Forest Data Analysis Report

Jakob Danel and Frederick Bruch

2023-12-19

Contents

1 Introduction

This report documents the analysis of forest data for different tree species.

2 Methods

2.1 Data acquisition

Our primary objective is to identify patches where one tree species exhibits a high level of dominance, striving to capture monocultural stands within the diverse forests of Nordrhein-Westfalia (NRW). Recognizing the practical challenges of finding true monocultures, we aim to identify patches where one species is highly dominant, enabling meaningful comparisons across different species.

The study is framed within the NRW region due to the availability of an easily accessible dataset. Our focus includes four prominent tree species in NRW: oak, beech, spruce, and pine, representing the most prevalent species in the region. To ensure the validity of our findings, we derive three patches for each species, thereby confirming that observed variables are characteristic of a particular species rather than a specific patch. Each patch is carefully selected to encompass an area of approximately 50-100 hectares and contain between 5,000 and 10,000 trees. Striking a balance between relevance and manageability, these patches avoid excessive size to enhance the likelihood of capturing varied species mixes and ensure compatibility with local hardware.

Specific Goals:

1. Retrieve patches with highly dominant tree species.
2. Minimize or eliminate the presence of human-made structures within the selected patches.

To achieve our goals, we utilized the waldmonitor dataset (Welle et al. 2022) and the map provided by (Blickensdoerfer 2022), both indicating dominant tree species in NRW. We identified patches of feasible size where both sources predicted the presence of a specific species. Further validation involved examining sentinel images of these forest regions to assess the evenness of structures, leaf color distribution, and the absence of significant human-made structures such as roads or buildings. The subsequent preprocessing steps, detailed in the following subsection, involved refining our selected patches and deriving relevant variables, such as tree distribution and density, to ensure that the chosen areas align with the desired research domains.

2.2 Preprocessing

In this research study, the management and processing of a large dataset are crucial considerations. The dataset's substantial size necessitates careful maintenance to ensure efficient handling. Furthermore, the data should be easily processable and editable to facilitate necessary corrections and precalculations within the context of our research objectives. To achieve our goals, we have implemented a framework that automatically derives data based on a shapefile, delineating areas of interest. The processed data and results of precalculations are stored in a straightforward manner to enhance accessibility. Additionally, we have designed functions that establish a user-friendly interface, enabling the execution of algorithms on subsets of the data, such as distinct species. These interfaces are not only directly callable by users but can also be integrated into other functions to automate processes. The overarching aim is to streamline the entire preprocessing workflow using a single script, leveraging only the shapefile as a basis. This subsection details the accomplishments of our R-package in realizing these goals, outlining the preprocessing steps undertaken and justifying their necessity in the context of our research.

The data are stored in a data subdirectory of the root directory in the format `species/location-name/tile-nam` To automate the matching of areas of interest with the catalog from the Land NRW[1], we utilize the intersecting tool developed by Heisig[2]. This tool, allows for the automatic retrieval and placement of data downloaded from the Land NRW catalog. To enhance data accessibility, we have devised an object that incorporates species, location name, and tile name (the NRW internal identifier) for each area This object facilitates the specification of the area to be processed. Additionally, we have defined an initialization function that downloads all tiles, returning a list of tile location objects for subsequent processing. A pivotal component of the package's preprocessing functionality is the map function, which iterates over a list of tile locations (effectively the entire dataset) and accepts a processing function as an argument. The subsequent paragraph outlines the specific preprocessing steps employed, all of which are implemented within the mapping function.

To facilitate memory-handling capabilities, each of the tiles, where one area can span multiple tiles, has been split into manageable chunks. We employed a 50x50m size for each tile, resulting in the division of original 1km x 1km files into 400 tiles. These tiles are stored in our directory structure, with each tile housed in a directory named after its tile name and assigned an id as the filename. Implementation-wise, the `lidr::catalog_retile` function was instrumental in achieving this segmentation. The resulting smaller chunks allow for efficient iteration during subsequent preprocessing steps.

The next phase involves reducing our data to the actual size by intersecting the tiles with the defined area of interest. Using the `lidR::merge_spatial` function, we intersect the area derived from the shapefile, removing all point cloud items outside this region. Due to our tile-wise approach, empty tiles may arise, and in such cases, those tiles are simply deleted.

Following the size reduction to our dataset, the next step involves correcting the `z` values. The `z` values in the data are originally relative to the ellipsoid used for referencing, but we require them to be relative to the ground. To achieve this, we utilize the `lidR::tin` function, which extrapolates a convex hull between all ground points (classified by the data provider) and calculates the z value based on this structure.

Subsequently, we aim to perform segmentation for each distinct tree, marking each item of the point cloud with a tree ID. We employ the algorithm described by Li et al. (2012), using parameters `li2012(dt1 = 2, dt2 = 3, R = 2, Zu = 10, hmin = 5, speed_up = 12)`. The meanings of these parameters are elucidated in Li et al.'s work (Li et al. 2012).

Finally, the last preprocessing step involves individual tree detection, seeking a single `POINT` object for each tree. The `lidR::lmf` function, an implementation of the tree data using a local

[1]https://www.opengeodata.nrw.de/produkte/geobasis/hm/3dm_l_las/3dm_l_las/, last visited 7th Dec 2023
[2]https://github.com/joheisig/GEDIcalibratoR, last visited 7th Dec 2023

maximum approach, is utilized for this purpose (Popescu and Wynne 2004). The results are stored in GeoPackage files within our data structure.

See Section 5.1 for the implementation of the preprocessing.

3 Results

3.1 Researched areas

```
library(ggplot2)
sf::sf_use_s2(FALSE)
patches <- sf::read_sf("research_areas.shp") |> sf::st_centroid()

de <- sf::read_sf("results/results/states_de/Bundesländer_2017_mit_Einwohnerzahl.shp") # S
nrw <- de[5,] |> sf::st_geometry()


ggplot() + geom_sf(data = nrw) +
    geom_sf(data = patches, mapping = aes(col = species))
```

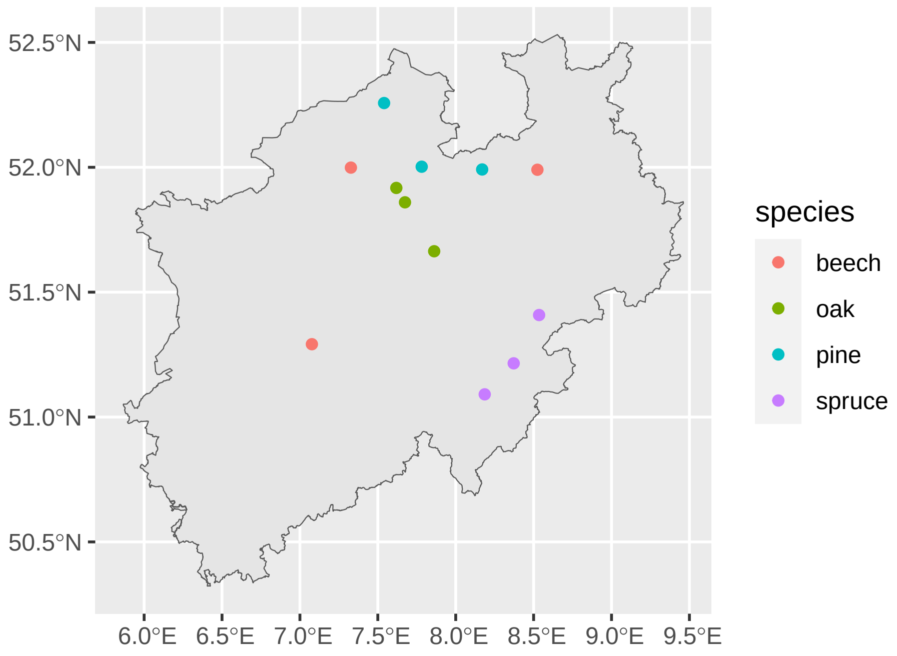

Figure 1: Locations of the different patches with the dominant species for that patch. The patches centroids are displayed on a basemap describing the borders from NRW.

We draw three patches for each species from different regions (see Table 1). We download the LiDAR data for those patches and runned all preprocessing steps as described. We than checked with certain derived parameters (e.g. tree heights, tree distributions or tree density) that all patches contain valid forest data. In that step we discovered, that in one patch some forest clearance took place in the near past. This patch was removed from the dataset and was replaced with a new one.

In our research, drawing patches evenly distributed across Nordrhein-Westfalia is inherently constrained by natural factors. Consequently, the patches for oak and pine predominantly originate from the Münsterland region, as illustrated in Figure 1. For spruce, the patches were derived from Sauerland, reflecting the prevalence of spruce forests in this specific region within NRW, as corroborated by Welle et al. (Welle et al. 2022) and Blickensdörfer et al. (Blickensdoerfer 2022). Beech patches, on the other hand, were generated from diverse locations within NRW. Across all patches, no human-made objects were identified, with the exception of small paths for pedestrians and forestry vehicles.

The distribution of area and detections is notable for each four species. Beech covers 69,791.9 hectares with a total of 5,954 detections, oak spans 63,232.49 hectares with 5,354 detections, pine extends across 72,862.4 hectares with 8,912 detections, and spruce encompasses 57,940.02 hectares with 8,619 detections. Both the amount of detections and the corresponding area exhibit a relatively uniform distribution across the diverse patches, as summarized in Table 1.

With the selected dataset described, we intentionally chose three patches for each four species that exhibit a practical and usable size for our research objectives. These carefully chosen patches align with the conditions essential for our study, providing comprehensive and representative data for in-depth analysis and meaningful insights into the characteristics of each tree species within the specified areas.

```r
shp <- sf::read_sf("research_areas.shp")
table <- lfa::lfa_get_all_areas()

sf::sf_use_s2(FALSE)
for (row in 1:nrow(table)) {
  area <-
    dplyr::filter(shp, shp$species == table[row, "specie"] &
                    shp$name == table[row, "area"])
  area_size <- area |> sf::st_area()
  point <- area |> sf::st_centroid() |> sf::st_coordinates()
  table[row,"point"] <- paste0("(",round(point[1], digits = 4),", ",round(point[2],digits

  table[row, "area_size"] = round(area_size,digits = 2) #paste0(round(area_size,digits = 2

  amount_det <- nrow(lfa::lfa_get_detection_area(table[row, "specie"], table[row, "area"])
  if(is.null(amount_det)){
    cat(nrow(lfa::lfa_get_detection_area(table[row, "specie"], table[row, "area"])),table[
  }
  table[row, "amount_detections"] = amount_det

  # table[row, "specie"] <- lfa::lfa_capitalize_first_char(table[row,"specie"])
  table[row, "area"] <- lfa::lfa_capitalize_first_char(table[row,"area"])
  }
table$area <- gsub("_", " ", table$area)
table$area <- gsub("ue", "ü", table$area)
table = table[,!names(table) %in% c("specie")]

knitr::kable(table, "html", col.names = c("Patch Name","Location","Area size (m²)","Amount
  kableExtra::kable_styling(
    bootstrap_options = c("striped", "hold_position", "bordered","responsive"),
    stripe_index = c(1:3,7:9),
    full_width = FALSE
  ) |>
  kableExtra::pack_rows("Beech", 1, 3) |>
  kableExtra::pack_rows("Oak", 4, 6) |>
  kableExtra::pack_rows("Pine", 7, 9) |>
  kableExtra::pack_rows("Spruce", 10, 12) |>
  kableExtra::column_spec(1, bold = TRUE)
```

Table 1: Summary of researched patches grouped by species, with their location, area and the amount of detected trees.

Patch Name	Location	Area size (m^2)	Amount tree detections
Beech			
Bielefeld brackwede	(8.5244, 51.9902)	161410.57	1443
Billerbeck	(7.3273, 51.9987)	185887.25	1732
Wülfenrath	(7.0769, 51.2917)	350621.21	2779
Oak			
Hamm	(7.8618, 51.6639)	269397.22	2441
Münster	(7.6187, 51.9174)	164116.61	1270
Rinkerode	(7.6744, 51.8598)	198811.09	1643
Pine			
Greffen	(8.1697, 51.9913)	49418.81	513
Mesum	(7.5403, 52.2573)	405072.85	5031
Telgte	(7.7816, 52.0024)	274132.34	3368
Spruce			

Patch Name	Location	Area size (m^2)	Amount tree detections
Brilon	(8.5352, 51.4084)	211478.20	3342
Oberhundem	(8.1861, 51.0909)	151895.53	2471
Osterwald	(8.3721, 51.2151)	216026.43	2806

specie	area	density ($1/m^2$)
beech	bielefeld_brackwede	0.0089399
beech	billerbeck	0.0093175
beech	wuelfenrath	0.0079259
oak	hamm	0.0090610
oak	muenster	0.0077384
oak	rinkerode	0.0082641
pine	greffen	0.0103807
pine	mesum	0.0124200
pine	telgte	0.0122860
spruce	brilon	0.0158030
spruce	oberhundem	0.0162678
spruce	osterwald	0.0129892

4 References

Blickensdoerfer, Lukas. 2022. "Dominant Tree Species for Germany (2017/2018)." *Waldatlas-Wald Und Waldnutzung*. Thünen Atlas. https://atlas.thuenen.de/layers/geonode:Dominant_Species_Class.

Li, Wenkai, Qinghua Guo, Marek Jakubowski, and Maggi Kelly. 2012. "A New Method for Segmenting Individual Trees from the Lidar Point Cloud." *Photogrammetric Engineering and Remote Sensing* 78 (January): 75–84. https://doi.org/10.14358/PERS.78.1.75.

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

Welle, Torsten, Lukas Aschenbrenner, Kevin Kuonath, Stefan Kirmaier, and Jonas Franke. 2022. "Mapping Dominant Tree Species of German Forests." *Remote Sensing* 14 (14). https://doi.org/10.3390/rs14143330.

5 Appendix

5.1 Script which can be used to do all preprocessing

Load the file with the research areas ::: {.cell}

```
sf <- sf::read_sf(here::here("research_areas.shp"))
print(sf)
```

```
Simple feature collection with 12 features and 3 fields
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: 7.071625 ymin: 51.0895 xmax: 8.539877 ymax: 52.25983
Geodetic CRS:  WGS 84
# A tibble: 12 x 4
      id species name                                                   geometry
   <dbl> <chr>   <chr>                                             <POLYGON [°]>
 1     1 oak     rinkerode          ((7.678922 51.85789, 7.675446 51.85752, 7.~
 2     2 oak     hamm               ((7.858955 51.66699, 7.866444 51.66462, 7.~
 3     3 oak     muenster           ((7.618908 51.9154, 7.617384 51.9172, 7.61~
 4     4 pine    greffen            ((8.168691 51.98965, 8.167178 51.99075, 8.~
 5     5 pine    telgte             ((7.779728 52.00662, 7.781616 52.00662, 7.~
 6     6 pine    mesum              ((7.534424 52.25499, 7.53378 52.25983, 7.5~
 7     7 beech   bielefeld_brackwede ((8.524749 51.9921, 8.528418 51.99079, 8.5~
 8     8 beech   wuelfenrath        ((7.071625 51.29256, 7.072311 51.29334, 7.~
 9     9 beech   billerbeck         ((7.324729 51.99783, 7.323548 51.99923, 7.~
10    11 spruce  brilon             ((8.532195 51.41029, 8.535027 51.41064, 8.~
11    12 spruce  osterwald          ((8.369328 51.21693, 8.371238 51.21718, 8.~
12    10 spruce  oberhundem         ((8.18082 51.08999, 8.180868 51.09143, 8.1~
```

:::

Init the project ::: {.cell}

```
library(lfa)
sf::sf_use_s2(FALSE)
locations <- lfa_init("research_areas.shp")
```

:::

Do all of the prprocessing steps ::: {.cell}

```
lfa_map_tile_locations(locations,retile,check_flag = "retile")
```

```
No further processing: flag retile is set!Function is already computed, no further computings
```

```
NULL
```

```
lfa_map_tile_locations(locations, lfa_intersect_areas, ctg = NULL, areas_sf = sf,check_fla
```

```
No further processing: flag intersect is set!Function is already computed, no further comput
```

```
NULL
```

```
lfa_map_tile_locations(locations, lfa_ground_correction, ctg = NULL,check_flag = "z_correc
```

```
No further processing: flag z_correction is set!Function is already computed, no further com
```

```
NULL
```

```
lfa_map_tile_locations(locations, lfa_segmentation, ctg = NULL,check_flag = "segmentation"
```

```
No further processing: flag segmentation is set!Function is already computed, no further com
```

```
NULL
```

```
lfa_map_tile_locations(locations, lfa_detection, catalog = NULL, write_to_file = TRUE,chec
```

```
No further processing: flag detection is set!Function is already computed, no further comput
```

```
NULL
```

:::

5.2 Documentation

5.2.1 `lfa_capitalize_first_char`

Capitalize First Character of a String

Arguments

Argument	Description
`input_string`	A single-character string to be processed.

Concept

String Manipulation

Description

This function takes a string as input and returns the same string with the first character capitalized. If the first character is already capitalized, the function does nothing. If the first character is not from the alphabet, an error is thrown.

Details

This function performs the following steps:

- Checks if the input is a single-character string.
- Verifies if the first character is from the alphabet (A-Z or a-z).
- If the first character is not already capitalized, it capitalizes it.
- Returns the modified string.

Keyword

alphabet

Note

This function is case-sensitive and assumes ASCII characters.

References

None

Seealso

This function is related to the basic string manipulation functions in base R.

Value

A modified string with the first character capitalized if it is not already. If the first character is already capitalized, the original string is returned.

Examples

```r
# Capitalize the first character of a string
capitalize_first_char("hello") # Returns "Hello"
capitalize_first_char("World") # Returns "World"

# Error example (non-alphabetic first character)
capitalize_first_char("123abc") # Throws an error
```

Usage

```r
lfa_capitalize_first_char(input_string)
```

5.2.2 `lfa_check_flag`

Check if a flag is set, indicating the completion of a specific process.

Arguments

Argument	Description
`flag_name`	A character string specifying the name of the flag file. It should be a descriptive and unique identifier for the process being checked.

Description

This function checks for the existence of a hidden flag file at a specified location within the working directory. If the flag file is found, a message is printed, and the function returns `TRUE` to indicate that the associated processing step has already been completed. If the flag file is not found, the function returns `FALSE` , indicating that further processing can proceed.

Value

A logical value indicating whether the flag is set (`TRUE`) or not (`FALSE`).

Examples

```r
# Check if the flag for a process named "data_processing" is set
lfa_check_flag("data_processing")
```

Usage

```r
lfa_check_flag(flag_name)
```

5.2.3 `lfa_create_tile_location_objects`

Create tile location objects

Author

Jakob Danel

Description

This function traverses a directory structure to find LAZ files and creates tile location objects for each file. The function looks into the the `data` directory of the repository/working directory. It then creates `tile_location` objects based on the folder structure. The folder structure should not be touched by hand, but created by `lfa_init_data_structure()` which builds the structure based on a shape file.

Seealso

`tile_location`

Value

A vector containing tile location objects.

Examples

```
lfa_create_tile_location_objects()

lfa_create_tile_location_objects()
```

Usage

```
lfa_create_tile_location_objects()
```

5.2.4 `lfa_detection`

Perform tree detection on a lidar catalog and optionally save the results to a file.

Arguments

Argument	Description
`catalog`	A lidar catalog containing point cloud data. If set to NULL, the function attempts to read the catalog from the specified tile location.
`tile_location`	An object specifying the location of the lidar tile. If catalog is NULL, the function attempts to read the catalog from this tile location.
`write_to_file`	A logical value indicating whether to save the detected tree information to a file. Default is TRUE.

Description

This function utilizes lidar data to detect trees within a specified catalog. The detected tree information can be optionally saved to a file in the GeoPackage format. The function uses parallel processing to enhance efficiency.

Value

A sf style data frame containing information about the detected trees.

Examples

```
# Perform tree detection on a catalog and save the results to a file
lfa_detection(catalog = my_catalog, tile_location = my_tile_location, write_to_file = TRUE
```

Usage

```
lfa_detection(catalog, tile_location, write_to_file = TRUE)
```

5.2.5 `lfa_download_areas`

Download areas based on spatial features

Arguments

Argument	Description
`sf_areas`	Spatial features representing areas to be downloaded. It must include columns like "species" "name" See details for more information.

Author

Jakob Danel

Description

This function initiates the data structure and downloads areas based on spatial features.

Details

The input data frame, `sf_areas` , must have the following columns:

- "species": The species associated with the area.
- "name": The name of the area.

The function uses the `lfa_init_data_structure` function to set up the data structure and then iterates through the rows of `sf_areas` to download each specified area.

Value

None

Examples

```r
lfa_download_areas(sf_areas)


# Example spatial features data frame
sf_areas <- data.frame(
species = c("SpeciesA", "SpeciesB"),
name = c("Area1", "Area2"),
# Must include also other attributes specialized to sf objects
# such as geometry, for processing of the download
)

lfa_download_areas(sf_areas)
```

Usage

```r
lfa_download_areas(sf_areas)
```

5.2.6 `lfa_download`

Download an las file from the state NRW from a specific location

Arguments

Argument	Description
`species`	The species of the tree which is observed at this location
`name`	The name of the area that is observed
`location`	An sf object, which holds the location information for the area where the tile should be downloaded from.

Description

It will download the file and save it to data/ list(list("html"), list(list(""))) / list(list("html"), list(list(""))) with the name of the tile

Value

The LASCatalog object of the downloaded file

Usage

```
lfa_download(species, name, location)
```

5.2.7 `lfa_get_detection_area`

Get Detection for an area

Arguments

Argument	Description
`species`	A character string specifying the target species.
`name`	A character string specifying the name of the tile.

Description

Retrieves the tree detection information for a specified species and tile.

Details

This function reads tree detection data from geopackage files within the specified tile location for a given species. It then combines the data into a single SF data frame and returns it. The function assumes that the tree detection files follow a naming convention with the pattern "_detection.gpkg".

Keyword

spatial

References

This function is part of the LiDAR Forest Analysis (LFA) package.

Seealso

`get_tile_dir`

Value

A Simple Features (SF) data frame containing tree detection information for the specified species and tile.

Examples

```r
# Retrieve tree detection data for species "example_species" in tile "example_tile"
trees_data <- lfa_get_detection_tile_location("example_species", "example_tile")

# Example usage:
trees_data <- lfa_get_detection_tile_location("example_species", "example_tile")

# No trees found scenario:
empty_data <- lfa_get_detection_tile_location("nonexistent_species", "nonexistent_tile")
# The result will be an empty data frame if no trees are found for the specified species a

# Error handling:
# In case of invalid inputs, the function may throw errors. Ensure correct species and til
```

Usage

```
lfa_get_detection_area(species, name)
```

5.2.8 `lfa_get_detections_species`

Retrieve detections for a specific species.

Arguments

Argument	Description
`species`	A character string specifying the target species.

Description

This function retrieves detection data for a given species from multiple areas.

Details

The function looks for detection data in the "data" directory for the specified species. It then iterates through each subdirectory (representing different areas) and consolidates the detection data into a single data frame.

Value

A data frame containing detection information for the specified species in different areas.

Examples

```
# Example usage:
detections_data <- lfa_get_detections_species("example_species")
```

Usage

```
lfa_get_detections_species(species)
```

5.2.9 `lfa_get_detections`

Retrieve aggregated detection data for multiple species.

Concept

data retrieval functions

Description

This function obtains aggregated detection data for multiple species by iterating through the list of species obtained from `lfa_get_species` . For each species, it calls `lfa_get_detections_species` to retrieve the corresponding detection data and aggregates the results into a single data frame. The resulting data frame includes columns for the species, tree detection data, and the area in which the detections occurred.

Keyword

aggregation

Seealso

`lfa_get_species` , `lfa_get_detections_species`

Other data retrieval functions: `lfa_get_species`

Value

A data frame containing aggregated detection data for multiple species.

Examples

```
lfa_get_detections()

# Retrieve aggregated detection data for multiple species
detections_data <- lfa_get_detections()
```

Usage

```
lfa_get_detections()
```

5.2.10 `lfa_get_flag_path`

Get the path to a flag file indicating the completion of a specific process.

Arguments

Argument	Description
`flag_name`	A character string specifying the name of the flag file. It should be a descriptive and unique identifier for the process being flagged.

Description

This function constructs and returns the path to a hidden flag file, which serves as an indicator that a particular processing step has been completed. The flag file is created in a designated location within the working directory.

Value

A character string representing the absolute path to the hidden flag file.

Examples

```
# Get the flag path for a process named "data_processing"
lfa_get_flag_path("data_processing")
```

Usage

```
lfa_get_flag_path(flag_name)
```

5.2.11 `lfa_get_species`

Get a list of species from the data directory.

Concept

data retrieval functions

Description

This function retrieves a list of species by scanning the “data” directory located in the current working directory.

Keyword

data

References

This function relies on the `list.dirs` function for directory listing.

Seealso

`list.dirs`

Other data retrieval functions: `lfa_get_detections`

Value

A character vector containing the names of species found in the “data” directory.

Examples

```
# Retrieve the list of species
species_list <- lfa_get_species()
```

Usage

```
lfa_get_species()
```

5.2.12 `lfa_ground_correction`

Correct the point clouds for correct ground imagery

Arguments

Argument	Description
`ctg`	An LASCatalog object. If not null, it will perform the actions on this object, if NULL inferring the catalog from the tile_location
`tile_location`	A tile_location type object holding the information about the location of the cataog. This is used to save the catalog after processing too.

Author

Jakob Danel

Description

This function is needed to correct the Z value of the point cloud, relative to the real ground height. After using this function to your catalog, the Z values can be seen as the real elevation about the ground. At the moment the function uses the `tin()` function from the `lidr` package. NOTE : The operation is inplace and can not be reverted, the old values of the point cloud will be deleted!

Value

A catalog with the corrected z values. The catalog is always stored at tile_location and holding only the transformed values.

Usage

```r
lfa_ground_correction(ctg, tile_location)
```

5.2.13 `lfa_init_data_structure`

Initialize data structure for species and areas

Arguments

Argument	Description
`sf_species`	A data frame with information about species and associated areas.

Description

This function initializes the data structure for storing species and associated areas.

Details

The input data frame, `sf_species` , should have at least the following columns:

- “species”: The names of the species for which the data structure needs to be initialized.
- “name”: The names of the associated areas.

The function creates directories based on the species and area information provided in the `sf_species` data frame. It checks whether the directories already exist and creates them if they don’t.

Value

None

Examples

```r
# Example species data frame
sf_species <- data.frame(
species = c("SpeciesA", "SpeciesB"),
name = c("Area1", "Area2"),
# Other necessary columns
)

lfa_init_data_structure(sf_species)

# Example species data frame
sf_species <- data.frame(
species = c("SpeciesA", "SpeciesB"),
name = c("Area1", "Area2"),
# Other necessary columns
)
```

```
lfa_init_data_structure(sf_species)
```

Usage

```
lfa_init_data_structure(sf_species)
```

5.2.14 `lfa_init`

Initialize LFA (LiDAR forest analysis) data processing

Arguments

Argument	Description
`sf_file`	A character string specifying the path to the shapefile containing spatial features of research areas.

Description

This function initializes the LFA data processing by reading a shapefile containing spatial features of research areas, downloading the specified areas, and creating tile location objects for each area.

Details

This function reads a shapefile (`sf_file`) using the `sf` package, which should contain information about research areas. It then calls the `lfa_download_areas` function to download the specified areas and `lfa_create_tile_location_objects` to create tile location objects based on Lidar data files in those areas. The shapefile MUST follow the following requirements:

- Each geometry must be a single object of type polygon
- Each entry must have the following attributes:
- species: A string describing the tree species of the area.
- name: A string describing the location of the area.

Value

A vector containing tile location objects.

Examples

```r
# Initialize LFA processing with the default shapefile
lfa_init()

# Initialize LFA processing with a custom shapefile
lfa_init("custom_areas.shp")

# Example usage with the default shapefile
lfa_init()

# Example usage with a custom shapefile
lfa_init("custom_areas.shp")
```

Usage

```r
lfa_init(sf_file = "research_areas.shp")
```

5.2.15 `lfa_intersect_areas`

Intersect Lidar Catalog with Spatial Features

Arguments

Argument	Description
`ctg`	A LAScatalog object representing the Lidar data to be processed.
`tile_location`	A tile location object representing the specific area of interest.
`areas_sf`	Spatial features defining areas.

Description

This function intersects a Lidar catalog with a specific area defined by spatial features.

Details

The function intersects the Lidar catalog specified by `ctg` with a specific area defined by the `tile_location` object and `areas_sf` . It removes points outside the specified area and returns a modified LAScatalog object.

The specified area is identified based on the `species` and `name` attributes in the `tile_location` object. If a matching area is not found in `areas_sf` , the function stops with an error.

The function then transforms the spatial reference of the identified area to match that of the Lidar catalog using `sf::st_transform` .

The processing is applied to each chunk in the catalog using the `identify_area` function, which merges spatial information and filters out points that are not classified as inside the identified area. After processing, the function writes the modified LAS files back to the original file locations, removing points outside the specified area.

If an error occurs during the processing of a chunk, a warning is issued, and the function continues processing the next chunks. If no points are found after filtering, a warning is issued, and NULL is returned.

Seealso

Other functions in the Lidar forest analysis (LFA) package.

Value

A modified LAScatalog object with points outside the specified area removed.

Examples

```r
# Example usage
lfa_intersect_areas(ctg, tile_location, areas_sf)

# Example usage
lfa_intersect_areas(ctg, tile_location, areas_sf)
```

Usage

```r
lfa_intersect_areas(ctg, tile_location, areas_sf)
```

5.2.16 `lfa_load_ctg_if_not_present`

Loading the catalog if it is not present

Arguments

Argument	Description
`ctg`	Catalog object. Can be NULL
`tile_location`	The location to look for the catalog tiles, if their are not present

Description

This function checks if the catalog is `NULL` . If it is it will load the catalog from the `tile_location`

Value

The provided ctg object if not null, else the catalog for the tiles of the tile_location.

Usage

```
lfa_load_ctg_if_not_present(ctg, tile_location)
```

5.2.17 `lfa_map_tile_locations`

Map Function Over Tile Locations

Arguments

Argument	Description
`tile_locations`	A list of tile location objects.
`map_function`	The mapping function to be applied to each tile location.
`...`	Additional arguments to be passed to the mapping function.

Description

This function applies a specified mapping function to each tile location in a list.

Details

This function iterates over each tile location in the provided list (`tile_locations`) and applies the specified mapping function (`map_function`) to each tile location. The mapping function should accept a tile location object as its first argument, and additional arguments can be passed using the ellipsis (`...`) syntax.

This function is useful for performing operations on multiple tile locations concurrently, such as loading Lidar data, processing areas, or other tasks that involve tile locations.

Seealso

The mapping function provided should be compatible with the structure and requirements of the tile locations and the specific task being performed.

Value

None

Examples

```
# Example usage
lfa_map_tile_locations(tile_locations, my_mapping_function, param1 = "value")

# Example usage
lfa_map_tile_locations(tile_locations, my_mapping_function, param1 = "value")
```

Usage

```
lfa_map_tile_locations(tile_locations, map_function, check_flag = NULL, ...)
```

5.2.18 `lfa_merge_and_save`

Merge and Save Text Files in a Directory

Arguments

Argument	Description
`input_directory`	The path to the input directory containing text files.
`output_name`	The name for the output file where the merged content will be saved.

Description

This function takes an input directory and an output name as arguments. It merges the textual content of all files in the specified directory into a single string, with each file's content separated by a newline character. The merged content is then saved into a file named after the output name in the same directory. After the merging is complete, all input files are deleted.

Details

This function reads the content of each text file in the specified input directory and concatenates them into a single string. Each file's content is separated by a newline character. The merged content is then saved into a file named after the output name in the same directory. Finally, all input files are deleted from the directory.

Seealso

`readLines` , `writeLines` , `file.remove`

Value

This function does not explicitly return any value. It prints a message indicating the successful completion of the merging and saving process.

Examples

```r
# Merge text files in the "data_files" directory and save the result in "merged_output"
lfa_merge_and_save("data_files", "merged_output")

# Merge text files in the "data_files" directory and save the result in "merged_output"
lfa_merge_and_save("data_files", "merged_output")
```

Usage

```r
lfa_merge_and_save(input_directory, output_name)
```

5.2.19 `lfa_rd_to_qmd`

Convert Rd File to Markdown

Arguments

Argument	Description
`rdfile`	The path to the Rd file or a parsed Rd object.
`outfile`	The path to the output Markdown file (including the file extension).
`append`	Logical, indicating whether to append to an existing file (default is FALSE).

Description

IMPORTANT NOTE: This function is nearly identical to the `Rd2md::Rd2markdown` function from the `Rd2md` package. We needed to implement our own version of it because of various reasons:

- The algorithm uses hardcoded header sizes (h1 and h2 in original) which is not feasible for our use-case of the markdown.
- We needed to add some Quarto Markdown specifics, e.g. to make sure that the examples will not be runned.
- We want to exclude certain tags from our implementation.

Details

For that reason we copied the method and made changes as needed and also added this custom documentation.

This function converts an Rd (R documentation) file to Markdown format (.md) and saves the converted file at the specified location. The function allows appending to an existing file or creating a new one. The resulting Markdown file includes sections for the function's name, title, and additional content such as examples, usage, arguments, and other sections present in the Rd file.

The function performs the following steps:

- Parses the Rd file using the Rd2md package.
- Creates a Markdown file with sections for the function's name, title, and additional content.
- Appends the content to an existing file if `append` is set to TRUE.
- Saves the resulting Markdown file at the specified location.

Seealso

`Rd2md::parseRd`

Value

This function does not explicitly return any value. It saves the converted Markdown file at the specified location as described in the details section.

Examples

```r
# Convert Rd file to Markdown and save it
lfa_rd_to_md("path/to/your/file.Rd", "path/to/your/output/file.md")

# Convert Rd file to Markdown and append to an existing file
lfa_rd_to_md("path/to/your/file.Rd", "path/to/existing/output/file.md", append = TRUE)
```

Usage

```r
lfa_rd_to_qmd(rdfile, outfile, append = FALSE)
```

5.2.20 `lfa_rd_to_results`

Convert Rd Files to Markdown and Merge Results

Description

This function converts all Rd (R documentation) files in the “man” directory to Markdown format (.qmd) and saves the converted files in the “results/appendix/package-docs” directory. It then merges the converted Markdown files into a single string and saves the merged content into a file named “docs.qmd” in the “results/appendix/package-docs” directory.

Details

The function performs the following steps:

- Removes any existing “docs.qmd” file in the “results/appendix/package-docs” directory.
- Finds all Rd files in the “man” directory.
- Converts each Rd file to Markdown format (.qmd) using the `lfa_rd_to_qmd` function.
- Saves the converted Markdown files in the “results/appendix/package-docs” directory.
- Merges the content of all converted Markdown files into a single string.
- Saves the merged content into a file named “docs.qmd” in the “results/appendix/package-docs” directory.

Seealso

`lfa_rd_to_qmd` , `lfa_merge_and_save`

Value

This function does not explicitly return any value. It performs the conversion, merging, and saving operations as described in the details section.

Examples

```
# Convert Rd files to Markdown and merge the results
lfa_rd_to_results()
```

Usage

```
lfa_rd_to_results()
```

5.2.21 `lfa_segmentation`

Segment the elements of an point cloud by trees

Arguments

Argument	Description
`ctg`	An LASCatalog object. If not null, it will perform the actions on this object, if NULL inferring the catalog from the tile_location
`tile_location`	A tile_location type object holding the information about the location of the catalog. This is used to save the catalog after processing too.

Author

Jakob Danel

Description

This function will try to to divide the hole point cloud into unique trees. Therefore it is assigning for each chunk of the catalog a `treeID` for each point. Therefore the algorithm uses the `li2012` implementation with the following parameters: `li2012(dt1 = 2, dt2 = 3, R = 2, Zu = 10, hmin = 5, speed_up = 12)` NOTE : The operation is in place and can not be reverted, the old values of the point cloud will be deleted!

Value

A catalog where each chunk has additional `treeID` values indicating the belonging tree.

Usage

```
lfa_segmentation(ctg, tile_location)
```

5.2.22 `lfa_set_flag`

Set a flag to indicate the completion of a specific process.

Arguments

Argument	Description
`flag_name`	A character string specifying the name of the flag file. It should be a descriptive and unique identifier for the process being flagged.

Description

This function creates a hidden flag file at a specified location within the working directory to indicate that a particular processing step has been completed. If the flag file already exists, a warning is issued.

Value

This function does not have a formal return value.

Examples

```r
# Set the flag for a process named "data_processing"
lfa_set_flag("data_processing")
```

Usage

```r
lfa_set_flag(flag_name)
```