing driver. By default SpatialPointsDataFrame objects are written into ESRI shapefiles with st_write. writeOGR can write into other file types, such as GeoPackage or GeoJSON and even as SQLite database objects. But it cannot add data into an existing SQLite database. Let's create our own driver for a SpatialPointsDataFrame. First we need a function able to write and append a SpatialPointsDataFrame into a SQLite database from the object and the path.

```
dbWrite_SpatialPointsDataFrame = function(x, path, name)
{
  x <- as.data.frame(x)
  con <- RSQLite::dbConnect(RSQLite::SQLite(), path)
  RSQLite::dbWriteTable(con, name, x, append = TRUE)
  RSQLite::dbDisconnect(con)
}
```

Then we create the driver. User-defined drivers supersede default drivers:
ctg@output_options$drivers$SpatialPointsDataFrame = list(
  write = dbWrite_SpatialPointsDataFrame,
  extension = ".sqlite",
  object = "x",
  path = "path",
  param = list(name = "layernname"))

Then to be sure that we do not write several .sqlite files, we don’t use templated filename.

opt_output_files(ctg) <- paste0(tempdir(), "/mysqlitefile")

And all the SpatialPointsDataFrame will be appended in a single database.

---

**lidR-parallelism**  
*Parallel computation in lidR*

**Description**

This document explain how to process point clouds taking advantage of parallel processing in the lidR package. The lidR package has two levels of parallelism, which is why it is difficult to understand how it works. This page aims to provide users with a clear overview of how to take advantage of multicore processing even if they are not comfortable with the parallelism concept.

**Algorithm-based parallelism**

When processing a point cloud we are applying an algorithm on data. This algorithm may or may not be natively parallel. In lidR some algorithms are fully computed in parallel, but some are not because they are not parallelizable, while some are only partially parallelized. It means that some portions of the code are computed in parallel and some are not. When an algorithm is natively parallel in lidR it is always a C++ based parallelization with OpenMP. The advantage is that the computation is faster without any consequence for memory usage because the memory is shared between the processors. In short, algorithm-based parallelism provides a significant gain without any cost for your R session and your system (but obviously there is a greater workload for the processors). By default lidR uses half of your cores but you can control this with `set_lidr_threads`. For example, the lmf algorithm is natively parallel. The following code is computed in parallel:

```r
las <- readLAS("file.las")
tops <- tree_detection(las, lmf(2))
```

However, as stated above, not all algorithms are parallelized or even parallelizable. For example, li2012 is not parallelized. The following code is computed in serial:

```r
las <- readLAS("file.las")
dtm <- segment_trees(las, li2012())
```

To know which algorithms are parallelized users can refer to the documentation or use the function `is.parallelised`.

```r
is.parallelised(lmf(2)) #> TRUE
is.parallelised(li2012()) #> FALSE
```
chunk-based parallelism

When processing a LAScatalog, the internal engine splits the dataset into chunks and each chunk is read and processed sequentially in a loop. But actually this loop can be parallelized with the future package. By default the chunks are processed sequentially, but they can be processed in parallel by registering an evaluation strategy. For example, the following code is evaluated sequentially:

```r
c tg <- readLAScatalog("folder/")
out <- grid_metrics(ctg, mean(Z))
```

But this one is evaluated in parallel with two cores:

```r
library(future)
plan(multisession, workers = 2L)
c tg <- readLAScatalog("folder/")
out <- grid_metrics(ctg, mean(Z))
```

With chunk-based parallelism any algorithm can be parallelized by processing several subsets of a dataset. However, there is a strong cost associated with this type of parallelism. When processing several chunks at a time, the computer needs to load the corresponding point clouds. Assuming the user processes one square kilometer chunks in parallel with 4 cores, then 4 chunks are loaded in the computer memory. This may be too much and the speed-up is not guaranteed since there is some overhead involved in reading several files at a time. Once this point is understood, chunk-based parallelism is very powerful since all the algorithms can be parallelized whether or not they are natively parallel. It also allows to parallelize the computation on several machines on the network or to work on a HPC.

Nested parallelism - part 1

Previous sections stated that some algorithms are natively parallel, such as lmf, and some are not, such as li2012. Anyway, users can split the dataset into chunks to process them simultaneously with the LAScatalog processing engine. Let’s assume that the user’s computer has four cores, what happens in this case:

```r
library(future)
plan(multisession, workers = 4L)
set_lidr_threads(4L)
c tg <- readLAScatalog("folder/")
out <- tree_detection(ctg, lmf(2))
```

Here the catalog will be split into chunks that will be processed in parallel. And each computation itself implies a parallelized task. This is a nested parallelism task and it is dangerous! Hopefully the lidR package handles such cases and chooses by default to give precedence to chunk-based parallelism. In this case chunks will be processed in parallel and the points will be processed serially by disabling OpenMP.

Nested parallelism - part 2

We explained rules of precedence. But actually the user can tune the engine more accurately. Let’s define the following function:
myfun = function(cluster, ...) {
  las <- readLAS(cluster)
  if (is.empty(las)) return(NULL)
  las <- normalize_height(las, tin())
  tops <- tree_detection(las, lmf(2))
  bbox <- extent(cluster)
  tops <- crop(tops, bbox)
  return(tops)
}

out <- catalog_apply(ctg, myfun, ws = 5)

This function uses two algorithms, one is partially parallelized (tin) and one is fully parallelized lmf. The user can manually use both OpenMP and future. By default the engine will give precedence to chunk-based parallelism because it works in all cases but the user can impose something else. In the following 2 workers are attributed to future and 2 workers are attributed to OpenMP.

plan(multisession, workers = 2L)
set_lidr_threads(2L)
catalog_apply(ctg, myfun, ws = 5)

The rule is simple. If the number of workers needed is greater than the number of available workers then OpenMP is disabled. Let suppose we have 4 cores:

# 2 chunks 2 threads: OK
plan(multisession, workers = 2L)
set_lidr_threads(2L)

# 4 chunks 1 threads: OK
plan(multisession, workers = 4L)
set_lidr_threads(1L)

# 1 chunks 4 threads: OK
plan(sequential)
set_lidr_threads(4L)

# 3 chunks 2 threads: NOT OK
# Needs 6 workers, OpenMP threads are set to 1 i.e. sequential processing
plan(multisession, workers = 3L)
set_lidr_threads(2L)

Complex computing architectures
For more complex processing architectures such as multiple computers controlled remotely or HPC a finer tuning might be necessary. Using

options(lidR.check.nested.parallelism = FALSE)

lidR will no longer check for nested parallelism and will never automatically disable OpenMP.
Description

This document explains how to process point-clouds taking advantage of different spatial indices available in the lidR package. lidR can use several types of spatial indexes to apply algorithms (that need a spatial indexing) as fast as possible. The choice of the spatial index depends on the type of point-cloud that is processed and the algorithm that is performed. lidR can use a grid partition, a voxel partition, a quadtree or an octree. See details.

Usage

sensor(las, h = FALSE)

sensor(las) <- value

index(las, h = FALSE)

index(las) <- value

Arguments

las An object of class LAS or LAScatalog.

h boolean. Human readable. Everything is stored as integers that are understood internally. Use h = TRUE for user readable output.

value integer or character. A code for referring to a sensor type or a spatial index type. Use one of "unknown", "als", "tls", "uav", "dap", "multispectral" for sensor type and one of "auto", "gridpartition", "voxelpartition", "quadtree", "octree" for spatial index type.

Details

From lidR (>= 3.1.0), a LAS object records the sensor used to sample the point-cloud (ALS, TLS, UAV, DAP) as well as the spatial index that must be used for processing the point cloud. This can be set manually by the user but the simplest is to use one of the read*LAS() functions. By default a point-cloud is associated to a sensor and the best spatial index is chosen on-the-fly depending on the algorithm applied. It is possible to force the use of a specific spatial index.

Information relative to the spatial indexing is stored in slot @index that contains a list with two elements:

• sensor: an integer that records the sensor type

• index: an integer that records the spatial index to be used

By default the spatial index code is 0 ("automatic") meaning that each function is free to choose a different spatial index depending on the recorded sensor. If the code is not 0 then each function will
be forced to use the spatial index that is imposed. This, obviously, applies only to functions that use spatial indexing.

**LAScatalog** objects also record such information that is automatically propagated to the LAS objects when processing.

Note: before version 3.1.0, point-clouds were all considered as ALS because lidR was originally designed for ALS. Consequently, for legacy and backwards-compatibility reasons, `readLAS()` and `readALSLAS()` are actually equivalent. `readLAS()` tags the point cloud with "unknown" sensor while `readALSLAS()` tags it with 'ALS'. Both behave the same and this is especially true compared with versions < 3.1. As a consequence, using `readLAS()` provides the same performance (no degradation) than in previous versions, while using one of the `read*LAS()` functions may improve the performance.

**Examples**

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

# By default the sensor and spatial index codes are 0
sensor(las)
index(las)

# Codes are used internally and not intended to be known by users
# Use h option for human readable output
sensor(las, h = TRUE)
index(las, h = TRUE)

# Modification of the sensor enables users to select a better spatial index
# when processing the point-cloud.
sensor(las) <- "tls"
sensor(las, h = TRUE)
index(las, h = TRUE)

# Modification of the spatial index forces users to choose one of the available
# spatial indexes.
index(las) <- "quadtree"
sensor(las, h = TRUE)
index(las, h = TRUE)

# The simplest way to take advantage of appropriate spatial indexing is
# to use one of the read*LAS() functions.
las <- readTLSLAS(LASfile)
sensor(las, h = TRUE)
index(las, h = TRUE)

# But for some specific point-clouds / algorithms it might be advisable to force
# the use of a specific spatial index to perform the computation faster
index(las) <- "voxelpartition"
index(las, h = TRUE)

# With a LAScatalog, spatial indexing information is propagated to the
# different chunks
```
ctg = readTLSLAScatalog(LASfile)
index(ctg) <- "voxelpartition"
sensor(ctg, h = TRUE)
index(ctg, h = TRUE)

# ==================
# PERFORMANCE TESTS
# ==================

## Not run:
# Performance tests on TLS
# ------------------------
# The package does not include TLS data
# so we can generate something that looks TLS-ish
# >>>>>>>>>>
X = runif(50, -25, 25)
Y = runif(50, -25, 25)
X = as.numeric(sapply(X, function(x) rnorm(2000, x, 2))
Y = as.numeric(sapply(Y, function(x) rnorm(2000, x, 2))
Z = abs(rnorm(length(Y), 10, 5))
veg = data.frame(X,Y,Z)
X = runif(5000, -30, 30)
Y = runif(5000, -30, 30)
Z = runif(5000, 0, 1)
ground = data.frame(X,Y,Z)
X = runif(30, -30, 30)
Y = runif(30, -30, 30)
Z = runif(30, 0, 30)
noise = data.frame(X,Y,Z)
las = LAS(rbind(ground, veg, noise))
# <<<<<<<<<<<<<
plot(las)

# If read with readALSLAS()
sensor(las) <- "als"
system.time(classify_noise(las, sor(20, 8)))
#> 1.5 sec

# If read with readTLSLAS()
sensor(las) <- "tls"
system.time(classify_noise(las, sor(20, 8)))
#> 0.6 sec

# Performance tests on ALS
# ------------------------

# The package does not include large ALS data
# so we can generate something that looks ALS-ish
# >>>>>>>>>>
X = runif(4e5, 0, 1000)
Y = runif(4e5, 0, 1000)
```r
Z = 40*sin(0.01*X) + 50*cos(0.005*Y) + abs(rnorm(length(Y), 10, 5))
veg = data.frame(X,Y,Z)
X = runif(100, 0, 1000)
Y = runif(100, 0, 1000)
Z = 40*sin(0.01*X) + 50*cos(0.005*Y) + abs(rnorm(length(Y), 10, 5)) + runif(100, 30, 70)
noise = data.frame(X,Y,Z)
las = LAS(rbind(veg, noise))
# <<<<<<<<<<<<<
plot(las)

# If read with readALSLAS()
sensor(las) <- "als"
system.time(classify_noise(las, sor(15, 8)))
#> 3 sec

# If read with readTLSLAS()
sensor(las) <- "tls"
system.time(classify_noise(las, sor(15, 8)))
#> 4.3 sec

## End(Not run)
```

---

**lidrpalettes**

<table>
<thead>
<tr>
<th><strong>Palettes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Create a vector of n contiguous (or not) colors</td>
</tr>
</tbody>
</table>

**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
Individual Tree Detection Algorithm

Description

This function is made to be used in `find_trees`. 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 by Popescu & Wynne (2004) (see references).

Usage

```r
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 units 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 point 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 window used to find the local maxima. Can be "square" or "circular".

References


See Also

Other individual tree detection algorithms: `manual()`

Examples

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-inside 481250 3812980 481300 3813050")
# point-cloud-based
# ===============

# 5x5 m fixed window size
ttops <- find_trees(las, lmf(5))
```
Individual Tree Detection Algorithm

Description

This function is made to be used in find_trees. 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 mouse button. Within the selected region the highest point will be flagged as 'tree top' in the scene. Once all the trees are labelled 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", button = "middle", ...)

Arguments

detected SpatialPointsDataFrame of already found tree tops that need manual correction.

radius numeric. Radius of the spheres displayed on the point cloud (aesthetic purposes only).
maxima

Point Cloud Decimation Algorithm

Description

These functions are made to be used in decimate_points. They implement algorithms that creates a grid with a given resolution and filters the point cloud by selecting the highest/lowest point within each cell.

Usage

```
highest(res = 1)
lowest(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_per_voxel(), random()
Examples

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

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

# Select the lowest point within each cell of an overlayed grid
thinned = decimate_points(las, lowest(4))
#plot(thinned)

merge_spatial

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 SpatialPolygons*, a sf data.frame or a Raster*.

- SpatialPolygons*, sf: it checks if the points belongs within each polygon. If the parameter attribute is the name of an attribute in the table of attributes 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

merge_spatial(las, source, attribute = NULL)

Arguments

las An object of class LAS
source An object of class SpatialPolygons* or sf or RasterLayer or a RasterStack or RasterBrick with RGB colors.
attribute 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**

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

las <- readLAS(LASfile, filter = "-keep_random_fraction 0.1")
lakes <- sf::st_read(shp)

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

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

---

**normalize_height**

*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 `unnormalize_height` to restore original point elevations.

**Usage**

```r
normalize_height(
  las,
  algorithm,
  na.rm = FALSE,
  use_class = c(2L, 9L),
  ...,  
  add_lasattribute = FALSE,
  Wdegenerated = TRUE
)
```
normalize_height(las)

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

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>las</td>
<td>An object of class LAS or LAScatalog.</td>
</tr>
<tr>
<td>algorithm</td>
<td>a spatial interpolation function. lidR have tin, kriging, knnidw or a RasterLayer representing a digital terrain model (can be computed with grid_terrain)</td>
</tr>
<tr>
<td>na.rm</td>
<td>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 na.rm = TRUE points with an elevation of NA are filtered. Be careful this creates a copy of the point cloud.</td>
</tr>
<tr>
<td>use_class</td>
<td>integer vector. By default the terrain is computed by using ground points (class 2) and water points (class 9). Relevant only for a normalisation without a raster DTM.</td>
</tr>
<tr>
<td>...</td>
<td>If algorithm is a RasterLayer, ... is propagated to extract. Typically one may use method = &quot;bilinear&quot;.</td>
</tr>
<tr>
<td>add_lasattribute</td>
<td>logical. By default the above see level elevation is retained in a new attribute. However this new attribute will be discarded at write time. If TRUE it is maintained as an extrabytes attribute. See also add_lasattribute.</td>
</tr>
<tr>
<td>Wdegenerated</td>
<td>logical. The function always check and remove degenerated ground points for computing the DTM to avoid unexpected behaviours such as infinite elevation. If TRUE a warning in thrown to alert about the presence of degenerated ground points.</td>
</tr>
<tr>
<td>e1</td>
<td>a LAS object</td>
</tr>
<tr>
<td>e2</td>
<td>RasterLayer representing a digital terrain model (can be computed with grid_terrain) or a spatial interpolation function. lidR has tin, kriging, and knnidw.</td>
</tr>
</tbody>
</table>

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.
- **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}`.
- **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

grid_terrain

Other normalize: normalize_intensity()

Examples

```r
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, filter = "-inside 273450 5274350 273550 5274450")
#plot(las)

# First option: use a RasterLayer as DTM
# ==============================================================
dtm <- grid_terrain(las, 1, knnidw(k = 6L, p = 2))
las <- normalize_height(las, dtm)
plot(dtm)
#plot(las)

# restore original elevations
las <- unnormalize_height(las)
```
# operator - can be used. This is equivalent to the previous
las <- las - dtm
#plot(las)

# restore original elevations
las <- unnormalize_height(las)

# Second option: interpolate each point (no discretization)
# ==============================================================
las <- normalize_height(las, tin())
#plot(las)

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

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

## End(Not run)

normalize_intensity  Normalize intensity

Description
Normalize intensity values using multiple methods.

Usage
normalize_intensity(las, algorithm)

Arguments
las  An object of class LAS or LAScatalog.
algorithm  an intensity normalizaton algorithm. lidR currently has range_correction.

Value
Returns an object of class LAS. The attribute 'Intensity' records the normalized intensity. An extra attribute named 'RawIntensity' records the original intensities.
**normalize_intensity**

**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**: No buffer needed. A buffer of 0 is used and cannot be changed
- **chunk alignment**: Align the processed chunks.
- **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}`.
- **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 normalize: `normalize_height()`

**Examples**

```r
# A valid file properly populated
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile)

# pmin = 15 because it is an extremely small file
# strongly decimated to reduce its size. There are
# actually few multiple returns
sensor <- track_sensor(las, Roussel2020(pmin = 15))

# Here the effect is virtually null because the size of
# the sample is too small to notice any effect of range
las <- normalize_intensity(las, range_correction(sensor, Rs = 2000))
```
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

subcircle  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).

na.fill  function. A function that implements an algorithm to compute spatial interpolation to fill the empty pixel often left by points-to-raster methods. lidR 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**

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

**Arguments**

- `thresholds` numeric. Set of height thresholds according to the Khosravipour et al. (2014) algorithm description (see references)
- `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. 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 `max_edge = 0` no trimming is done (see examples).
- `subcircle` 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**


**See Also**

Other digital surface model algorithms: `dsmtin()`, `p2r()`

**Examples**

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
poi = "-drop_z_below 0 -inside 481280 3812940 481330 3812990"
las <- readLAS(LASfile, filter = poi)
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 = clip_polygon(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(las, 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**

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

## S4 method for signature 'LAS,missing'

```r
plot(
  x,
  y,
  color = "Z",
  colorPalette = "auto",
  bg = "black",
  trim = Inf,
  backend = "rgl",
```
clear_artifacts = TRUE,
nbits = 16,
axis = FALSE,
legend = FALSE,
add = FALSE,
voxel = FALSE,
...
)

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

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

**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 `heat.colors`, `topo.colors`, `terrain.colors` or similar functions. Default is "auto" providing an automatic coloring depending on the argument `color`.

- **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 `trim` will be plotted with the highest color.

- **backend**  
  character. Can be "rgl" or "lidRviewer". If "rgl" is chosen the display relies on the `rgl` package. If "lidRviewer" is chosen it relies on the `lidRviewer` package, which is much more efficient and can handle million of points using less memory. `lidRviewer` is not available on CRAN yet and should be installed from github (see. [https://github.com/Jean-Romain/lidRviewer](https://github.com/Jean-Romain/lidRviewer)).

- **clear_artifacts**  
  logical. It is a known and documented issue that the 3D visualisation with `rgl` displays artifacts. The points look aligned and/or regularly spaced in some view angles. This is because `rgl` 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 colours 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.

- **axis**  
  logical. Display axis on XYZ coordinates.

- **legend**  
  logical. Display a gradient colour legend.
add

If FALSE normal behaviour otherwise must be the output of a prior plot function to enable the alignment of a second point cloud.

voxel

boolean or numeric. Displays voxels instead of points. Useful to render the output of `voxelize_points`, for example. However it is computationally demanding to render and can easily take 15 seconds for 10000 voxels. It should be reserved for small scenes. If boolean the voxel resolution is guessed automatically. Otherwise users can provide the size of the voxels. To reduce the rendering time, an internal optimization removes voxels that are not visible when surrounded by other voxels.

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.

overlaps

logical. Highlight the overlaps between files.

Examples

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

plot(las)
plot(las, color = "Intensity")

# If outliers break the color range, use the trim parameter
plot(las, color = "Intensity", trim = 150)

plot(las, color = "Classification")

# This dataset is already tree segmented
plot(las, color = "treeID")

# single file LAScatalog using data provided in lidR
t = readLAScatalog(LASfile)
plot(t)
plot(t, map = T, map.types = "Esri.WorldImagery")

## End(Not run)
```

plot.lasmetrics3d

Plot voxelized LiDAR data

Description

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

Usage

```r
## S3 method for class 'lasmetrics3d'
plot(x, y, ...)
```
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. It works only with rgl i.e. backend = "rgl" which is the default.

Usage

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

add_dtm3d(x, dtm, ...)

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

add_flightlines3d(x, flightlines, z = "Z", ...)

Arguments

dtm An object of the class RasterLayer

bg The color for the background. Default is black.

Examples

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

voxels = voxel_metrics(lidar, list(Imean = mean(Intensity)), res = 5)
plot(voxels, color = "Imean", colorPalette = heat.colors(50), trim=60)

## End(Not run)
plot_metrics

Computes metrics for each plot of a ground inventory

Description

Computes metrics for each plot of a ground inventory by 1. clipping the plots inventories, 2. computing a user’s metrics to each plot, and 3. combining spatial data and metrics into one data.frame ready for statistical modelling. ‘plot_metrics’ is basically a seamless wrapper around clip_roi, cloud_metrics, ‘cbind’ and adequate processing settings.

Usage

plot_metrics(las, func, geometry, ...)
Arguments

las
   An object of class LAS or LAScatalog.
func
   formula. An expression to be applied to each cell (see also grid_metrics).
geometry
   a spatial object. Many types are supported (see also clip_roi)
   ...
   optional supplementary options (see also clip_roi)

Value

An ‘sp’ or ‘sf’ object depending on the input with all the metrics for each plot bound with the
original input.

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 plot_metrics function (in bold). For more
details see the LAScatalog engine documentation:
   • chunk size: Not relevant here.
   • chunk buffer: Not relevant here.
   • chunk alignment: Not relevant here.
   • progress: Displays a progress estimate.
   • output files: plots are extracted in memory.
   • select: Read only the attributes of interest.
   • filter: Read only the points of interest.

See Also

Other metrics: cloud_metrics(), grid_metrics(), point_metrics(), tree_metrics(), voxel_metrics()
Examples

```r
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
SHPfile <- system.file("extdata", "efi_plot.shp", package="lidR")
las <- readLAS(LASfile)
inventory <- sf::st_read(SHPfile, quiet = TRUE)
inventory # contains an ID and a Value Of Interest (VOI) per plot
M <- plot_metrics(las, ~list(q85 = quantile(Z, probs = 0.85)), inventory, radius = 11.28)
model <- lm(VOI ~ q85, M)
M <- plot_metrics(las, .stdmetrics_z, inventory, radius = 11.28)

## Not run:
# Works with polygons as well
inventory <- sf::st_buffer(inventory, 11.28)
plot(las@header)
plot(sf::st_geometry(inventory), add = TRUE)
M <- plot_metrics(las, .stdmetrics_z, inventory)
## End(Not run)
```

Ground Segmentation Algorithm

**Description**

This function is made to be used in `classify_ground`. 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**

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

**Arguments**

- `ws` 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).
- `th` 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.
point_metrics

References


See Also

Other ground segmentation algorithms: `csf()`

Examples

```r
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn", filter = "-inside 273450 5274350 273550 5274450")
ws <- seq(3,12, 3)
th <- seq(0.1, 1.5, length.out = length(ws))
las <- classify_ground(las, pmf(ws, th))
#plot(las, color = "Classification")
```

Description

Computes a series of user-defined descriptive statistics for a LiDAR dataset for each point. This function is very similar to `grid_metrics` but computes metrics for each point based on its k-nearest neighbours or its sphere neighbourhood.

Usage

```r
point_metrics(las, func, k, r, xyz = FALSE, filter = NULL, ...)
```

Arguments

- `las`: An object of class LAS
- `func`: formula. An expression to be applied to each point neighbourhood (see section "Parameter func").
- `k, r`: integer and numeric respectively for k-nearest neighbours and radius of the neighborhood sphere. If k is given and r is missing, computes with the knn, if r is given and k is missing computes with a sphere neighborhood, if k and r are given computes with the knn and a limit on the search distance.
- `xyz`: logical. Coordinates of each point are returned in addition to each metric. If `filter = NULL` coordinates are references to the original coordinates and do not occupy additional memory. If `filter != NULL` it obviously takes memory.
- `filter`: formula of logical predicates. Enables the function to run only on points of interest in an optimized way. See examples.
- `...`: unused.
Details

When the neighbourhood is knn the user-defined function is fed with the current processed point and its k-1 neighbours. The current point being considered as the 1-neighbour with a distance 0 to the reference point. The points are ordered by distance to the central point. When the neighbourhood is a sphere the processed point is also included in the query but points are coming in a random order.

Performances

It is important to bear in mind that this function is very fast for the feature it provides i.e. mapping a user-defined function at the point level using optimized memory management. However, it is still computationally demanding.

To help users to get an idea of how computationally demanding this function is, let’s compare it to grid_metrics. Assuming we want to apply mean(Z) on a 1 km² tile with 1 point/m² with a resolution of 20 m (400 m² cells), then the function mean is called roughly 2500 times (once per cell). On the contrary, with point_metrics, mean is called 1000000 times (once per point). So the function is expected to be more than 400 times slower in this specific case (but it does not provide the same feature).

This is why the user-defined function is expected to be well-optimized, otherwise it might drastically slow down this already heavy computation. See examples.

Last but not least, grid_metrics() relies on the data.table package to compute a user-defined function in each pixel. point_metrics() relies on a similar method but with a major difference: it does not rely on data.table and thus has not been tested over many years by thousands of people. Please report bugs, if any.

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.

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

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

```r
point_metrics(las, ~f(Z), k = 8)
point_metrics(las, ~f(Intensity), k = 5)
```

Everything that works in grid_metrics should also work in point_metrics but sometimes might be meaningless. For example, computing the quantile of elevation does not really makes sense here.

See Also

Other metrics: cloud_metrics(), grid_metrics(), plot_metrics(), tree_metrics(), voxel_metrics()
Examples

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

# Read only 0.5 points/m^2 for the purposes of this example
las = readLAS(LASfile, filter = "-thin_with_grid 2")

# Computes the eigenvalues of the covariance matrix of the neighbouring
# points and applies a test on these values. This function simulates the
# 'shp_plane()' algorithm from 'segment_shape()'
plane_metrics1 = function(x,y,z, th1 = 25, th2 = 6) {
  xyz <- cbind(x,y,z)
  cov_m <- cov(xyz)
  eigen_m <- eigen(cov_m)$value
  is_planar <- eigen_m[2] > (th1*eigen_m[3]) && (th2*eigen_m[2]) > eigen_m[1]
  return(list(planar = is_planar))
}

# Apply a user-defined function
M <- point_metrics(las, ~plane_metrics1(X,Y,Z), k = 25)
#> Computed in 6.3 seconds

# We can verify that it returns the same as 'shp_plane'
las <- segment_shape(las, shp_plane(k = 25), "planar")
#> Computed in 0.1 seconds
all.equal(M$planar, las$planar)

# At this stage we can be clever and find that the bottleneck is
# the eigenvalue computation. Let's write a C++ version of it with
# Rcpp and RcppArmadillo
Rcpp::sourceCpp(code = "
#include <RcppArmadillo.h>

// [Rcpp::depends(RcppArmadillo)]
// [Rcpp::export]
SEXP eigen_values(arma::mat A) {
  arma::mat coeff;
  arma::mat score;
  arma::vec latent;
  arma::princomp(coeff, score, latent, A);
  return(Rcpp::wrap(latent));
}

plane_metrics2 = function(x,y,z, th1 = 25, th2 = 6) {
  xyz <- cbind(x,y,z)
  eigen_m <- eigen_values(xyz)
  is_planar <- eigen_m[2] > (th1*eigen_m[3]) && (th2*eigen_m[2]) > eigen_m[1]
  return(list(planar = is_planar))
}

M <- point_metrics(las, ~plane_metrics2(X,Y,Z), k = 25)
```

---

point_metrics

113
all.equal(M$planar, las$planar)
# Here we can see that the optimized version is way better but is still 5-times slower
# because of the overhead of calling R functions and switching back and forth from R to C++.

# Use the filter argument to process only first returns
M1 <- point_metrics(las, ~plane_metrics2(X,Y,Z), k = 25, filter = ~ReturnNumber == 1)
dim(M1) # 13894 instead of 17182 previously.

# is a memory-optimized equivalent to:
first = filter_first(las)
M2 <- point_metrics(first, ~plane_metrics2(X,Y,Z), k = 25)
all.equal(M1, M2)

## End(Not run)

---

**Summary and Print for LAS* objects**

**Description**

Summary and Print for LAS* objects

**Usage**

```r
print(x, ...)
```

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

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

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

## S3 method for class 'lidRAlgorithm'
```r
print(x, ...)
```

**Arguments**

... Unused

object, x A LAS* object or other lidR related objects.
projection

Get or set the projection of a LAS* object

Description

Get or set the projection of a LAS* object.

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

wkt(object) <- value

## S4 method for signature 'LASheader'
wkt(obj)

## S4 replacement method for signature 'LASheader'
wkt(object) <- value

## S4 method for signature 'LAS'
wkt(obj)

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

## S4 method for signature 'LASheader'
projection(x, asText = TRUE)

## S4 method for signature 'LAS'
projection(x, asText = TRUE)

## S4 replacement method for signature 'LAS'
projection(x) <- value
### S4 method for signature 'LAScatalog'
projection(x, asText = TRUE)

### S4 replacement method for signature 'LAScatalog'
projection(x) <- value

### S4 method for signature 'LASheader'
crs(x, asText = FALSE)

### S4 method for signature 'LAS'
crs(x, asText = FALSE)

### S4 replacement method for signature 'LAS'
crs(x, ...) <- value

### S4 method for signature 'LAScatalog'
crs(x, asText = FALSE)

### S4 replacement method for signature 'LAScatalog'
crs(x, ...) <- value

#### Arguments

- **object, x, obj** An object of class LAS or eventually LASheader (regular users don’t need to manipulate LASheader objects).
- **...** Unused.
- **value** A CRS object or a proj4string string or WKT string or an EPSG code.
- **asText** logical. If TRUE, the projection is returned as text. Otherwise a CRS object is returned.

#### Details

There are two ways to store the CRS of a point cloud in a LAS file:

- Store an EPSG code (for LAS 1.0 to 1.3)
- Store a WKT string (for LAS 1.4)

On the other hand, R spatial packages use a proj4string to store the CRS (but the ecosystem is moving to WKT). This is why the CRS is duplicated in a LAS object. The information belongs within the header in a format that can be written in a LAS file and in the slot proj4string in a format that can be understood by R packages.

- **projection<-**: assigns a CRS from a CRS (sp), a crs (sf), a WKT string, a proj4string or an epsg code. It updates the header of the LAS object either with the EPSG code for LAS formats < 1.4 or with a WKT string for LAS format 1.4 and updates the proj4string slot.
- **projection**: returns the CRS in sp format
- **crs and crs<-** are equivalent to projection and projection<-
random

- epsg<-, wkt<-: legacy functions superseded by projection<-
- sf::st_crs return the CRS in sf format.
- epsg: reads the epsg code from the header.
- wkt: reads the WKT string from the header.

Examples

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

# Get the EPSG code stored in the header (returns 0 if not recorded)
epsg(las)

# Get the WKT string stored in the header (LAS >= 1.4)
wkt(las)

# Uses the EPSG code
projection(las) <- 26919
sf::st_crs(las)$input
```

---

random

**Point Cloud Decimation Algorithm**

Description

This function is made to be used in `decimate_points`. 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 `retrieve_pulses` first)

See Also

Other point cloud decimation algorithms: `homogenize`, `maxima`, `random_per_voxel()`
**random_per_voxel**

**Point Cloud Decimation Algorithm**

**Description**

This function is made to be used in `decimate_points`. It implements an algorithm that creates a 3D grid with a given resolution and filters the point cloud by randomly selecting n points within each voxel.

**Usage**

```r
random_per_voxel(res = 1, n = 1)
```

**Arguments**

- `res` numeric. The resolution of the voxel grid used to filter the point cloud.
- `n` integer. The number of points to select.

**See Also**

Other point cloud decimation algorithms: `homogenize()`, `maxima`, `random()`.

**Examples**

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

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

```r
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz")
thinned <- decimate_points(las, random_per_voxel(8, 1))
#plot(thinned)
```
**range_correction**

*Intensity normalization algorithm*

**Description**

This function is made to be used in `normalize_intensity`. It corrects intensity with a range correction according to the formula (see references):

\[
I_{\text{norm}} = I_{\text{obs}} \left( \frac{R}{R_s} \right)^f
\]

To achieve the range correction the position of the sensor must be known at different discrete times. Using the ‘gpstime’ of each point, the position of the sensor is interpolated from the reference and a range correction is applied.

**Usage**

```r
range_correction(sensor, Rs, f = 2.3, gpstime = "gpstime", elevation = "Z")
get_range(las, sensor, gpstime = "gpstime", elevation = "Z")
```

**Arguments**

- `sensor` : `SpatialPointsDataFrame` object containing the coordinates of the sensor at different timepoints t. The time and elevation are stored as attributes (default names are ‘gpstime’ and ‘Z’). It can be computed with `track_sensor`.
- `f` : numeric. Exponent. Usually between 2 and 3 in vegetation contexts.
- `gpstime`, `elevation` : character. The name of the attributes that store the gpstime of the position and the elevation of the sensor respectively. If `elevation = NULL` the Z coordinates are searched in the third column of the coordinates matrix of the `SpatialPointsDataFrame`. This is useful if read from a format that supports 3 coordinates points.
- `las` : an object of class LAS. `get_range()` is a regular function documented here for convenience.

**References**

Examples

# A valid file properly populated
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile)

# pmin = 15 because it is an extremely tiny file
# strongly decimated to reduce its size. There are
# actually few multiple returns
sensor <- track_sensor(las, Roussel2020(pmin = 15))

# Here the effect is virtually null because the size of
# the sample is too small to notice any effect of range
las <- normalize_intensity(las, range_correction(sensor, Rs = 2000))

# This might be useful for some applications
R = get_range(las, sensor)

---

rbind

Merge LAS* objects

Description

Merge LAS* objects

Usage

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

## S3 method for class 'LAScatalog'
rbind(...)

Arguments

... LAS or LAScatalog objects

---

readLAS

Read .las or .laz files
Description

Reads .las or .laz files into 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. LAS formats 1.1 to 1.4 are supported. Point Data Record Format 0 to 10 are supported.

readLAS is the original function and always works. Using one of the read*LAS functions adds information to the returned object to register a point-cloud type. Registering the correct point type may improve the performance of some functions by enabling users to select an appropriate spatial index. See spatial indexing. Notice that by legacy and for backwards-compatibility reasons, readLAS() and readALSLAS() are equivalent because lidR was originally designed for ALS and thus the original function readLAS() was (supposedly) used for ALS. Reading a TLS dataset with readLAS() instead of readTLSLAS() is perfectly valid and performs similarly to versions <= 3.0.0, with neither performance degradation nor improvements.

Usage

readLAS(files, select = "*", filter = "")
readALSLAS(files, select = "*", filter = "")
readTLSLAS(files, select = "*", filter = "")
readUAVLAS(files, select = "*", filter = "")
readDAPLAS(files, select = "*", filter = "")
readMSLAS(files1, files2, files3, 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).
files1, files2, files3 characters. Path(s) to one or several a file(s). Each argument being one channel (see section 'Multispectral data').

Details

Select: the 'select' argument specifies the data that 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, s - synthetic flag, k - keypoint flag, w - withheld flag, o - overlap flag (format 6+), 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, C - scanner channel (format 6+), W - Full waveform. Also numbers from 1 to 9 for the extra bytes data numbers 1 to 9. 0 enables all extra bytes to be loaded and '*' is the wildcard that
enables everything to be loaded from the LAS file.
Note that x, y, z are implicit and always loaded. ‘xyzia’ is equivalent to ‘ia’.

**Filter:** the ‘filter’ argument allows filtering of the point cloud while reading files. This is much more efficient than `filter_poi` 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: `readLAS(filter = "-help")`. (see also `rlas::read.las`). From rlas v1.3.6 the transformation commands can also be passed via the argument `filter`.

**Value**

A LAS object

**Full waveform**

With most recent versions of the ‘rlas’ package, full waveform (FWF) can be read and ‘lidR’ provides some compatible functions. However, the support of FWF is still a work-in-progress in the ‘rlas’ package. How it is read, interpreted and represented in R may change. Consequently, tools provided by ‘lidR’ may also change until the support of FWF becomes mature and stable in ‘rlas’. See also `rlas::read.las`.

Remember that FWF represents an insanely huge amount of data. It terms of memory it is like having between 10 to 100 times more points. Consequently, loading FWF data in R should be restricted to relatively small point clouds.

**Multispectral data**

Multispectral laser data are often stored in 3 different files. If this is the case `readMSLAS` reads the .las or .laz files of each channel and merges them into an object of class `LAS` and takes care of attributing an ID to each channel. If the multispectral point cloud is already stored in a single file leave `file2` and `file3` missing.

**Examples**

```r
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 attributes is also possible (all except intensity and angle)
las = readLAS(LASfile, select = "* -i -a")
```

---

**readLAScatalog**

*Create an object of class LAScatalog*
Description

Create an object of class LASCatalog from a folder or a collection of filenames. A LASCatalog is a representation of a collection 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 appropriate functions. Careful reading of the LASCatalog class documentation is required to use the LASCatalog class correctly.

readLAScatalog is the original function and always works. Using one of the read*LAScatalog functions adds information to the returned object to register a point-cloud type. Registering the correct point type may improve the performance of some functions by enabling users to select an appropriate spatial index. See spatial indexing. Notice that by legacy and for backwards-compatibility reasons readLAScatalog() and readALSLAScatalog() are equivalent because lidR was originally designed for ALS and thus the original function readLAScatalog() was (supposedly) used for ALS.

Usage

```r
readLAScatalog(
  folder,
  progress = TRUE,
  select = "*",
  filter = "",
  chunk_size = 0,
  chunk_buffer = 30,
  ...
)
```

```r
readALSLAScatalog(
  folder,
  progress = TRUE,
  select = "*",
  filter = "",
  chunk_size = 0,
  chunk_buffer = 30,
  ...
)
```

```r
readTLSLAScatalog(
  folder,
  progress = TRUE,
  select = "*",
  filter = "",
  chunk_size = 0,
  chunk_buffer = 30,
  ...
)
```
readUAVLAScatalog(
  folder,
  progress = TRUE,
  select = "*",
  filter = "",
  chunk_size = 0,
  chunk_buffer = 30,
  ...
)

readDAPLAScatalog(
  folder,
  progress = TRUE,
  select = "*",
  filter = "",
  chunk_size = 0,
  chunk_buffer = 30,
  ...
)

catalog(folder, ...)

Arguments

folder string. The path of a folder containing a set of las/laz files. Can also be a vector of file paths.

progress, select, filter, chunk_size, chunk_buffer
Easily accessible processing options tuning. See LAScatalog-class and catalog_options_tools.

... Extra parameters to list.files. Typically recursive = TRUE.

Value

A LAScatalog object

Examples

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

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

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

# Internal engine will align the 500 x 500 m chunks 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)

---

readLASHeader  

**Read a .las or .laz file header**

**Description**

Reads a .las or .laz file header into an object of class `LASHeader`. This function strictly reads the header while the function `readLAS` can alter the header to fit the actual data loaded.

**Usage**

```r
readLASHeader(file)
```

**Arguments**

- `file` characters. Path to one file.

**Value**

A LASHeader object

**Examples**

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

print(header)
plot(header)

## Not run:
plot(header, mapview = TRUE)
## End(Not run)
```
retrieve_pulses

Retrieve individual pulses, flightlines or scanlines

Description

Retrieve each individual pulse, individual flightline or individual scanline and assigns a number to each point. The LAS object must be properly populated according to LAS specifications otherwise users could find unexpected outputs.

Usage

retrieve_pulses(las)

retrieve_flightlines(las, dt = 30)

retrieve_scanlines(las)

Arguments

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

Details

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

retrieve_scanlines 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 is used to retrieve each scanline. An attribute scanlineID is added in the LAS object

retrieve_flightlines 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 <- retrieve_pulses(las)
las
Description

This function is made to be used in track_sensor. It implements an algorithm from Roussel et al. 2020 (see reference) for sensor tracking using multiple returns to estimate the positioning of the sensor by computing the intersection in space of the lines passing through the first and last returns.

Usage

Roussel2020(interval = 0.5, pmin = 50)

Arguments

- **interval**: numeric. Interval used to bin the gps times and group the pulses to compute a position at a given timepoint t.
- **pmin**: integer. Minimum number of pulses needed to estimate a sensor position. For a given interval, the sensor position is not computed if the number of pulses is lower than pmin.

Details

When multiple returns from a single pulse are detected, the sensor computes their positions as being in the center of the footprint and thus all aligned. Because of that behavior, a line drawn between and beyond those returns must cross the sensor. Thus, several consecutive pulses emitted in a tight interval (e.g. 0.5 seconds) can be used to approximate an intersection point in the sky that corresponds to the sensor position given that the sensor carrier hasn’t moved much during this interval. A weighted least squares method gives an approximation of the intersection by minimizing the squared sum of the distances between the intersection point and all the lines.

References

Roussel Jean-Romain, Bourdon Jean-Francois, Achim Alexis, (2020) Range-based intensity normalization of ALS data over forested areas using a sensor tracking method from multiple returns (preprint) Retrieved from eartharxiv.org/k32qw

Examples

```r
# A valid file properly populated
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)

# pmin = 15 because it is an extremely tiny file
```
# strongly decimated to reduce its size. There are
# actually few multiple returns
flightlines <- track_sensor(las, Roussel2020(pmin = 15))

plot(las@header)
plot(flightlines, add = TRUE)

---

**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 (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**

```r
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**


**Examples**

```r
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)
```
segment_shapes

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)

segment_shapes

Eigenvalues-based features at the point level

Description

Functions that compute, for each point, the eigenvalues of the covariance matrix of the neighbouring points. The eigenvalues are later used either to segment linear/planar points or to compute derived metrics (see Details).

Usage

segment_shapes(las, algorithm, attribute = "Shape", filter = NULL)

point_eigenvalues(las, k, r, xyz = FALSE, metrics = FALSE, filter = NULL)

Arguments

las An object of class LAS or LAScatalog.
algorithm An algorithm for shape detection. lidR has: shp_plane, shp_hplane and shp_line.
attribute character. The name of the new column to add into the LAS object.
filter formula of logical predicates. Enables the function to run only on points of interest in an optimized way. See the examples.
k, r integer and numeric respectively for k-nearest neighbours and radius of the neighborhood sphere. If k is given and r is missing, computes with the knn, if r is given and k is missing computes with a sphere neighbourhood, if k and r are given computes with the knn and a limit on the search distance.
xyz logical. Returns the XYZ coordinates of each points instead of IDs.
metrics logical. Compute additional metrics such as curvature, linearity, planarity based on the eigenvalues.
segment_shapes

Details

All the functions documented here can be reproduced with `point_metrics`. However, `point_metrics` is a versatile and multipurpose function that is not as fast as is possible because it calls user-defined R code and that implies computational overheads. These functions are parallelized plain C++ versions of tools users can build with `point_metrics` and are consequently 10-times faster.

**segment_shape** The points that meet a given criterion based on the eigenvalue are labelled as approximately coplanar/colinear or any other shape supported.

**point_eigenvalues** Computes the eigenvalues of the covariance matrix and computes associated metrics following Lucas et al, 2019 (see references). It is equivalent to `point_metrics(las, stdshapemetrics)` but much faster

Value

**segment_shape** A LAS object with a new column named after the argument attribute that indicates those points that are part of a neighborhood that is approximately of the shape searched (TRUE) or not (FALSE).

**point_eigenvalues** A data.frame like `point_metrics`

References


Examples

```r
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile, filter = "-keep_random_fraction 0.5")

# Use the eigenvalues to estimate if points are part of a local plan
las <- segment_shapes(las, shp_plane(k = 15), "Coplanar")
#plot(las, color = "Coplanar")

# Computes the eigenvalue of each point
M <- point_eigenvalues(las, k = 15)
M

## Not run:
# Drop ground point at runtime
las <- segment_shapes(las, shp_plane(k = 15), "Coplanar", filter = ~Classification != 2L)
#plot(las, color = "Coplanar")

## End(Not run)
```
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

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

Arguments

- `las`: An object of class LAS or LAScatalog.
- `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.
segment_trees

Individual tree segmentation

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.

Examples

```r
## 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 <- segment_snags(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 <- filter_poi(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)
```
Usage

segment_trees(las, algorithm, attribute = "treeID", uniqueness = "incremental")

Arguments

las An object of class LAS or LAScatalog.
algorithm function. An algorithm of individual tree segmentation. lidR has: dalponte2016, watershed, li2012 and silva2016. More experimental algorithms may be found in the package lidRplugins.
attribute 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".
uniqueness character. A method to compute a unique ID. Can be 'incremental', 'gpstime' or 'bitmerge'. See section 'Uniqueness'. This feature must be considered as 'experimental'.

Value

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

Uniqueness

By default the tree IDs are numbered from 1 to n, n being the number of trees found. The problem with such incremental numbering is that, while it ensures a unique ID is assigned for each tree in a given point-cloud, it also guarantees duplication of tree IDs in different tiles or chunks when processing a LAScatalog. This is because each chunk/file is processed independently of the others and potentially in parallel on different computers. Thus, the index always restarts at 1 on each chunk/file or chunk. Worse, in a tree segmentation process, a tree that is located exactly between 2 chunks/files will have two different IDs for its two halves.

This is why we introduced some uniqueness strategies that are all imperfect and that should be seen as experimental. Please report any troubleshooting. Using a uniqueness-safe strategy ensures that trees from different files will not share the same IDs. Moreover, it also means that two halves of a tree on the edge of a processing chunk will be assigned the same ID.

incremental Number from 0 to n. This method does not ensure uniqueness of the IDs. This is the legacy method.
gpstime This method uses the gpstime of the highest point of a tree (apex) to create a unique ID. This ID is not an integer but a 64-bit decimal number, which is suboptimal but at least it is expected to be unique if the gpstime attribute is consistent across files. If inconsistencies with gpstime are reported (for example gpstime records the week time and was reset to 0 in a coverage that takes more than a week to complete), there is a (low) probability of getting ID attribution errors.
bitmerge This method uses the XY coordinates of the highest point (apex) of a tree to create a single 64-bit number with a bitwise operation. First, XY coordinates are converted to 32-bit integers using the scales and offsets of the point cloud. For example, if the apex is at (10.32, 25.64) with a scale factor of 0.01 and an offset of 0, the 32-bit integer coordinates are X = 1032 and Y = 2564. Their binary representations are, respectively, (here displayed
as 16 bits) 0000010000001000 and 0001010000000100. X is shifted by 32 bits and becomes a 64-bit integer. Y is kept as-is and the binary representations are unionized into a 64-bit integer like (here displayed as 32 bit) 00000100000010000000101000000100 that is guaranteed to be unique. However R does not support 64-bit integers. The previous steps are done at C++ level and the 64-bit binary representation is reinterpreted into a 64-bit decimal number to be returned in R. The IDs thus generated are somewhat weird. For example, the tree ID 00000100000010000000100000001000 which is 67635716 if interpreted as an integer becomes 3.34164837074751323479078607289E-316 if interpreted as a decimal number. This is far from optimal but at least it is guaranteed to be unique if all files have the same offsets and scale factors.

All the proposed options are suboptimal because they either do not guarantee uniqueness in all cases (inconsistencies in the collection of files), or they imply that IDs are based on non-integers or meaningless numbers. But at least it works and deals with some of the limitations of R.

**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.
- **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\}.
- **select**: The function will write files equivalent to the original ones. Thus select = "*" and cannot be changed.
- **filter**: Read only points of interest.
set_lidr_threads

Set or get number of threads that lidR should use

Examples

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

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

------

set_lidr_threads Set or get number of threads that lidR should use

Description

Set and get number of threads to be used in lidR functions that are parallelized with OpenMP. 0 means to utilize all CPU available. get_lidr_threads() returns the number of threads that will be used. This affects lidR package but also the data.table package by internally calling setDTthreads because several functions of lidR rely on data.table but it does not change R itself or other packages using OpenMP.

Usage

set_lidr_threads(threads)

get_lidr_threads()

Arguments

threads Positive scalar. Default 0 means use all CPU available. Values > 1 mean using n cores, values in [0, 1] mean using a fraction of the cores e.g. 0.5 = half.

See Also

lidR-parallelism

shape_detection Algorithms for shape detection of the local point neighborhood

Description

These functions are made to be used in segment_shapes. They implement algorithms for local neighborhood shape estimation.
Usage

shp_plane(th1 = 25, th2 = 6, k = 8)

shp_hplane(th1 = 25, th2 = 6, th3 = 0.98, k = 8)

shp_line(th1 = 10, k = 8)

Arguments

th1, th2, th3 numeric. Threshold values (see details)

k integer. Number of neighbours used to estimate the neighborhood.

Details

In the following, \(a_1, a_2, a_3\) denote the eigenvalues of the covariance matrix of the neighbouring points in ascending order. \(th_1, th_2, th_3\) denote a set of threshold values. Points are labelled TRUE if they meet the following criteria. FALSE otherwise.

**shp_plane** Detection of plans based on criteria defined by Limberger & Oliveira (2015) (see references). A point is labelled TRUE if the neighborhood is approximately planar, that is:

\[
a_2 > (th_1 \times a_1) \text{ and } a_2 > a_3
\]

**shp_hplane** The same as 'plane' but with an extra test on the orientation of the Z vector of the principal components to test the horizontality of the surface.

\[
a_2 > (th_1 \times a_1) \text{ and } a_2 > a_3 \text{ and } |Z| > th_3
\]

In theory \(|Z|\) should be exactly equal to 1. In practice 0.98 or 0.99 should be fine

**shp_line** Detection of lines inspired by the Limberger & Oliveira (2015) criterion. A point is labelled TRUE if the neighborhood is approximately linear, that is:

\[
th_1 \times a_2 < a_3 \text{ and } th_1 \times a_1 < a_3
\]

References

Individual Tree Segmentation Algorithm

Description

This function is made to be used in `segment_trees`. 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 `find_trees` 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 `segment_trees` 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/")
tttops = find_trees(chm, lmf(3))
crowns = silva2016(chm, tttops)()
```

References

See Also

Other individual tree segmentation algorithms: `dalponte2016()`, `li2012()`, `watershed()`
Other raster based tree segmentation algorithms: `dalponte2016()`, `watershed()`

Examples

```r
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
poi <- "-drop_z_below 0 -inside 481280 3812940 481320 3812980"
las <- readLAS(LASfile, select = "xyz", filter = poi)
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 <- find_trees(chm, lmf(4, 2))
las <- segment_trees(las, silva2016(chm, ttops))
#plot(las, color = "treeID", colorPalette = col)
```

smooth_height

Smooth a point cloud

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 `unsmooth_height`

Usage

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

Arguments

```r
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
filter_surfacepoints, for example to develop new algorithms.

Value

An object of the class LAS.

Examples

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

las <- filter_surfacepoints(las, 1)
#plot(las)

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

las <- unsmooth_height(las)
#plot(las)
```

Description

This function is made to be used in classify_noise. It implements an algorithm for outliers (noise)
segmentation based on Statistical Outliers Removal (SOR) methods first described in the PCL li-
brary and also implemented in CloudCompare (see references). For each point, it computes
the mean distance to all its k-nearest neighbours. The points that are farther than the average distance
plus a number of times (multiplier) the standard deviation are considered noise.

Usage

```r
sor(k = 10, m = 3, quantile = FALSE)
```

Arguments

- **k** numeric. The number of neighbours
- **m** numeric. Multiplier. The maximum distance will be: avg distance + m * std de-
  viation. If quantile = TRUE, m becomes the quantile threshold.
- **quantile** boolean. Modification of the original SOR to use a quantile threshold instead
  of a standard deviation multiplier. In this case the maximum distance will be:
  quantile(distances,probs = m)
References

https://pointclouds.org/documentation/tutorials/statistical_outlier.html

See Also

Other noise segmentation algorithms: ivf()

Examples

LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, filter = "-inside 273450 5274350 273550 5274450")

# Add some artificial outliers because the original
data set is 'clean'
set.seed(314)
id = round(runif(20, 0, npoints(las)))
set.seed(42)
err = runif(20, -50, 50)
las$Z[id] = las$Z[id] + err

las <- classify_noise(las, sor(15,7))

stdmetrics

Predefined standard metrics functions

Description

Predefined metrics functions intended to me used in *_metrics function such as grid_metrics, hexbin_metrics, cloud_metrics, tree_metrics, voxel_metrics or point_metrics. Each function comes with a convenient shortcuts for lazy coding. The lidR package aims to provide an easy way to compute user-defined metrics rather than to provide them. However, for efficiency and to save time, sets of standard metrics have been predefined (see details). Every function can be computed by every *_metrics functions however stdmetrics* are more pixel-based metrics, stdtreemetrics are more tree-based metrics and stdshapemetrics are more point-based metrics. For example the metric zmean computed by stdmetrics_z makes sense when computed at the pixel level but brings no information at the voxel level.

Usage

stdmetrics(x, y, z, i, rn, class, dz = 1, th = 2, zmin = 0)

stdmetrics_z(z, dz = 1, th = 2, zmin = 0)

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

stdmetrics_rn(rn, class = NULL)
stdmetrics

stdmetrics_pulse(pulseID, rn)
stdmetrics_ctrl(x, y, z)
stdtreemetrics(x, y, z)
stdshapemetrics(x, y, z)

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.
An object of class formula of length 2.
An object of class formula of length 2.
An object of class formula of length 2.
An object of class formula of length 2.
An object of class formula of length 2.
An object of class formula of length 2.
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 behaviour (see examples)

\[
stdmetrics \text{ is a combination of } stdmetrics_ctrl + stdmetrics_z + stdmetrics_i + stdmetrics_rn
\]
\[
stdtreemetrics \text{ is a special function that works with tree_metrics. Actually, it won’t fail with other functions but the output makes more sense if computed at the individual tree level.}
\]
\[
stdshapemetrics \text{ is a set of eigenvalue based feature described in Lucas et al, 2019 (see references).}
\]

References


See Also

cloud_metrics grid_metrics hexbin_metrics voxel_metrics tree_metrics point_metrics
Examples

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile, select = "*", filter = "-keep_random_fraction 0.5")

# All the predefined metrics
m1 <- grid_metrics(las, ~stdmetrics(X,Y,Z,Intensity,ReturnNumber,Classification,dz=1), res = 40)

# Convenient shortcut
m2 <- grid_metrics(las, .stdmetrics, res = 40)

# Basic metrics from intensities
m3 <- grid_metrics(las, ~stdmetrics_i(Intensity), res = 40)

# All the metrics from intensities
m4 <- grid_metrics(las, ~stdmetrics_i(Intensity, Z, Classification, ReturnNumber), res = 40)

# Convenient shortcut for the previous example
m5 <- grid_metrics(las, .stdmetrics_i, res = 40)

# 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(  # Num of returns above 2 divided by num of 1st returns
    above2aboven1st = x,
    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), res = 40)

# Users can write their own convenient shortcuts like this:
.myMetrics = ~myMetrics(Z, Intensity, ReturnNumber)
m11 <- grid_metrics(las, .myMetrics, res = 40)

---

Spatial Interpolation Algorithm
Description

This function is made to be used in grid_terrain or normalize_height. 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(..., extrapolate = knnidw(3, 1, 50))

Arguments

... unused
extrapolate 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 by default using knnidw.

See Also

Other spatial interpolation algorithms: knnidw(), kriging()

Examples

LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile, filter = "-inside 273450 5274350 273550 5274450")

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

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

track_sensor

Reconstruct the trajectory of the LiDAR sensor using multiple returns

Description

Use multiple returns to estimate the positioning of the sensor by computing the intersection in space of the line passing through the first and last returns. To work, this function requires a dataset where the 'gpstime', 'ReturnNumber', 'NumberOfReturns' and 'PointSourceID' attributes are properly populated, otherwise the output may be incorrect or weird. For LAScatalog processing it is recommended to use large chunks and large buffers (e.g. a swath width). The point cloud must not be normalized.
track_sensor

Usage

```r
track_sensor(
  las,
  algorithm,
  extra_check = TRUE,
  thin_pulse_with_time = 0.001,
  multi_pulse = FALSE
)
```

Arguments

- **las**: An object of class LAS or LAScatalog.
- **algorithm**: function. An algorithm to compute sensor tracking. lidR implements Roussel2020 and Gatziolis2019 (see respective documentation and examples).
- **extra_check**: boolean. Datasets are rarely perfectly populated, leading to unexpected errors. Time-consuming checks of data integrity are performed. These checks can be skipped as they account for an significant proportion of the computation time. See also section ‘Tests of data integrity’.
- **thin_pulse_with_time**: numeric. In practice, it is not useful to compute the position using all multiple returns. It is more computationally demanding but not necessarily more accurate. This keeps only one pulse every $x$ seconds. Set to 0 to use all multiple returns. Use 0 if the file has already been read with `filter = "-thin_pulses_with_time 0.001"`.
- **multi_pulse**: logical. TRUE only for systems with multiple pulses. Pulse ID must be recorded in the UserData attribute.

Value

A SpatialPointsDataFrame with the Z elevation stored in the table of attributes. Information about the time interval and the score of the positioning (according to the method used) are also in the table of attributes.

Test of data integrity

In theory, sensor tracking is a simple problem to solve as long as each pulse is properly identified from a well-populated dataset. In practice, many problems may arise from datasets that are populated incorrectly. Here is a list of problems that may happen. Those with a * denote problems already encountered and internally checked to remove weird points:

- 'gpstime' does not record the time at which pulses were emitted and thus pulses are not identifiable
- *A pulse (two or more points that share the same gpstime) is made of points from different flightlines (different PointSourceID). This is impossible and denotes an improperly populated PointSourceID attribute.
- 'ReturnNumber' and 'NumberOfReturns' are wrongly populated with either some ReturnNumber > NumberOfReturn or several first returns by pulses
For a given time interval, when weird points are not filtered, the position is not computed for this interval.

**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.
- **progress**: Displays a progression estimation.
- **output_files**: Saving intermediate results is disabled in 'sensor_tracking' because the output must be post-processed as a whole.
- **laz_compression**: write las or laz files
- **select**: is not supported. It is set by default to "xyzrntp"
- **filter**: Read only points of interest. By default it uses "-drop_single" and "-thin_pulses_with_time" to reduce the number of points loaded.

**Author(s)**

Jean-Francois Bourdon & Jean-Romain Roussel

**Examples**

```r
# A valid file properly populated
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile,
    select = "xyzrntp",
    filter = "-drop_single -thin_pulses_with_time 0.001")
#plot(las)

# pmin = 15 because it is an extremely small file
```
tree_metrics

Compute metrics for each tree

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 = ~max(Z), attribute = "treeID")

Arguments

las An object of class LAS or LAScatalog.
func formula. An expression to be applied to each tree. It works like in grid_metrics voxel_metrics or delineate_crowns and computes, in addition to tree locations a set of metrics for each tree.
attribute character. The column name of the attribute 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:
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.
- **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\}.
- **select**: Load only attributes of interest.
- **filter**: Read only points of interest.

**See Also**

Other metrics: `cloud_metrics()`, `grid_metrics()`, `plot_metrics()`, `point_metrics()`, `voxel_metrics()`
Examples

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

# NOTE: This dataset is already segmented
# plot(las, color = "treeID", colorPalette = pastel.colors(200))

# Default computes only Z max
metrics = tree_metrics(las)

# User-defined metrics - mean height and mean intensity for each tree
metrics = tree_metrics(las, ~list(Zmean = mean(Z), Imean = 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 classify_ground 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

- \( b \) numeric. This is the parameter \( b \) in Zhang et al. (2003) (eq. 4 and 5).
- \( dh_0 \) numeric. This is \( dh_0 \) in Zhang et al. (2003) (eq. 7).
- \( dh_{max} \) numeric. This is \( dh_{max} \) in Zhang et al. (2003) (eq. 7).
- \( s \) numeric. This is \( s \) in Zhang et al. (2003) (eq. 7).
- \( max\_ws \) numeric. Maximum window size to be used in filtering ground returns. This limits the number of windows created.
- \( exp \) 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:

\[
th_k = s \ast (w_k - w_{k-1}) \ast c + th_0
\]

Because the function `classify_ground` 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


Examples

```r
p = util_makeZhangParam()
```
**VCI**  

**Vertical Complexity Index**

**Description**

A fixed normalization of the entropy function (see references)

**Usage**

\[ \text{VCI}(z, \text{zmax}, \text{by} = 1) \]

**Arguments**

- \( z \) vector of \( z \) coordinates
- \( \text{zmax} \) numeric. Used to turn the function \( \text{entropy} \) to the function \( \text{vci} \).
- \( \text{by} \) numeric. The thickness of the layers used (height bin)

**Value**

A number between 0 and 1

**References**


**See Also**

\[ \text{entropy} \]

**Examples**

\[ z <- \text{runif}(10000, 0, 10) \]
\[ \text{VCI}(z, \text{by} = 1, \text{zmax} = 20) \]

\[ z <- \text{abs}(\text{rnorm}(10000, 10, 1)) \]
\[ # \text{expected to be closer to 0}. \]
\[ \text{VCI}(z, \text{by} = 1, \text{zmax} = 20) \]
voxelize_points  
**Voxelize a point cloud**

**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 $\text{mean}(\text{Intensity})$. Other attributes cannot be aggregated and are lost.

**Usage**

```r
voxelize_points(las, res)
```

**Arguments**

- `las` An object of class LAS or LAScatalog.
- `res` numeric. The resolution of the voxels. $\text{res} = 1$ for a $1x1x1$ cubic voxels. Optionally $\text{res} = c(1,2)$ 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.
• **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), \{ORIGINALLFILENAME\}.

• **select**: The function will write files equivalent to the original ones. Thus select = "*" and cannot be changed.

• **filter**: Read only points of interest.

**Examples**

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

las2 = voxelize_points(las, 2)
#plot(las2)
```

**voxel_metrics**

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**

```r
voxel_metrics(las, func, res = 1, ..., all_voxels = FALSE)
```

**Arguments**

- `las` An object of class LAS.
- `func` formula. An expression to be applied to each voxel (see also `grid_metrics`).
- `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).
- `...` Unused
- `all_voxels` boolean. By default the function returns only voxels that contain 1 or more points. Empty voxels do not exist as the metrics are undefined. If `all_voxels = TRUE` all the voxels are returned and metrics are NA for voxels with 0 points.

**Value**

It returns a `data.table` containing the metrics for each voxel. The table has the class `lasmetrics3d` enabling easier plotting. It also has an attribute `res` that stores the resolution.
See Also

Other metrics: `cloud_metrics()`, `grid_metrics()`, `plot_metrics()`, `point_metrics()`, `tree_metrics()`

Examples

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

# Cloud of points is voxelized with a 8-meter resolution and in each voxel
# the number of points is computed.
vm <- voxel_metrics(las, ~length(Z), 8)

# Cloud of points is voxelized with a 8-meter resolution and in each voxel
# the mean intensity of points is computed.
vm <- voxel_metrics(las, ~mean(Intensity), 8)
#plot(vm, color = "V1", colorPalette = heat.colors(50), trim = 60)

# Define your own metric function
myMetrics = function(i)
{
  ret = list(
    npoints = length(i),
    imean = mean(i)
  )

  return(ret)
}

voxels <- voxel_metrics(las, ~myMetrics(Intensity), 8)
#plot(voxels, color = "imean", colorPalette = heat.colors(50), trim = 60)
#etc.

attr(voxels, "res")
```

---

**watershed**

Individual Tree Segmentation Algorithm

**Description**

This function is made to be used in `segment_trees`. 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 and has been removed because imager is an orphaned package.
watershed

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 ?EBImage::watershed.

`ext` numeric. see ?EBImage::watershed.

`treetops` SpatialPointsDataFrame. Can be computed with `find_trees` or read from an external shapefile.

`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 `segment_trees` 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 = find_trees(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

```r
## Not run:
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
poi <- "-drop_z_below 0 -inside 481280 3812940 481320 3812980"
las <- readLAS(LASfile, select = "xyz", filter = poi)
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)
las <- segment_trees(las, watershed(chm))
plot(las, color = "treeID", colorPalette = col)
```
Snags Segmentation Algorithm

Description

This function is made to be used in `segment_snags`. It implements an algorithm 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

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

Arguments

- **neigh_radii**: 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.
- **low_int_thrsh**: numeric. The lower intensity threshold filtering value. See Wing et al. (2015) page 171. Default is 50.
- **uppr_int_thrsh**: numeric. The upper intensity threshold filtering value. See Wing et al. (2015) page 171. Default is 170.
- **pt_den_req**: numeric. Point density requirement based on plot-level point density defined classes. See Wing et al. (2015) page 172. Default is 3.
- **BBPRthrsh_mat**: 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 >= 55%).

Author(s)

Implementation by Andrew Sánchez Meador & Jean-Romain Roussel

References


Examples

LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
# Wing also included -keep_single
poi ="-keep_first -inside 481260 3812920 481310 3812960"
las <- readLAS(LASfile, select = "xyzi", filter = poi)

# 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 <- segment_snags(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 <- filter_poi(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, index = FALSE)

**Arguments**

- **las**: an object of class LAS.
- **file**: character. A character string naming an output file.
- **index**: boolean. Also write a lax file to index the points in the files

**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 = clip_rectangle(las, 684850, 5017850, 684900, 5017900)
writeLAS(subset, tempfile(fileext = ".laz"))
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

```r
## 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 'LAS,numeric,ANY'
x[i]

## 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

- `x`: A LAS* object
- `name`: A literal character string or a name (possibly backtick quoted).
- `value`: typically an array-like R object of a similar class as `x`.
- `i`: string, name of elements to extract or replace.
- `j`: Unused.
- `...`: Unused
- `drop`: Unused

Examples

```r
## 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

* cast
  as.spatial, 10
* datasets
  asprs, 10
  stdmetrics, 140
* digital surface model algorithms
  dsmtin, 45
  p2r, 102
  pitfree, 103
* filters
  filter_duplicates, 49
  filter_poi, 50
  filter_surfacepoints, 51
  filters, 48
* ground segmentation algorithms
  csf, 34
  pmf, 110
* individual tree detection algorithms
  lmf, 93
  manual, 94
* individual tree segmentation algorithms
  dalponte2016, 35
  li2012, 83
  silva2016, 137
  watershed, 154
* las utilities
  las_check, 80
  las_utilities, 81
* lidrpalettes
  lidrpalettes, 92
* metrics
  cloud_metrics, 32
  grid_metrics, 60
  plot_metrics, 108
  point_metrics, 111
  tree_metrics, 147
  voxel_metrics, 153
* noise segmentation algorithms
  ivf, 69
* normalize
  normalize_height, 97
  normalize_intensity, 100
* point cloud decimation algorithms
  homogenize, 66
  maxima, 95
  random, 117
  random_per_voxel, 118
* point-cloud based tree segmentation algorithms
  li2012, 83
* range
  track_sensor, 144
* raster based tree segmentation algorithms
  dalponte2016, 35
  silva2016, 137
  watershed, 154
* snags segmentation algorithms
  wing2015, 156
* spatial interpolation algorithms
  knnidw, 70
  kriging, 70
  tin, 143
- ,LAS,RasterLayer-method
  (normalize_height), 97
- ,LAS,lidRAlgorithm-method
  (normalize_height), 97
  .stdmetrics (stdmetrics), 140
  .stdmetrics_ctrl (stdmetrics), 140
  .stdmetrics_i (stdmetrics), 140
  .stdmetrics_pulse (stdmetrics), 140
  .stdmetrics_pn (stdmetrics), 140
  .stdmetrics_z (stdmetrics), 140
  .stdshapemetrics (stdmetrics), 140
  .stdtreemetrics (stdmetrics), 140
[,LAS,numerics,ANY-method
  ($<-$,LAS-method), 159
[,LAScatalog,ANY,ANY-method

161
add_attribute, 6
add_dtm3d (plot_3d), 107
add_flightlines3d (plot_3d), 107
add_lasattribute, 98
add_lasattribute (add_attribute), 6
add_lasattribute_manual (add_attribute), 6
add_lasrgb (add_attribute), 6
add_treetops3d (plot_3d), 107
appropriate functions, 123
area, 8, 117
area, LAS-method (area), 8
area, LAScatalog-method (area), 8
area, LASheader-method (area), 8
as.list.LASheader, 9
as.spatial, 10
available options, 21, 24, 26, 28, 30, 37,
40, 49, 57, 59, 61, 64, 98, 101, 109,
131, 134, 146, 148, 152
bbox, 13, 30
catalog (readLAScatalog), 122
catalog processing options, 18
catalog_apply, 5, 12, 85
catalog_boundaries, 16
catalog_intersect, 18
catalog_makechunks, 18
catalog_options_tools, 19, 124
catalog_retile, 21
catalog_sapply (catalog_apply), 12
catalog_select, 23
classify_ground, 24, 34, 70, 110, 149, 150
classify_noise, 25, 69, 139
classify_poi, 27
clip, 29
clip_circle (clip), 29
clip_polygon (clip), 29
clip_rectangle (clip), 29
clip_roi, 108, 109
clip_roi (clip), 29
classify_poi, 27
clip_transsect (clip), 29
cloud_metrics, 32, 62, 108, 109, 112, 140,
142, 148, 154
concaveman, 16, 33, 39, 64
count_not_quantized (las_utilities), 81
crs, 24, 34, 111
crs, LAS-method (projection), 115
crs, LAScatalog-method (projection), 115
crs, LASHeader-method (projection), 115
crs-class, 72
crs, LAS-method (projection), 115
crs, LAScatalog-method (projection), 115
dalponte2016, 35, 83, 133, 138, 155
data.table, 72, 73
decimate_points, 37, 51, 66, 95, 117, 118
delineate_crowns, 39, 147
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
density, LASHeader-method (area), 8
deprecated, 41
dsmtin, 45, 57, 102, 103
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41
density (area), 8
density, LAS-method (area), 8
density, LAScatalog-method (area), 8
deprecated, 41
density, LASHeader-method (area), 8
deprecated, 41

INDEX

filter_firstofmany (filters), 48
filter_ground (filters), 48
filter_last (filters), 48
filter_nth (filters), 48
filter_poi, 49, 50, 52, 122
filter_single (filters), 48
filter_surfacepoints, 49–51, 51, 139
filters, 48, 50–52
find_localmaxima, 52
find_trees, 36, 52, 53, 93, 94, 137, 155
forest.colors (lidrpalettes), 92
fwrite, 84
gap_fraction_profile, 55, 72
Gatziolis2019, 56, 145
get_lidr_threads (set_lidr_threads), 135
get_range (range_correction), 119
grid_canopy, 36, 45, 57, 102, 103, 137, 155
grid_density, 58
grid_metrics, 14, 32, 39, 58, 60, 62, 109, 111, 112, 140, 142, 147, 148, 153, 154
gridTerrain, 63, 70, 98, 99, 102, 144
height.colors (lidrpalettes), 92
hexbin_metrics, 140, 142
hexbin_metrics (deprecated), 41
highest, 37, 51
highest (maxima), 95
homogenize, 37, 66, 95, 117, 118
index (lidR-spatial-index), 89
index<- (lidR-spatial-index), 89
interpret_waveform, 67
is, 68
is.parallelised, 86
is.quantized (las_utilities), 81
ivf, 26, 69, 140
knnidw, 64, 70, 71, 98, 102, 144
krige, 70
kriging, 64, 70, 70, 98, 102, 144
LAD, 32, 55, 61, 71
LAS, 6, 7, 24, 26, 27, 29, 37, 39, 48–51, 57, 59, 60, 64, 80, 89, 98, 100, 104, 109, 121, 122, 129, 131, 133, 145, 147, 152, 158
LAS (LAS-class), 72
LAS-class, 72
las_check, 80, 82
las_quantize (las_utilities), 81
las_reoffset (las_utilities), 81
las_rescale (las_utilities), 81
las_update (las_utilities), 81
las_utilities, 80, 81
lasadd, 41
lasadddata (deprecated), 41
lasaddextrabytes (deprecated), 41
lasaddextrabytes_manual (deprecated), 41
LASBRIDGE (asprs), 10
LASBUILDING (asprs), 10
LAScatalog, 12, 18–21, 23, 24, 26–30, 37, 39, 40, 49, 51, 57, 59–61, 64, 80, 89, 90, 98, 100, 101, 104, 109, 121, 123, 129, 131, 133, 134, 145–148, 152
LAScatalog class, 12
LAScatalog class documentation, 123
LAScatalog-class, 31, 74, 84, 124
lascheck, 41
lascheck (deprecated), 41
lasclip, 41
lasclip (deprecated), 41
lasclipCircle (deprecated), 41
lasclipPolygon (deprecated), 41
lasclipRectangle (deprecated), 41
lasdetectshape, 41
lasdetectshape (deprecated), 41
lasfilter, 41
lasfilter (deprecated), 41
lasfilterdecimate (deprecated), 41
lasfilterduplicates (deprecated), 41
lasfilterfirst (deprecated), 41
lasfilterfirstlast (deprecated), 41
lasfilterfirstofmany (deprecated), 41
lasfilterground (deprecated), 41
lasfilterlast (deprecated), 41
lasfilternth (deprecated), 41
lasfiltersingle (deprecated), 41
lasfiltersurfacepoints, 41
lasfiltersurfacepoints (deprecated), 41
lasflightline, 41
lasflightline (deprecated), 41
LASGROUND (asprs), 10
lasground, 41
lasground (deprecated), 41
LASHeader, 72, 73, 78, 104, 125
LASHeader-class, 79
LASHIGHVEGETATION (asprs), 10
LASKEYPOINT (asprs), 10
LASLOWPOINT (asprs), 10
LASLOWVEGETATION (asprs), 10
LASMEDIUMVEGETATION (asprs), 10
lasmergespatial, 41
lasmergespatial (deprecated), 41
LASNOISE (asprs), 10
LASNONCLASSIFIED (asprs), 10
lasnormalize, 41
lasnormalize (deprecated), 41
laspulse, 41
laspulse (deprecated), 41
LASRAIL (asprs), 10
lasrangeposition, 41
lasrangeposition (deprecated), 41
lasremoveextrabytes (deprecated), 41
lasreoffset, 41
lasreoffset (deprecated), 41
lasrescale, 41
lasrescale (deprecated), 41
LASROADSURFACE (asprs), 10
lasscanline (deprecated), 41
lasscanlines, 41
lassmooth, 41
lassmooth (deprecated), 41
lassnags, 41
lassnags (deprecated), 41
lastransform (deprecated), 41
LASTRANSFORMATIONTOWER (asprs), 10
lastrees, 41
lastrees (deprecated), 41
LASUNCLASSIFIED (asprs), 10
lasunnormalize (deprecated), 41
lasunsmooth (deprecated), 41
lasvexelize, 41
lasvexelize (deprecated), 41
LASWATER (asprs), 10
LASWIRECONDUCTOR (asprs), 10
LASWIREGUARD (asprs), 10
li2012, 37, 83, 86, 87, 133, 138, 155
lidR (lidR-package), 5
lidR-LAScatalog-drivers, 77, 84

lidR-package, 5
lidR-parallelism, 86, 135
lidR-spatial-index, 89
lidrpalettes, 92
list.files, 124
lwf, 52, 53, 86, 87, 93, 95
lowest, 37
lowest (maxima), 95

manual, 53, 93, 94
mapview, 104, 105
matrix, 30
maxima, 66, 95, 117, 118
mcwatershed (watershed), 154
merge Spatial, 96

normalize_height, 70, 97, 101, 144
normalize_intensity, 99, 100, 119
npoints (area), 8
npoints, LAS-method (area), 8
npoints, LAScatalog-method (area), 8
npoints, LASHeader-method (area), 8

opt_chunk_alignment, 76
opt_chunk_alignment
  (catalog_options_tools), 19
opt_chunk_alignment<-
  (catalog_options_tools), 19
opt_chunk_buffer, 76
opt_chunk_buffer
  (catalog_options_tools), 19
opt_chunk_buffer<-
  (catalog_options_tools), 19
opt_chunk_size, 76
opt_chunk_size (catalog_options_tools), 19
opt_chunk_size<-
  (catalog_options_tools), 19
opt_filter, 77
opt_filter (catalog_options_tools), 19
opt_filter<-(catalog_options_tools), 19
opt_independent_files
  (catalog_options_tools), 19
opt_independent_files<-
  (catalog_options_tools), 19
opt_laz_compression
  (catalog_options_tools), 19
opt_laz_compression<-
  (catalog_options_tools), 19
opt_merge(catalog_options_tools), 19
opt_merge<-(catalog_options_tools), 19
opt_output_files, 77
opt_output_files
  (catalog_options_tools), 19
opt_output_files<-(
  (catalog_options_tools), 19
opt_progress, 75
opt_progress(catalog_options_tools), 19
opt_progress<-(catalog_options_tools), 19
opt_restart<-(catalog_options_tools), 19
opt_select, 77
opt_select(catalog_options_tools), 19
opt_select<-(catalog_options_tools), 19
opt_stop_early, 76
opt_stop_early(catalog_options_tools), 19
opt_stop_early<-(
  (catalog_options_tools), 19
opt_wall_to_wall, 76
opt_wall_to_wall
  (catalog_options_tools), 19
opt_wall_to_wall<-(
  (catalog_options_tools), 19
p2r, 45, 57, 102, 103
pastel.colors(lidrpalettes), 92
pitfree, 45, 57, 102, 103
plot, 23, 95, 104, 105, 107
plot, LAS, missing-method (plot), 104
plot, LAScatalog, missing-method (plot), 104
plot, LASheader, missing-method (plot), 104
plot, lasmetrics3d, 106
plot_3d, 107
plot_dtm3d (plot_3d), 107
pmf, 24, 35, 110
point_eigenvalues (segment_shapes), 129
point_metrics, 32, 62, 109, 111, 130, 140, 142, 148, 154
points3d, 105
Polygon, 30
print, 114
print, LAS-method (print), 114
print.lidRAlgorithm (print), 114
projection, 115
projection, LAS-method (projection), 115
projection, LAScatalog-method (projection), 115
projection, LASheader-method (projection), 115
projection<-, LAS-method (projection), 115
projection<-, LAScatalog-method (projection), 115
quantize (las_utilities), 81
random, 37, 66, 95, 117, 118
random.colors (lidrpalettes), 92
random_per_voxel, 37, 66, 95, 117, 118
range_correction, 100, 119
raster::extent, 47
RasterLayer, 98
rbind, 120
read+LAS(), 89
readALSLAS (readLAS), 120
readALSLAScatalog (readLAScatalog), 122
readDAPLAS (readLAS), 120
readDAPLAScatalog (readLAScatalog), 122
readLAS, 13, 74, 75, 77, 120, 125
readLAScatalog, 75, 122
readLASheader, 125
readMSLAS (readLAS), 120
readTLSLAS (readTLS), 120
readTLSLAScatalog (readTLScatalog), 122
readUAVLAS (readUAV), 120
readUAVLAScatalog (readUAVcatalog), 122
remove_lasattribute (add_attribute), 6
retrieve_flightlines (retrieve_pulses), 126
retrieve_pulses, 66, 117, 126
retrieve_scanlines (retrieve_pulses), 126
rlas::header_create(), 78
rlas::read.las, 67, 122
Roussel2020, 127, 145
rumple_index, 128
segment_shapes, 129, 135
segment_snags, 131, 156
segment_trees, 35, 36, 83, 132, 137, 154, 155
sensor (lidR-spatial-index), 89
sensor<-(lidR-spatial-index), 89
sensor_tracking, 41
sensor_tracking (deprecated), 41
set_lidr_threads, 3, 86, 135
setDThreads, 135
shape_detection, 135
shp_hplane, 129
shp_hplane (shape_detection), 135
shp_line, 129
shp_line (shape_detection), 135
shp_plane, 129
shp_plane (shape_detection), 135
silva2016, 37, 83, 133, 137, 138, 155
SimpleFeature, 30
smooth_height, 138
sor, 26, 69, 139
Spatial, 73, 74, 159
spatial indexing, 72, 73, 75, 121, 123
SpatialPoints, 30
SpatialPointsDataFrame, 30
SpatialPolygons, 30
SpatialPolygonsDataFrame, 30
spheres3d, 108
st_write, 84, 85
stdmetrics, 32, 61, 140
stdmetrics_ctrl (stdmetrics), 140
stdmetrics_i (stdmetrics), 140
stdmetrics_pulse (stdmetrics), 140
stdmetrics_rn (stdmetrics), 140
stdmetrics_tree, 148
stdmetrics_z (stdmetrics), 140
stdshapemetrics (stdmetrics), 140
stdtreemetrics (stdmetrics), 140
storable_coordinate_range
(las_utilities), 81
summary (print), 114
summary, LAS-method (print), 114
summary, LAScatalog-method (print), 114
surface3d, 108
tin, 64, 70, 71, 98, 102, 143
track_sensor, 56, 119, 127, 144
tree_detection, 41
tree_detection (deprecated), 41
tree_hull, 41
tree_hulls (deprecated), 41
tree_metrics, 32, 39, 62, 109, 112, 140, 142, 147, 154
unnormalize_height, 97
unnormalize_height (normalize_height), 97
unsmooth_height (smooth_height), 138
util_makeZhangParam, 110, 149
VCI, 32, 46, 61, 151
vgm, 71
voxel_metrics, 32, 39, 62, 109, 112, 140, 142, 147, 148, 153
voxelize_points, 106, 152
watershed, 37, 83, 133, 138, 154
wing2015, 131, 156
wkt, LAS-method (projection), 115
wkt, LASheader-method (projection), 115
wkt<-> (projection), 115
wkt<->, LAS-method (projection), 115
wkt<->, LASheader-method (projection), 115
writeLAS, 7, 84, 158
writeRaster, 84