Package: lidR (via r-universe)

October 8, 2024

Type Package

Title Airborne LiDAR Data Manipulation and Visualization for Forestry Applications

Version 4.1.2

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/r-lidar/lidR

BugReports https://github.com/r-lidar/lidR/issues

License GPL-3

Depends R (>= 3.5.0), methods

Imports classInt, data.table (>= 1.12.0), glue, grDevices, lazyeval, Rcpp (>= 1.0.3), rgl, rlas (>= 1.5.0), sf, stats, stars, terra (>= 1.5-17), tools, utils

Suggests EBImage, future, geometry, gstat, raster, RCSF, RMCC, rjson, mapview, mapedit, progress, sp, testthat (>= 2.1.0), knitr, rmarkdown

RoxygenNote 7.3.1

LinkingTo BH (>= 1.72.0),Rcpp,RcppArmadillo

Encoding UTF-8

ByteCompile true

VignetteBuilder knitr

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), Benoît St-Onge [cph] (Author of the 'chm_prep' function)

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

Repository CRAN

Date/Publication 2024-07-09 08:40:02 UTC

Contents

`lidR-package` *lidR: airborne LiDAR for forestry applications*

Description

lidR provides a set of tools to manipulate airborne LiDAR data in forestry contexts. The package works with .las or .laz files. The toolbox includes algorithms for DSM, CHM, DTM, ABA, normalisation, tree detection, tree segmentation, tree delineation, colourization, validation and other tools, as well as a processing engine to process broad LiDAR coverage 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.raster.default` The functions that return a raster are raster agnostic meaning that they work either with rasters from packages ‘raster‘, ‘stars‘ or ‘terra‘. By default they return rasters from ‘stars‘. Can be one of "raster", "stars" or "terra". Default: "terra"

`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:

- Florian De Boissieu (Fixed bugs and improved catalog features) [contributor]
- Andrew Sánchez Meador (Implemented wing2015() for segment_snags()) [contributor]
- Bourdon Jean-François (Contributed to Roussel2020() for track_sensor()) [contributor]

- Gatziolis Demetrios (Implemented Gatziolis2019() for track_sensor()) [contributor]
- Leon Steinmeier (Contributed to parallelization management) [contributor]
- Stanislaw Adaszewski (Author of the C++ concaveman code) [copyright holder]
- Benoît St-Onge (Author of the 'chm_prep' function) [copyright holder]

See Also

Useful links:

- https://github.com/r-lidar/lidR
- Report bugs at https://github.com/r-lidar/lidR/issues

`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)

add_lasnir(las, NIR)

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, NIR	integer. RGB and NIR values. Values are automatically scaled to 16 bits if they are coded on 8 bits (0 to 255).

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

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

print(las)
print(header(las))

x <- 1:30

las <- add_attribute(las, x, "mydata")
print(las)        # The las object has a new attribute called "mydata"
print(header(las)) # 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(header(las)) # 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_attribute(las, x, "newattr")
las <- add_lasattribute(las, name = "newattr", desc = "Amplitude")
print(header(las))

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

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

---

aggregate *Metric derivation at different levels of regularization*

Description

`template_metrics()` computes a series of user-defined descriptive statistics for a LiDAR dataset within each element of a template. Depending on the template it can be for each pixel of a raster (area-based approach), or each polygon, or each segmented tree, or on the whole point cloud. Other functions are convenient and simplified wrappers around `template_metrics()` and are expected to be the actual functions used. See Details and Examples.

Usage

```
cloud_metrics(las, func, ...)

crown_metrics(
  las,
```

```
  func,
  geom = "point",
  concaveman = c(3, 0),
  attribute = "treeID",
  ...
)

hexagon_metrics(las, func, area = 400, ...)

pixel_metrics(las, func, res = 20, start = c(0, 0), ...)

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

polygon_metrics(las, func, geometry, ...)

template_metrics(las, func, template, filter = NULL, by_echo = "all", ...)

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

Arguments

las	An object of class LAS or LAScatalog.
func	formula or expression. An expression to be applied to each element of the template (see section "Parameter func").
...	propagated to template_metrics i.e. filter and by_echo. pixel_metrics() also supports pkg = "terra|raster|stars" to get an output in SpatRaster, Raster* or stars format. Default is getOption("lidR.raster.default").
geom	character. Geometry type of the output. Can be 'point', 'convex', 'concave' or 'bbox'.
concaveman	numeric. Only if type = "concave". Vector with the two parameters of the function concaveman.
attribute	character. The column name of the attribute containing tree IDs. Default is "treeID"
area	numeric. Area of the hexagons
res	numeric. The resolution of the output. Can optionally be a RasterLayer or a stars or a SpatRaster. In that case the raster is used as the template.
start	vector of x and y coordinates for the reference raster. Default is (0,0) meaning that the grid aligns on (0,0). Not consiered if res is a raster
geometry	A spatial vector object. sp and sf' objects are supported. plot_metrics() supports point and polygons but polygon_metrics() supports only polygons.
radius	numeric. If the geometry is spatial points a radius must be defined. Support one radius or a vector of radii for variable plot sizes.
template	can be of many types and corresponds to the different levels of regularization. RasterLayer/stars/SpatRaster, sf/sfc (polygons), numeric, bbox, NULL. The metrics are computed for each element of the template. See examples.

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

`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.

Details

`pixel_metrics` Area-based approach. Computes metrics in a square tessellation. The output is a raster.

`hexagon_metrics` Computes metrics in an hexagon tessellation. The output is a `sf/sfc_POLYGON`

`plot_metrics` Computes metrics for each plot of a ground inventory by 1. clipping the plot inventories with clip_roi, 2. computing the user's metrics for each plot with cloud_metrics, and 3. combining spatial data and metrics into one data.frame ready for statistical modelling with `cbind`. The output is of the class of the input.

`cloud_metrics` Computes a series of user-defined descriptive statistics for an entire point cloud. The output is a `list`

`crown_metrics` 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. The output can be spatial points or spatial polygons (`sf/sfc_POINT` or `sf/sfc_POLYGON`)

`voxel_metrics` Is a 3D version of `pixel_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. The output is a `data.frame`

`point_metrics` Is a bit more complex and is documented in point_metrics

Value

Depends on the function, the template and the number of metrics. Can be a `RasterLayer`, a `RasterBrick`, a `stars`, a `SpatRaster` a `sf/sfc`, a `list`, a `SpatialPolygonDataFrame`, or a `data.table`. Functions are supposed to return an object that is best suited for storing the level of regularization needed.

Parameter `func`

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

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

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

```
pixel_metrics(las, f(Z), res = 20)
voxel_metrics(las, f(Intensity), res = 2)
```

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. `template_metrics` will dispatch the point cloud in the user's function.

Examples

```
LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las <- readLAS(LASfile, filter = "-keep_random_fraction 0.5")
col <- sf::sf.colors(15)
fun1 <- ~list(maxz = max(Z))
fun2 <- ~list(q85 = quantile(Z, probs = 0.85))

set_lidr_threads(1) ; data.table::setDTthreads(1) # for cran only

# ================
# CLOUD METRICS
# ================

cloud_metrics(las, .stdmetrics_z)

# ================
# PIXEL METRICS
# ================

m <- pixel_metrics(las, fun1, 20)
#plot(m, col = col)

# ================
# PLOT METRICS
# ================

shpfile <- system.file("extdata", "efi_plot.shp", package="lidR")
inventory <- sf::st_read(shpfile, quiet = TRUE)
inventory # contains an ID and a Value Of Interest (VOI) per plot

m <- plot_metrics(las, fun2, inventory, radius = 11.28)
#plot(header(las))
#plot(m["q85"], pch = 19, cex = 3, add = TRUE)


# Works with polygons as well
inventory <- sf::st_buffer(inventory, 11.28)
#plot(header(las))
#plot(sf::st_geometry(inventory), add = TRUE)
m <- plot_metrics(las, .stdmetrics_z, inventory)
#plot(m["zq85"], pch = 19, cex = 3, add = TRUE)


# ================
# VOXEL METRICS
# ================

m <- voxel_metrics(las, length(Z), 8)
```

```
m <- voxel_metrics(las, mean(Intensity), 8)
#plot(m, color = "V1", colorPalette = heat.colors(50), trim = 60)
#plot(m, color = "V1", colorPalette = heat.colors(50), trim = 60, voxel = TRUE)

# ================
# CROWN METRICS
# ================

# Already tree-segmented point cloud
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
trees <- readLAS(LASfile, filter = "-drop_z_below 0")

metrics <- crown_metrics(trees, .stdtreemetrics)
#plot(metrics["Z"], pch = 19)

metrics <- crown_metrics(trees, .stdtreemetrics, geom = "convex")
#plot(metrics["Z"])

metrics <- crown_metrics(trees, .stdtreemetrics, geom = "bbox")
#plot(metrics["Z"])


metrics <- crown_metrics(trees, .stdtreemetrics, geom = "concave")
#plot(metrics["Z"])

# ================
# ARGUMENT FILTER
# ================

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

# Compute using only some points: optimized
# faster and uses less memory. No intermediate object
metrics = pixel_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 = pixel_metrics(las, mean(Z), 20)

# ================
# ARGUMENT BY_ECHO
# ================

func = ~list(avgI = mean(Intensity))
echo = c("all", "first","multiple")

# func defines one metric but 3 are computed respectively for: (1) all echo types,
# (2) for first returns only and (3) for multiple returns only
metrics <- pixel_metrics(las, func, 20, by_echo = echo)
#plot(metrics, col = heat.colors(25))
```

```
cloud_metrics(las, func, by_echo = echo)

## Not run:
# ================
# TEMPLATE METRICS
# ================

# a raster as template
template <- raster::raster(extent(las), nrow = 15, ncol = 15)
raster::crs(template) <- crs(las)
m <- template_metrics(las, fun1, template)
#plot(m, col = col)

# a sfc_POLYGON as template
sfc <- sf::st_as_sfc(st_bbox(las))
template <- sf::st_make_grid(sfc, cellsize = 20, square = FALSE)
m <- template_metrics(las, fun1, template)
#plot(m)

# a bbox as template
template <- st_bbox(las) + c(50,30,-50,-70)
plot(sf::st_as_sfc(st_bbox(las)), col = "gray")
plot(sf::st_as_sfc(template), col = "darkgreen", add = TRUE)
m <- template_metrics(las, fun2, template)
print(m)

# ================
# CUSTOM METRICS
# ================

# Define a function that computes custom metrics
# in an R&D perspective.
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)
}

# example with a stars template
template <- stars::st_as_stars(st_bbox(las), dx = 10, dy = 10)
m <- template_metrics(las, myMetrics(Z, Intensity), template)
#plot(m, col = col)

## End(Not run)
```

---

as *Transform to a list*

Description

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

Usage

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

Arguments

x	A LASheader object
...	unused

---

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
```

LASROADSURFACE

LASWIREGUARD

LASWIRECONDUCTOR

LASTRANSMISSIONTOWER

LASBRIGDE

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.

Examples

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

print(LASGROUND)
```

`catalog_apply` *LAScatalog processing engine*

Description

This function gives users access to the LAScatalog processing engine. It allows the application of a user-defined routine over a collection of LAS/LAZ files. The LAScatalog processing engine 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_map()` is a simplified version of `catalog_apply()` that suits for 90% of use cases.

`catalog_sapply()` is a previous attempt to provide simplified version of `catalog_apply()`. Use `catalog_map()` instead.

Usage

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

catalog_sapply(ctg, FUN, ..., .options = NULL)

catalog_map(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 neighbouring 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'. `catalog_map()` takes care of removing automatically the results computed in the buffered area to avoid unexpected output with duplicated entries or conflict between values computed twice. It does that in predefined way that may not suit all cases. `catalog_apply()` does not remove the buffer and leave users free to handle this in a custom way. This is why `catalog_apply()` is more complex but gives more freedom to build new applications.

Buffered data

The LAS objects loaded in theses functions 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` of `catalog_apply` 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 file collection
- The function readLAS can be used with a `LAScluster`
- The function extent or st_bbox can be used with a `LAScluster` and they return the bounding box of the chunk without the buffer. It must be used to clip the output and remove the buffered region (see examples).

A user-defined function must be templated like this:

```
myfun <- function(chunk, ...) {
   # Load the chunk + buffer
   las <- readLAS(chunk)
   if (is.empty(las)) return(NULL)

   # do something
   output <- do_something(las, ...)

   # remove the buffer of the output
   bbox <- bbox(chunk)
   output <- remove_buffer(output, bbox)
   return(output)
}
```

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 chunk was empty.

`catalog_map` is much simpler (but less versatile). It automatically takes care of reading the chunk and checks if the point cloud is empty. It also automatically crop the buffer. The way it crops the buffer suits for most cases but for some special cases it may be advised to handle this in a more specific way i.e. using `catalog_apply()`. For `catalog_map()` the first argument is a `LAS` and the template is:

```
myfun <- function(las, ...) {
   # do something
   output <- do_something(las, ...)
   return(output)
}
```

.options

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 `buffer = 0` is forbidden. Users can add the same constraints to protect against inappropriate options. The `.options` argument is a `list` that allows users to tune the behaviour of the processing engine.

- `drop_null = FALSE`: not intended to be used by regular users. The engine does not remove NULL outputs
- `need_buffer = TRUE`: the function complains if the buffer is 0.
- `need_output_file = TRUE` the function complains if no output file template is provided.
- `raster_alignment = ...` the function checks the alignment of the chunks. This option is important if the output is a raster. See below for more details.
- `automerge = TRUE` by default the engine returns a `list` with one item per chunk. If `automerge = TRUE`, it tries to merge the outputs into a single object: a `Raster*`, a `Spatial*`, a `LAS*`, an `sf`, a `stars` similarly to other functions of the package. This is a fail-safe option so in the worst case, if the merging fails, the `list` is returned. This is activated by `catalog_map` making it simpler.
- `autoread = TRUE`. Introduced in v3.0.0 this option enables to get rid of the first steps of the function i.e `readLAS()` and `if (is.empty())`. In this case the function must take two objects as input, first a `LAS` object and second a `bbox` from `sf`. This is activated by `catalog_map` making it simpler.
- `autocrop = TRUE`. Introduced in v4.0.0 this option enables to get rid of the buffer crop steps of the function i.e `something <- remove_buffer(something, bbox)`. In this case the function must take one `LAS` object as input. This is activated by `catalog_map` making it simpler.

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

```
# check if 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))
```

Examples

```
# More examples might be avaible in the official lidR vignettes or
# on the github book <https://jean-romain.github.io/lidRbook/>

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

# 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 users.
my_tree_detection_method <- function(chunk, ws)
{
  # The chunk argument is a LAScluster object. The users do 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(chunk)
  if (is.empty(las)) return(NULL)

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

  # ttops is an sf object 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  <- st_bbox(chunk)
  ttops <- sf::st_crop(ttops, bbox)
  return(ttops)
}

# 2. Build a collection of file
# (here, a single file LAScatalog for the purposes of this simple example).
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
ctg <- readLAScatalog(LASfile)
plot(ctg)

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

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

plot(output)
```

```
## ========================================================================
## Example 1: simplified. There is nothing that requires special data
## manipulation in the previous example. Everything can be handled automatically
##========================================================================

# 1. Build the user-defined function that analyzes a point cloud.
my_tree_detection_method <- function(las, ws)
{
  # Find the tree tops using a user-developed method
  # (here simply a LMF for the example).
  ttops <- locate_trees(las, lmf(ws))
  return(ttops)
}

# 2. Build a project
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
ctg <- readLAScatalog(LASfile)
plot(ctg)

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

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


## Not run:
## ================================================
## Example 2: compute a rumple index on surface points
## ================================================

rumple_index_surface = function(las, res)
{
  las    <- filter_surfacepoints(las, 1)
  rumple <- pixel_metrics(las, ~rumple_index(X,Y,Z), res)
  return(rumple)
}

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

opt_chunk_buffer(ctg) <- 1
opt_chunk_size(ctg)   <- 140     # small because this is a dummy example.
opt_select(ctg)       <- "xyz"   # read only the coordinates.
```

```
opt    <- list(raster_alignment = 20) # catalog_apply will adjust the chunks if required
output <- catalog_map(ctg, rumple_index_surface, res = 20, .options = opt)

plot(output, col = height.colors(25))

## End(Not run)
```

`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 st_concave_hull function. When all the hulls are computed it updates the LAScatalog to set the true polygons instead of the bounding boxes.

Usage

```
catalog_boundaries(ctg, ...)
```

Arguments

`ctg`	A LAScatalog
`...`	propagated to st_concave_hull

Value

A LAScatalog with true boundaries

Non-supported LAScatalog options

The options ‘select‘, ‘output files‘, ‘chunk size‘, ‘chunk buffer‘, ‘chunk alignment‘ are not supported and not respected in ‘catalog_boundaries*‘ because the function must always process by file, without buffer and knows which attributes to load.

Examples

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

`catalog_intersect` *Subset a LAScatalog*

Description

Subset a LAScatalog interactively using the mouse. 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.

Usage

```
catalog_intersect(
  ctg,
  y,
  ...,
  subset = c("subset", "flag_unprocessed", "flag_processed")
)

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

Arguments

`ctg`	A LAScatalog
`y`	'bbox', 'sf', 'sfc', 'Extent', 'Raster*', 'Spatial*' objects
`...`	ignored
`subset`	character. By default it subsets the collection It is also possible to flag the files to maintain the collection as a whole but process only a subset of its content. `flag_unprocessed` flags the files that will not be processed. `flag_processed` flags the files that will be processed.
`mapview`	logical. If `FALSE`, use R base plot instead of mapview (no pan, no zoom, see also plot)

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)
```

`catalog_retile` *Retile a LAScatalog*

Description

Splits or merges files to reshape the original files collection (.las or .laz) into smaller or larger files. It also enables the addition or removal of a buffer around the tiles. Internally, the function reads and writes the chunks defined by the internal processing options of a LAScatalog. 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). **This function is does not load the point cloud into R memory** It streams from input file(s) to output file(s) and can be applied to large point-cloud with low memory computer.

Note that this function is not actually very useful because `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

Non-supported LAScatalog options

The option `select` is not supported and not respected because it always preserves the file format and all the attributes. `select = "*"` is imposed internally.

Examples

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

# Create a new set of 200 x 200 m.las files with first returns only
opt_chunk_buffer(ctg) <- 0
opt_chunk_size(ctg) <- 200
opt_filter(ctg) <- "-keep_first"
opt_chunk_alignment(ctg) <- c(275, 90)
opt_output_files(ctg) <- paste0(tempdir(), "/retile_{XLEFT}_{YBOTTOM}")
```

```
# preview the chunk pattern
plot(ctg, chunk = TRUE)

newctg = catalog_retile(ctg)

plot(newctg)

# Create a new set of 200 x 200 m.las files
# but extended with a 50 m buffer in the folder

opt_chunk_buffer(ctg) <- 25
opt_chunk_size(ctg) <- 200
opt_filter(ctg) <- ""
opt_chunk_alignment(ctg) <- c(275, 90)
opt_output_files(ctg) <- paste0(tempdir(), "/{XLEFT}_{YBOTTOM}_buffered")
newctg = catalog_retile(ctg)

plot(newctg)


## Not run:
# 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"
opt_output_files(ctg) <- paste0(tempdir(), "/{ORIGINALFILENAME}_first_2m")
newctg = catalog_retile(ctg)

## End(Not run)
```

`classify` *Classify points*

Description

Classify points that meet some criterion and/or that belong in a region of interest. The functions updates the attribute `Classification` of the LAS object according to las specifications

Usage

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

classify_noise(las, algorithm)

classify_poi(
  las,
  class,
```

```
  poi = NULL,
  roi = NULL,
  inverse_roi = FALSE,
  by_reference = FALSE
)
```

Arguments

las	An object of class LAS or LAScatalog.
algorithm	An algorithm for classification. lidR has has: sor, ivf for noise classification, and pmf, csf, mcc for ground classification (see respective documentation).
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 attributes 'ReturnNumber' or 'NumberOfReturns' are absent, 'last_returns' is turned to FALSE automatically.
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*, from sp or a sf/sfc_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).

Details

classify_noise Classify points as 'noise' (outliers) with several possible algorithms. lidR has: sor, ivf. The points classified as 'noise' are assigned a value of 18.

classify_ground Classify points as 'ground' with several possible algorithms. lidR has pmf, csf and mcc. The points classified as 'ground' are assigned a value of 2

classify_poi Classify points that meet some logical criterion and/or that belong in a region of interest with class of choice.

Non-supported LAScatalog options

The option select is not supported and not respected because it always preserves the file format and all the attributes. select = "*" is imposed internally.

Examples

```
# ===============
# Classify ground
# ===============

if (require(RCSF, quietly = TRUE))
{
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn", filter = "-inside 273450 5274350 273550 5274450")
```

```
# (Parameters chosen mainly for speed)
mycsf <- csf(TRUE, 1, 1, time_step = 1)
las <- classify_ground(las, mycsf)
#plot(las, color = "Classification")
}

# ===============
# Classify noise
# ===============

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

# Add 20 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

# Using IVF
las <- classify_noise(las, ivf(5,2))
#plot(las, color = "Classification")

# Remove outliers using filter_poi()
las_denoise <- filter_poi(las, Classification != LASNOISE)

# ===============
# Classify POI
# ===============

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 collection of files (LAScatalog object).

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. spatial points, spatial polygons in sp or sf/sfc format, Extent, bbox, 2x2 matrix
...	in clip_roi: optional supplementary options (see supported geometries). Unused in other functions
xleft, ybottom, xright, ytop	numeric. coordinates of one or several rectangles.
xpoly, ypoly	numeric. x coordinates of a polygon.
xcenter, ycenter	numeric. x coordinates of on or several disc centres.
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.

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, or a LAScatalog if the queries are immediately written into files without loading anything in R.

Non-supported LAScatalog options

The option chunk size, buffer, chunk alignment and select are not supported by clip_* because they are meaningless in this context.

Examples

```
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)

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

plot(tr1, axis = TRUE, clear_artifacts = FALSE)

## End(Not run)
```

---

decimate_points *Decimate a LAS object*

Description

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.

Non-supported LAScatalog options

The option 'select' is not supported and not respected because it always preserves the file format and all the attributes. 'select = "*"' is imposed internally.
The options 'chunk buffer' is not supported and not respected because it is not needed.

Examples

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

# Select points randomly to reach an overall density of 1
thinned1 <- decimate_points(las, random(1))
#plot(rasterize_density(las))
#plot(rasterize_density(thinned1))

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

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

---

deprecated *Deprecated functions in lidR*

Description

These functions are provided for compatibility with older versions of lidR but are deprecated. They will progressively print a message, throw a warning and eventually be removed. The links below point to the documentation of the new names. In version 4 they now throw an error. In version 4.1 they ill be removed definitively.

lasadd lascheck lasclip lasdetectshape lasfilter 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)

lasrangecorrection(
  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")

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(...)

filter_surfacepoints(las, res)

## S3 method for class 'LAS'
filter_surfacepoints(las, res)

## S3 method for class 'LAScatalog'
filter_surfacepoints(las, res)
```

Arguments

las	See the new functions that replace the old ones
geometry	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`, `xoffset`, `yoffset`, `zoffset`
See the new functions that replace the old ones

`size`, `method`, `shape`, `sigma`
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

`dsm_pitfree` *Digital Surface Model Algorithm*

Description

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

Usage

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

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).
highest	bool. By default it keeps only the highest point per pixel before to triangulate to decrease computation time. If highest = FALSE all first returns are used.

References

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

See Also

Other digital surface model algorithms: dsm_point2raster, dsm_tin

Examples

```
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 <- rasterize_canopy(las, res = 0.5, dsmtin())
plot(chm, col = col)

# Khosravipour et al. pitfree algorithm
chm <- rasterize_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)

# Because the TIN interpolation is done within the convex hull of the point cloud
# dummy pixels are interpolated that are correct according to the interpolation
# method used, but meaningless in our CHM
chm <- rasterize_canopy(las2, res = 0.5, pitfree(max_edge = c(0, 1.5)))
plot(chm, col = col)
```

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

## End(Not run)
```

`dsm_point2raster` *Digital Surface Model Algorithm*

Description

This function is made to be used in rasterize_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 rasterize_terrain for more details).

See Also

Other digital surface model algorithms: `dsm_pitfree`, `dsm_tin`

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 <- rasterize_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 <- rasterize_canopy(las, res = 0.5, p2r(0.2))
plot(chm, col = col)
```

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

## End(Not run)
```

---

dsm_tin *Digital Surface Model Algorithm*

Description

This function is made to be used in rasterize_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, highest = TRUE)
```

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).
highest	bool. By default it keeps only the highest point per pixel before to triangulate to decrease computation time. If highest = FALSE all first returns are used.

See Also

Other digital surface model algorithms: dsm_pitfree, dsm_point2raster

Examples

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

# Basic triangulation and rasterization of first returns
chm <- rasterize_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)
```

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

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

## End(Not run)
```

`dtm_idw` *Spatial Interpolation Algorithm*

Description

This function is made to be used in rasterize_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

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

Arguments

k	integer. Number of k-nearest neighbours. Default 10.
p	numeric. Power for inverse-distance weighting. Default 2.
rmax	numeric. Maximum radius where to search for knn. Default 50.

See Also

Other dtm algorithms: dtm_kriging, dtm_tin

Examples

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

#plot(las)

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

#plot(dtm)
#plot_dtm3d(dtm)
```

`dtm_kriging` *Spatial Interpolation Algorithm*

Description

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

Usage

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

Arguments

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

See Also

Other dtm algorithms: `dtm_idw`, `dtm_tin`

Examples

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

plot(las)

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

plot(dtm)
plot_dtm3d(dtm)

## End(Not run)
```

dtm_tin *Spatial Interpolation Algorithm*

Description

This function is made to be used in rasterize_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 can be performed with another algorithm.

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 knnidw by default.

See Also

Other dtm algorithms: dtm_idw, dtm_kriging

Examples

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

#plot(las)

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

#plot(dtm)
#plot_dtm3d(dtm)
```

engine *Functions for the LAScatalog processing engine not meant to be called directly by users*

Description

Functions for the LAScatalog processing engine not meant to be called directly by users. They are exported for debugging and to simplify export of internal functions when processing in parallel

Usage

```
engine_apply(
  .CHUNKS,
  .FUN,
  .PROCESSOPT,
  .OUTPUTOPT,
  .GLOBALS = NULL,
  .AUTOREAD = FALSE,
  .AUTOCROP = FALSE,
  ...
)

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

engine_crop(x, bbox)

engine_merge(ctg, any_list, ...)

engine_write(x, path, drivers)
```

Arguments

.CHUNKS	list. list of LAScluster
.FUN	function. function that respects a template (see catalog_apply)
.PROCESSOPT	list. Processing option
.OUTPUTOPT	list. Output option
.GLOBALS	list. Force export of some object in workers
.AUTOREAD	bool. Enable autoread
.AUTOCROP	bool. Enable autocrop
...	parameters of .FUN
ctg	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.
x	LAS, Raster, stars, SpatRaster,sf, sfc, Spatial
bbox	bbox
any_list	list of LAS, Raster, stars, SpatRaster, sf, sfc, Spatial, data.frame
path	strings
drivers	list. Drivers of a LAScatalog

See Also

Other LAScatalog processing engine: engine_options

Other LAScatalog processing engine: engine_options

---

engine_options *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

```
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

ctg	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.

See Also

Other LAScatalog processing engine: engine

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/folder/templated_filename_{XBOTTOM}_{ID}"
summary(ctg)
```

Extract *Extract or Replace Parts of a LAS* Object*

Description

Operators acting on LAS* objects. However, some have modified behaviors to prevent some irrelevant modifications. Indeed, a LAS* object cannot contain anything, as the content is restricted by the LAS specifications. If a user attempts to use one of these functions inappropriately an informative error will be thrown.

Usage

```
## S4 method for signature 'LAS'
x$name

## S4 method for signature 'LAS,ANY,missing'
x[[i, j, ...]]

## 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 'LAS,logical,ANY'
x[i]

## S4 method for signature 'LAS,sf,ANY'
x[i]

## S4 method for signature 'LAS,sfc,ANY'
x[i]

## S4 method for signature 'LAScatalog'
x$name
```

```
## S4 method for signature 'LAScatalog,ANY,missing'
x[[i, j, ...]]

## S4 method for signature 'LAScatalog,ANY,ANY'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'LAScatalog,logical,ANY'
x[i]

## S4 method for signature 'LAScatalog,sf,ANY'
x[i]

## S4 method for signature 'LAScatalog,sfc,ANY'
x[i]

## S4 replacement method for signature 'LAScatalog,ANY,ANY'
x[[i, j]] <- value

## S4 replacement method for signature 'LAScatalog'
x$name <- value

## S4 method for signature 'LASheader'
x$name

## S4 replacement method for signature 'LASheader'
x$name <- value

## S4 method for signature 'LASheader,ANY,missing'
x[[i, j, ...]]

## S4 replacement method for signature 'LASheader,character,missing'
x[[i]] <- value
```

Arguments

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

Examples

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

```
las$Intensity
las[["Z"]]
las[["Number of points by return"]]

## Not run:
las$Z = 2L
las[["Z"]] = 1:10
las$NewCol = 0
las[["NewCol"]] = 0

## End(Not run)
```

---

filters *Filter points of interest*

Description

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

Usage

```
filter_poi(las, ...)

filter_first(las)

filter_firstlast(las)

filter_firstofmany(las)

filter_ground(las)

filter_last(las)

filter_nth(las, n)

filter_single(las)

filter_duplicates(las)

## S3 method for class 'LAS'
filter_duplicates(las)

## S3 method for class 'LAScatalog'
filter_duplicates(las)
```

Arguments

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

Details

- `filter_poi` Select points of interest based on custom logical criteria.
- `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.
- `filter_duplicates` **Removes** the duplicated points (duplicated by XYZ)

Value

An object of class LAS

Non-supported LAScatalog options

The option `select` is not supported and not respected because it always preserves the file format and all the attributes. `select = "*"` is imposed internally.

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)
```

gnd_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

```
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

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

See Also

Other ground segmentation algorithms: gnd_mcc, gnd_pmf

Examples

```
if (require(RCSF, quietly = TRUE))
{
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")
}
```

---

gnd_mcc *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 Multiscale Curvature Classification. This method is a strict implementation of the MCC algorithm made by Evans and Hudak. (2007) (see references) that relies on the authors' original source code written and exposed to R via the the RMCC package.

Usage

```
mcc(s = 1.5, t = 0.3)
```

Arguments

s	numeric. Scale parameter. The optimal scale parameter is a function of 1) the scale of the objects (e.g., trees) on the ground, and 2) the sampling interval (post spacing) of the lidar data.
t	numeric. Curvature threshold

Details

There are two parameters that the user must define, the scale (s) parameter and the curvature threshold (t). The optimal scale parameter is a function of 1) the scale of the objects (e.g., trees) on the ground, and 2) the sampling interval (post spacing) of the lidar data. Current lidar sensors are capable of collecting high density data (e.g., 8 pulses/m2) that translate to a spatial sampling frequency (post spacing) of 0.35 m/pulse (1/sqrt(8 pulses/m2) = 0.35 m/pulse), which is small relative to the size of mature trees and will oversample larger trees (i.e., nominally multiple returns/tree). Sparser lidar data (e.g., 0.25 pulses/m2) translate to a post spacing of 2.0 m/pulse (1/sqrt(0.25 pulses/m2) = 2.0 m/pulse), which would undersample trees and fail to sample some smaller trees (i.e., nominally <1 return/tree).

Therefore, a bit of trial and error is warranted to determine the best scale and curvature parameters to use. Select a las tile containing a good variety of canopy and terrain conditions, as it's likely the parameters that work best there will be applicable to the rest of your project area tiles. As a starting point: if the scale (post spacing) of the lidar survey is 1.5 m, then try 1.5. Try varying it up or down by 0.5 m increments to see if it produces a more desirable digital terrain model (DTM) interpolated from the classified ground returns in the output file. Use units that match the units of the lidar data. For example, if your lidar data were delivered in units of feet with a post spacing of 3 ft, set the scale parameter to 3, then try varying it up or down by 1 ft increments and compare the resulting interpolated DTMs. If the trees are large, then it's likely that a scale parameter of 1 m (3 ft) will produce a cleaner DTM than a scale parameter of 0.3 m (1 ft), even if the pulse density is 0.3 m (1 ft). As for the curvature threshold, a good starting value to try might be 0.3 (if data are in meters; 1 if data are in feet), and then try varying this up or down by 0.1 m increments (if data are in meters; 0.3 if data are in feet).

References

Evans, Jeffrey S.; Hudak, Andrew T. 2007. A multiscale curvature algorithm for classifying discrete return LiDAR in forested environments. IEEE Transactions on Geoscience and Remote Sensing. 45(4): 1029-1038.

See Also

Other ground segmentation algorithms: gnd_csf, gnd_pmf

Examples

```
if (require(RMCC, quietly = TRUE))
{
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzrn", filter = "-inside 273450 5274350 273550 5274450")

las <- classify_ground(las, mcc(1.5,0.3))
#plot(las, color = "Classification")
}
```

---

gnd_pmf *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

```
pmf(ws, th)

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

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.
b	numeric. This is the parameter b in Zhang et al. (2003) (eq. 4 and 5).
dh0	numeric. This is dh_0 in Zhang et al. (2003) (eq. 7).
dhmax	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

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

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

$$w_k = 2kb + 1$$

or

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

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

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

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

References

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

See Also

Other ground segmentation algorithms: gnd_csf, gnd_mcc

Examples

```
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")
```

---

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

```
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. 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.

Examples

```
## Not run:
LASfile <- system.file("extdata", "fwf.laz", package="rlas")
fwf <- readLAS(LASfile)
las <- interpret_waveform(fwf)
x <- plot(fwf, size = 3, pal = "red")
plot(las, color = "Amplitude", bg = "white", add = x, size = 2)

## End(Not run)
```

`is` *A set of boolean tests on objects*

Description

`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)

g <- pitfree()
is.parallelised(g)

ctg <- readLAScatalog(LASfile)
is.indexed(ctg)
```

---

itd_lmf *Individual Tree Detection Algorithm*

Description

This function is made to be used in locate_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

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

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. By default function takes 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. This can be controled with the ‘ws_args‘ parameter
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".
ws_args	list. Named list of argument for the function ‘ws‘ if ‘ws‘ is a function.

References

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

See Also

Other individual tree detection algorithms: itd_manual

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyzi", filter = "-inside 481250 3812980 481300 3813050")

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

# 5x5 m fixed window size
ttops <- locate_trees(las, lmf(5))

#plot(las) |> add_treetops3d(ttops)

# variable windows size
f <- function(x) { x * 0.07 + 3}
ttops <- locate_trees(las, lmf(f))

#plot(las) |> add_treetops3d(ttops)

# Very custom variable windows size
f <- function(x, y, z) { x * 0.07 + y * 0.01 + z}
ws_args <- list(x = "Z", y = "Intensity", z = 3)
ttops <- locate_trees(las, lmf(f, ws_args = ws_args))

# ============
```

```
# raster-based
# ============

chm <- rasterize_canopy(las, res = 1, p2r(0.15), pkg = "terra")
ttops <- locate_trees(chm, lmf(5))

plot(chm, col = height.colors(30))
plot(sf::st_geometry(ttops), add = TRUE, col = "black", cex = 0.5, pch = 3)

# variable window size
f <- function(x) { x * 0.07 + 3 }
ttops <- locate_trees(chm, lmf(f))

plot(chm, col = height.colors(30))
plot(sf::st_geometry(ttops), add = TRUE, col = "black", cex = 0.5, pch = 3)
```

`itd_manual` *Individual Tree Detection Algorithm*

Description

This function is made to be used in locate_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.
This algorithm does not preserve tree IDs from `detected` **and renumber all trees. It also looses all attributes**

Usage

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

Arguments

`detected`	`SpatialPoints*` or sf/sfc_POINT‘ with 2 or 3D points of already found tree tops that need manual correction. Can be NULL
`radius`	numeric. Radius of the spheres displayed on the point cloud (aesthetic purposes only).
`color`	character. Colour of the spheres displayed on the point cloud (aesthetic purposes only).
`button`	Which button to use for selection. One of "left", "middle", "right". lidR using left for rotation and right for dragging using one of left or right will disable either rotation or dragging
`...`	supplementary parameters to be passed to plot.

See Also

Other individual tree detection algorithms: itd_lmf

Examples

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

# Full manual tree detection
ttops = locate_trees(las, manual())

# Automatic detection with manual correction
ttops = locate_trees(las, lmf(5))
ttops = locate_trees(las, manual(ttops))

## End(Not run)
```

---

its_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 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', 'SpatRaster' or 'stars'. Canopy height model. Can be computed with rasterize_canopy or read from an external file.

treetops	'SpatialPoints*' or 'sf/sfc_POINT' with 2D or 3D coordinates. Can be computed with locate_trees or read from an external file
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 treetops contains an attribute with the ID for each tree, the name of this attribute. This way, original IDs will be preserved.

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:

```
chm <- raster("chm.tif")
ttops <- locate_trees(chm, lmf(3))
crowns <- dalponte2016(chm, ttops)()
```

References

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

See Also

Other individual tree segmentation algorithms: its_li2012, its_silva2016, its_watershed

Other raster based tree segmentation algorithms: its_silva2016, its_watershed

Examples

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

ttops <- locate_trees(chm, lmf(4, 2))
las   <- segment_trees(las, dalponte2016(chm, ttops))
#plot(las, color = "treeID", colorPalette = col)
```

`its_li2012` *Individual Tree Segmentation Algorithm*

Description

This functions is made to be used in segment_trees. It implements an algorithm for tree segmentation based on Li et al. (2012) (see reference). This method is a growing region method working at the point cloud level. It is an implementation by lidR authors, from the original paper, as close as possible from the original description. However we added a parameter `hmin` to prevent over-segmentation for objects that are too low. This algorithm is known to be slow because it has an algorithmic complexity worst that O(n^2).

Usage

```
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.
`dt2`	numeric. Threshold number 2. See reference page 79 in Li et al. (2012). Default is 2.
`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.
`hmin`	numeric. Minimum height of a detected tree. Default is 2.
`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 by restricting the number of comparisons to perform. The lower the value, the faster the method. Default is 10.

References

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

See Also

Other individual tree segmentation algorithms: `its_dalponte2016`, `its_silva2016`, `its_watershed`

Examples

```
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)

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

---

its_silva2016 *Individual Tree Segmentation Algorithm*

Description

This functions is made to be used in segment_trees. It implements an algorithm for tree segmentation based on Silva et al. (2016) (see reference). This is a simple method based on seed + voronoi tesselation (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', 'SpatRaster' or 'stars'. Canopy height model. Can be computed with rasterize_canopy or read from an external file.
`treetops`	'SpatialPoints*' or 'sf/sfc_POINT' with 2D or 3D coordinates. Can be computed with locate_trees or read from an external file
`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 `treetops` contains an attribute with the ID for each tree, the name of this attribute. This way, original IDs will be preserved.

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("chm.tif")
ttops <- locate_trees(chm, lmf(3))
crowns <- silva2016(chm, ttops)()
```

References

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

See Also

Other individual tree segmentation algorithms: `its_dalponte2016`, `its_li2012`, `its_watershed`

Other raster based tree segmentation algorithms: `its_dalponte2016`, `its_watershed`

Examples

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

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

`its_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. It is based on the bioconductor package `EBImage`. You need to install this package to run this method (see its github page). Internally, the function EBImage::watershed is called.

Usage

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

Arguments

`chm`	'RasterLayer', 'SpatRaster' or 'stars'. Canopy height model. Can be computed with rasterize_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.

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("chm.tif")
crowns <- watershed(chm)()
```

See Also

Other individual tree segmentation algorithms: `its_dalponte2016`, `its_li2012`, `its_silva2016`

Other raster based tree segmentation algorithms: `its_dalponte2016`, `its_silva2016`

Examples

```
## 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)

# Using raster because focal does not exist in stars
chm <- rasterize_canopy(las, res = 0.5, p2r(0.3), pkg = "raster")
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)

## End(Not run)
```

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(), crs = sf::NA_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.
crs	crs object of class crs from sf
check	logical. Conformity tests while building the object.
index	list with two elements list(sensor = 0L, index = 0L). See spatial indexing
...	internal use

Details

A LAS object contains 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, Y, Z (numeric)
- gpstime (numeric)
- Intensity (integer)
- ReturnNumber, NumberOfReturns (integer)
- ScanDirectionFlag (integer)
- EdgeOfFlightline (integer)
- Classification (integer)
- Synthetic_flag,Keypoint_flag, Withheld_flag (logical)
- ScanAngleRank/ScanAngle (integer/numeric)
- UserData (integer)
- PointSourceID (integer)
- R,G,B, 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

crs Object of class crs from sf.

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.

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 <- header(las)

# 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 collection of .las or .laz files*

Description

A LAScatalog object is a representation of a collection of las/laz files. A LAScatalog is a way to manage and batch process a lidar coverage. 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.

Details

A LAScatalog contains an sf object to store the geometry and metadata. It is extended with 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 collection. 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

`data` sf. An sf `data.frame` with the bounding box of each file as well as all the information read from the header of each LAS/LAZ file.

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

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

`output_options` list. A list that contains some settings describing how the collection 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 subareas 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 chunk pattern is build using the file pattern. The chunks are expecting to be wall-to-wall coverage, which means that `chunk_size = 0` has meaning only if the files are not overlapping. 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 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 users to restart a computation that failed without reprocessing everything. See opt_restart<-. Technically, this option may be used for partial processing of a collection, but it generally should not be. 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 a file 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/{ORIGINALFILNAME}_normalized"
```

This option will generate as many filenames as needed with custom names for each file. The allowed templates are `{XLEFT}`, `{XRIGHT}`, `{YBOTTOM}`, `{YTOP}`, `{ID}`, `{XCENTER}`,`{YCENTER}`, `{ORIGINALFILENAME}`. See opt_output_files.

- **drivers**: list. This contains all the drivers required to seamlessly write `Raster*`, `SpatRaster`, `stars`, `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. See opt_merge.

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

```
## Not run:
# Build a catalog
ctg <- readLAScatalog("filder/to/las/files/", filter =  "-keep_first")


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

# We can seamlessly use lidR functions
hmean <- pixel_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 <- rasterize_terrain(ctg, 1, tin())

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

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

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

```
dtm <- rasterize_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 functions catalog_apply
# and catalog_map
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),
                    EdgeOfFlightline = c(1L, 0L, 0L),
                    Classification = c(1L, 1L, 1L),
                    ScanAngleRank = c(-21L, -21L, -21L),
                    UserData = c(32L, 32L, 32L),
                    PointSourceID = c(17L, 17L, 17L))

header = LASheader(data)
header

# 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, a `list` in the slot `@VLR` with the data read from the Variable Length Records and a `list` in the slot `EVLR` with the data read from the Extended Variable Lenght 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 LAScatalog object and prints a report.

For a LAS object it checks:

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

For a LAScatalog object it checks:

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

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

Usage

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

Arguments

las	An object of class LAS or LAScatalog.
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 LAScatalog 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)
```

`las_compression` *Compression of the point cloud*

Description

Package rlas 1.6.0 supports compact representation of non populated attributes. For example `UserData` is usually populated with zeros (not populated). Yet it takes 32 bits per point to store each 0. With rlas 1.6.0 it can now use 644 bits no matter the number of points loaded if it is not populated or populated with a unique value.

Usage

```
las_is_compressed(las)

las_size(las)
```

Arguments

`las`	A LAS object.

Details

`las_is_compressed` test each attributes and returns a named vector with TRUE if the attribute is compressed FALSE otherwise.

`las_size` returns the true size of a LAS object by considering the compression. `object.size` from base R does not account for ALTREP and consequently cannot measure properly the size of a LAS object

Examples

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

format(object.size(las), units = "MB")
format(las_size(las), units = "MB")
```

las_utilities *LAS utilities*

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)

header(las)

payload(las)

phb(las)

vlr(las)

evlr(las)
```

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: `las_check()`

Examples

```
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
# $<- already perform such operations on-the-fly. Thus the following
# syntax must be preferred and returns valid objects. Previous tools
# were only intended to be used in very specific cases.
las$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)
```

`lidR-LAScatalog-drivers` *LAScatalog drivers*

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 defaut `LAS` objects are stored in .las files with writeLAS, raster objects are stored in .tif files using native write function for `raster`, `stars` or `terra`, `Spatial*` objects are stored in .shp files with after coercion to `sf` with st_write, `sf` object are stored to `.gpkg` 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 slot `@output_options` 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

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

And the driver to write a `LAS` is

```
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 shapefile instead of a GeoPackage, one must change the `sf` driver:

```
ctg@output_options$drivers$sf$extension <- ".shp"
```

To write a `Raster*` 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`,`sf`, `Raster`, `stars`, `SpatRaster` 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 `@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 `sf` objects are written into GeoPackage with st_write. `st_write` can also wite in GeoJSON and even in SQLlite database objects. But it cannot add data into an existing SQLlite database. Let's create our own driver for a `sf`. First we need a function able to write and append a `sf` into a `SQLlite` database from the object and the path.

```
dbWrite_sf = function(x, path, name)
{
 x <- sf::st_drop_geometry(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$drivers$sf = list(
 write = dbWrite_sf,
 extension = ".sqlite",
 object = "x",
 path = "path",
 param = list(name = "layername"))
```

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 `sf` will be appended in a single database. To preserve the geometry one can

---

`lidR-parallelism` *Parallel computation in lidR*

Description

This document explains 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:

```
las  <- readLAS("file.las")
tops <- locate_trees(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:

```
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.

```
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. 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:

```
ctg <- readLAScatalog("folder/")
out <- pixel_metrics(ctg, ~mean(Z))
```

But this one is evaluated in parallel with two cores:

```
library(future)
plan(multisession, workers = 2L)
ctg <- readLAScatalog("folder/")
out <- pixel_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:

```
library(future)
plan(multisession, workers = 4L)
set_lidr_threads(4L)
ctg <- readLAScatalog("folder/")
out <- locate_trees(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(las, ws, ...)
{
  las  <- normalize_height(las, tin())
  tops <- locate_tree(las, lmf(ws))
  return(tops)
}

out <- catalog_map(ctg, myfun, ws = 5)
```

This function used 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_map(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 a 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.

`lidR-spatial-index` *Spatial index*

Description

This document explains how to process point-clouds taking advantage of different spatial indexes 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 used 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

```
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)
```

```
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)
```

`locate_trees` *Individual tree detection*

Description

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

Usage

```
locate_trees(las, algorithm, uniqueness = "incremental")
```

Arguments

`las`	An object of class `LAS` or `LAScatalog`. Can also be a raster from `raster`, `stars` or `terra` 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

`locate_trees` returns an sf object with POINT Z geometries. The table of attributes contains a column `treeID` with an individual ID for each tree. The height of the trees (`Z`) are also repeated in the table of attribute to be analysed as an attribute and not as a coordinate.

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. 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. It also ensures 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.

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, select = "xyz", filter = "-inside 481250 3812980 481300 3813030")

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

#plot(las) |> add_treetops3d(ttops)
```

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*, an sf/sfc, a Raster*, a stars, or a SpatRaster.

- SpatialPolygons*, sf and sfc: 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, single band stars or single layer SpatRaster: it attributes to each point the value found in each pixel of the raster.
- RasterStack, RasterBrick, multibands stars or multilayer SpatRaster must have 3 layers 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 sfc or RasterLayer or RasterStack or RasterBrick or stars.
attribute	character. The name of an attribute in the table of attributes or the name of a new column in the LAS object. Not relevant for RGB colorization.

Value

a LAS object

Examples

```
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, quiet = TRUE)

# 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.
names(las)
```

```
forest <- filter_poi(las, inlakes == FALSE)
#plot(forest)

# 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.
names(las)
```

`noise_ivf` *Noise Segmentation Algorithm*

Description

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 x 3 x 3 = 27 voxels.

Usage

```
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: `noise_sor`

Examples

```
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))
```

noise_sor *Noise Segmentation Algorithm*

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 library 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

```
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 deviation`. 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
https://www.cloudcompare.org/doc/wiki/index.php?title=SOR_filter

See Also

Other noise segmentation algorithms: noise_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
# dataset 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))
```

normalize *Normalize point cloud*

Description

Normalize elevation or intensity values using multiple methods.

Usage

```
normalize_height(las, algorithm, use_class = c(2L, 9L), dtm = NULL, ...)

unnormalize_height(las)

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

normalize_intensity(las, algorithm)
```

Arguments

las	An object of class LAS or LAScatalog.
algorithm	(1) An algorithm for spatial interpolation. lidR has tin, kriging, knnidw or a raster representing a digital terrain model. (2) An algorithm for intensity normalization. lidR currently has range_correction.
use_class	integer vector. By default the terrain is computed by using ground points (class 2) and water points (class 9). Relevant only for a normalization without a raster DTM.
dtm	raster. If dtm is provided, then the DTM is used in place of ground points. This is different than providing a DTM in algorithm. If algorithm = dtm the dtm is subtracted naively. If algorithm = tin() and dtm = raster the ground points are not used and the DTM is interpolated as if it were made of regularly-spaced ground points.
...	normalized_height() supports add_lasattribute= TRUE to add the elevation above see level as an extra byte attribute and Wdegenerated = FALSE to silence the warning about degenerated ground points.
e1	a LAS object
e2	A raster representing a digital terrain model in format from raster, stars or terra..

Details

normalize_height Subtract digital terrain model (DTM) from a LiDAR point cloud to create a dataset normalized with the ground at 0. The DTM can be a raster, but 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. A new attribute 'Zref' records the former elevation values, which enables the use of unnormalize_height to restore original point elevations.

normalize_intensity Normalize intensity values using multiple methods. The attribute 'Intensity' records the normalized intensity. An extra attribute named 'RawIntensity' records the original intensities.

Non-supported LAScatalog options

The option `select` is not supported and not respected because it always preserves the file format and all the attributes. `select = "*"` is imposed internally.

Examples

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

# ====================
# Normalize elevation
# ====================

# First option: use a raster as DTM
# ---------------------------------------

dtm <- rasterize_terrain(las, 1, knnidw(k = 6L, p = 2))
nlas <- normalize_height(las, dtm)

# restore original elevations
las <- unnormalize_height(nlas)

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

# restore original elevations
las <- unnormalize_height(las)

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

nlas <- normalize_height(las, tin())

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

# 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))
```

`nstdmetrics` *Predefined non standard metrics*

Description

Functions and metrics from the literature. See details and references

Usage

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

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

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

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

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

Arguments

`x`	A `RasterLayer`, a `stars` a `SpatRaster` or a vector of x coordinates.
`y`	numeric. If `x` is a vector of coordinates: the associated y coordinates.
`z`	vector of positive z coordinates
`...`	unused
`z0`	numeric. The bottom limit of the profile
`k`	numeric. is the extinction coefficient
`by, dz`	numeric. The thickness of the layers used (height bin)
`zmax`	numeric. Maximum elevation for an entropy normalized to zmax.

Details

rumple_index 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.

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

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

entropy 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.

VCI Vertical Complexity Index. A fixed normalization of the entropy function from van Ewijk et al. (2011) (see references)

Value

numeric. The computed Rumple index.

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

A number between 0 and 1

A number between 0 and 1

References

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

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

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

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

Examples

```
x <- runif(20, 0, 100)
y <- runif(20, 0, 100)

if (require(geometry, quietly = TRUE))
{
# Perfectly flat surface, rumple_index = 1
z <- rep(10, 20)
rumple_index(x, y, z)

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

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

# Measure of roughness is scale-dependent
rumple_index(x, y, z)
rumple_index(x/10, y/10, z)
}
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")
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")
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)
z <- runif(10000, 0, 10)
```

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

z <- abs(rnorm(10000, 10, 1))

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

`old_spatial_packages` *Older R Spatial Packages*

Description

lidR 4.0.0 no longer uses the `sp` and `raster` packages. New functions are based on `sf`, `terra` and `stars`. However, to maintain backward compatibility the old functions from v<4.0.0 were preserved.

`rgdal` and `rgeos` will be retired on Jan 1st 2024. The `raster` and `sp` packages are based on `rgdal` and `rgeos`. `lidR` was based on `raster` and `sp` because it was created before the `sf`, `terra` and `stars` packages. This means that sooner or later users and packages that are still based on old R spatial packages will run into trouble. According to Edzer Pebesma, Roger Bivand:

R users who have been around a bit longer, in particular before packages like `sf` *and* `stars` *were developed, may be more familiar with older packages like* `maptools`*,* `sp`*,* `rgeos`*, and* `rgdal`*. A fair question is whether they should migrate existing code and/or existing R packages depending on these packages. The answer is: yes (see reference).*

The following functions are not formally deprecated but users should definitely move their workflow to modern spatial packages. lidR will maintain the old functions as long as it does not generate issues on CRAN. So, it might be until Jan 1st 2024 or later, who knows...

Usage

```
as.spatial(x)

## S3 method for class 'LAS'
as.spatial(x)

## S3 method for class 'LAScatalog'
as.spatial(x)

tree_metrics(las, func = ~list(Z = max(Z)), attribute = "treeID", ...)

grid_canopy(las, res, algorithm)

grid_density(las, res = 4)

grid_terrain(
```

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

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

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

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

Arguments

x, las	an object of class LAS*
func	see template_metrics
attribute, type	see crown_metrics
...	ignored
res, start	see pixel_metrics
algorithm	see rasterize_canopy, rasterize_terrain
full_raster, use_class, Wdegenerated, is_concave, keep_lowest	see rasterize_density
filter, by_echo	see template_metrics
uniqueness	see crown_metrics
concavity, length_threshold	see concaveman

References

Edzer Pebesma, Roger Bivand Spatial Data Science with applications in R https://keen-swartz-3146c4.netlify.app/older.html

---

`pitfill_stonge2008` *Pits and spikes filling*

Description

Pits and spikes filling for raster. Typically used for post-processing CHM. This algorithm is from St-Onge 2008 (see reference).

Usage

```
pitfill_stonge2008(
  x,
  lap_size = 3L,
  thr_lap = 0.1,
  thr_spk = -0.1,
  med_size = 3L,
  dil_radius = 0L
)
```

Arguments

`x`	raster. SpatRaster, RasterLayer, stars.
`lap_size`	integer. Size of the Laplacian filter kernel (integer value, in pixels).
`thr_lap`	numeric. Threshold Laplacian value for detecting a cavity (all values above this value will be considered a cavity). A positive value.
`thr_spk`	numeric. Threshold Laplacian value for detecting a spike (all values below this value will be considered a spike). A negative value.
`med_size`	integer. Size of the median filter kernel (integer value, in pixels).
`dil_radius`	integer. Dilation radius (integer value, in pixels).

References

St-Onge, B., 2008. Methods for improving the quality of a true orthomosaic of Vexcel UltraCam images created using alidar digital surface model, Proceedings of the Silvilaser 2008, Edinburgh, 555-562. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=81365288221f3ac34b51a82e2cfed8d58defb10e

Examples

```
LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile)
chm <- rasterize_canopy(las, 0.5, dsmtin())
sto <- pitfill_stonge2008(chm)

#terra::plot(c(chm, sto), col = lidR::height.colors(25))
```

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()' from package 'mapview'. If the coordinate reference system (CRS) of the LAScatalog is non empty, the plot can be displayed on top of base maps (satellite data, elevation, street, and so on).

Plot displays a LASheader object exactly like it displays a LAScatalog object.

Usage

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

## S4 method for signature 'LAS,missing'
plot(
  x,
  y,
  ...,
  color = "Z",
  pal = "auto",
  bg = "black",
  breaks = "pretty",
  nbreaks = "auto",
  backend = "rgl",
  clear_artifacts = TRUE,
  axis = FALSE,
  legend = FALSE,
  add = FALSE,
  voxel = FALSE,
  NAcol = "lightgray",
  mapview = 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, ...)

height.colors(n)

forest.colors(n)
```

```
random.colors(n)

pastel.colors(n)
```

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.
pal	palette function, similar to heat.colors, or palette values. Default is "auto" providing an automatic coloring depending on the attribute color
bg	The color for the background. Default is black.
breaks	either a numeric vector with the actual breaks, or a name of a method accepted by the style argument of classIntervals
nbreaks	Number of colors breaks.
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).
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.
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.
NAcol	a color for NA values.
mapview	logical. If FALSE the catalog is displayed in a regular plot from R base. Since v4.0.4 'mapview = TRUE' is also possible with LAS objects.

chunk_pattern	logical. Display the current chunk pattern used to process the catalog.
overlaps	logical. Highlight the overlaps between files.
n	The number of colors (> 1) to be in the palette

Examples

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

plot(las)
plot(las, color = "Intensity")
plot(las, color = "ScanAngleRank", pal = rainbow)

# If outliers break the color range, use the breaks parameter
las$Intensity[150] <- 1000L
plot(las, color = "Intensity")
plot(las, color = "Intensity", breaks = "quantile", nbreaks = 50)

plot(las, color = "Classification")

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


# single file LAScatalog using data provided in lidR
ctg = readLAScatalog(LASfile)
plot(ctg)
plot(ctg, 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

```
## S3 method for class 'lasmetrics3d'
plot(x, y, ...)
```

Arguments

x	An object of the class lasmetrics3d
y	Unused (inherited from R base)
...	Supplementary parameters for plot. The function internally uses the same plot function than LAS objects.

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_3d *Add a spatial object to a point cloud scene*

Description

Add a raster (‘raster‘, ‘stars‘ ‘terra‘) object that represents a digital terrain model or a ‘SpatialPointsDataFrame‘ or ‘sf‘ 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 offset 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‘ or ‘stars‘ or ‘SpatRaster‘
bg	The color for the background. Default is black.

clear_artifacts
: logical. It is a known and documented issue that 3D visualisation with rgl displays artifacts. The points and lines are inaccurately positioned in the space and thus the rendering may look false or weird. This is because 'rgl' computes with single precision 'float'. To fix this, the objects are shifted to (0,0) to reduce the number of digits needed to represent their coordinates. The drawback is that the objects are not plotted at their actual coordinates.

...
: Supplementary parameters for surface3d or spheres3d.

x
: The output of the function plot used with a LAS object.

ttops
: A 'SpatialPointsDataFrame' or 'sf/sfc' that contains tree tops coordinates.

z
: character. The name of the attribute that contains the height of the tree tops or of the flightlines. Only for XY geometries Ignored if the input have XYZ geometries

flightlines
: A 'SpatialPointsDataFrame' or 'sf' that contains flightlines coordinates.

Examples

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

dtm <- rasterize_terrain(las, algorithm = tin())
ttops <- locate_trees(las, lmf(ws = 5))

plot_dtm3d(dtm)

x <- plot(las)
add_dtm3d(x, dtm)
add_treetops3d(x, ttops)

plot(las) |> add_dtm3d(dtm) |> add_treetops3d(ttops)

## End(Not run)
```

---

plugins *Plugin system*

Description

Tools to build plugin functions for lidR

Usage

```
plugin_dsm(f, omp = FALSE)

plugin_dtm(f, omp = FALSE)
```

```
plugin_gnd(f, omp = FALSE)

plugin_decimate(f, omp = FALSE)

plugin_shape(f, omp = FALSE)

plugin_snag(f, omp = FALSE)

plugin_track(f, omp = FALSE)

plugin_nintensity(f, omp = FALSE)

plugin_outliers(f, omp = FALSE)

plugin_itd(f, omp = FALSE, raster_based = FALSE)

plugin_its(f, omp = FALSE, raster_based = FALSE)
```

Arguments

f	a function
omp	logical is the function natively parallized with OpenMP
raster_based	logical. For ITS and ITD algorithms, is the method raster-based or or point-cloud-based?

Examples

```
## Not run:
mba <- function(n = 1, m = 1, h = 8, extend = TRUE) {
  f <- function(las, where) {
    res <- MBA::mba.points(las@data, where, n, m , h, extend)
    return(res$xyz.est[,3])
  }

  f <- plugin_dtm(f)
  return(f)
}

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

dtm = rasterize_terrain(las, algorithm = mba())

## End(Not run)
```

point_metrics *Point-based metrics*

Description

Computes a series of user-defined descriptive statistics for a LiDAR dataset for each point based on its k-nearest neighbours or its sphere neighbourhood.

Usage

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

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

Arguments

las	An object of class LAS
func	formula. An expression to be applied to each point neighbourhood (see also template_metrics).
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. Otherwise an ID referring to each point.
filter	formula of logical predicates. Enables the function to run only on points of interest in an optimized way. See examples.
...	unused.
metrics	logical. Compute metrics or not
coeffs	logical. Principal component coefficients are returned

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.

`point_eigenmetrics` 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 because it is optimized and parallelized internally.

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 pixel_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.

Examples

```
## 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_shapes(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)
#> Computed in 0.5 seconds

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++.

M <- point_eigenvalues(las, k = 25)
is_planar = M$eigen_medium > (25*M$eigen_smallest) & (6*M$eigen_medium) > M$eigen_largest

# 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.

## End(Not run)
```

`print.LAS` *Tools inherited from base R for LAS* objects*

Description

Tools inherited from base R for LAS* objects

Usage

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

## S3 method for class 'LAScatalog'
```

```
print(x, ...)

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

## S3 method for class 'raster_template'
print(x, ...)

## S3 method for class 'LAS'
summary(object, ...)

## S3 method for class 'LAScatalog'
summary(object, ...)

## S3 method for class 'LAS'
dim(x)

## S3 method for class 'LAScatalog'
dim(x)

ncol.LAS(x)

nrow.LAScatalog(x)

## S3 method for class 'LAS'
names(x)

## S3 method for class 'LASheader'
names(x)

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

npoints(x, ...)

density(x, ...)

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

## S4 method for signature 'LASheader'
density(x, ...)

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

Arguments

x	a LAS* object

...	LAS* objects if it is the sole argurment (e.g. in rbind())
object	A LAS* object or other lidR related objects.

`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):

$$I_{norm} = I_{obs} \left(\frac{R}{Rs} \right)^f$$

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

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

get_range(las, sensor, gpstime = "gpstime", elevation = "Z")
```

Arguments

sensor	'SpatialPointsDataDrame' or 'sf' 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'). Z can also come from the geometry if the input records XYZ coordinates. It can be computed with track_sensor.
Rs	numeric. Range of reference.
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 the input contains 3 coordinates points, 'elevation' is not considered.
las	an object of class LAS. `get_range()` is a regular function documented here for convenience.

References

Gatziolis, D. (2011). Dynamic Range-based Intensity Normalization for Airborne, Discrete Return Lidar Data of Forest Canopies. Photogrammetric Engineering & Remote Sensing, 77(3), 251–259. https://doi.org/10.14358/pers.77.3.251

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)
```

`rasterize` *Rasterize a point cloud*

Description

Rasterize a point cloud in different ways to compute a DTM, a CHM or a density map. Most raster products can be computed with pixel_metrics but some are more complex and require dedicated and optimized functions. See Details and Examples.

Usage

```
rasterize_canopy(las, res = 1, algorithm = p2r(), ...)

rasterize_density(las, res = 4, ...)

rasterize_terrain(
  las,
  res = 1,
  algorithm = tin(),
  use_class = c(2L, 9L),
  shape = "convex",
  ...
)
```

Arguments

`las`	An object of class LAS or LAScatalog.
`res`	numeric. The size of a grid cell in point cloud coordinates units. Can also be `RasterLayer` or a `stars` or a `SpatRaster` used as layout.

algorithm	function. A function that implements an algorithm to compute a digital surface model or a digital terrain model. lidR implements p2r, dsmtin, pitfree for digital surface models, and knnidw, tin, and kriging for digital terrain models (see respective documentation and examples).
...	Use pkg = "terra|raster|stars" to get an output in SpatRaster, RasterLayer or stars format. Default is getOption("lidR.raster.default").
use_class	integer vector. By default the terrain is computed by using ground points (class 2) and water points (class 9).
shape	By default the interpolation is made only within the "convex" hull of the point cloud to get a DTM with the shape of the point cloud. This prevents meaningless interpolations where there is no data. It can also be "concave" or "bbox". It can also be an sfc to define a polygon in which to perform the interpolation.

Details

rasterize_terrain 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.

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

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

Value

RasterLayer or a stars or a SpatRaster depending on the settings.

Non-supported LAScatalog options

The option select is not supported and not respected in rasterize_* because it is internally known what is best to select.
The option chunk_buffer is not supported and not respected in rasterize_canopy and rasterize_density because it is not necessary.

Examples

```
# ====================
# Digital Terrain Model
# ====================

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

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

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

plot(dtm1, col = gray(0:25/25))
plot(dtm2, col = gray(0:25/25))
plot(dtm3, col = gray(0:25/25))
plot_dtm3d(dtm1)
plot_dtm3d(dtm2)
plot_dtm3d(dtm3)

## End(Not run)

# =====================
# Digital Surface Model
# =====================

LASfile <- system.file("extdata", "MixedConifer.laz", package="lidR")
las <- readLAS(LASfile, filter = "-inside 481280 3812940 481330 3812990")
col <- height.colors(15)

# Points-to-raster algorithm with a resolution of 1 meter
chm <- rasterize_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 <- rasterize_canopy(las, res = 0.5, p2r(0.2))
plot(chm, col = col)

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

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

# ====================
# Digital Density Map
# ====================

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

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

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

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 1.0 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'). 'files2' and 'files3' can be missing.

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 multisprectral point cloud is already stored in a single file leave `file2` and `file3` missing.

Examples

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

# Negation of 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

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

readALSLAScatalog(folder, ...)

readTLSLAScatalog(folder, ...)

readUAVLAScatalog(folder, ...)

readDAPLAScatalog(folder, ...)

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 engine_options.

... : Extra parameters to list.files. Typically recursive = TRUE. Propagates also to readLAScatalog

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/folder/of/las/>")

# 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.
# Files will be named e.g.
# filename_256000_1.ext
# filename_257000_2.ext
# filename_258000_3.ext
# ...
opt_output_files(ctg) <- "/path/filename_{XBOTTOM}_{ID}"

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

```
readLASheader(file)
```

Arguments

file	characters. Path to one file.

Value

A LASheader object

Examples

```
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

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

---

sample_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

```
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: sample_maxima, sample_per_voxel, sample_random

Examples

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

# Select points randomly to reach an homogeneous density of 1
thinned <- decimate_points(las, homogenize(1,5))
plot(rasterize_density(thinned, 10))
```

---

sample_maxima *Point Cloud Decimation Algorithm*

Description

These functions are made to be used in decimate_points. They implement algorithms that create 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: sample_homogenize, sample_per_voxel, sample_random

Other point cloud decimation algorithms: sample_homogenize, sample_per_voxel, sample_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)
```

`sample_per_voxel` *Point Cloud Decimation Algorithm*

Description

This functions 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

```
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: `sample_homogenize`, `sample_maxima`, `sample_random`

Examples

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

sample_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: sample_homogenize, sample_maxima, sample_per_voxel

Examples

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

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

segment *Segment a point cloud*

Description

Segment a point cloud using different methods. segment_* functions add a new attribute to the point cloud to label each point. They segment either individual trees, snags, or geometrical features.

Usage

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

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

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

Arguments

las	An object of class LAS or LAScatalog.
algorithm	function. An algorithm for segmentation. For individual tree segmentation, lidR has dalponte2016, watershed, li2012, and silva2016. More experimental algorithms may be found in the package lidRplugins. For snag segmentation, lidR has wing2015. For geometry segmentation, lidR has shp_plane, shp_hplane, and shp_line.
attribute	character. The returned LAS object as a new attribute (in a new column). This parameter controls the name of the new attribute.
filter	formula of logical predicates. Enables the function to run only on points of interest in an optimized way. See the examples.
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'.

Details

`segment_trees` 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.

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

`segment_snags` Snag segmentation using several possible algorithms. 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 is implemented, which uses LiDAR intensity thresholds and specified neighbourhoods to differentiate bole and branch from foliage points.

Non-supported LAScatalog options

The option `select` is not supported and not respected because it always preserves the file format and all the attributes. `select = "*"` is imposed internally.

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. 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. It also ensures 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.

Examples

```
# ==============
# Segment trees
# ==============

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")

# ==============
# Segment shapes
# ==============

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")

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

# ==============
# Segment snags
# ==============

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)
```

`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 neighbourhood*

Description

These functions are made to be used in segment_shapes. They implement algorithms for local neighbourhood 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)

shp_hline(th1 = 10, th2 = 0.02, k = 8)

shp_vline(th1 = 10, th2 = 0.98, 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. $|Z_1|, |Z_2|, |Z_3|$ denote the norm of the Z component of first, second and third axis of the decomposition. 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:

$$a_2 > (th_1 \times a_1) \& (th_2 \times a_2) > a_3$$

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.

$$a_2 > (th_1 \times a_1) \& (th_2 \times a_2) > a_3 \& |Z_3| > th_3$$

In theory $|Z_3|$ 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 neighbourhood is approximately linear, that is:

$$th_1 \times a_2 < a_3 \& th_1 \times a_1 < a_3$$

shp_hline Detection of horizontal lines inspired by the Limberger & Oliveira (2015) criterion. A point is labelled TRUE if the neighbourhood is approximately linear and horizontal, that is:

$$th_1 \times a_2 < a_3 \& th_1 \times a_1 < a_3 \& |Z_1| < th_2$$

In theory $|Z_1|$ should be exactly equal to 0. In practice 0.02 or 0.01 should be fine

shp_vline Detection of vertical lines inspired by the Limberger & Oliveira (2015) criterion. A point is labelled TRUE if the neighbourhood is approximately linear and vertical, that is:

$$th_1 \times a_2 < a_3 \& th_1 \times a_1 < a_3 \& |Z_1| > th_2$$

In theory $|Z_1|$ should be exactly equal to 1. In practice 0.98 or 0.99 should be fine

References

Limberger, F. A., & Oliveira, M. M. (2015). Real-time detection of planar regions in unorganized point clouds. Pattern Recognition, 48(6), 2043–2053. https://doi.org/10.1016/j.patcog.2014.12.020

Examples

```
# Generating some data
n = 400
xplane = runif(n,0,6)
yplane = runif(n,0,6)
zplane = xplane + 0.8 * yplane + runif(n, 0, 0.1)
plane = data.frame(X = xplane, Y = yplane, Z = zplane)

xhplane = runif(n,5,15)
yhplane = runif(n,0,10)
zhplane = 5 + runif(n, 0, 0.)
hplane = data.frame(X = xhplane, Y = yhplane, Z = zhplane)

tline = 1:n
```

```
xline = 0.05*tline
yline = 0.01*tline
zline = 0.02*tline + runif(n, 0, 0.1)
line = data.frame(X = xline, Y = yline, Z = zline)

thline = 1:n
xhline = 0.05*thline + runif(n, 0, 0.05)
yhline = 10 - 0.01*thline + runif(n, 0, 0.05)
zhline = 3 + runif(n, 0, 0.05)
hline = data.frame(X = xhline, Y = yhline, Z = zhline)

tvline = 1:n
xvline = 5 + runif(n, 0, 0.05)
yvline = 5 + runif(n, 0, 0.05)
zvline = 0.02*tline
vline = data.frame(X = xvline, Y = yvline, Z = zvline)

las <- rbind(plane, line, hplane, hline, vline)
las <- LAS(las)

las <- segment_shapes(las, shp_plane(k = 20), "plane")
las <- segment_shapes(las, shp_hplane(k = 20), "hplane")
las <- segment_shapes(las, shp_line(k = 20), "line")
las <- segment_shapes(las, shp_hline(k = 20), "hline")
las <- segment_shapes(las, shp_vline(k = 20), "vline")

#plot(las)
#plot(las, color = "plane")
#plot(las, color = "hplane")
#plot(las, color = "line")
#plot(las, color = "hline")
#plot(las, color = "vline")
```

`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

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

```
  sigma = size/6
)

unsmooth_height(las)
```

Arguments

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

Details

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

Value

An object of the class LAS.

Examples

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

las <- decimate_points(las, highest(1))
#plot(las)

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

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

---

snag_wing2015 *Snags Segmentation Algorithm*

Description

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

Usage

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

Arguments

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

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

Examples

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

stdmetrics *Predefined standard metrics functions*

Description

Predefined metrics functions intended to me used in `*_metrics` function such as pixel_metrics, cloud_metrics, crown_metrics, voxel_metrics and so on. 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_pulse(pulseID, rn)

stdmetrics_ctrl(x, y, z)

stdtreemetrics(x, y, z)

stdshapemetrics(x, y, z)

.stdmetrics

.stdmetrics_z

.stdmetrics_i

.stdmetrics_rn

.stdmetrics_pulse

.stdmetrics_ctrl

.stdtreemetrics
```

`.stdshapemetrics`

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.
`zmin`	numeric. Lower bound of the integral for zpcumx metrics. See wiki page and Wood et al. (2008) reference.
`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` is a combination of `stdmetrics_ctrl` + `stdmetrics_z` + `stdmetrics_i` + `stdmetrics_rn`

`stdtreemetrics` is a special function that works with crown_metrics. Actually, it won't fail with other functions but the output makes more sense if computed at the individual tree level.

`stdshapemetrics` is a set of eigenvalue based feature described in Lucas et al, 2019 (see references).

References

M. Woods, K. Lim, and P. Treitz. Predicting forest stand variables from LiDAR data in the Great Lakes – St. Lawrence forest of Ontario. The Forestry Chronicle. 84(6): 827-839. https://doi.org/10.5558/tfc84827-6

Lucas, C., Bouten, W., Koma, Z., Kissling, W. D., & Seijmonsbergen, A. C. (2019). Identification of Linear Vegetation Elements in a Rural Landscape Using LiDAR Point Clouds. Remote Sensing, 11(3), 292.

Examples

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

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

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

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

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

# Convenient shortcut for the previous example
m5 <- pixel_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(
     above2aboven1st = x,       # Num of returns above 2 divided by num of 1st returns
     zimean  = mean(z*i),       # Mean products of z by intensity
     zsqmean = sqrt(mean(z^2))  # Quadratic mean of z
   )

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

m10 <- pixel_metrics(las, ~myMetrics(Z, Intensity, ReturnNumber), res = 40)

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

---

st_area *Surface covered by a LAS* object*

Description

Surface covered by a LAS* object. The surface covered by a point cloud is mathematically 0. To compute non zero values the function uses different strategies. The area is computed based on the number of occupied cells, or on the area of the convex hull of the points depending on the density and the size of the point cloud. The result is necessarily an approximation that depends on the method used.

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 area covered because some regions are sampled twice. For a LASheader it is computed with the bounding box. As a consequence, for the same file st_area applied on a LASheader or on a LAS can return slightly different values. st_area() extends sf:st_area(), area() extends raster:area(). area() is provided for backward compatibility.

Usage

```
## S3 method for class 'LAS'
st_area(x, ...)

## S3 method for class 'LASheader'
st_area(x, ...)

## S3 method for class 'LAScatalog'
st_area(x, ...)

area(x, ...)
```

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

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

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

Arguments

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

Value

numeric. A number in the same units as the coordinate reference system.

---

st_bbox *Bounding box of a LAS* object*

Description

Bounding box of a LAS* object. st_bbox() extends sf, and ext() extends terra. The values returned are similar to their parent functions.

Usage

```
## S3 method for class 'LAS'
st_bbox(obj, ...)

## S3 method for class 'LASheader'
st_bbox(obj, ...)

## S3 method for class 'LAScatalog'
st_bbox(obj, ...)

## S3 method for class 'LAScluster'
st_bbox(obj, ...)

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

## S4 method for signature 'LASheader'
ext(x, ...)
```

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

## S4 method for signature 'LAScluster'
ext(x, ...)
```

Arguments

obj, x	An object of class LAS*.
...	unused

Value

A bbox from sf, or a SpatExtent from terra.

Examples

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

st_bbox(las)
ext(las)
```

---

st_coordinates *Coordinates of a LAS* object in a matrix form*

Description

Retrieve coordinates of a LAS* object in matrix form. It creates a copy of the coordinates because of the coercion from data.frame to matrix. This function inherits sf::st_coordinates

Usage

```
## S3 method for class 'LAS'
st_coordinates(x, z = TRUE, ...)

## S3 method for class 'LAScatalog'
st_coordinates(x, ...)
```

Arguments

x	A LAS* object
z	bool. Return XY or XYZ matrix
...	unused.

Value

matrix

Examples

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

---

st_crs *Get or set the projection of a LAS* object*

Description

Get or set the projection of a LAS* object. st_crs() extends sf:st_crs(), projection() and crs() extend raster:projection() and raster:crs(). projection() and crs() are provided for backward compatibility. For epsg() and wkt(), see details.

Usage

```
## S3 method for class 'LAS'
st_crs(x, ...)

## S3 method for class 'LAScatalog'
st_crs(x, ...)

## S3 method for class 'LASheader'
st_crs(x, ...)

## S3 method for class 'LAScluster'
st_crs(x, ...)

## S3 replacement method for class 'LAS'
st_crs(x) <- value

## S3 replacement method for class 'LASheader'
st_crs(x) <- value

## S3 replacement method for class 'LAScatalog'
st_crs(x) <- value

projection(x, asText = TRUE)

projection(x) <- value

crs(x, asText = FALSE)

crs(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 method for signature 'LAScluster'
crs(x, asText = FALSE)

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

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

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(obj)

wkt(obj) <- value

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

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

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

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

Arguments

x, object, obj	An object of class LAS*
...	Unused.
value	A CRS or a crs 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 WTK string (for LAS 1.4)

On the other hand, R spatial packages use a crs object to store the CRS. 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 crs, and also in a format that can be understood by other R packages.

- st_crs return the CRS in sf format.
- st_crs<-: 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
- epsg: reads the epsg code from the header.
- wkt: reads the WKT string from the header.

Value

A st_crs() returns a sf::crs. projection() and crs() return a sp::CRS and should no longer be used.

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)

# Overwrite the CRS (but does not reproject)
st_crs(las) <- 26918
las
```

st_hull *Concave and convex hulls for LAS objects*

Description

Concave and convex hulls for LAS objects. `st_convex_hull` extends `sf::st_convex_hull` for LAS objects. Both functions return a `sfc_POLYGON`. `concaveman` is very a fast 2D concave hull algorithm for a set of points.

Usage

```
st_concave_hull(x, method = "concaveman", ...)

## S3 method for class 'LAS'
st_convex_hull(x)

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

Arguments

x, y	An object of class LAS or XY coordinates of points in case of `concaveman`. This can be specified as two vectors x and y, a 2-column matrix x, a list with two components, etc.
`method`	string. currently supports "concaveman".
`...`	Propagate to the method.
`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 below this threshold, it stops being considered for further detailed processing. Higher values result in simpler shapes.

Details

The concaveman 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.

Value

A `sfc_POLYGON` from `sf` or a `data.frame` in the case of `concaveman`

References

Park, J.-S & Oh, S.-J. (2013). A New Concave Hull Algorithm and Concaveness Measure for n-dimensional Datasets. Journal of Information Science and Engineering. 29. 379-392.

Examples

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

LASfile <- system.file("extdata", "Megaplot.laz", package="lidR")
las = readLAS(LASfile, filter = "-drop_z_below 1")
hull = st_concave_hull(las, length_threshold = 10)
plot(hull)
```

---

st_transform *Transform or convert coordinates of LAS objects*

Description

Transform or convert coordinates of LAS objects `st_transform()` extends `sf::st_transform()`

Usage

```
## S3 method for class 'LAS'
st_transform(x, crs, ...)
```

Arguments

`x`	An object of class LAS
`crs`	crs object from sf or CRS object from sp
`...`	additional arguments `scale` and `xoffset`, `yoffset`, `zoffset` for the output LAS objects. It is not mandatory but recommended to consider providing such information. Otherwise it will be guessed automatically which might not be the best choice.

Value

A LAS object

Examples

```
LASfile <- system.file("extdata", "example.laz", package="rlas")
las = readLAS(LASfile)
st_crs(las)$Name
st_bbox(las)
tlas <- sf::st_transform(las, sf::st_crs(26918))
st_crs(tlas)$Name
st_bbox(tlas)
```

---

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.

Usage

```
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

An sf object with POINT Z geometries. Information about the time interval and the score of the positioning (according to the method used) are also in the table of attributes.

Non-supported LAScatalog options

The option 'select' is not supported and not respected because it is internally known what is the best to select
The option 'output_files' is not supported and not respected because the output must be post-processed as a whole

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.

Author(s)

Jean-Francois Bourdon & Jean-Romain Roussel

Examples

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

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

plot(las@header)
plot(sf::st_geometry(flightlines), add = TRUE)

#plot(las) |> add_flightlines3d(flightlines, radius = 10)
```

```
## Not run:
# With a LAScatalog "-drop_single" and "-thin_pulses_with_time"
# are used by default
ctg = readLAScatalog("folder/")
flightlines <- track_sensor(ctg,  Roussel2020(pmin = 15))
plot(flightlines)

## End(Not run)
```

`track_sensor_gatziolis2019` *Sensor tracking algorithm*

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

```
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 Gaziolis and Jean-Romain Roussel

References

Gatziolis, D., & McGaughey, R. J. (2019). Reconstructing Aircraft Trajectories from Multi-Return Airborne Laser-Scanning Data. Remote Sensing, 11(19), 2258.

Examples

```
# A valid file properly populated
LASfile <- system.file("extdata", "Topography.laz", package="lidR")
las = readLAS(LASfile)
flightlines <- track_sensor(las, Gatziolis2019())

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

`track_sensor_roussel2020` *Sensor tracking algorithm*

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 https://doi.org/10.31223/osf.io/k32qw

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
flightlines <- track_sensor(las, Roussel2020(pmin = 15))

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

---

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 `mean(Intensity)`. Other attributes cannot be aggregated and are lost.

Usage

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

Arguments

`las`	An object of class LAS or LAScatalog.
`res`	numeric. The resolution of the voxels. `res = 1` for a 1x1x1 cubic voxels. Optionally `res = c(1,2)` for non-cubic voxels (1x1x2 cuboid voxel).

Value

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

Examples

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

las2 = voxelize_points(las, 5)
#plot(las2, voxel = TRUE)
```

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"))
```

Index