

Package: arbor (via r-universe)

October 13, 2026

Type Package

Title Tree Segmentation in MLS/TLS Point Clouds

Version 1.1.2

Author Jean-Romain Roussel

Maintainer Jean-Romain Roussel <info@r-lidar.com>

Description Semantic segmentation, instance segmentaion and
Quantitative Structure Model in MLS and TLS Point Clouds.

License GPL (>= 3)

URL <https://github.com/r-lidar/arbor>

BugReports <https://github.com/r-lidar/arbor/issues>

Encoding UTF-8

LazyData true

LinkingTo Rcpp

Depends R (>= 4.1.0)

Imports data.table, graphics, lidR (>= 4.2.2), methods, Rcpp, rgl, sf,
utils

Suggests testthat (>= 3.0.0), terra

Config/roxygen2/version 8.1.0

Config/build/compilation-database true

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

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

Date/Publication 2026-10-13 23:20:27 UTC

RemoteUrl <https://github.com/r-lidar/arbor>

RemoteRef HEAD

RemoteSha c714348e61950ad2e4172ac662c8ffbfefd6639b

Contents

[.qsf	3
%in%,LAS-method	3
add_single_tree_ground	4
ARBORTREE	4
available_allometries	5
extract_tree_context	5
find_seeds	6
flagging	7
foliage.colors	8
hybrid_homogeneization	8
install_cmd_tools	8
parameters	9
plot.qsf	9
print.qsf	10
print.qsm	11
qsf	11
qsf_filter	12
qsf_log	15
qsf_read	16
qsf_segment_semantic	17
qsf_treemap	17
qsf_write	18
qsm	20
qsm_dbh	21
qsm_read	22
qsm_stats	22
qsm_subset	23
qsm_volume	24
qsm_write	25
resolve_oversegmentation	26
segment_ground	26
segment_instance	27
segment_semantic	28
st_crs	28
trajectory_range	29
transfer_attributes	30
wood_likelihood	31

Index

32

[.qsf	<i>Subset a qsf object</i>
-------	----------------------------

Description

Subsets a ‘qsf’ object while preserving its class.

Usage

```
## S3 method for class 'qsf'
x[i, ...]
```

Arguments

- x A ‘qsf’ object.
- i Index specifying the elements to extract.
- ... Additional arguments passed to ‘[’.

Value

A ‘qsf’ object containing the selected QSMs.

%in%,LAS-method	<i>Value Matching for qsf Objects</i>
-----------------	---------------------------------------

Description

Determines if elements of a qsf object exist within a las object.

Usage

```
## S4 method for signature 'LAS'
x %in% table
```

Arguments

- x An object of class LAS.
- table An object of class qsf.

Value

A logical vector indicating matching elements.

`add_single_tree_ground`

*Adds a set of synthetic ground points below a single-tree point cloud.
Useful to perform arbor operations on already externally isolated trees*

Description

Adds a set of synthetic ground points below a single-tree point cloud. Useful to perform arbor operations on already externally isolated trees

Usage

```
add_single_tree_ground(las, n = 1000)
```

Arguments

<code>las</code>	A LAS object containing a single-tree point cloud.
<code>n</code>	integer. Number of ground points

Value

A LAS object containing the original points plus `n` synthetic ground points.

ARBORTREE

Constants

Description

Built-in variables. See the [Arbor book](#) for more details.

Usage

ARBORTREE

ARBORLOW

ARBORUNDERSTORY

ARBORBUFFER

Details

- ARBORTREE is used in the UserData attributes. It flags points that are valid trees. Usually, all points of interest are flagged as ARBORTREE.
- ARBORLOW is used in the UserData attributes. It flags low points that are not processed by Arbor at all.
- ARBORUNDERSTORY is used in the UserData attributes. It flags small trees that are not processed to build QSMs. See [flag_small_trees](#).
- ARBORBUFFER is used in the UserData attributes. It flags points in the buffer area. See [flag_buffer](#).

available_allometries *Available allometric models*

Description

Show available allometric models

Usage

```
available_allometries()
```

Value

A data.frame containing available models and their reference URLs:

model Model name

url DOI or reference link

Examples

```
available_allometries()
```

extract_tree_context *Extract the local tree context from a LAS object*

Description

Extracts points belonging to trees surrounding a target tree, based on different spatial context definitions (contact, crown extent, or root zone). This is useful for analysing neighbourhood interactions, competition, or structural context around individual trees.

Usage

```
extract_tree_context(las, tree, exclude_tree = FALSE)
```

Arguments

las	A LAS object (from lidR) containing a segmented forest point cloud with a treeID attribute.
tree	Either a numeric tree ID corresponding to treeID in las, or a LAS object containing points from a single tree.
exclude_tree	Logical. If TRUE, the target tree itself is removed from the returned context.

Value

A LAS object containing points from neighbouring trees that form the selected context.

See Also

Other experimental: [qsf_segment_semantic\(\)](#)

Examples

```
## Not run:
las <- readLAS("segmented_forest.las")

tree <- lidR::filter_poi(las, treeID == 10)
context <- extract_tree_context(las, tree, exclude_tree = TRUE)
x <- plot_instance(context)
plot(tree, add = x, pal = "red")

## End(Not run)
```

find_seeds	<i>Find seeds to perform instance segmentation</i>
------------	--

Description

In order to perform instance segmentation with [segment_instance](#) we need some seeds with reference treeIDs. Thus function finds the seeds. See the [Arbor book](#) for mode details.

Usage

```
find_seeds(las, params = arbor_parameters_default)
```

Arguments

las	A LAS object from lidR.
params	list See parameters .

Description

Utilities to flag trees that should be excluded from downstream processing. These functions assign values to the UserData attribute of the point cloud in order to later filter data on the fly. See the [Arbor book](#)

Usage

```
flag_small_trees(las, max_height = 2)
```

```
flag_buffer(las, seeds, buffer = -5)
```

Arguments

las	A LAS object from lidR containing segmented trees.
max_height	Numeric. Trees with a height lower than this threshold (in meters) are flagged.
seeds	A LAS object containing tree seeds, typically generated with find_seeds . If missing, an internal routine estimates seed positions from the lowest point of each tree.
buffer	Numeric value (in meters) used to shrink the convex hull before filtering trees. Default is -5, which excludes trees within 5 meters of the boundary. Can also be an sf polygon object for custom clipping.

Details

`flag_small_trees()` flags trees below a given height threshold, which are often poorly segmented or correspond to understory vegetation.

`flag_buffer()` flags trees located near the edges of a point cloud by clipping trees whose seed points fall outside a buffered polygon.

See the [Arbor book](#) for more details.

Value

A modified LAS object with flagged trees assigned negative treeID values.

See Also

[qsf](#), [find_seeds](#)

foliage.colors	<i>Color palette for semantic segmentation plot</i>
----------------	---

Description

Color palette for semantic segmentation plot

Usage

```
foliage.colors
```

hybrid_homogeneization	<i>Homogenization of the Point Cloud</i>
------------------------	--

Description

Homogenization of the point cloud using a hybrid approach that includes Barycentric Voxel Decimation and reinjection of random points. See the [Arbor book](#) for more details.

Usage

```
hybrid_homogeneization(las, res = 0.02)
```

Arguments

las	LAS object from lidR
res	Voxel resolution.

install_cmd_tools	<i>Use some arbor features from a terminal</i>
-------------------	--

Description

Install the required files to be able to run some simple arbor commands from a terminal.

Usage

```
install_cmd_tools()
```

parameters

*Parameters***Description**

The built-in internal variable ‘arbor_parameters_default’ contains all the parameters used by the arbor functions. The variable itself is a nested list that is intentionally not documented. There are numerous parameters controlling thresholds, neighborhoods, clustering, noise filtering, and pathfinder configuration. If users had to adjust and understand all of them, the software would quickly become unusable. While every parameter can be configured, we deliberately avoid documenting them to prevent endless tweaking. The pipeline MUST work reliably out of the box with the default parameters. See the [Arbor book](https://r-lidar.github.io/arbor_book/)

Usage

```
arbor_parameters_default
```

Examples

```
params = arbor_parameters_default
params$global$cut_above_ground <- 0.25
```

plot.qsf

*Plot QSM Data in 3D***Description**

Plots the QSM data as a series of connected cylinders, with color-coding based on branch attributes. ‘colorize_trees’ assigns a RGB value per tree for rendering in third party softwares.

Usage

```
## S3 method for class 'qsf'
plot(x, ..., color = "branch_order", pal = "auto", add = NULL)

## S3 method for class 'qsm'
plot(x, ...)

plot_qsm(
  x,
  add = NULL,
  color = "branch_order",
  skeleton = TRUE,
  cylinder = TRUE,
  pal = "auto",
```

```

    ...
  )

  plot_likelihood(las, dtm = NULL, ...)

  plot_semantic(las, dtm = NULL, ...)

  plot_instance(las, dtm = NULL, ...)

  plot_semantic_instance(las, dtm = NULL, ...)

  plot_passage(las, dtm = NULL, th = 0, ...)

  colorize_trees(las, darken_foliage = TRUE)

```

Arguments

x	A QSM of QSF
...	propagated to lidR::plot
color	The attribute for color mapping.
pal	Color palette
add	A numeric vector for translation offsets. Like in the lidR package.
skeleton, cylinder	boolean. plot the skeleton or the cylinders or both.
las	LAS object
dtm	SpatRaster DTM
th	threshold to remove points with few passages
darken_foliage	boolean. In addition to one color per tree, the foliage is darkened the same color that its corresponding tree to display semantic and instance segmentation in one rendering.
qsm	A QSM

print.qsf

Print a Quantitative Structure Forest (QSF)

Description

Displays a short summary of a Quantitative Structure Forest followed by the standard data.table print output.

Usage

```

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

```

Arguments

x A qsm object.
 ... Unused (for S3 compatibility).

Value

Invisibly returns x.

print.qsm	<i>Print a Quantitative Structure Model (QSM)</i>
-----------	---

Description

Displays a short summary of a Quantitative Structure Model followed by the standard `data.table` print output.

Usage

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

Arguments

x A qsm object.
 ... Unused (for S3 compatibility).

Value

Invisibly returns x.

qsf	<i>Quantitative Structural Forest</i>
-----	---------------------------------------

Description

Batch processing of QSM models with parallel execution. Trees that are flagged with `flag_buffer` or `flag_small_trees` are excluded from the computation. See also the [Arbor book](#) for more details.

Usage

```
qsf(las, min_height = 2, params = arbor_parameters_default)
```

Arguments

las	A point cloud with semantic and instance segmentation computed.
min_height	numeric. Default: any instance higher than 2 m generates a QSM. This is pretty equivalent to flag_small_trees , but forces a height limit even if the user did not call flag_small_trees , thus ensuring that no QSM lower than 2 m is computed.
params	list. See parameters .

Value

A qsf object.

See Also

[qsm](#)

qsf_filter	<i>Filter a qsf object</i>
------------	----------------------------

Description

`qsf_filter()` subsets a qsf object, retaining only the trees that satisfy the requested criteria. Filters compose with AND semantics: a tree is retained only if it satisfies every criterion supplied.

A handful of `qsf_filter_*`() wrappers cover the most common triage needs on top of `qsf_filter()`: pulling out clean ("ok") trees, trees carrying any warning, error or message, or (with `qsf_filter_flagged()`) anything with a log entry at all. Each one still takes `...`, so it composes with the other `qsf_filter()` criteria (`dbh`, `height`, ...) in a single call. See details section for more in depth details

Usage

```
qsf_filter(
  qsf,
  code = NULL,
  type = NULL,
  invert = FALSE,
  flag = NULL,
  dbh = NULL,
  height = NULL,
  ...
)

qsf_filter_ok(qsf, ...)

qsf_filter_sapling(qsf, ...)

qsf_filter_nomeasure(qsf, ...)
```

```

qsf_filter_broken(qsf, ...)

qsf_filter_flagged(qsf, code = NULL, type = NULL, ...)

qsf_filter_warnings(qsf, code = NULL, ...)

qsf_filter_errors(qsf, code = NULL, ...)

qsf_filter_messages(qsf, code = NULL, ...)

```

Arguments

qsf	A qsf object.
code	character. One or more qsf_log codes (e.g. <code>c("W0", "E1")</code>). By default, trees carrying any of these codes are kept . Use <code>invert = TRUE</code> to instead exclude trees carrying these codes. In <code>qsf_filter_ok()</code> , <code>qsf_filter_warnings()</code> , <code>qsf_filter_errors()</code> and <code>qsf_filter_messages()</code> , code narrows the match to specific codes within that function's type; left <code>NULL</code> (default) it matches any code of that type.
type	character. One or more log types: "warning", "message", "error". Same semantics as code, and combined with code (if both given, an entry must match both to be considered flagged). Only exposed on <code>qsf_filter()</code> and <code>qsf_filter_flagged()</code> ; the other wrappers fix type for you.
invert	logical. If <code>FALSE</code> (default), trees matching code/type are kept. If <code>TRUE</code> , matching trees are dropped. Only exposed on <code>qsf_filter()</code> .
flag	logical. Filters on the mere <i>presence</i> of a log entry, regardless of its code or type (independent of code/type, combined with them by AND like every other filter here). <code>FALSE</code> keeps only trees with no log entry at all (clean/"good" QSMs). <code>TRUE</code> keeps only trees with at least one log entry, of any code or type. Default <code>NULL</code> : no filtering on this criterion. This is what you want when you need "everything with nothing flagged" without having to enumerate every possible type. Only exposed on <code>qsf_filter()</code> .
dbh	numeric(2). Keep only trees with a DBH (see qsm_dbh) inside <code>c(min, max)</code> , in meters.
height	numeric(2). Keep only trees with a height (see qsm_height) inside <code>c(min, max)</code> , in meters.
...	In <code>qsf_filter()</code> : unused, reserved for additional filters in the future. In <code>qsf_filter_*</code> () wrappers: passed on to <code>qsf_filter()</code> (e.g. <code>dbh = c(0.2, Inf)</code> , <code>height = c(0, 30)</code>), so a wrapper can be chained with the other filters in a single call.

Details

- `qsf_filter_ok()` retains QSMs with no log entry at all. They are assumed to be good QSMs.
- `qsf_filter_sapling()` retains QSMs for which no measurements were performed according to the height allometry rules (see [Arbor Book](#)). Under Arbor's definition, they are considered saplings.

- `qsf_filter_nomeasure()` retains QSMs for which no measurements were found because the QSM algorithm failed to measure anything. However, these trees are not saplings. Saplings are different. For sapling the algorithm does not even try, on purpose.
- `qsf_filter_broken()` retains QSMs for which Arbor automatically detected that they correspond to broken trees.
- `qsf_filter_flagged()` retains QSMs with at least one log entry, of any kind.
- `qsf_filter_warnings()` retains QSMs carrying at least one warning.
- `qsf_filter_errors()` retains QSMs carrying at least one error.
- `qsf_filter_messages()` retains QSMs carrying at least one plain message.

Value

A `qsf` object containing only the selected QSMs.

See Also

[qsf_log](#)

Examples

```
f <- system.file("extdata", "oak-plantation.qsf", package="arbor")
qsf <- qsf_read(f)

# qsf_filter_*(): quick selectors
qsf_filter_ok(qsf)           # trees with no log entry at all
qsf_filter_sapling(qsf)      # saplings according to arbor's definition
qsf_filter_flagged(qsf)      # trees with at least one log entry, any kind
qsf_filter_warnings(qsf)     # trees carrying at least one warning
qsf_filter_warnings(qsf, "W2") # ... specifically warning code W2
qsf_filter_messages(qsf)     # trees carrying at least one plain message

# qsf_filter(): the general-purpose filter
# Keep only trees flagged with code W0 or W3
qsf_filter(qsf, code = c("W0", "W3"))

# Exclude trees flagged with code W2 or W3, while keeping trees with DBH > 20 cm
qsf_filter(qsf, code = c("W2", "W3"), invert = TRUE, dbh = c(0.2, Inf))

# Trees with no log entry at all ("good" QSMs)
qsf_filter(qsf, flag = FALSE)

# Trees with at least one log entry, of any kind
qsf_filter(qsf, flag = TRUE)
```

qsf_log

*QSF log***Description**

Use `qsf_log` after [qsf](#) to get a structured summary of the log messages recorded during the QSM reconstruction of every tree in a 'qsf' object.

Usage

```
qsf_log(qsf)
```

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

Arguments

<code>qsf</code>	<code>qsf</code>
<code>x</code>	A 'qsf_log' object, as returned by qsf_log .
<code>...</code>	Unused (for S3 compatibility).

Value

A named list, one element per distinct log entry. Elements are named by their 'code' (e.g. "W0"). Each element is a list with:

- 'code': the log code (e.g. "W0"), or 'NA' for legacy, uncoded messages.
- 'type': "warning", "message", "error", or 'NA' if unknown.
- 'label': the short, human readable label extracted from the message.
- 'message': an example of the full message text.
- 'index': the indices, within 'qsf', of the trees carrying this log entry.
- 'treeID': the tree IDs of the trees carrying this log entry.

Codes

Messages recorded on a QSM start with a code '[<L><n>]', where '<L>' is one of 'W' (warning), 'M' (message) or 'E' (error) and '<n>' is a numeric identifier, immediately followed by '[Short label]' and the free text explanation, e.g. "[W0] [No wood point] This tree has no point labelled as wood". This is fully backward compatible: messages recorded before this convention was introduced have no code ('code = NA') and are grouped by their short label instead, exactly as before.

See Also

[qsf_filter_qsm_message](#)

Examples

```
f <- system.file("extdata", "oak-plantation.qsf", package="arbor")
qsf <- qsf_read(f)
logs <- qsf_log(qsf)
print(logs)
logs$W2
logs$W2$treeID
logs$W3
```

qsf_read

Read a QSF Manifest File

Description

Reads a .qsf manifest file produced by [qsf_write](#) (whenever "qsm" is among the requested formats) and reconstructs the Quantitative Structural Forest (QSF) it describes.

Usage

```
qsf_read(path)
```

Arguments

path A string giving the path to the .qsf manifest file.

Details

A .qsf file is a lightweight, standalone manifest that indexes a collection of .qsm files by relative path - conceptually similar to a virtual raster mosaic (VRT). Currently only "virtual" manifests are supported: the manifest itself holds no tree data, only relative references to the .qsm files sitting next to it, which are read in turn.

Value

A qsf object, i.e. a list of qsm objects.

See Also

[qsf_write](#)

Examples

```
f <- system.file("extdata", "oak-plantation.qsf", package="arbor")
oak <- qsf_read(f)
## Not run:
plot(oak)

## End(Not run)
```

qsf_segment_semantic	<i>Semantic segmentation of a tree point cloud using a QSM</i>
----------------------	--

Description

Performs an experimental semantic segmentation of a single-tree point cloud using distances derived from a Quantitative Structure Model (QSM). Points are labelled as foliage or non-foliage based on their proximity to QSM elements.

Usage

```
qsf_segment_semantic(las, qsf)
```

Arguments

las	A LAS object containing points from a single tree
qsf	A Quantitative Structure Forest associated with las.

See Also

Other experimental: [extract_tree_context\(\)](#)

qsf_treemap	<i>Convert a Quantitative Structure Forest (QSF) to an sf treemap</i>
-------------	---

Description

Extracts global statistics from each Quantitative Structure Model (QSM) contained in a Quantitative Structure Forest (QSF) object and returns them as an **sf** point object. Each point corresponds to the root position of an individual tree.

Usage

```
qsf_treemap(qsf)
```

Arguments

qsf	A Quantitative Structure Forest (QSF)
-----	---------------------------------------

Value

An **sf** object with one feature per QSM. Attributes correspond to global statistics returned by [qsm_stats](#), and geometry is defined by the root coordinates (X_root, Y_root, Z_root).

qsf_write

*Write a QSF to Files***Description**

Writes a Quantitative Structure Forest (QSF), i.e. a collection of QSMs to files in QSM, OBJ, PLY, STL, CSV, or TXT format.

Usage

```
qsf_write(qsf, path, format = c("qsm", "obj"), binary = TRUE)
```

Arguments

qsf	A QSF object to be written.
path	A string giving the output directory path (directory mode), or a file path ending in .obj, .ply, or .stl for single-file mode (see Details).
format	A character vector specifying the export format(s) (e.g., "qsm", "obj", "ply", "stl", "csv", "txt"). Defaults to c("qsm", "obj"). Ignored when path specifies a single-file output.
binary	Logical. Indicates whether to output binary or ASCII where supported. Defaults to TRUE. For stl in particular this also controls whether trees stay distinguishable in single-file mode - see Details.

Details

The function operates in two distinct modes:

- **Directory mode (recommended):** If path is a directory (or has no file extension), each QSM in the forest is written into its own separate file inside path/format/, named after the QSM's name (or its id if it has no name). Multiple formats can be exported at once by passing several values in format. Every tree is always fully separate and identifiable in this mode, for every format, because each tree gets its own file.
- **Single-file mode:** If path points to a specific file path (i.e., its extension is non-empty), a single file is written containing all QSMs together. The file format is inferred strictly from the file extension of path, and the format argument is ignored. This is only supported for the obj, ply, and stl mesh formats. qsm, csv, and txt do not support combining several trees into one file and will raise an error if used this way. Even among the supported formats, how well individual trees stay distinguishable inside the combined file varies a lot (see "Limits of single-file mode" below), so this mode is best reserved for cases where you specifically need one file to hand to some other piece of software (e.g. importing a whole forest into Blender in one go).

Supported formats:

- .qsm: Native binary format. Directory mode only. Also produces a sibling .qsf manifest file that indexes every .qsm file written, by relative path (see [qsf_read](#)).

- .obj, .ply, or .stl: Mesh formats. Available in both directory and single-file mode.
- .csv or .txt: ASCII table formats. Directory mode only.

Value

Returns TRUE invisibly upon completion.

Limits of single-file mode

When several QSMs are combined into one obj, ply, or stl file, each format has a different (and sometimes very limited) way of keeping individual trees identifiable:

- **OBJ:** Full support. Each QSM becomes its own named object inside the file. This is NOT respected by all OBJ readers. E.g. Blender respects the OBJ specifications and load individual trees but not CloudCompare that loads one massive undifferentiated mesh.
- **PLY:** Partial support. All QSMs share one vertex/face list (a single mesh), but every face carries a qsm_id integer property, and the header records one comment per QSM with its id and name. Software that lets you inspect or filter by custom face properties can recover the original trees; a generic PLY viewer will just show one undifferentiated mesh.
- **STL (ASCII, binary = FALSE):** Weak, non-standard support. Each QSM is written as its own solid <name> ... endsolid <name> block. This is a common convention, not part of the STL standard, so support is inconsistent: some tools import every block as a separate named object, others only read the first block, and others merge everything into one mesh. Always test with your target software before relying on this.
- **STL (binary, binary = TRUE, the default):** No support at all. Binary STL has no concept of an object name or grouping - it is just a flat, unlabelled list of triangles. Combining multiple QSMs into one binary STL file merges them into a single anonymous mesh with no way to tell which triangle came from which tree. Use binary = FALSE (ASCII STL), or switch to obj or ply, if keeping trees separate matters.

See Also

[qsf_read](#)

Examples

```
## Not run:
# Single-file mode: maybe the simplest use case. Writes forest.qsf file that references
# the .qsm files in the subdirectory "qsm/". In this case it does produces numerous
# files because .qsf file format only contains metadata.
qsf_write(qsf, "forest.qsf")

# Directory mode: one file per tree.
# Writes forest/qsm/<name>.qsm and forest/obj/<name>.obj for every tree.
# Writes also path/to/directory/directory.qsf to reference all the .qsm file
qsf_write(qsf, "path/to/directory/", format = c("qsm", "obj"))

# Directory mode: several formats at once, ASCII where applicable.
qsf_write(qsf, "forest", format = c("ply", "csv"), binary = FALSE)
```

```
# Single-file mode: with OBJ format is the safest choice. Write a single OBJ file. Every
# tree keeps its name as a separate 'o' object. Be careful that NOT all software will
# import as such. Blender supports it not CloudCompare.
qsf_write(qsf, "forest_all_trees.obj")

# Single-file mode: with PLY format. One mesh, but trees can still be told apart via the
# per-face 'qsm_id' property if your software can filter on it.
qsf_write(qsf, "forest_all_trees.ply")

## End(Not run)
```

qsm

Generate a QSM from a single tree point cloud

Description

This function processes a tree point cloud to generate a Quantitative Structure Model (QSM) See the [Arbor book](#) for mode details.

Usage

```
qsm(tree, params = arbor_parameters_default)
```

Arguments

tree	A LAS object from lidR containing a single tree point cloud. Only the point labelled as wood will be used for QSM.
params	list See parameters .

See Also

[qsm_write](#) [qsm_read](#) [qsm_dbh](#) [qsm_stats](#)

Examples

```
f <- system.file("extdata", "tree_qsm.laz", package="arbor")
tree <- lidR::readLAS(f)
qsm = qsm(tree)
## Not run:
x = plot_semantic(tree)
plot_qsm(qsm, add = x, color = "branch_order", cylinder = TRUE)

## End(Not run)
```

qsm_dbh*Extract diameter at breast height (DBH) from a QSM*

Description

Extract the diameter at breast height (DBH) from a Quantitative Structure Model (QSM).

Usage

```
qsm_dbh(qs, bh = 1.3)

add_dbh3d(x, df, col = "red", lwd = 3)
```

Arguments

qs	A QSM or QSF
bh	Numeric. Breast height (in meters). Default is 1.30.
x	An alignment offset returned by a ‘plot’ function.
df	the output of ‘qsm_dbh()’
col, lwd	render parameters

Value

A data.frame with:

- dbh: DBH in meter
- pos: xyz position of the center
- normal: normal vector of the circle

Examples

```
f <- system.file("extdata", "tree_qsm.laz", package="arbor")
tree <- lidR::readLAS(f)
qsm <- qsm(tree)
ans <- qsm_dbh(qsm)
## Not run:
plot_semantic(tree) |> add_dbh3d(ans)
plot_qsm(qsm) |> add_dbh3d(ans)
## End(Not run)
```

qsm_read	<i>Read QSM Data from File</i>
----------	--------------------------------

Description

Loads QSM segment data from a CSV or QSM files.

Usage

qsm_read(x)

Arguments

x A character string specifying the file path.

Value

A data frame of QSM segment data.

qsm_stats	<i>Compute summary statistics from a QSM</i>
-----------	--

Description

Computes structural statistics from a Quantitative Structure Model (QSM), including volume distributions by radius and branch order, stem taper, and global tree metrics such as DBH, height, and total volume. Optionally, statistics can be complemented with information derived from a LAS point cloud of the tree and visual diagnostics can be displayed.

Usage

qsm_stats(qs, ..., display = FALSE)

Arguments

qs A QSM or QSF produced by [qsm](#) or [qsf](#). If 'qs' is a 'QSF' the output is less comprehensive to fit in a 'data.frame'

... passed to [qsm_dbh](#).

display Logical. If 'TRUE', diagnostic plots are displayed, including volume distributions, stem profile, and vertical projections.

Details

The function computes:

- Volume and cylinder counts by radius class (2 cm bins).
- Volume distribution and cumulative percentage by branch order.
- Stem taper profile from the main axis.
- Global tree statistics such as DBH, total volume, and height.

When a ‘LAS’ object is supplied, metrics are derived directly from the point cloud and QSM rather than the QSM alone.

Value

A named list with the following elements:

- stats_by_order: A ‘data.table’ with volume and cumulative percentage by branch order.
- stats_by_radius: A ‘data.table’ with total volume and cylinder count by radius class.
- stats_global: A ‘data.frame’ containing global tree metrics (coordinates, DBH, volume, height).
- stem_taper: A ‘data.frame’ describing stem diameter as a function of distance to root.

See Also

[qsm_dbh](#), [qsm](#)

Examples

```
f <- system.file("extdata", "tree_qsm.laz", package="arbor")
tree <- lidR::readLAS(f)
qsm <- qsm(tree)
ans <- qsm_stats(qsm, display = TRUE)
```

qsm_subset

Subset a QSM or QSF

Description

Subset a QSM or QSF retaining only the stem, the merchantable or other options.

qsm_stem() retains only the stem of the QSM (branching_order = 1). Applying the function to a QSF is like a loop on each QSM.

qsm_merchantable() retains only the merchantable parts of the QSM (radius > merchantable_radius with additional graph validity constraints). Applying the function to a QSF is like a loop on each QSM.

qsf_merchantable() retains only the merchantable trees of the QSF (DBH > merchantable_radius). It is not equivalent to qsm_merchantable() as it does not subset the QSMs, it subsets the QSF. The entire trees are retains.

Usage

```

qsm_stem(qs)

qsm_merchantable(qs, merchantable_radius = 0.045, merchantable_length = 1)

qsf_merchantable(qsf, merchantable_radius = 0.045)

qsm_nostump(qs, stump_height = 0.15)

```

Arguments

qs	A QSM or QSF
merchantable_radius	The radius considered merchantable. Default 4.5 cm (9 cm diameter) which is the standard in Canada.
merchantable_length	The length considered merchantable. Default 1 m which is the standard in Canada.
qsf	A QSF
stump_height	The height to remove from the root. Default 15 cm

qsm_volume

Simple tools for QSM

Description

Simple tools for QSM.

Usage

```

qsm_volume(qs)

qsm_height(qs)

qsm_message(qsm, short = FALSE)

qsm_treeid(qs)

## S3 method for class 'qsm'
qsm_treeid(qs)

## S3 method for class 'qsf'
qsm_treeid(qs)

```

Arguments

qs	A QSM or QSF
qsm	A QSM
short	bool. Shorter message: only the human readable label is returned (the optional leading '[<code>]' tag, see qsf_log , is stripped as well).

`qsm_write`*Write a QSM to File*

Description

Writes a Quantitative Structure Model (QSM) to a file in either OBJ, PLY or CSV format, based on the file extension.

Usage

```
qsm_write(qsm, file, binary = TRUE)
```

Arguments

qsm	A QSM object to be written.
file	A string giving the path to the output file. The file extension determines the format (e.g., ".qsm", ".ply", ".obj", ".csv", ".txt", ".stl").
binary	Boolean. Used if the format supports ASCII or binary

Details

Supported formats:

- .qsm: writes the QSM as native binary format
- .ply or .obj or .stl: writes the QSM as a mesh file
- .csv or .txt: writes the QSM as ASCII table

Examples

```
## Not run:  
qsm_write(qsm, "tree.qsm")  
qsm_write(qsm, "tree.obj")  
qsm_write(qsm, "tree.csv")  
  
## End(Not run)
```

 resolve_oversegmentation

Resolve over-segmentation in a tree instance segmentation

Description

Detects and merges tree instances that were split into multiple treeIDs during segmentation (over-segmentation) and merges them back into single, consistent instances. This module is already executed by [segment_instance](#).

Over-segmentation is detected by scanning the point cloud layer by layer across a range of height-above-ground (HAG) values. Within each layer, points are clustered (DBSCAN) and a circular stem cross-section is fitted to each cluster. When a well-fitted circle's inlier points are supported by more than one treeID, this is evidence that two (or more) instances actually belong to the same stem. Evidence is accumulated across all layers into a matching table of (reference, candidate) treeID pairs, each backed by a count of how many circles support the pair and how many inlier points came from each side. Chains of matches (e.g. 1-2, 2-3) are resolved with a union-find so that every id in a chain collapses onto a single canonical id.

Usage

```
resolve_oversegmentation(las)
```

Arguments

las	A LAS object. Must already carry a treeID attribute (from a prior instance segmentation) and a HAG (height above ground) attribute, since detection only operates within a configured HAG range.
-----	--

Value

The input LAS object, with treeID values remapped wherever over-segmentation was detected and merged. Instances that were not found to be over-segmented are left untouched.

See Also

[segment_instance](#)

 segment_ground

Ground Segmentation

Description

Classifies ground points and computes height above ground. See the [Arbor book](#) for mode details.

Usage

```
segment_ground(las, params = arbor_parameters_default)
```

Arguments

las	A LAS object (from the lidR package) containing the point cloud data
params	list See parameters .

See Also

[find_seeds](#), [segment_semantic](#), [segment_instance](#)

segment_instance	<i>Individual Tree Instance Segmentation</i>
------------------	--

Description

This function performs individual tree instance segmentation by constructing a point-based network and computing the least-cost path for each point to predefined seed points. The goal is to assign each point in the point cloud a treeID corresponding to its nearest seed, based on the least-cost path computed through the network. The process incorporates both spatial and heuristic constraints to segment trees effectively. See the [Arbor book](#) for more details.

Usage

```
segment_instance(las, seeds, params = arbor_parameters_default)
```

Arguments

las	A LAS object (from the lidR package) containing the point cloud data. It must have an attribute foliage, which can be generated using prior semantic segmentation.
seeds	An sf object containing points that represent tree seeds. Each point must have xyz coordinates and a treeID. Seeds can be generated using find_seeds .
params	list See parameters .

Value

A LAS object with an additional attribute treeID, indicating the ID of the tree to which each point belongs.

See Also

[find_seeds](#), [segment_semantic](#)

segment_semantic	<i>Wood-Foliage Semantic Segmentation</i>
------------------	---

Description

This function performs wood-foliage semantic segmentation by constructing a network of points using k-nearest neighbors (KNN). A pathfinding algorithm navigates this network to identify the tree skeleton by finding the least-cost path from every point in space to the ground. Each point is classified as wood or foliage based on two main criteria: (1) Proximity to the detected skeleton and (2) wood likelihood exceeding a given threshold. The function [wood_likelihood](#) must be applied first. The point cloud must have an attribute 'hag' (see [height_above_ground](#)) Finally, a connected component step removes small clusters incorrectly classified as wood.

See the [Arbor book](#) for mode details.

Usage

```
segment_semantic(las, params = arbor_parameters_default)
```

Arguments

las	A LAS object.
params	list See parameters .

Value

A LAS object with an additional attribute foliage. 0 - wood, 1 and 2 foliage. The UserData attribute is also assigned two constant: ARBORLOW for points below the cut threshold and ARBORTREE for all others points

st_crs	<i>Get or set the projection of a QSM and QSF objects</i>
--------	---

Description

Get or set the projection of a QSM and QSF objects

Usage

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

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

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

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

Arguments

x	a QSM or QSF
...	Unused.
value	see sf::st_crs

trajectory_range	<i>Trajectory-based range filtering utilities</i>
------------------	---

Description

These functions implement a simple workflow to:

1. read trajectory file,
2. compute point-to-sensor range using the trajectory,
3. filter points based on range and height above ground.

They are designed to be used together, but can also be called independently.

read_trajectory: Reads a trajectory file and converts it to an sf point object. The first column is assumed to be the GPS time and is renamed to `gpstime`. The file must contain columns named `x`, `y`, and `z`.

add_range: Computes the distance between each point and the sensor position derived from a trajectory, and stores it as a new LAS attribute named `range`.

filter_range: Filters a point cloud by removing points that are too far to the sensor and below a given height above ground. For example, if `distance = 15` and `hag_max = 4`, points that are beyond 15 m of the sensor *and* lower than 4 m above ground are removed. Points farther than 15 m from the sensor but higher than 4 m above ground, are retained. This prevents the removal of canopy point.

Usage

```
read_trajectory(file, dt = 0.5)

add_range(las, traj)

filter_range(las, distance = 15, hag_max = 4)
```

Arguments

file	Character. Path to the ASCII trajectory file
dt	Numeric. Keep a record every dt seconds. Default 0.5.
las	A LAS object.
traj	An sf trajectory object as returned by read_trajectory .
distance	Numeric. Maximum allowed distance to the sensor.
hag_max	Numeric. Maximum allowed height above ground.

Value

An sf object with point geometry and a gpstime attribute.
 A LAS object with an additional attribute range.
 A filtered LAS object.

transfer_attributes	<i>Transfer attributes between point clouds</i>
---------------------	---

Description

Transfers one or more attributes from a decimated (sparser) point cloud to a denser point cloud using a 1-nearest neighbor (1-NN) approach. Each point in the target point cloud receives the attribute value of its nearest neighbor in the source point cloud.

Usage

```
transfer_attributes(from, to, attributes)
```

Arguments

from	A 'LAS' object representing the source (typically decimated) point cloud containing the attributes to transfer.
to	A 'LAS' object representing the target (typically denser) point cloud that will receive the transferred attributes.
attributes	A character vector specifying the names of the attributes to transfer from 'from' to 'to'.

Details

The function performs a 1-NN search between the 'from' and 'to' point clouds using 'lidR::knnx()'. For each attribute listed in 'attributes', the values are assigned from the nearest point in 'from'. If available, the function also propagates the corresponding attribute description from the Extra Bytes metadata ('VLR\$Extra_Bytes').

Value

A ‘LAS’ object identical to ‘to’, but with the requested attributes added.

See Also

[lidR::knnx()], [lidR::add_lasattribute()]

Examples

```
## Not run:
las <- transfer_attributes(las_decimated, las, attributes = "treeID", "foliage")

## End(Not run)
```

wood_likelihood	<i>Compute Local Wood Likelihood</i>
-----------------	--------------------------------------

Description

This function calculates the local anisotropy of each point in a LAS object using a k-nearest neighbors (KNN) approach. See the [Arbor book](https://r-lidar.github.io/arbor_book/) for more details.

Usage

```
wood_likelihood(las, params = arbor_parameters_default)
```

Arguments

las	A LAS object from lidR.
params	list See parameters .

Index

* experimental

- extract_tree_context, [5](#)
 - qsf_segment_semantic, [17](#)
- [.qsf, [3](#)
- %in%, LAS-method, [3](#)

- add_dbh3d (qsm_dbh), [21](#)
- add_range (trajectory_range), [29](#)
- add_single_tree_ground, [4](#)
- arbor_parameters_default (parameters), [9](#)
- ARBORBUFFER (ARBORTREE), [4](#)
- ARBORLOW (ARBORTREE), [4](#)
- ARBORTREE, [4](#)
- ARBORUNDERSTORY (ARBORTREE), [4](#)
- available_allometries, [5](#)

- colorize_trees (plot.qsf), [9](#)

- extract_tree_context, [5](#)
- extract_tree_context(), [17](#)

- filter_range (trajectory_range), [29](#)
- find_seeds, [6](#), [7](#), [27](#)
- flag_buffer, [5](#), [11](#)
- flag_buffer (flagging), [7](#)
- flag_small_trees, [5](#), [11](#), [12](#)
- flag_small_trees (flagging), [7](#)
- flagging, [7](#)
- foliage.colors, [8](#)

- height_above_ground, [28](#)
- hybrid_homogeneization, [8](#)

- install_cmd_tools, [8](#)

- parameters, [6](#), [9](#), [12](#), [20](#), [27](#), [28](#), [31](#)
- plot.qsf, [9](#)
- plot.qsm (plot.qsf), [9](#)
- plot_instance (plot.qsf), [9](#)
- plot_likelihood (plot.qsf), [9](#)
- plot_passage (plot.qsf), [9](#)

- plot_qsm (plot.qsf), [9](#)
- plot_semantic (plot.qsf), [9](#)
- plot_semantic_instance (plot.qsf), [9](#)
- print.qsf, [10](#)
- print.qsf_log (qsf_log), [15](#)
- print.qsm, [11](#)

- qsf, [7](#), [11](#), [15](#), [22](#)
- qsf_filter, [12](#), [15](#)
- qsf_filter_broken (qsf_filter), [12](#)
- qsf_filter_errors (qsf_filter), [12](#)
- qsf_filter_flagged (qsf_filter), [12](#)
- qsf_filter_messages (qsf_filter), [12](#)
- qsf_filter_nomeasure (qsf_filter), [12](#)
- qsf_filter_ok (qsf_filter), [12](#)
- qsf_filter_sapling (qsf_filter), [12](#)
- qsf_filter_warnings (qsf_filter), [12](#)
- qsf_log, [13](#), [14](#), [15](#), [15](#), [25](#)
- qsf_merchantable (qsm_subset), [23](#)
- qsf_read, [16](#), [18](#), [19](#)
- qsf_segment_semantic, [17](#)
- qsf_segment_semantic(), [6](#)
- qsf_treemap, [17](#)
- qsf_write, [16](#), [18](#)
- qsm, [12](#), [20](#), [22](#), [23](#)
- qsm_dbh, [13](#), [20](#), [21](#), [22](#), [23](#)
- qsm_height, [13](#)
- qsm_height (qsm_volume), [24](#)
- qsm_merchantable (qsm_subset), [23](#)
- qsm_message, [15](#)
- qsm_message (qsm_volume), [24](#)
- qsm_nostump (qsm_subset), [23](#)
- qsm_read, [20](#), [22](#)
- qsm_stats, [17](#), [20](#), [22](#)
- qsm_stem (qsm_subset), [23](#)
- qsm_subset, [23](#)
- qsm_treeid (qsm_volume), [24](#)
- qsm_volume, [24](#)
- qsm_write, [20](#), [25](#)

read_trajectory, [30](#)
read_trajectory (trajectory_range), [29](#)
resolve_oversegmentation, [26](#)

segment_ground, [26](#)
segment_instance, [6](#), [26](#), [27](#), [27](#)
segment_semantic, [27](#), [28](#)
st_crs, [28](#)
st_crs<-.qsf (st_crs), [28](#)
st_crs<-.qsm (st_crs), [28](#)

trajectory_range, [29](#)
transfer_attributes, [30](#)

wood_likelihood, [28](#), [31](#)