Package: lasR (via r-universe)

September 16, 2024

Type Package

Version 0.10.2

Title Fast and Pipeable Airborne LiDAR Data Tools

Description Fast and pipeable airborne lidar processing tools. Read/write 'las' and 'laz' files, computation of metrics in area based approach, point filtering, normalization, individual tree segmentation and other manipulations in a powerful and versatile processing chain.

URL https://github.com/r-lidar/lasR

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

License GPL-3 + file LICENSE

Depends R (>= 3.6.0)

Imports methods, utils, stats

Suggests knitr, rmarkdown, sf, terra, testthat (>= 3.0.0),

RoxygenNote 7.3.1

SystemRequirements C++17, GDAL (>= 2.2.3), GEOS (>= 3.4.0), PROJ (>= 4.9.3), sqlite3, GNU make

Encoding UTF-8

Language en-US

Config/testthat/edition 3

VignetteBuilder knitr

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

RemoteUrl https://github.com/r-lidar/lasR

RemoteRef HEAD

RemoteSha ab345f06f9df34629c964d2bf68396ec1578826e

Contents

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

Description

lasR provides a set of tools to process efficiently 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 efficiently.

Author(s)

Maintainer: Jean-Romain Roussel `<info@r-lidar.com>` [copyright holder]

Other contributors:

- Martin Isenburg (Is the author of the included LASlib and LASzip libraries) [copyright holder]
- Benoît St-Onge (Is the author of the included 'chm_prep' function) [copyright holder]
- Niels Lohmann (Is the author of the included json parser) [copyright holder]
- Volodymyr Bilonenko (Is the author of the included delaunator triangulation) [copyright holder]
- State Key Laboratory of Remote Sensing Science, Institute of Remote Sensing Science and Engineering, Beijing Normal University (Is the copyright holder of the included CSF) [copyright holder]

See Also

Useful links:

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

`add_extrabytes` *Add attributes to a LAS file*

Description

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. This stages adds an extra bytes attribute to the points. Values are zeroed: the underlying point cloud is edited to support a new extrabyte attribute. This new attribute can be populated later in another stage

Usage

```
add_extrabytes(data_type, name, description, scale = 1, offset = 0)
```

Arguments

`data_type`	character. The data type of the extra bytes attribute. Can be "uchar", "char", "ushort", "short", "uint", "int", "uint64", "int64", "float", "double".
`name`	character. The name of the extra bytes attribute to add to the file.
`description`	character. A short description of the extra bytes attribute to add to the file (32 characters).
`scale`, `offset`	numeric. The scale and offset of the data. See LAS specification.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

Examples

```
f <- system.file("extdata", "Example.las", package = "lasR")
fun <- function(data) { data$RAND <- runif(nrow(data), 0, 100); return(data) }
pipeline <- reader_las() +
  add_extrabytes("float", "RAND", "Random numbers") +
  callback(fun, expose = "xyz")
exec(pipeline, on = f)
```

---

add_rgb — *Add RGB attributes to a LAS file*

Description

Modifies the LAS format to convert into a format with RGB attributes. Values are zeroed: the underlying point cloud is edited to be transformed in a format that supports RGB. RGB can be populated later in another stage. If the point cloud already has RGB, nothing happens, RGB values are preserved.

Usage

```
add_rgb()
```

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

Examples

```
f <- system.file("extdata", "Example.las", package="lasR")

pipeline <- add_rgb() + write_las()
exec(pipeline, on = f)
```

`callback` *Call a user-defined function on the point cloud*

Description

Call a user-defined function on the point cloud. The function receives a `data.frame` with the point cloud. Its first input must be the point cloud. If the function returns anything other than a `data.frame` with the same number of points, the output is stored and returned at the end. However, if the output is a `data.frame` with the same number of points, it updates the point cloud. This function can, therefore, be used to modify the point cloud using a user-defined function. The function is versatile but complex. A more comprehensive set of examples can be found in the online tutorial.

Usage

```
callback(fun, expose = "xyz", ..., drop_buffer = FALSE, no_las_update = FALSE)
```

Arguments

`fun`	function. A user-defined function that takes as first argument a `data.frame` with the exposed point cloud attributes (see examples).
`expose`	character. Expose only attributes of interest to save memory (see details).
`...`	parameters of function `fun`
`drop_buffer`	bool. If false, does not expose the point from the buffer.
`no_las_update`	bool. If the user-defined function returns a data.frame, this is supposed to update the point cloud. Can be disabled.

Details

In `lasR`, the point cloud is not exposed to R in a `data.frame` like in lidR. It is stored internally in a C++ structure and cannot be seen or modified directly by users using R code. The `callback` function is the only stage that allows direct interaction with the point cloud by **copying** it temporarily into a `data.frame` to apply a user-defined function.

expose: the 'expose' argument specifies the data that will actually be exposed to R. For example, 'xyzia' means that the x, y, and z coordinates, the intensity, and the scan angle will be exposed. 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+) Also numbers from 1 to 9 for the extra bytes data numbers 1 to 9. 'E' enables all extra bytes to be loaded. '*' is the wildcard that enables everything to be exposed from the LAS file.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

See Also

write_las

Examples

```
f <- system.file("extdata", "Topography.las", package = "lasR")

# There is no function in lasR to read the data in R. Let's create one
read_las <- function(f)
{
  load <- function(data) { return(data) }
  read <- reader_las()
  call <- callback(load, expose = "xyzi", no_las_update = TRUE)
  return (exec(read + call, on = f))
}
las <- read_las(f)
head(las)

convert_intensity_in_range <- function(data, min, max)
{
  i <- data$Intensity
  i <- ((i - min(i)) / (max(i) - min(i))) * (max - min) + min
  i[i < min] <- min
  i[i > max] <- max
  data$Intensity <- as.integer(i)
  return(data)
}

read <- reader_las()
call <- callback(convert_intensity_in_range, expose = "i", min = 0, max = 255)
write <- write_las()
pipeline <- read + call + write
ans <- exec(pipeline, on = f)

las <- read_las(ans)
head(las)
```

`chm` *Canopy Height Model*

Description

Create a Canopy Height Model using triangulate and rasterize.

Usage

```
chm(res = 1, tin = FALSE, ofile = tempfile(fileext = ".tif"))
```

Arguments

`res`	numeric. The resolution of the raster.
`tin`	bool. By default the CHM is a point-to-raster based methods i.e. each pixel is assigned the elevation of the highest point. If `tin = TRUE` the CHM is a triangulation-based model. The first returns are triangulated and interpolated.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

See Also

triangulate rasterize

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
pipeline <- reader_las() + chm()
exec(pipeline, on = f)
```

`classify_with_csf` *Classify ground points*

Description

Classify points using the Cloth Simulation Filter by Zhang et al. (2016) (see references) that relies on the authors' original source code. If the point cloud already has ground points, the classification of the original ground point is set to zero. This stage modifies the point cloud in the pipeline but does not produce any output.

Usage

```
classify_with_csf(
  slope_smooth = FALSE,
  class_threshold = 0.5,
  cloth_resolution = 0.5,
  rigidness = 1L,
  iterations = 500L,
  time_step = 0.65,
  ...,
  class = 2L,
  filter = "-keep_last"
)
```

Arguments

- `slope_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.
- `...` Unused
- `class` integer. The classification to attribute to the points. Usually 2 for ground points.
- `filter` the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

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)

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
pipeline = classify_with_csf(TRUE, 1 ,1, time_step = 1) + write_las()
ans = exec(pipeline, on = f, progress = TRUE)
```

`classify_with_ivf` *Classify noise points*

Description

Classify points using Isolated Voxel Filter (IVF). The stage identifies points that have only a few other points in their surrounding 3 x 3 x 3 = 27 voxels and edits the points to assign a target classification. Used with class 18, it classifies points as noise. This stage modifies the point cloud in the pipeline but does not produce any output.

Usage

```
classify_with_ivf(res = 5, n = 6L, class = 18L)
```

Arguments

`res`	numeric. Resolution of the voxels.
`n`	integer. The maximal number of 'other points' in the 27 voxels.
`class`	integer. The class to assign to the points that match the condition.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

`classify_with_sor` *Classify noise points*

Description

Classify points using the 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 neighbors. The points that are farther than the average distance plus a number of times (multiplier) the standard deviation are considered noise.

Usage

```
classify_with_sor(k = 8, m = 6, class = 18L)
```

Arguments

`k`	numeric. The number of neighbours
`m`	numeric. Multiplier. The maximum distance will be: avg distance + m * std deviation
`class`	integer. The class to assign to the points that match the condition.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

`delete_points` *Filter and delete points*

Description

Remove some points from the point cloud. This stage modifies the point cloud in the pipeline but does not produce any output.

Usage

```
delete_points(filter = "")
```

Arguments

`filter`	the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

Examples

```
f <- system.file("extdata", "Megaplot.las", package="lasR")
read <- reader_las()
filter <- delete_points(keep_z_above(4))

pipeline <- read + summarise() + filter + summarise()
exec(pipeline, on = f)
```

---

dtm *Digital Terrain Model*

Description

Create a Digital Terrain Model using triangulate and rasterize.

Usage

```
dtm(res = 1, add_class = NULL, ofile = temptif())
```

Arguments

`res`	numeric. The resolution of the raster.
`add_class`	integer. By default it triangulates using ground and water points (classes 2 and 9). It is possible to provide additional classes.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

See Also

triangulate rasterize

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
pipeline <- reader_las() + dtm()
exec(pipeline, on = f)
```

exec — *Process the pipeline*

Description

Process the pipeline. Every other functions of the package do nothing. This function must be called on a pipeline in order to actually process the point-cloud. To process in parallel using multiple cores, refer to the multithreading page.

Usage

```
exec(pipeline, on, with = NULL, ...)
```

Arguments

`pipeline`	a pipeline. A serie of stages called in order
`on`	Can be the paths of the files to use, the path of the folder in which the files are stored, the path to a virtual point cloud file or a `data.frame` containing the point cloud. It supports also a `LAScatalog` or a `LAS` objects from `lidR`.
`with`	list. A list of options to control how the pipeline is executed. This includes options to control parallel processing, progress bar display, tile buffering and so on. See set_exec_options for more details on the available options.
`...`	The processing options can be explicitly named and passed outside the `with` argument. See set_exec_options

See Also

multithreading set_exec_options

Examples

```
## Not run:
f <- paste0(system.file(package="lasR"), "/extdata/bcts/")
f <- list.files(f, pattern = "(?i)\\.la(s|z)$", full.names = TRUE)

read <- reader_las()
tri <- triangulate(15)
dtm <- rasterize(5, tri)
lmf <- local_maximum(5)
met <- rasterize(2, "imean")
pipeline <- read + tri + dtm + lmf + met
ans <- exec(pipeline, on = f, with = list(progress = TRUE))

## End(Not run)
```

filters *Point filters*

Description

lasR uses LASlib/LASzip, the library developed by Martin Isenburg to read and write LAS/LAZ files. Thus, the flags that are available in `LAStools` are also available in `lasR`. Filters are strings to put in the `filter` arguments of the `lasR` algorithms. The list of available strings is accessible with `filter_usage`. For convenience, the most useful filters have an associated function that returns the corresponding string.

Usage

```
keep_class(x)

drop_class(x)

keep_first()

drop_first()

keep_ground()

keep_ground_and_water()

drop_ground()

keep_noise()

drop_noise()

keep_z_above(x)

drop_z_above(x)

keep_z_below(x)

drop_z_below(x)

drop_duplicates()

filter_usage()

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

## S3 method for class 'laslibfilter'
```

```
e1 + e2
```

Arguments

`x`	numeric or integer as a function of the filter used.
`...`	Unused.
`e1, e2`	lasR objects.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
filter_usage()
gnd = keep_class(c(2,9))
reader_las(gnd)
triangulate(filter = keep_ground())
rasterize(1, "max", filter = "-drop_z_below 5")
```

`filter_with_grid` *Select highest or lowest points*

Description

Select and retained only highest or lowest points per grid cell

Usage

```
filter_with_grid(res, operator = "min", filter = "")
```

Arguments

`res`	numeric. The resolution of the grid
`operator`	string. Can be min or max to retain lowest or highest points
`filter`	the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

`focal` *Calculate focal ("moving window") values for each cell of a raster*

Description

Calculate focal ("moving window") values for each cell of a raster using various functions. NAs are always omitted; thus, this stage effectively acts as an NA filler. The window is always circular. The edges are handled by adjusting the window.

Usage

```
focal(raster, size, fun = "mean", ofile = temptif())
```

Arguments

`raster`	LASRalgorithm. A stage that produces a raster.
`size`	numeric. The window size **in the units of the point cloud**, not in pixels. For example, 2 means 2 meters or 2 feet, not 2 pixels.
`fun`	string. Function to apply. Supported functions are 'mean', 'median', 'min', 'max', 'sum'.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

Value

This stage produces a raster. The path provided to 'ofile' is expected to be '.tif' or any other format supported by GDAL.

Examples

```
f <- system.file("extdata", "Topography.las", package = "lasR")

chm = rasterize(2, "zmax")
chm2 = lasR:::focal(chm, 8, fun = "mean")
chm3 = lasR:::focal(chm, 8, fun = "max")
pipeline <- reader_las() + chm + chm2 + chm2
ans = exec(pipeline, on = f)

terra::plot(ans[[1]])
terra::plot(ans[[2]])
terra::plot(ans[[3]])
```

`geometry_features` *Compute pointwise geometry features*

Description

Compute pointwise geometry features based on local neighborhood. Each feature is added into an extrabyte attribute. The names of the extrabytes attributes (if recorded) are `coeff00`, `coeff01`, `coeff02` and so on, `lambda1`, `lambda2`, `lambda3`, `anisotropy`, `planarity`, `sphericity`, `linearity`, `omnivariance`, `curvature`, `eigensum`, `angle`, `normalX`, `normalY`, `normalZ` (recorded in this order). There is a total of 23 attributes that can be added. It is strongly discouraged to use them all. All the features are recorded with single precision floating points yet computing them all will triple the size of the point cloud. This stage modifies the point cloud in the pipeline but does not produce any output.

Usage

```
geometry_features(k, r, features = "")
```

Arguments

`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.
`features`	String. Geometric feature to export. Each feature is added into an extrabyte attribute. Use 'C' for the 9 principal component coefficients, 'E' for the 3 eigenvalues of the covariance matrix, 'a' for anisotropy, 'p' for planarity, 's' for sphericity, 'l' for linearity, 'o' for omnivariance, 'c' for curvature, 'e' for the sum of eigenvalues, 'i' for the angle (inclination in degrees relative to the azimuth), and 'n' for the 3 components of the normal vector. Notice that the uppercase labeled components allow computing all the lowercase labeled components. Default is "". In this case, the singular value decomposition is computed but serves no purpose. The order of the flags does not matter and the features are recorded in the order mentioned above.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

References

Hackel, T., Wegner, J. D., & Schindler, K. (2016). Contour detection in unstructured 3D point clouds. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1610-1618).

Examples

```
f <- system.file("extdata", "Example.las", package = "lasR")
pipeline <- geometry_features(8, features = "pi") + write_las()
ans <- exec(pipeline, on = f)
```

hulls — *Contour of a point cloud*

Description

This stage uses a Delaunay triangulation and computes its contour. The contour of a strict Delaunay triangulation is the convex hull, but in lasR, the triangulation has a `max_edge` argument. Thus, the contour might be a convex hull with holes. Used without triangulation it returns the bouding box of the points.

Usage

```
hulls(mesh = NULL, ofile = tempgpkg())
```

Arguments

`mesh`	NULL or LASRalgorithm. A `triangulate` stage. If NULL take the bounding box of the header of each file.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

Value

This stage produces a vector. The path provided to ‘ofile‘ is expected to be ‘.gpkg‘ or any other format supported by GDAL. Vector stages may produce geometries with Z coordinates. Thus, it is discouraged to store them in formats with no 3D support, such as shapefiles.

See Also

triangulate

Examples

```
f <- system.file("extdata", "Topography.las", package = "lasR")
read <- reader_las()
tri <- triangulate(20, filter = keep_ground())
contour <- hulls(tri)
pipeline <- read + tri + contour
ans <- exec(pipeline, on = f)
plot(ans)
```

load_raster *Load a raster for later use*

Description

Load a raster from a disk file for later use. For example, load a DTM to feed the transform_with stage or load a CHM to feed the pit_fill stage. The raster is never loaded entirely. Internally, only chunks corresponding to the currently processed point cloud are loaded. Be careful: internally, the raster is read as float no matter the original datatype.

Usage

```
load_raster(file, band = 1L)
```

Arguments

`file`	character. Path to a raster file.
`band`	integer. The band to load. It reads and loads only a single band.

Examples

```
r <- system.file("extdata/bcts", "bcts_dsm_5m.tif", package = "lasR")
f <- paste0(system.file(package = "lasR"), "/extdata/bcts/")
f <- list.files(f, pattern = "(?i)\\.la(s|z)$", full.names = TRUE)

# In the following pipeline, neither load_raster nor pit_fill process any points.
# The internal engine is capable of knowing that, and the LAS files won't actually be
# read. Yet the raster r will be processed by chunk following the LAS file pattern.
rr <- load_raster(r)
pipeline <- rr + pit_fill(rr)
ans <- exec(pipeline, on = f, verbose = FALSE)
```

local_maximum *Local Maximum*

Description

The Local Maximum stage identifies points that are locally maximum. The window size is fixed and circular. This stage does not modify the point cloud. It produces a derived product in vector format. The function `local_maximum_raster` applies on a raster instead of the point cloud

Usage

```
local_maximum(
  ws,
  min_height = 2,
  filter = "",
  ofile = tempgpkg(),
  use_attribute = "Z",
  record_attributes = FALSE
)

local_maximum_raster(
  raster,
  ws,
  min_height = 2,
  filter = "",
  ofile = tempgpkg()
)
```

Arguments

`ws`	numeric. Diameter of the moving window used to detect the local maxima in the units of the input data (usually meters).
`min_height`	numeric. Minimum height of a local maximum. Threshold below which a point cannot be a local maximum. Default is 2.
`filter`	the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.
`use_attribute`	character. By default the local maximum is performed on the coordinate Z. Can also be the name of an extra bytes attribute such as 'HAG' if it exists. Can also be 'Intensity' but there is probably no use case for that one.
`record_attributes`	The coordinates XYZ of points corresponding to the local maxima are recorded. It is also possible to record the attributes of theses points such as the intensity, return number, scan angle and so on.
`raster`	LASRalgorithm. A stage that produces a raster.

Value

This stage produces a vector. The path provided to 'ofile' is expected to be '.gpkg' or any other format supported by GDAL. Vector stages may produce geometries with Z coordinates. Thus, it is discouraged to store them in formats with no 3D support, such as shapefiles.

Examples

```
f <- system.file("extdata", "MixedConifer.las", package = "lasR")
read <- reader_las()
lmf <- local_maximum(5)
ans <- exec(read + lmf, on = f)
ans

chm <- rasterize(1, "max")
lmf <- local_maximum_raster(chm, 5)
ans <- exec(read + chm + lmf, on = f)
# terra::plot(ans$rasterize)
# plot(ans$local_maximum, add = T, pch = 19)
```

`metric_engine` *Metric engine*

Description

The metric engine is an internal tool that allow to derive any metric from a set of points by parsing a string. It is used by rasterize, summarise as well as other functions. Each string is composed of two parts separated by an underscore. The first part is the attribute on which the metric must be computed (e.g., z, intensity, classification). The second part is the name of the metric (e.g., `mean`, `sd`, `cv`). A string thus typically looks like `"z_max"`, `"intensity_min"`, `"z_mean"`, `"classification_mode"`. For more details see the sections 'Attribute' and 'Metrics' respectively.

Details

Be careful: the engine supports any combination of `attribute_metric` strings. While they are all computable, they are not all meaningful. For example, `c_mode` makes sense but not `z_mode`. Also, all metrics are computed with 32-bit floating point accuracy, so `x_mean` or `y_sum` might be slightly inaccurate, but anyway, these metrics are not supposed to be useful.

Attribute

The available attributes are accessible via a single letter or via their lowercase name: t - gpstime, a - angle, i - intensity, n - numberofreturns, r - returnnumber, c - classification, s - synthetic, k - keypoint, w - withheld, o - overlap (format 6+), u - userdata, p - pointsourceid, e - edgeofflightline, d - scandirectionflag, R - red, G - green, B - blue, N - nir.
Be careful to the typos: attributes are non failing features. If the attribute does not exist `NaN` is returned. Thus `intesity_mean` return `NaN` rather than failing.

Metrics

The available metric names are: `count`, `max`, `min`, `mean`, `median`, `sum`, `sd`, `cv`, `pX` (percentile), `aboveX`, and `mode`. Some metrics have an attribute + name + a parameter `X`, such as `pX` where `X` can be substituted by a number. Here, `z_pX` represents the Xth percentile; for instance, `z_p95` signifies the 95th percentile of z. `z_aboveX` corresponds to the percentage of points above `X` (sometimes called canopy cover).

It is possible to call a metric without the name of the attribute. In this case, z is the default. e.g. `mean` equals `z_mean`

Extrabytes attribute

The core attributes are x, y, z, classification, intensity, and so on. Some point clouds have extra attributes called extrabytes attributes. In this case, metrics can be derived the same way using the names of the extra attributes. Be careful of typos. The attributes are not checked internally because of the extrabytes attributes. For example, if a user requests: `ntensity_mean`, this could be a typo or the name of an extra attribute. Because extrabytes are never failing, `ntensity_mean` will return `NaN` rather than an error.

Examples

```
metrics = c("z_max", "i_min", "r_mean", "n_median", "z_sd", "c_sd", "t_cv", "u_sum", "z_p95")
f <- system.file("extdata", "Example.las", package="lasR")
p <- summarise(metrics = metrics)
r <- rasterize(5, operators = metrics)
ans <- exec(p+r, on = f)
ans$summary$metrics
ans$rasterize
```

`multithreading` *Parallel processing tools*

Description

`lasR` uses OpenMP to paralellize the internal C++ code. `set_parallel_strategy()` globally changes the strategy used to process the point clouds. `sequential()`, `concurrent_files()`, `concurrent_points()`, and `nested()` are functions to assign a parallelization strategy (see Details). `has_omp_support()` tells you if the `lasR` package was compiled with the support of OpenMP which is unlikely to be the case on MacOS.

Usage

```
set_parallel_strategy(strategy)

unset_parallel_strategy()

get_parallel_strategy()
```

```
ncores()

half_cores()

sequential()

concurrent_files(ncores = half_cores())

concurrent_points(ncores = half_cores())

nested(ncores = ncores()/4L, ncores2 = 2L)

has_omp_support()
```

Arguments

`strategy`	An object returned by one of `sequential()`, `concurrent_points()`, `concurrent_files()` or `nested()`.
`ncores`	integer. Number of cores.
`ncores2`	integer. Number of cores. For `nested` strategy `ncores` is the number of concurrent files and `ncores2` is the number of concurrent points.

Details

There are 4 strategies of parallel processing:

sequential No parallelization at all: `sequential()`

concurrent-points Point cloud files are processed sequentially one by one. Inside the pipeline, some stages are parallelized and are able to process multiple points simultaneously. Not all stages are natively parallelized. E.g. `concurrent_points(4)`

concurrent-files Files are processed in parallel. Several files are loaded in memory and processed simultaneously. The entire pipeline is parallelized, but inside each stage, the points are processed sequentially. E.g. `concurrent_files(4)`

nested Files are processed in parallel. Several files are loaded in memory and processed simultaneously, and inside some stages, the points are processed in parallel. E.g. `nested(4,2)`

`concurrent-files` is likely the most desirable and fastest option. However, it uses more memory because it loads multiple files. The default is `concurrent_points(half_cores())` and can be changed globally using e.g. `set_parallel_strategy(concurrent_files(4))`

Examples

```
## Not run:
f <- paste0(system.file(package="lasR"), "/extdata/bcts/")
f <- list.files(f, pattern = "(?i)\\.la(s|z)$", full.names = TRUE)

pipeline <- reader_las() + rasterize(2, "imean")
```

```
ans <- exec(pipeline, on = f, progress = TRUE, ncores = concurrent_files(4))

set_parallel_strategy(concurrent_files(4))
ans <- exec(pipeline, on = f, progress = TRUE)

## End(Not run)
```

`normalize` *Normalize the point cloud*

Description

Normalize the point cloud using triangulate and transform_with. It triangulates the ground points and then applies `transform_with` to linearly interpolate the elevation of each point within each triangle.

Usage

```
normalize(extrabytes = FALSE)
```

Arguments

`extrabytes`	bool. If FALSE the coordinate Z of the point cloud is modified and becomes the height above ground (HAG). If TRUE the coordinate Z is not modified and a new extrabytes attribute named 'HAG' is added to the point cloud.

See Also

triangulate transform_with

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
pipeline <- reader_las() + normalize() + write_las()
exec(pipeline, on = f)
```

`pit_fill` *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

```
pit_fill(
  raster,
  lap_size = 3L,
  thr_lap = 0.1,
  thr_spk = -0.1,
  med_size = 3L,
  dil_radius = 0L,
  ofile = temptif()
)
```

Arguments

`raster`	LASRalgorithm. A stage that produces a raster.
`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).
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

Value

This stage produces a raster. The path provided to ‘ofile‘ is expected to be ‘.tif‘ or any other format supported by GDAL.

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=81365288221f3ac34b51a

Examples

```
f <- system.file("extdata", "MixedConifer.las", package="lasR")

reader <- reader_las(filter = keep_first())
tri <- triangulate()
chm <- rasterize(0.25, tri)
pit <- pit_fill(chm)
u <- exec(reader + tri + chm + pit, on = f)

chm <- u[[1]]
```

```
sto <- u[[2]]

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

`rasterize` *Rasterize a point cloud*

Description

Rasterize a point cloud using different approaches. This stage does not modify the point cloud. It produces a derived product in raster format.

Usage

```
rasterize(res, operators = "max", filter = "", ofile = temptif(), ...)
```

Arguments

`res`	numeric. The resolution of the raster. Can be a vector with two resolutions. In this case it does not correspond to the x and y resolution but to a buffered rasterization. (see section 'Buffered' and examples)
`operators`	Can be a character vector. "min", "max" and "count" are accepted as well as many others (see section 'Operators'). Can also rasterize a triangulation if the input is a LASRalgorithm for triangulation (see examples). Can also be a user-defined expression (see example and section 'Operators').
`filter`	the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.
`...`	`default_value` numeric. When rasterizing with an operator and a filter (e.g. `-keep_z_above 2`) some pixels that are covered by points may no longer contain any point that pass the filter criteria and are assigned NA. To differentiate NAs from non covered pixels and NAs from covered pixels without point that pass the filter, the later case can be assigned another value such as 0.

Value

This stage produces a raster. The path provided to 'ofile' is expected to be '.tif' or any other format supported by GDAL.

Operators

If `operators` is a string or a vector of strings: read metric_engine to see the possible strings Below are some examples of valid calls:

```
rasterize(10, c("max", "count", "i_mean", "z_p95"))
rasterize(10, c("z_max", "c_count", "intensity_mean", "p95"))
```

If `operators` is an R user-defined expression, the function should return either a vector of numbers or a `list` containing atomic numbers. To assign a band name to the raster, the vector or the `list` must be named accordingly. The following are valid operators:

```
f = function(x) { return(mean(x)) }
g = function(x,y) { return(c(avg = mean(x), med = median(y))) }
h = function(x) { return(list(a = mean(x), b = median(x))) }
rasterize(10, f(Intensity))
rasterize(10, g(Z, Intensity))
rasterize(10, h(Z))
```

Buffered

If the argument `res` is a vector with two numbers, the first number represents the resolution of the output raster, and the second number represents the size of the windows used to compute the metrics. This approach is called Buffered Area Based Approach (BABA).

In classical rasterization, the metrics are computed independently for each pixel. For example, predicting a resource typically involves computing metrics with a 400 square meter pixel, resulting in a raster with a resolution of 20 meters. It is not possible to achieve a finer granularity with this method.

However, with buffered rasterization, it is possible to compute the raster at a resolution of 10 meters (i.e., computing metrics every 10 meters) while using 20 x 20 windows for metric computation. In this case, the windows overlap, essentially creating a moving window effect.

This option does not apply when rasterizing a triangulation, and the second value is not considered in this case.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
read <- reader_las()
tri  <- triangulate(filter = keep_ground())
dtm  <- rasterize(1, tri) # input is a triangulation stage
avgi <- rasterize(10, mean(Intensity)) # input is a user expression
chm  <- rasterize(2, "max") # input is a character vector
pipeline <- read + tri + dtm + avgi + chm
ans <- exec(pipeline, on = f)
ans[[1]]
ans[[2]]
ans[[3]]
```

```
# Demonstration of buffered rasterization

# A good resolution for computing point density is 5 meters.
c0 <- rasterize(5, "count")

# Computing point density at too fine a resolution doesn't make sense since there is
# either zero or one point per pixel. Therefore, producing a point density raster with
# a 2 m resolution is not feasible with classical rasterization.
c1 <- rasterize(2, "count")

# Using a buffered approach, we can produce a raster with a 2-meter resolution where
# the metrics for each pixel are computed using a 5-meter window.
c2  <- rasterize(c(2,5), "count")

pipeline = read + c0 + c1 + c2
res <- exec(pipeline, on = f)
terra::plot(res[[1]]/25)  # divide by 25 to get the density
terra::plot(res[[2]]/4)   # divide by 4 to get the density
terra::plot(res[[3]]/25)  # divide by 25 to get the density
```

`reader_las` *Initialize the pipeline*

Description

This is the first stage that must be called in each pipeline. The stage does nothing and returns nothing if it is not associated to another processing stage. It only initializes the pipeline. `reader_las()` is the main function that dispatches into to other functions. `reader_las_coverage()` processes the entire point cloud. `reader_las_circles()` and `reader_las_rectangles()` read and process only some selected regions of interest. If the chosen reader has no options i.e. using `reader_las()` it can be omitted.

Usage

```
reader_las(filter = "", ...)

reader_las_coverage(filter = "", ...)

reader_las_circles(xc, yc, r, filter = "", ...)

reader_las_rectangles(xmin, ymin, xmax, ymax, filter = "", ...)
```

Arguments

`filter`	the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings

are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

`...` passed to other readers

`xc, yc, r` numeric. Circle centres and radius or radii.

`xmin, ymin, xmax, ymax`
numeric. Coordinates of the rectangles

Examples

```
f <- system.file("extdata", "Topography.las", package = "lasR")

pipeline <- reader_las() + rasterize(10, "zmax")
ans <- exec(pipeline, on = f)
# terra::plot(ans)

pipeline <- reader_las(filter = keep_z_above(1.3)) + rasterize(10, "zmean")
ans <- exec(pipeline, on = f)
# terra::plot(ans)

# read_las() with no option can be omitted
ans <- exec(rasterize(10, "zmax"), on = f)
# terra::plot(ans)

# Perform a query and apply the pipeline on a subset
pipeline = reader_las_circles(273500, 5274500, 20) + rasterize(2, "zmax")
ans <- exec(pipeline, on = f)
# terra::plot(ans)

# Perform a query and apply the pipeline on a subset with 1 output files per query
ofile = paste0(tempdir(), "/*_chm.tif")
pipeline = reader_las_circles(273500, 5274500, 20) + rasterize(2, "zmax", ofile = ofile)
ans <- exec(pipeline, on = f)
# terra::plot(ans)
```

`region_growing` *Region growing*

Description

Region growing for individual tree segmentation based on Dalponte and Coomes (2016) algorithm (see reference). Note that this stage strictly performs segmentation, while the original method 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

```
region_growing(
  raster,
  seeds,
```

```
  th_tree = 2,
  th_seed = 0.45,
  th_cr = 0.55,
  max_cr = 20,
  ofile = temptif()
)
```

Arguments

`raster`	LASRalgoritm. A stage producing a raster.
`seeds`	LASRalgoritm. A stage producing points used as seeds.
`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 data units)**. Default is 20. **BE CAREFUL** this algorithm exists in the `lidR` package and this parameter is in pixels in `lidR`.
`ofile`	character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

Value

This stage produces a raster. The path provided to 'ofile' is expected to be '.tif' or any other format supported by GDAL.

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.

Examples

```
f <- system.file("extdata", "MixedConifer.las", package="lasR")

reader <- reader_las(filter = keep_first())
chm <- rasterize(1, "max")
lmx <- local_maximum_raster(chm, 5)
tree <- region_growing(chm, lmx, max_cr = 10)
u <- exec(reader + chm + lmx + tree, on = f)

# terra::plot(u$rasterize)
```

```
# plot(u$local_maximum, add = T, pch = 19, cex = 0.5)
# terra::plot(u$region_growing, col = rainbow(150))
# plot(u$local_maximum, add = T, pch = 19, cex = 0.5)
```

`sampling_voxel` *Sample the point cloud*

Description

Sample the point cloud, keeping one random point per pixel or per voxel or perform a poisson sampling. This stages modify the point cloud in the pipeline but do not produce any output.

Usage

```
sampling_voxel(res = 2, filter = "", ...)

sampling_pixel(res = 2, filter = "", ...)

sampling_poisson(distance = 2, filter = "", ...)
```

Arguments

`res`	numeric. pixel/voxel resolution
`filter`	the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.
`...`	unused
`distance`	numeric. Minimum distance between points for poisson sampling.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
read <- reader_las()
vox <- sampling_voxel(5)
write <- write_las()
pipeline <- read + vox + write
exec(pipeline, on = f)
```

set_crs *Set the CRS of the pipeline*

Description

Assign a CRS in the pipeline. This stage **does not** reproject the data. It assigns a CRS. This stage affects subsequent stages of the pipeline and thus should appear close to reader_las to assign the correct CRS to all stages.

Usage

```
set_crs(x)
```

Arguments

x	integer or string. EPSG code or WKT string understood by GDAL

Examples

```
# expected usage
hmax = rasterize(10, "max")
pipeline = reader_las() + set_crs(2949) + hmax

# fancy usages are working as expected. The .tif file is written with a CRS, the .gpkg file with
# another CRS and the .las file with yet another CRS.
pipeline = set_crs(2044) + hmax + set_crs(2004) + local_maximum(5) + set_crs(2949) + write_las()
```

set_exec_options *Set global processing options*

Description

Set global processing options for the exec function. By default, pipelines are executed without a progress bar, processing one file at a time sequentially. The following options can be passed to the `exec()` function in four ways. See details.

Usage

```
set_exec_options(
  ncores = NULL,
  progress = NULL,
  buffer = NULL,
  chunk = NULL,
  ...
)

unset_exec_option()
```

Arguments

- `ncores` — An object returned by one of `sequential()`, `concurrent_points()`, `concurrent_files()`, or `nested()`. See multithreading. If `NULL` the default is `concurrent_points(half_cores())`. If a simple integer is provided it corresponds to `concurrent_files(ncores)`.
- `progress` — boolean. Displays a progress bar.
- `buffer` — numeric. Each file is read with a buffer. The default is NULL, which does not mean that the file won't be buffered. It means that the internal routine knows if a buffer is needed and will pick the greatest value between the internal suggestion and this value.
- `chunk` — numeric. By default, the collection of files is processed by file (`chunk = NULL` or `chunk = 0`). It is possible to process in arbitrary-sized chunks. This is useful for e.g., processing collections with large files or processing a massive `copc` file.
- `...` — Other internal options not exposed to users.

Details

There are 4 ways to pass processing options, and it is important to understand the precedence rules:

The first option is by explicitly naming each option. This option is deprecated and used for convenience and backward compatibility.

```
exec(pipeline, on = f, progress = TRUE, ncores = 8)
```

The second option is by passing a `list` to the `with` argument. This option is more explicit and should be preferred. The `with` argument takes precedence over the explicit arguments.

```
exec(pipeline, on = f, with = list(progress = TRUE, chunk = 500))
```

The third option is by using a `LAScatalog` from the `lidR` package. A `LAScatalog` already carries some processing options that are respected by the `lasR` package. The options from a `LAScatalog` take precedence.

```
exec(pipeline, on = ctg, ncores = 4)
```

The last option is by setting global processing options. This has global precedence and is mainly intended to provide a way for users to override options if they do not have access to the `exec()` function. This may happen when a developer creates a function that executes a pipeline internally, and users cannot provide any options.

```
set_exec_options(progress = TRUE, ncores = concurrent_files(2))
exec(pipeline, on = f)
```

See Also

multithreading

sort_points — *Sort points in the point cloud*

Description

This stage sorts the points by scanner channel, GPStime, and return number in order to maximize LAZ compression. An optional second sorting step can be added to also sort points spatially. In this case, a grid of 50 meters is applied, and points are sorted by scanner channel, GPSTime, and return number within each cell of the grid. This increases data locality, speeds up spatial queries, but may slightly increases the final size of the files when compressed in LAZ format compared to the optimal compression.

Usage

```
sort_points(spatial = TRUE)
```

Arguments

`spatial` Boolean indicating whether to add a spatial sorting stage.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
exec(sort_points(), on = f)
```

stop_if_outside — *Stop the pipeline if a conditionally*

Description

Stop the pipeline conditionally. The stages after a 'stop_if' stage are skipped if the condition is met. This allows to process a subset of the dataset of to skip some stages conditionally. This DOES NOT stop the computation. In only breaks the pipeline for the current file/chunk currently processed. (see exemple)

Usage

```
stop_if_outside(xmin, ymin, xmax, ymax)
```

Arguments

`xmin, ymin, xmax, ymax`
numeric. bounding box

Examples

```
# Collection of 4 files
f <- system.file("extdata", "bcts/", package="lasR")

# This bounding box encompasses only one of the four files
stopif = stop_if_outside(884800, 620000, 885400, 629200)

read = reader_las()
hll = hulls()
tri = triangulate(filter = keep_ground())
dtm = rasterize(1, tri)

# reads the 4 files but 'tri' and 'dtm' are computed only for one file because stopif
# allows to escape the pipeline outside the bounding box
pipeline = read + hll + stopif + tri + dtm
ans1 <- exec(pipeline, on = f)
plot(ans1$hulls$geom, axes = TRUE)
terra::plot(ans1$rasterize, add = TRUE)

# stopif can be applied before read. Only one file will actually be read and processed
pipeline = stopif + read + hll + tri + dtm
ans2 <- exec(pipeline, on = f)
plot(ans2$hulls$geom, axes = TRUE)
terra::plot(ans1$rasterize, add = TRUE, legend = FALSE)
```

---

summarise *Summary*

Description

Summarize the dataset by counting the number of points, first returns and other metrics for the **entire point cloud**. It also produces an histogram of Z and Intensity attributes for the **entiere point cloud**. It can also compute some metrics **for each file or chunk** with the same metric engine than rasterize. This stage does not modify the point cloud. It produces a summary as a `list`.

Usage

```
summarise(zwbin = 2, iwbin = 50, metrics = NULL, filter = "")
```

Arguments

`zwbin, iwbin`	numeric. Width of the bins for the histograms of Z and Intensity.
`metrics`	Character vector. "min", "max" and "count" are accepted as well as many others (see metric_engine). If `NULL` nothing is computed. If something is provided these metrics are computed for each chunk loaded. A chunk might be a file but may also be a plot (see examples).

`filter` the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
read <- reader_las()
pipeline <- read + summarise()
ans <- exec(pipeline, on = f)
ans

# Compute metrics for each plot
read = reader_las_circles(c(273400, 273500), c(5274450, 5274550), 11.28)
metrics = summarise(metrics = c("z_mean", "z_p95", "i_median", "count"))
pipeline = read + metrics
ans = exec(pipeline, on = f)
ans$metrics
```

`temporary_files` *Temporary files*

Description

Convenient functions to create temporary file with a given extension.

Usage

```
temptif()

tempgpkg()

tempshp()

templas()

templaz()
```

Value

string. Path to a temporary file.

Examples

```
tempshp()
templaz()
```

tools — *Tools inherited from base R*

Description

Tools inherited from base R

Usage

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

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

## S3 method for class 'LASRpipeline'
e1 + e2

## S3 method for class 'LASRpipeline'
c(...)
```

Arguments

`x, e1, e2`	lasR objects
`...`	lasR objects. Is equivalent to +

Examples

```
algo1 <- rasterize(1, "max")
algo2 <- rasterize(4, "min")
print(algo1)
pipeline <- algo1 + algo2
print(pipeline)
```

transform_with — *Transform a point cloud using another stage*

Description

This stage uses another stage that produced a Delaunay triangulation or a raster and performs an operation to modify the point cloud. This can typically be used to build a normalization stage This stage modifies the point cloud in the pipeline but does not produce any output.

Usage

```
transform_with(stage, operator = "-", store_in_attribute = "")
```

Arguments

- `stage` — LASRpipeline. A stage that produces a triangulation or a raster.
- `operator` — string. '-' and '+' are supported.
- `store_in_attribute` — string. Use an extra bytes attribute to store the result.

Value

This stage transforms the point cloud in the pipeline. It consequently returns nothing.

See Also

triangulate write_las

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")

# There is a normalize pipeline in lasR but let's create one almost equivalent
mesh  <- triangulate(filter = keep_ground())
trans <- transform_with(mesh)
pipeline <- mesh + trans + write_las()
ans <- exec(pipeline, on = f)
```

`triangulate` *Delaunay triangulation*

Description

Delaunay triangulation. Can be used to build a DTM, a CHM, normalize a point cloud, or any other application. This stage is typically used as an intermediate process without an output file. This stage does not modify the point cloud.

Usage

```
triangulate(max_edge = 0, filter = "", ofile = "", use_attribute = "Z")
```

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. If max_edge = 0, no trimming is done (see examples).
- `filter` — the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

`ofile` character. Full outputs are always stored on disk. If `ofile = ""` then the stage will not store the result on disk and will return nothing. It will however hold partial output results temporarily in memory. This is useful for stage that are only intermediate stage.

`use_attribute` character. By default the triangulation is performed on the coordinate Z. Can also be the name of an extra bytes attribute such as 'HAG' if it exists. Can also be 'Intensity'.

Value

This stage produces a vector. The path provided to 'ofile' is expected to be '.gpkg' or any other format supported by GDAL. Vector stages may produce geometries with Z coordinates. Thus, it is discouraged to store them in formats with no 3D support, such as shapefiles.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
read <- reader_las()
tri1 <- triangulate(25, filter = keep_ground(), ofile = tempgpkg())
filter <- "-keep_last -keep_random_fraction 0.1"
tri2 <- triangulate(filter = filter, ofile = tempgpkg())
pipeline <- read + tri1 + tri2
ans <- exec(pipeline, on = f)
#plot(ans[[1]])
#plot(ans[[2]])
```

`write_las` *Write LAS or LAZ files*

Description

Write a LAS or LAZ file at any step of the pipeline (typically at the end). Unlike other stages, the output won't be written into a single large file but in multiple tiled files corresponding to the original collection of files.

Usage

```
write_las(
  ofile = paste0(tempdir(), "/*.las"),
  filter = "",
  keep_buffer = FALSE
)
```

Arguments

ofile — character. Output file names. The string must contain a wildcard * so the wildcard can be replaced by the name of the original tile and preserve the tiling pattern. If the wildcard is omitted, everything will be written into a single file. This may be the desired behavior in some circumstances, e.g., to merge some files.

filter — the 'filter' argument allows filtering of the point-cloud to work with points of interest. The available filters are those from LASlib and can be found by running filter_usage. For a given stage when a filter is applied, only the points that meet the criteria are processed. The most common strings are "-keep_first", "-keep_class 2", "drop_z_below 2". For more details see filters.

keep_buffer — bool. The buffer is removed to write file but it can be preserved.

Examples

```
f <- system.file("extdata", "Topography.las", package="lasR")
read <- reader_las()
tri  <- triangulate(filter = keep_ground())
normalize <- tri + transform_with(tri)
pipeline <- read + normalize + write_las(paste0(tempdir(), "/*_norm.las"))
exec(pipeline, on = f)
```

---

write_lax — *Write spatial indexing .lax files*

Description

Creates a .lax file for each `.las` or `.laz` file of the processed datase. A .lax file contains spatial indexing information. Spatial indexing drastically speeds up tile buffering and spatial queries. In lasR, it is mandatory to have spatially indexed point clouds, either using .lax files or .copc.laz files. If the processed file collection is not spatially indexed, a `write_lax()` file will automatically be added at the beginning of the pipeline (see Details).

Usage

```
write_lax(embedded = FALSE, overwrite = FALSE)
```

Arguments

embedded — boolean. A .lax file is an auxiliary file that accompanies its corresponding las or laz file. A .lax file can also be embedded within a laz file to produce a single file.

overwrite — boolean. This stage does not create a new spatial index if the corresponding point cloud already has a spatial index. If TRUE, it forces the creation of a new one. `copc.laz` files are never reindexed with `lax` files.

Details

When this stage is added automatically by `lasR`, it is placed at the beginning of the pipeline, and las/laz files are indexed **on-the-fly** when they are used. The advantage is that users do not need to do anything; it works transparently and does not delay the processing. The drawback is that, under this condition, the stage cannot be run in parallel. When this stage is explicitly added by the users, it can be placed anywhere in the pipeline but will always be executed first before anything else. All the files will be indexed first in parallel, and then the actual processing will start. To avoid overthinking about how it works, it is best and simpler to run `exec(write_lax(), on = files)` on the non indexed point cloud before doing anything with the point cloud.

Examples

```
## Not run:
exec(write_lax(), on = files)

## End(Not run)
```

---

write_vpc — *Write a Virtual Point Cloud*

Description

Borrowing the concept of virtual rasters from GDAL, the VPC file format references other point cloud files in virtual point cloud (VPC)

Usage

```
write_vpc(ofile, absolute_path = FALSE, use_gpstime = FALSE)
```

Arguments

`ofile`	character. The file path with extension .vpc where to write the virtual point cloud file
`absolute_path`	boolean. The absolute path to the files is stored in the tile index file.
`use_gpstime`	logical. To fill the datetime attribute in the VPC file, it uses the year and day of year recorded in the header. These attributes are usually NOT relevant. They are often zeroed and the official signification of these attributes corresponds to the creation of the LAS file. There is no guarantee that this date corresponds to the acquisition date. If `use_gpstime = TRUE`, it will use the gpstime of **the first point** recorded in each file to compute the day and year of acquisition. This works only if the GPS time is recorded as Adjusted Standard GPS Time and not with GPS Week Time.

References

https://www.lutraconsulting.co.uk/blog/2023/06/08/virtual-point-clouds/
https://github.com/PDAL/wrench/blob/main/vpc-spec.md

Examples

```
## Not run:
pipeline = write_vpc("folder/dataset.vpc")
exec(pipeline, on = "folder")

## End(Not run)
```

Index