LiDAR Analysis in R and rLiDAR for Forestry Applications

Book · April 2018

CITATIONS
0

4 authors:

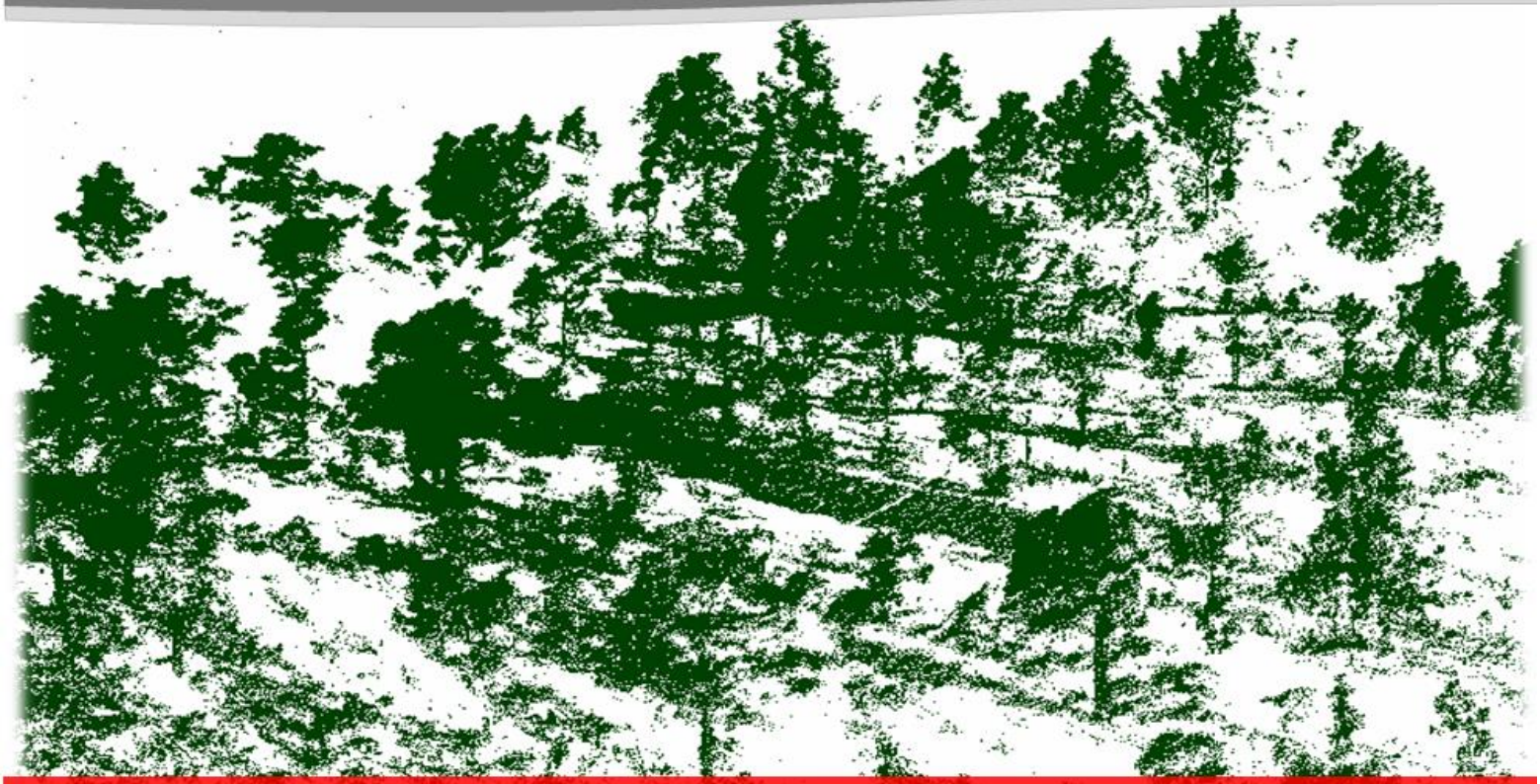
Carlos Alberto Silva
NASA
72 PUBLICATIONS **111** CITATIONS
SEE PROFILE

Carine Klauberg
US Forest Service
53 PUBLICATIONS **104** CITATIONS
SEE PROFILE

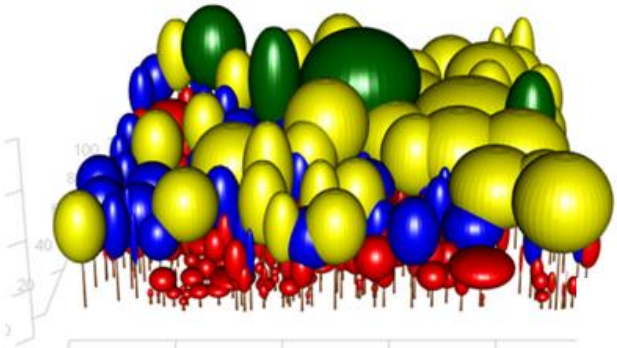
Mikey Mid Mohan
North Carolina State University
4 PUBLICATIONS **17** CITATIONS
SEE PROFILE

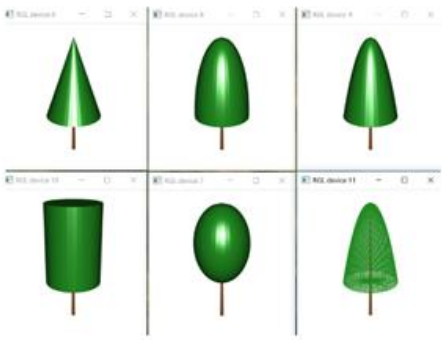
Benjamin C. Bright
US Forest Service
25 PUBLICATIONS **185** CITATIONS
SEE PROFILE

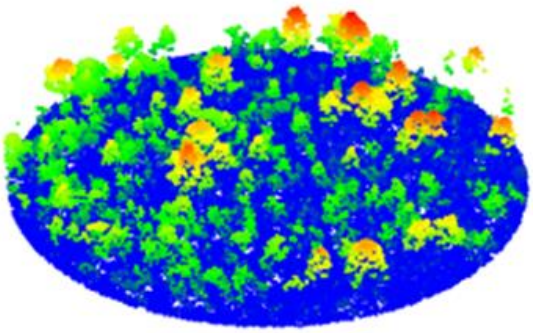
NRS 404/504 – Lidar Remote Sensing for Environmental Monitoring



LiDAR Analysis in R and rLiDAR for Forestry Applications







Carlos Alberto Silva
Carine Klauberg
Midhun Mohan
Benjamin C. Bright

Moscow, ID
Summer 2018

Table of Contents

Chapter 1: Introduction to LiDAR Remote Sensing

Benjamin C. Bright
USDA Forest Service
Rocky Mountain Research Station-RMRS
benjamincbright@fs.fed.us

1.1 What is LiDAR and how does it work?

Light detection and ranging (LiDAR) is a remote sensing system that uses lasers to measure distances between a sensor and targets of interest. Because LiDAR systems use lasers to interact with targets, LiDAR is considered a form of active remote sensing, as opposed to passive remote sensing, such as sensors aboard satellites that passively measure reflected sunlight. LiDAR instruments calculate distances, or ranges, by using the simple formula:

$$d = 0.5 \times c \times t$$

where d is the distance between the sensor and the target, c is the speed of light (3×10^8 m s^{-1}), and t is the time required for a laser pulse to travel from the instrument, bounce off the target of interest, and return to the instrument.

In airborne LiDAR systems, a typical LiDAR instrument consists of a high frequency laser that has the ability to emit tens of thousands of laser pulses per second, from which corresponding range measurements can be derived. Rate of laser pulse emission is quantified and referred to as the "pulse repetition frequency". Pulses are delivered by scanning, so that pulses are emitted within a set number of degrees from nadir; each laser pulse and associated returned energy has an associated "scan angle", which describes the angle from nadir of the pulse. For this reason, LiDAR is frequently described as "laser scanning".

Other important airborne LiDAR parameters include laser wavelength, beam divergence, and laser footprint size. Laser wavelength for terrestrial applications is usually in the near-infrared around 1064 nm; green LiDAR with wavelengths around 532 nm is also used for bathymetry. Beam divergence, usually reported in units of milliradians, quantifies how much laser energy diverges or widens with distance from emission. Laser footprint diameter describes the diameter of the circular area of the target illuminated by the laser pulse, and is described with the formula:

$$f = \frac{h}{cos2(\Theta)}\gamma$$

where f is the footprint diameter, h is altitude of the LiDAR sensor, Θ is scan angle of the laser pulse, and γ is beam divergence (Baltsavias, 1999; Jensen, 2007).

1.2 LiDAR platforms

LiDAR platforms can be divided into two categories: airborne and terrestrial.

1.2.1 Airborne

LiDAR sensors for earth applications are frequently flown aboard fixed-wing aircraft at relatively low elevations (~1000 m above ground level), although helicopters have also been used, and unmanned aerial vehicles (UAVs) are becoming more popular as LiDAR platforms. LiDAR sensors have also been put aboard satellites; the Geoscience Laser Altimeter System (GLAS) is one such spaceborne LiDAR system.

1.2.2 Terrestrial

For terrestrial LiDAR acquisitions, the LiDAR instrument remains in a fixed location, often mounted on a tripod, and rotates to scan an area around that fixed location. The area scanned via terrestrial LiDAR is smaller than that scanned by airborne LiDAR, but the density of returns is greater.

1.3 LiDAR products

1.3.1 LiDAR point clouds

For most airborne LiDAR collections, the precise location of the LiDAR sensor is tracked via the Global Positioning System (GPS) throughout the acquisition. In this way, collections of range measurements can be georeferenced, so that x,y,z points refer to locations on the earth. Collections of range measurements are typically delivered as "point clouds"; points are also frequently called "returns" as they represent energy that has returned to the sensor after reflecting from a target. For a given pulse, LiDAR sensors are often able to record more than one return; "return number" refers to the order with which the return was detected by the sensor. Each return often has an associated "return intensity", which describes the amount of energy detected by the sensor. LiDAR point clouds are typically delivered in LAS format.

1.3.2 Digital terrain models

Point cloud classification, in which returns are classified by the type of surface from which the laser pulse reflected, is a vital processing step in most applications of LiDAR data. One frequently performed classification of airborne lidar in earth applications is that of distinguishing between ground and non-ground (often vegetation or buildings) returns. Classification of ground and non-ground returns is usually performed via algorithms, which can be area or application specific. For example, the Multiscale Curvature Classification algorithm was developed for classifying returns as ground and non-ground in forested environments (Evans and Hudak, 2007). LiDAR vendors often perform classification of returns before delivering data, and LiDAR data available in repositories are frequently already classified.

Once classified, ground returns can then be interpolated into a digital terrain model (DTM, Figure 1b), typically in raster format. The creation of a triangulated irregular network (TIN) can be, but is not always, an intermediate step in converting point measurements to raster format. DTMs can then be used to create additional raster products, such as slope or aspect.

1.3.3 LiDAR metrics

In natural environments, most other returns that are not classified as ground returns, i.e. non-ground returns, can be thought of as vegetation returns, where lidar pulses have reflected off vegetation. LiDAR metrics, in which various statistics of vegetation returns are calculated, are often used to characterize and model forest and tree characteristics. Before creating LiDAR metrics, non-ground returns must be normalized from height above ellipsoid or sea level, to height above ground. This is done by subtracting the height of the DTM from the return height for each non-ground or vegetation return.

Similar to how ground return information can be summarized into raster format, vegetation returns can be "binned" into raster format to create LiDAR metrics. For example, returns might be binned into a grid with a resolution of 30 m, and then various statistics for the returns in each bin can be calculated and output to raster format. An example of a frequently used LiDAR metric is mean vegetation height, where a raster grid cell takes the value of the mean height of all the returns within that grid cell (Figure 1c).

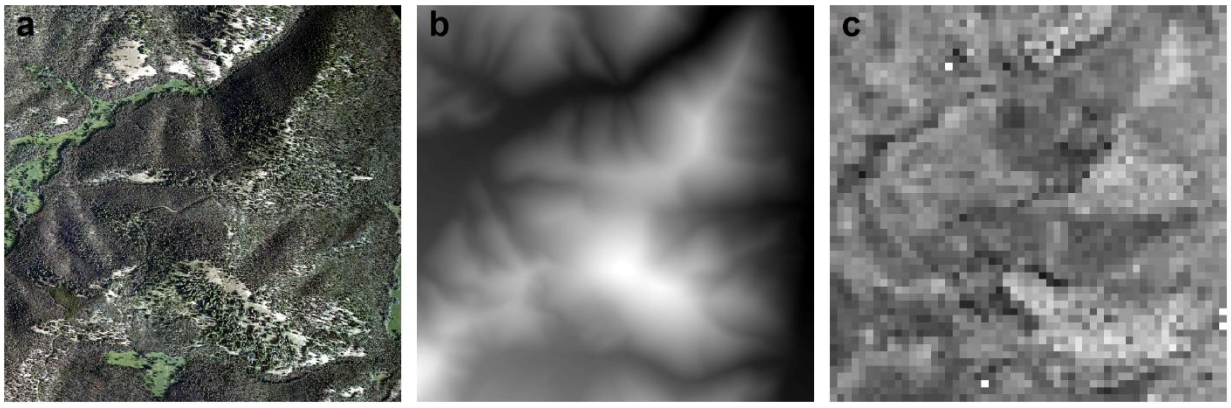


Figure 1. True-color aerial imagery (a), lidar-derived digital terrain model (DTM) (b), and lidar-derived mean vegetation height (c) of a bark beetle-affected forest in central Idaho. Lighter grayscale tones indicate higher DTM and mean vegetation height values.

1.4 LiDAR for forest applications

Maltamo et al. (2014) gives a thorough and current overview of how LiDAR is being used for forest applications. Lefsky et al. (2002) also provides a good overview, some history, and some ecological applications of LiDAR. Some applications of LiDAR for

forestry include modeling and mapping of tree density, biomass, volume, fuel, age, structure, and wildlife habitat. Using LiDAR to distinguish between tree type, species, and health has also been demonstrated. LiDAR forestry applications can be distinguished as single-tree or area approaches. Single-tree approaches use LiDAR to detect and measure characteristics of single trees, whereas area approaches seek to characterize multiple trees. LiDAR data is often used in conjunction with imagery from passive sensors, where the spectral information from imagery complements three dimensional information provided by LiDAR.

1.5 LiDAR data sources

LiDAR data is expensive to acquire. However, the availability and amount of LiDAR data has increased as the popularity of LiDAR as increased. Some sources of raw LiDAR data as well as derived LiDAR products such as DTMs, include OpenTopography (http://www.opentopography.org/), the USGS Earth Explorer (https://earthexplorer.usgs.gov/), and the National Oceanic and Atmospheric Administration (https://coast.noaa.gov/dataviewer/#/). Local repositories also exist and are worth seeking out if an area of interest is known; an example of a local repository is the Idaho LiDAR Consortium (https://www.idaholidar.org/), which has complied downloadable LiDAR raw data and derived products for LiDAR acquisitions in the state of Idaho. Many such online repositories of LiDAR data exist.

Summary

Light detection and ranging (LiDAR) is an active remote sensing technology that uses lasers to create three-dimensional point clouds. Airborne LiDAR flown aboard fixed-wing aircraft is the most commonly used type of LiDAR acquired for forest applications. Digital terrain models (DTMs), which describe the bare-earth surface, and lidar metrics, which describe vegetation structure, are two commonly used LiDAR products. Applications of LiDAR in forests include mapping forest characteristics such as tree density, biomass, fuels, and wildlife habitat. LiDAR data is expensive to acquire, but is becoming increasing available to the public in online repositories.

Questionnaire

- What is the difference between active and passive remote sensing?
- In LiDAR terminology, what is pulse repetition frequency, scan angle, laser wavelength, beam divergence, and laser footprint size?
- What platforms are used for LiDAR?
- How do LiDAR pulses differ from LiDAR returns?
- What do LiDAR return number and return intensity refer to?
- How is a digital terrain model (DTM) derived from LiDAR?
- Why is it important to normalize non-ground or vegetation returns before producing LiDAR metrics? How is height normalization done?
- What are some applications of LiDAR in forests?

References

Baltsavias, E.P. 1999. Airborne Laser Scanning: Basic Relations and Formulas. *ISPRS Journal of Photogrammetry & Remote Sensing*, 54: 199-214.

Evans, J.S. and Hudak, A.T. 2007. A multiscale curvature algorithm for classifying discrete return LiDAR in forested environments. *Geoscience and Remote Sensing*, 45(4): 1029-1038. Software available at https://sourceforge.net/p/mcclidar/wiki/Home/.

Jensen, J.R. 2007. Remote Sensing of the Environment: An Earth Resource Perspective, 2nd Edition, Upper Saddle River, New Jersey: Prentice-Hall, 592 p.

Lefsky, M.A., Cohen, W.B., Parker, G.G., and Harding, D.J. 2002. Lidar remote sensing for ecosystem studies. *BioScience*, 52(1): 19-30.

Maltamo, M., Naesset, E. and Vauhkonen, J. (Eds.) 2014. Forestry Applications of Airborne Laser Scanning: Concepts and Case Studies, Dordrecht, New York: Springer, 464 p.

Chapter 2: R programming

Midhun Mohan
Department of Forestry and Environmental Resources
North Carolina State University
mmohan2@ncsu.edu

2.1 What is R?

R is an open source programming language developed by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, for statistical computing, software development, visualization, and data analysis. It is based on the S language developed at Bell laboratories in the 1980s [1] and is currently developed by the R Development Core Team.

Similar to other interpreted languages, users can access the R environment through a command-line interpreter. The R platform is widely used among statisticians and data miners due to its statistical analysis capabilities, sophisticated visualization techniques, fully programmable environment, ease of data transferability, active chain of worldwide collaborators, comprehensive technical documentation, and its usability over all the major operating systems.

2.2 R and Rstudio installation

As mentioned previously one of the core advantages of R environment is that it is a free, open-source software, available for commonly used operating systems, including Windows, Mac OS X, and Linux systems. R and R studio system-specific instructions for installation are given below. RStudio is an open-source integrated development environment (IDE) for R and you need to install R first before installing RStudio.

For Windows:

Go to Comprehensive R Archive Network (CRAN), click Download R for Windows, click Base, and download the installer for the latest R version. Once it is downloaded, right-click

the installer file, and select Run as Administrator from the pop-up menu. Select the language you want to be used and choose all the defaults as listed in the installer instructions.

For Mac:

Go to Comprehensive R Archive Network (CRAN), click Download R for MacOS OX, click Base, and download the installer for the latest R version (if you are using an older version of Mac OS X such as 10.6 Snow Leopard, you will also find older versions of R). Once it is downloaded, right-click the installer file, and select Run as Administrator from the pop-up menu. Select the language you want to be used and choose all the defaults as listed in the installer instructions.

For Linux:

Go to Comprehensive R Archive Network (CRAN), click Download R for Linux, select your Linux distribution, and run the associated commands in the terminal for installing for the latest R version as given in the installation file. If your current Linux system is not compatible with the available R distributions, you might have to compile R from source code as explained in the R FAQ (Frequently Asked Questions) list.

Installing R-Studio:

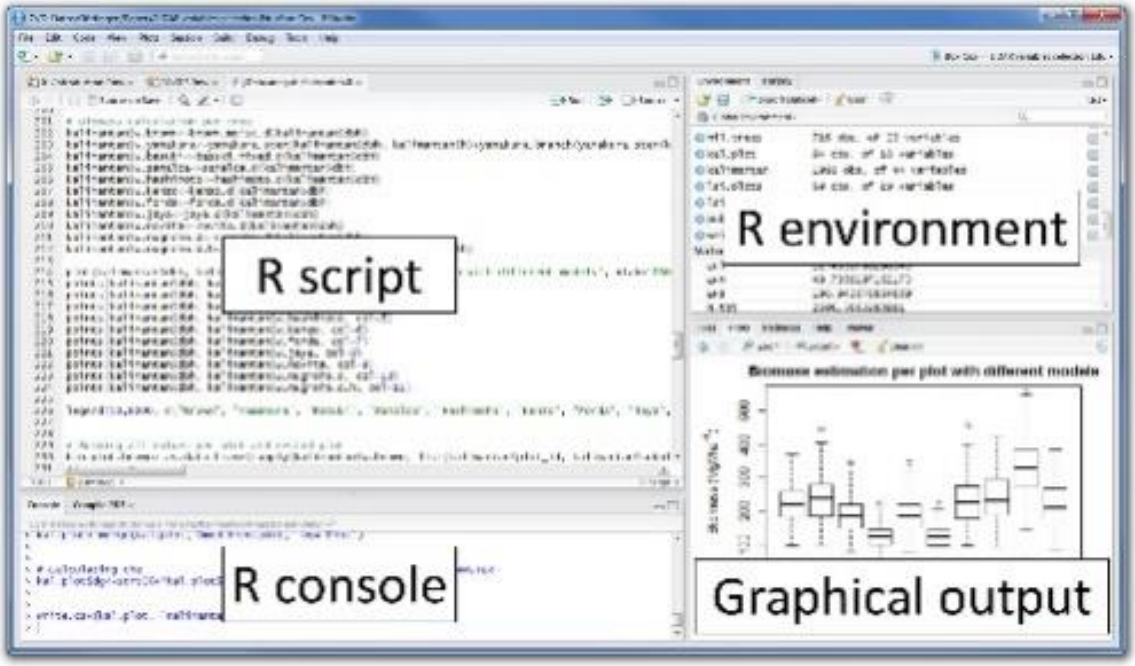
Once you have finished installing the R, go to www.rstudio.com, click on RStudio and select RStudio Desktop. Choose the version of RStudio you want and download the appropriate installer for your operating system. Once download the complete, run the installation file and choose all the default options. If you wish, you can also configure the RStudio to select the R version you want to use by choosing Tools -> Options. Make the required changes. Consult the RStudio troubleshooting guide if you encounter any technical difficulties during the installation or configuration process.

2.3 R and Rstudio interface

R is Dynamic and user-friendly

RStudio is just an open-source integrated development environment (IDE)

RStudio contains an R command line interface but with a code editor, help text, a workspace browser, and graphic output.



2.4 R packages

2.4.1 What is R package?

One of the key strengths of R is its internationally collaborative development environment comprised of renowned computational statisticians and computer engineers. Because of that thousands of user-contributed extension packages are available in R, out of which many are tailored for specific research domains. The base R you are using already has several pre-installed packages for getting you started with a variety of statistical analyses and graphics.

A R package can be defined as a collection of R functions, data, and compiled codes in a well-defined format. There are more than 4500+ available for download in CRAN. The packages are stored in a directory, referred to as “library”.

2.4.2 How to install a R package?

There are multiple ways of installing a specific package. If you have your own package stored in a folder named My-packages” in your C: Drive, you can use the following code for installation.

```
> install.packages("package_name", lib="/C:/My-packages/")
```

You can alos install packages by selecting the “install packages” option available under the tools menu in the taskbar. In the provided window you can select the repository, list the package name (s) and select the library folder for installing the package.

For built-in packages

```
> install.packages(“package_name”)
```

You need to use the “library” function for loading the package

```
> library(packagename)
```

For more details use the help() comannd

```
> help(install.packages)
```

Otherwise you can activate the package by manually selecting it from the graphics output window. After this you can start using all the commands and functionalities of a given package. The package will be active in the R environment unless you remove or uninstall it. However, you need to load the library in each R-section.

For removing a R package, use the following syntax:

```
>  remove.packages(“package_name”)
```

2.5 Data structure

Before getting in to the definition of Data Structure, let us first focus on understanding the basic building block called “variable”.

Consider the following example

Example 1:

```
> x – y =3 + 4
```

Here we see that the equation has some names (x and y), which hold values (i.e., data). These kind of placeholders, that represent and hold data area referred to as variables in R language. Data Structure refers to a specific format assigned for organizing and storing these kinds of data and information so that they can be used efficiently.

In the above example the variables x and y can be anything ranging from real numbers to binary digits to integral entities. For solving the equation, we need to assign the variables to a specific kind of values they can take i.e., we need to know how the data structure is stored in memory, and this is referred to as **Data Type**. We can use the *Typeof()* function for

understanding the storage mode of an R object. Another important element of R language is class. Class of an objects tells you what information is being contained in it and how this can be implemented into other functions. For finding the class of an R object we use the *Class()* function. Also, for inserting comments while writing R-code you can use "#" symbol. Everything written after the # symbol would be ignored during execution of the code.

2.5.1 Vector.

The simplest and foundational data structure in R is the vector. There are two kind of vectors namely atomic vectors and lists. In an atomic vector, all the elements should be of the same type, whereas a list can have any different types of elements. Type, Length. and Attributes are some properties that are common to both the vector types. The Typeof() command as mentioned earlier gives an idea on the data type, the length() command tells you about how many elements are contained in the vector and the attributes() command provides additional arbitrary metadata. There are several ways to develop vectors in R and the following examples covers all the commonly used methods.

The following examples returns a sequence/vector of numbers

Example 2:

```
> 1:10
```

Output:

```
[1]  1  2  3  4  5  6  7  8  9 10
```

Example 3:

```
> seq(1, 10, 1) #R functions are case sensitive
```

Output:

```
[1]  1  2  3  4  5  6  7  8  9 10
```

Example 4:

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Here "c" stands for combine, and the combine function combines the input parameters into a single vector.

Output:

```
[1]  1  2  3  4  5  6  7  8  9 10
```

You can also have strings assigned to a variable using the combine function

Example 5:

```
> v <- c("a", "B", "c")
> v
```

Output:

```
[1] "a" "B" "c"
```

Within these vectors, elements have to be of the same type (i.e. numeric, strings, etc.), otherwise few might be converted to a common format maintain homogeneity. In R you can also access values inside a vector by indexing.

Example 6:

```
> v[3]
```

Output:

```
[1] "c"
```

Example 7:

```
> v[1:3]
```

Output:

```
[1] "a" "B" "c"
```

Example 8:

```
> v[-1]                                    #everything except the
first element
```

Output:

```
[1] "B" "c"
```

2.5.2 Matrices

Matrices are two-dimensional vectors and can be created using the matrix() function.

Example 9:

```
> m <- matrix(data = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), nrow = 2, ncol = 5)
> m
```

output:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

2.5.3 Arrays

Arrays are multi-dimensional vectors and can be created using the array() function.

Example 10:

```
> a1 = array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), dim = c(2, 5))
> a2 = array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim = c(2, 3, 2))
> a1
> a2
> a1[1, 1]                                    #first row and first
column
> a1[1, ]                                     #first row and every single
column
```

Output:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

[1] 1

[1] 1 3 5 7 9
```

2.5.4 List

A list is a generic vector containing other R objects. These elements can even be heterogenous unlike atomic vectors. Also, within a list, each element can have a specific name and you can use indexing to filter results just like we did with arrays.

Example 11:

```
> l <- list(element1 = “One”, element2 = 2)
> l[1]
> l[1:2]
```

Output:

```
$element1
[1] "One"

$element2
[1] 2
```

Few other ways to access information

Example 12:

```
> l$element1
> l[“element1”]
```

Output:

```
[1] "One"

[1] "One"
```

It is also possible to create a list of lists.

Example 13:

```
> l1 = list(e1=1, e2=2, e3=3)
> l2 = list(e4=4, e5=5)
> l3 = list(List1 = l1, List2 = l2)
> l3
```

Output:

```
$List1
$List1$e1
[1] 1

$List1$e2
[1] 2
```

```
$List1$e3
[1] 3


$List2
$List2$e4
[1] 4

$List2$e5
[1] 5
```

2.5.5 Data frames

A data frame is a two-dimensional array like structure having a list of named vectors. In this case, all the vectors inside a data frame function has to be of equal lengths, and individual elements inside each vector has to be of the same type. You can create a data frame using the data.frame function. Data frames are considered very useful as they let you load data easily from large databases as well as from files of .csv and xls formats.

Example 14:

```
> df <- data.frame(c(1, 2, 3), c("one", "two", "three"))
> df
```

Output:

```
  c.1..2..3. c..one....two....three..
1          1                      one
2          2                      two
3          3                    three
```

2.6 Conditional Statements

A conditional statement checks whether a condition is true or false, and perform a certain action based on whether the condition is true or false. This is very useful for enhancing decision making and usually we use logical operators (such as <, >, <=, >=, ==, !=, |, &) for checking a condition. Most commonly used conditional statements are as follows:

2.6.1 if()

The syntax of if() statement is:

```
if (logical_condition) {
        statement
}
```

In this case, the operation is done only when the initial condition is met.

Example 15:

```
> x <- 2
> if(x > 0){
+        print("x is Positive")
+ }                                                    #Here R adds the "+" symbol automatically
```

Output:

```
[1] "x is Positive"
```

One notable disadvantage with the if statements is that they do not work with vectorised arguments

Example 16:

```
> Animal <- c("Zebra", "Owl", "Whale", "Owl" )
> if(Animal == "Owl") {                              #note that here it is "==" as opposed to "="
+        print ("I Fly!")
+ }
```

Output:

```
Warning message:
In if (animal == "Owl") { :
  the condition has length > 1 and only the first element will be used
```

Here only the first element is being checked i.e. the code executes only for "Zebra". Hence we need to use ifelse() or Loop statements for overcoming the drawbacks of if statement, and you will get to know more about them later in this chapter.

2.6.2 else()

The else() statement is an extension to the if() statement and is only evaluated if the condition is FALSE. They are also not good with vectorised arguments as only the first element would be evaluated during execution.

```
if (logical_condition) {
        statement
} else {
        different statement
}
```

Example 17:

```
> x <- -3
```

```
> if(x > 0){
+   print("x is Positive")
+ } else {
+   print("x is not Positive")
+ }
```

Output:

```
[1] "x is not Positive"
```

2.6.3 ifelse()

An ifelse() statement can be used instead of separate if and else statement and more efficient while working with vectorised arguments. The syntax of the ifelse() statement is as follows:

ifelse (logical_condition, if TRUE, Statement if FALSE)

Example 18:

```
> Animal <- c("Zebra", "Owl", "Whale", "Owl" )
> ifelse(Animal == "Owl", "I Fly!", "I can't Fly!")
```

Output:

```
[1] "I can't Fly!" "I Fly!"      "I can't Fly!" "I Fly!"
```

Here while executing the code, whenever an "Owl" was encountered the output was "I Fly!" and for "Zebra" and "Whale", the code printed "I can't Fly!" as the output.

2.6.4 Nested if...else ()

We can also nest as many if else statements as we want in R by combined use of if(), else if() and else() statements. The syntax for a nested if…else() statement is as follows:

```
if (logical_condition1) {
        statement1
} else if (logical_condition2) {
        statement2
} else if (logical_condition3) {
        statement3
} else
        statement4
```

Example 19:

```
> x <- 0
> if(x > 0){
+   print("x is Positive")
+ } else if(x < 0){
```

```
+   print("x is Negative")
+ } else {
+   print("x is Zero")
+ }
```

Output:

```
[1] "x is Zero"
```

2.6.5 switch()

Apart from multiple if else statements there is also something called the "Switch" Statement. A switch statement generally allows a variable to be tested for a condition against a list of values.

Switch (logical_condition, case1, case2, case3....)

Here, each case represents a specific value and the variable being switched on is checked for each case.

Example 20:

```
> Ability <- "swim"
> switch(Ability,
+        "run"={
+          print("Zebra is running!")
+        }
+        ,"fly"={
+          print("Owl is flying!")
+        }
+          , "swim"={
+            print("Whale is swimming!")
+        }
+           , print("Not enough data"))
```

If we have ability as anything other than "run", "fly" or "swim" we will get the value listed in the final option i.e., "Not enough data"

Output:

```
[1] "Whale is swimming!"
```

Switch() statement can also be used for printing an output by refering its index

Example 21:

```
> switch(3, "Zebra", "Owl", "Whale", "Owl")
```

Output:

```
[1] "Whale"
```

Also, you should keep in mind that indexing in R language starts from 1 and not 0.

2.7 Loops

For performing same operations again and again, we can implement Loops statements in R.

2.7.1 for loops

One of the most common type of loops is the for loops:

```
for (index variable in vector) {
        statements
}
```

Here the index variable can be anything, though "i" is the most commonly used variable. The code inside the curly brackets would be repeated once for each element of the specified vector.

Example 22:

```
> for (i in 1:4) {
+   print (i ^ 2)
+ }
```

Output:

```
[1] 1
[1] 4
[1] 9
[1] 16
```

To make the output of the loop reusable we can store it in a variable and use it later used in a function or for performing other calculations. In the following example, we first define a empty variable storage and use it as a container to store the output from the loop.

Example 23:

```
> storage = c()
> for(i in 1:4){
+   storage[i]=i^2
+ }
> storage
> mean(storage)
```

Output:

```
[1]  1  4  9 16

[1] 7.5
```

Now let us take one of our previous examples from the if statement and see how looping solves the problem of vectorised arguments

Example 24:

```
> for(i in 1:4){
+   if(Animal[i]=="Owl")
+     print("I Fly!")
+   else
+     print("N/A")
+     }
```

Output:

```
[1] "N/A"
[1] "I Fly!"
[1] "N/A"
[1] "I Fly!
```

Just like nested if…else statements we can also have nested for loops

```
For (variable1 in vector1) {
        For (variable2 in vector2) {
                For (variable n in vector n) {
                        statements
                } #end of nth loop
        } #end of 2nd loop
} #end of 1st loop
```

Example 25:

```
> for(i in 1:2){
+   for(j in 1:3){
+     print(i+j)
+   }
+ }
```

Output:

```
[1] 2
[1] 3
[1] 4
[1] 3
[1] 4
[1] 5
```

Two other commands that are very useful within loops are the “next” command and “break” command. “Next” command is used to skip the current iteration of the loop without terminating it. However, you have to be very careful on where you insert the print statement or else your results might vary as you can see from the following examples.

Example 26:

```
> for (i in 1:5) {
+   if (i >= 4)
```

```
+     next                    #next statement before the print statement
+   print (i)
+ }
```

Output:

```
[1] 1
[1] 2
[1] 3
```

Example 27:

```
> for (i in 1:5) {
+   if (i >= 4)
+   print (i)
+   next                      #next statement after the print statement
+ }
```

Output:

```
[1] 4
[1] 5
```

Break command let the control flow jump outside the loop. Following example demonstrates its usage:

Example 28:

```
> x <- 1:5
> for (value in x) {
+ if (value == 4) {
+ break
+   }
+   print (value)
+ }
```

Output:

```
[1] 1
[1] 2
[1] 3
```

2.7.3 while loops

In case of while loops we do not have to specify the number of iterations in the loop. We just want the loop to keep running until some statement is TRUE.

```
While (logical_condition) {
        statement
}
```

Example 29:

```
> x <- 1
> while (x < 5) {                                    #conditional stateme
nt
+   print (x)
+   x <- x + 1
+ }
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
```

Here also you should be careful on where you insert the print statement or else your output will vary like in the example below.

Example 30:

```
> while (x < 5) {
+   x <- x + 1
+   print (x)
+ }
```

Output:

```
[1] 2
[1] 3
[1] 4
[1] 5
```

In case, if you insert the print statement outside the loop, you will get only the last value.

Example 31:

```
> x <- 1
> while (x < 5) {
+   x <- x + 1
+ }
> print (x)
```

Output:

```
[1] 5
```

As we did with for loops, you can store the value of while loops for later usage and have them nested if required. You can also nest a for loop inside a while loop and vice versa. However, we should be extra careful while working with While loops as you can often create infinite loops.

Example 32:

```
> storage <- c ()
> x <- 1
```

```
> while (x < 10) {

+   storage = c (storage, x)
+   x <- x + 1
+ }
> storage
```

Output:

```
[1]  1  4  9 16  1  2  3  4  5  6  7  8  9
```

Example 33:

```
> i<-1
> j<-1
> while(i<=2){
+   while(j<=3){
+     print(c(i,j))
+     j=j+1
+   }
+   i=i+1
+ }
```

Output:

```
[1] 1 1
[1] 1 2
[1] 1 3
```

2.7.4 repeat loops

Even though For loops and While loops are the most commonly used loop statements, R also offers you another kind of loops called Repeat loops. These loops are very useful in case of unconditional statements.

Example 34:

```
> x<-1
> repeat {
+   print(x)
+   x = x+1
+   if (x == 5){
+     break
+   }
+ }
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
```

2.8 Functions

Functions are R objects created using the function () directive and they have their class as "function". In several cases, functions can also be functions of functions i.e., they can be passed as arguments to other functions as well as nested within a function. These are very helpful for avoiding repetition of codes in multiple areas. The return value of a function is the last expression in the function body to be evaluated.

2.8.1 Creating a new function

For creating a new function, follow the syntax:

```
new_function <- function(parameters ){
        statements
        return(object)
}
```

Here, the object returned can be of any data type. However, objects are local to a particular funtion. Instead of return you can also use print command for getting the output.

Example 35:

```
> avg_func <- function(x) {
+   sum.x = sum(x)
+   length.x = length(x)
+   avg.x = sum.x/length.x
+   return(avg.x)
+ }
> avg_func(c(1,2,3,4))
```

Output:

```
[1] 2.5
```

Here, you have to make sure that the parameter is sent as a single vector otherwise you will get error

Example 36:

```
> avg_func(1, 2, 3, 4)
```

Output:

```
Error in avg_func(1, 2, 3, 4) : unused arguments (2, 3, 4)
```

2.9 Read and write files

Data often comes in various formats and R uses the working directory for reading and writing these data. Two of the commonly used data types in R are text (ASCII) file and comma separated value file (.csv). R can also read files in other formats such as SAS, SPSS and SQL. However, considering the scope of this text book we would be focusing our examples on ASCII and .csv files.

2.9.1 Data input

First, we have to set the working directory using the setwd() command

```
> setwd ("C:/data")
```

For finding the current directory, the command getwd() can be used and use read.csv command for reading a csv file.

Example 37:

```
read.csv("filename.csv") -> new_data
```

Here we assigned the data in filename.csv to a variable "new_data" and if you check the class of this variable you can see that it is imported as a data frame by default. If we are calling a file outside the directory we will have to specify the entire file path inside the brackets

Example 38:

```
read.csv("C:/data/new/new_file.csv")
```

we can also create a data frame right away using the read.table option. This is useful if you wish to read the data in tabular form. The following example shows how to import a text file named "filename.txt" from the directory.

Example 39:

```
read.table("filename.txt", fill = T, header = F)
```

Here "fill = T" command makes sure that missing value errors are avoided and "header=F" command let R know that the first row is not considered as variable name line.

If the columns in the datasets are the same you can combine vectors, matrices or data frames using the rbind(). Another recently developed function for loading datasets is the "fread" function. It works similar to the table function but is faster and more convenient. For importing excel files such as .xlsx files you will need to install additional packages.

.

2.9.2 Data output

You can write the data you have into a different file using the write.csv function.

Example 40:

```
write.csv (new_data, exported_file)
```

Now we can see that a new file named "exported_file" will be generated in the directory. However, we can see that there is an additional column containing the row numbers in the file, which is redundant. This is because R by default include the row names when it exports a data frame. For removing the additional row, we can modify our code by turning off the row names.

```
write.csv (new_data, exported_file, row.names = F)
```

Now we can see that the exported file looks the way we had wanted (fig 2). Instead if you just want to store R objects in a file, you can use the save function.

Example 41:

```
save (new_data, file="new_data.Rdata")
```

2.10 Basic statistics

There are a lot of built-in functions in R that helps you in performing all the basic statistical operations and here we would be going through a few of the most commonly used commands.

2.10.1 Mean

For finding mean we use the "mean" function.

Example 42:

```
> a <- 1:10
> mean(a)
```

Output:

```
[1] 5.5
```

2.10.2 Standard deviation and variance

For finding standard deviation we use the "sd" function and for variance, we use "var" function.

Example 43:

```
> a <- 1:10
```

```
> sd(a)
```

Output:

```
[1] 3.02765
```

Example 44:

```
> a <- 1:10
> var(a)
```

Output:

```
[1] 9.166667
```

2.10.3 Range

For finding range we make use of max and min function in R.

Example 45:

```
> a <- 1:10
> range <- max (a) - min (a)
> range
```

Output:

```
[1] 9
```

2.10.4 Percentiles

For determining the percentiles, we use the quantile function in R and the syntax is:

> quantile(a, n)

Here "n" stands for the nth percentile of the observation variable a and it is the value that cuts off the first n percent of the data values when it is sorted in ascending order.

The following code gives you the 5^{th} percentile, 25^{th} percentile, 50^{th} percentile (i.e., median) and 99^{th} percentile of a.

Example 46:

```
> a <- 1:10
> quantile(a, c(0.05, 0.25, 0.5, .99))
```

Output:

```
  5%  25%  50%  99%
1.45 3.25 5.50 9.91
```

You can also use the quartile() function in R if you are only interested in finding the quartiles.

2.10.5 Summary

You can get the min, max, 1st quartile, median, 3rd quartile and mean in one step using the summary() command.

Example 47:

```
> a <- 1:10
> summary (a)
```

Output:

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

2.11 Graphics

One of the reasons why R is very common among the academic and professional workers is because of the exceptional graphical capabilities it offers. R provides you with number of graphical visualizations tools and packages, out of which, several of them are loaded by default. You can easily create, replicate, and modify beautiful publication quality graphs and plots with few lines of R code. Few of the most popular R packages on visualizations are ggplot2, ggvis and lattice. The following sections will provide you an introduction to the basic graphic capabilities offered in R programming environment.

2.11.1 Histogram and density plots

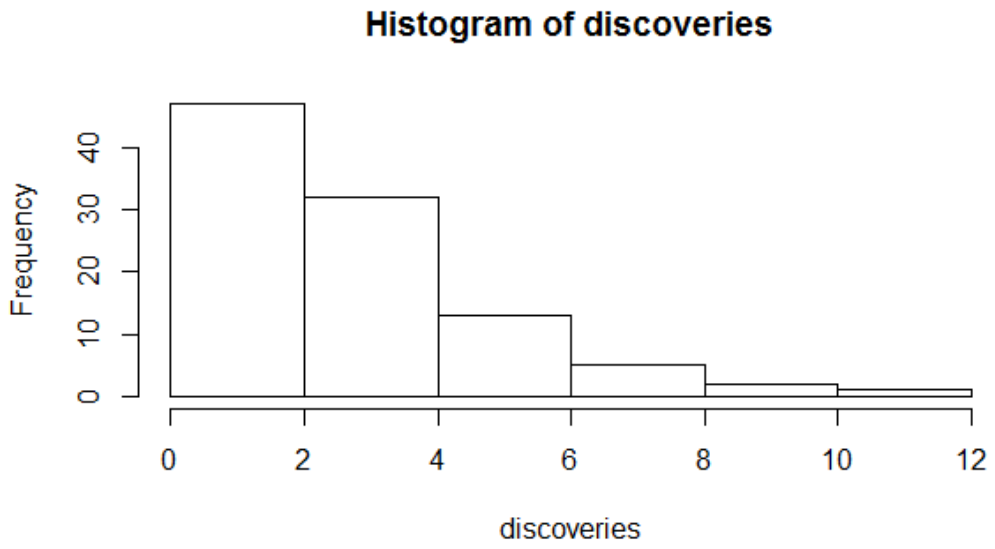
A histogram is a visual representation of the distribution of a dataset that allows you to see the spread of data by separating information into bins along with its respective frequency. In this way you can also identify the possible outliers within your dataset. A histogram comprises of an x-axis, a y-axis and various bars (bins) of different heights. The y-axis tells about the frequency of x-axis values occurrence in the data, while the bars group ranges of values or continuous categories on the x-axis.

For generating histograms, we use the “hist” function available in R. The following example illustrates the usage of hist function in generating histogram for a built-in R dataset “discoveries”. The plot commands help produces the plot in the graphics plot window.

Example 48:

```
> hist(discoveries)
```

Output Image:

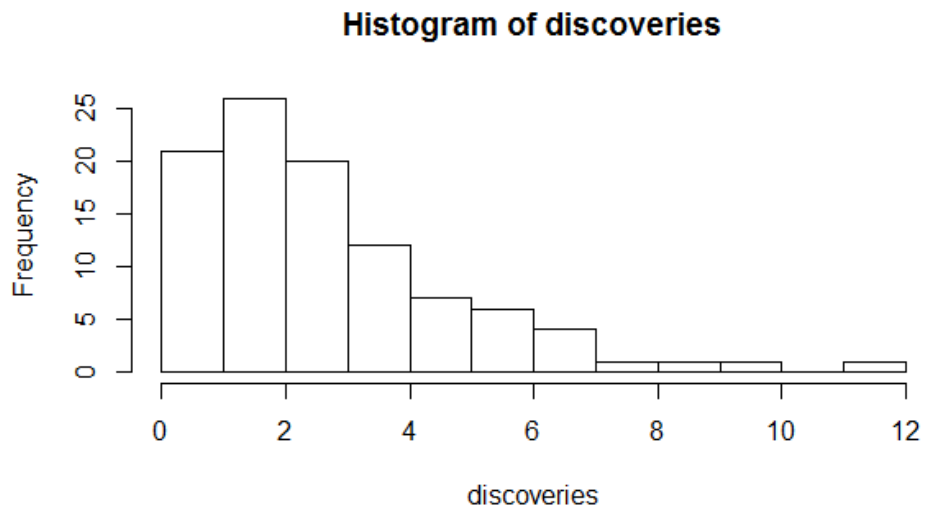


you can also increase the number of bins using the breaks function

Example 49:

```
> hist(discoveries, breaks = 10)
```

Output Image:

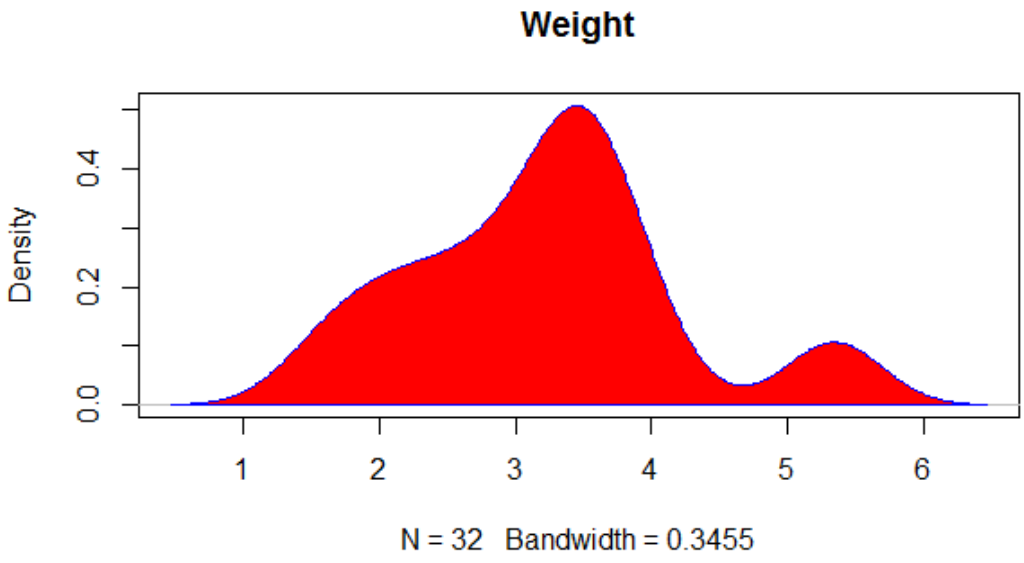


Histograms are based on estimating a local density; in their case, points are local to each other if they fall into the same bin. However, bins are a rather inflexible way of implementing local density estimation. Hence often kernel density plots are considered as alternatives and we can create density plots in R easily using the “density()” function.

Example 50:

```
> d <- density(mtcars$wt)                                    #mtcars is a built-in
R >
plot(d, main = "Weight")
polygon(d, col = "red", border= "blue")

#col command for fill colour and border command for border colour
```

Output Image:



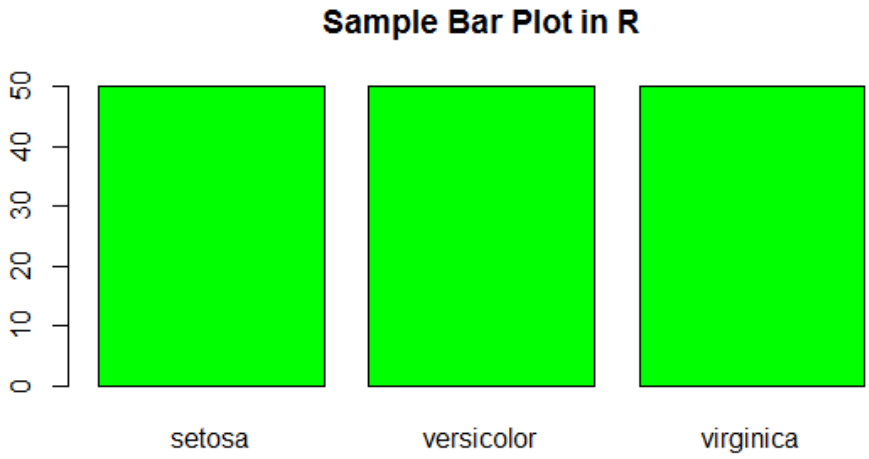
2.11.2 Bar plots

Bar plots are primarily used to graphically represent the distribution of a categorical variable and we can use barplot function in R for this purpose.

Example 51:

```
> bp <- table(iris$Species)                                    #iris is a built-in R dataset
> barplot(bp, main = "Sample Bar Plot in R", col = "green")

#main function helps you include a title name
```

Output:



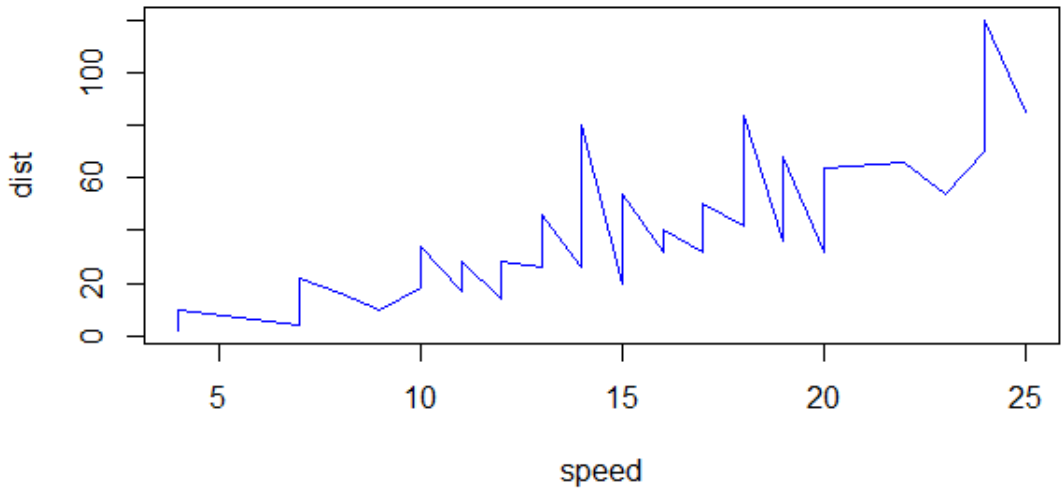
2.11.3 Line Charts

We can create line charts using the plot() function. By defining the type as “l” we specify that we want a line chart. Instead if we assign type as “o”, then we will get over-plotted points in addition to the lines.

Example 52:

```
> plot(cars, type="l", col="blue")
```

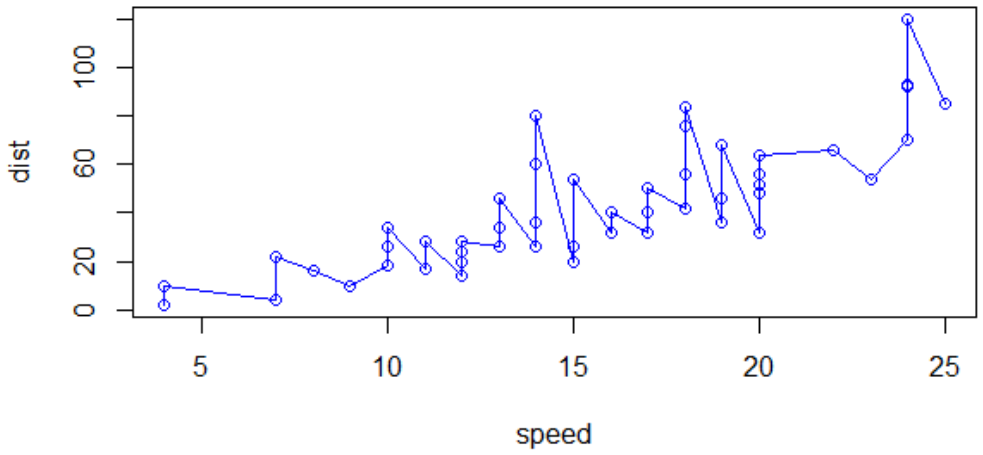
Output Image:



Example 53:

```
> plot(cars, type="o", col="blue")
```

Output Image:



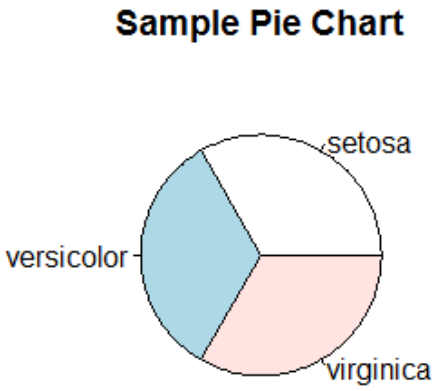
2.11.4 Pie Charts

We can create pie charts using the pie command and as earlier we can add a title using the main() argument.

Example 54:

```
> pc <- table(iris$Species)
> pie(pc, main = "Sample Pie Chart")
```

Output image:

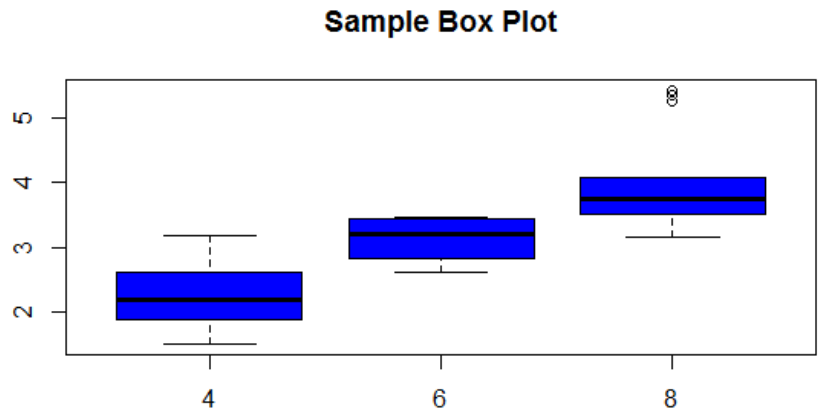


2.11.5 Boxplots

Box plots can be created for individual variables or variables by group using the boxplot() command.

Example 55:

```
> boxplot(mtcars$wt~mtcars$cyl, main ="Sample Box Plot", col ="blue")
```



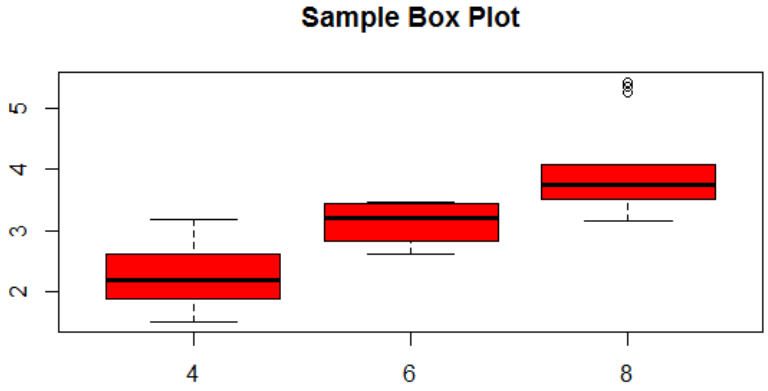
While using multiple variables instead of mentioning the data name each time you can instead use the following syntax where data is listed separately:

```
> boxplot(variable1~variable2, data = data_name)
```

Example 56:

```
> boxplot(wt~cyl, data = mtcars, main = "Sample Box Plot", col ="red")
```

Output Image:



2.11.6 Scatter Plots

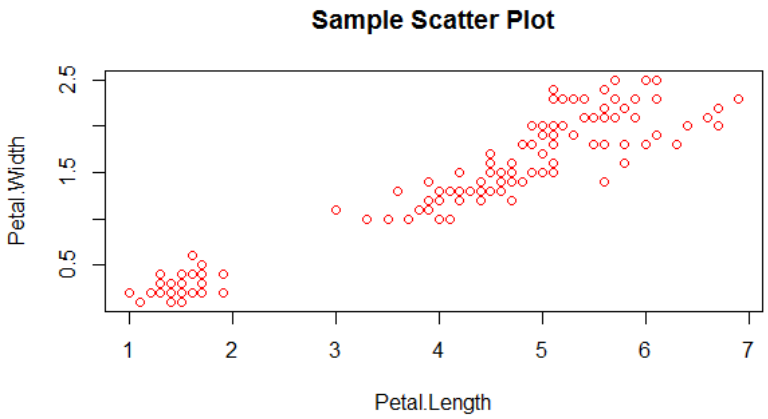
Scatter plots are a way to represent two or more numeric variables coming from same data set and their relationship with one another. They are usually made in conjunction with correlation or regression analysis. For creating the scatterplots also we use the “plot()” function but the syntax is bit different than that of line charts.

> plot(variable1~variable2, data = data_name)

Example 57:

> plot(Petal.Width~Petal.Length, main = “Sample Scatter Plot”, data = iris, col = “red”)

Output Image:

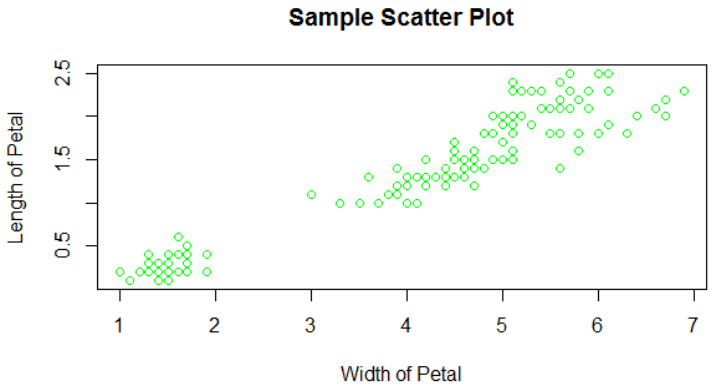


By default, R shows the name in x and y axis as listed in the dataset, but we can customize the labels using xlab and ylab functions.

Example 58:

>plot(Petal.Width~Petal.Length, data = iris, main = "Sample Scatter Plot", xlab = "Petal Width", ylab = "Petal Length", col ="green")

Output Image:



2.11.7 Exporting graphics

In addition to exploring the datasets visually through plots, you can also export and publish your results in R. For that you can save your plots to a file in R and then import this graphic

document into another file having formats such as JPG, PNG or PDF. Finally, you can close the function using the dev.off() command.

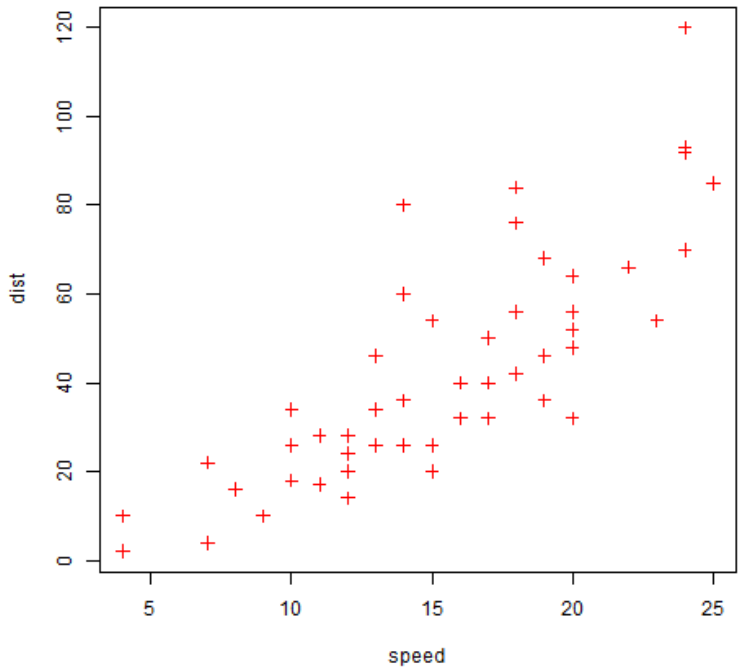
First, we need to set the working directory where we want our file to be generated.

```
> setwd("C:/Users/CASMike/Desktop”)
```

After that we use the “png() function to generate a file named cars_plot.png in our working directory. Instead of png(), you can use pdf() and jpg() function as well depending on your requirements.

```
> png(filename="cars_plot.jpeg")
> plot(dist ~ speed, data = cars, col = "red", pch = 3)
> dev.off()
```

Output file:



2.11.8 Clearing Plots in R/RStudio

Once you have exported all your results, you can clear the plot section using the following code:

```
dev.off(dev.list()["RStudioGD"])
```

Questionnaire

1. - What are some advantages of using Rstudio over R?
2. - How can you debug and test R programming code?
3. - What do you think is the best graphical representation for categorical variables?
4. - How to create Histograms for grouped data? How to further calculate medians and quartiles from the histograms?
5. - Can you think of some spatial applications using R programming language?
6. - Do you think there is a way that you can build intereactive web applications in R? (refer: Rshiny)
7. - How to scrape data from web using R?
8. - Can you come up with some packages for performing Visual Analysis in R?
9. - How can you create a R package?
10. - How can you reduce the complexity of multiple nested groups? Check out the function “lapply()”
11. - How to vectorize ensted loops in R?
12. - How to add elements to apre-existing list in R?
13. - How can we address the missing values in R language?
14. - List the homogenous and heterogenous data structures available in R
15. - How can you create a table in R language without using external files?
16. - What is the memory limit in R?
17. - When we concatenate a number and a character, what will be the class of the resulting vector?
18. - How to save multiple plots to jpg in R?
19. - Can you write a code that changes the color and symbol of variables in a scatter plot based on grouping?
20. - What do you think the output of the function `Z <- X*Y` be, if:

```
X <- 5:7
Y <- c(3, 2)
```

21. - Find and interpret the output of the following code

```
> repeat {
+   print(x)
+   x = x+1
+   if (x <= 5){
+     break
+   }
+ }
```

References

1. Paradis, E. (2002). R for Beginners
2. Rossiter, D. G. (2012). Introduction to the R Project for Statistical Computing for use at ITC. International Institute for Geo-information Science & Earth Observation (ITC), Enschede (NL), 3, 3-6.
3. Ihaka, R., & Gentleman, R. (1996). R: a language for data analysis and graphics. Journal of computational and graphical statistics, 5(3), 299-314.
4. R Core Team. R: A Language and Environment for Statistical Computing; R Foundation for Statistical Computing:
Vienna, Austria, 2017. Available online: http://www.R-project.org (accessed on 20 October 2016).
5. Matloff, N. (2011). The art of R programming. No Starch Pess, 3.
6. Cotton, R. (2013). Learning R: A Step-by-Step Function Guide to Data Analysis. " O'Reilly Media, Inc.".

Chapter 3 rLiDAR: An R package for reading, processing and visualizing lidar data

Carlos Alberto Silva

Carine Klauberg

USDA Forest Service

Rocky Mountain Research Station-RMRS

Carlos_engflorestal@outlook.com

Carine_klauberg@hotmail.com

3.1 Package description

The R package rLiDAR is an open-source tool for reading, processing and visualizing small sets of airborne lidar data. It was developed in 2014 and made publicly available on the Comprehensive R Archive Network (CRAN) in 2015. The rLiDAR package presents eight functions that allow ecologists, forest managers and scientists to i) import and visualize lidar data (e.g. Figure A1a); ii) smooth, detect and delineate individual trees on the lidar-derived canopy height model (e.g., Figure A1b-d), iii) compute lidar metrics at plot and crown levels (e.g., Figure A1d-g), and iv) plot virtual forest stands (e.g., Figure A1h1-h3). Since we developed the rLiDAR, additional open-source R packages for lidar data processing and visualization have been developed, such as lidR (Roussel et al. 2017) and ForestTools (Plowright 2017). rLiDAR was specifically developed to support the analysis presented in the Silva et al. (2016) with the goal of providing and testing a new framework for imputing individual tree attributes from field and lidar data in longleaf pine forests. However, the rLiDAR has general applicability to other forests in other ecosystems, and we encourage users to test it broadly.

```
#=============================================================#
# Install and load the rLiDAR package
#=============================================================#
install.packages("rLiDAR")
require(rLiDAR)
```

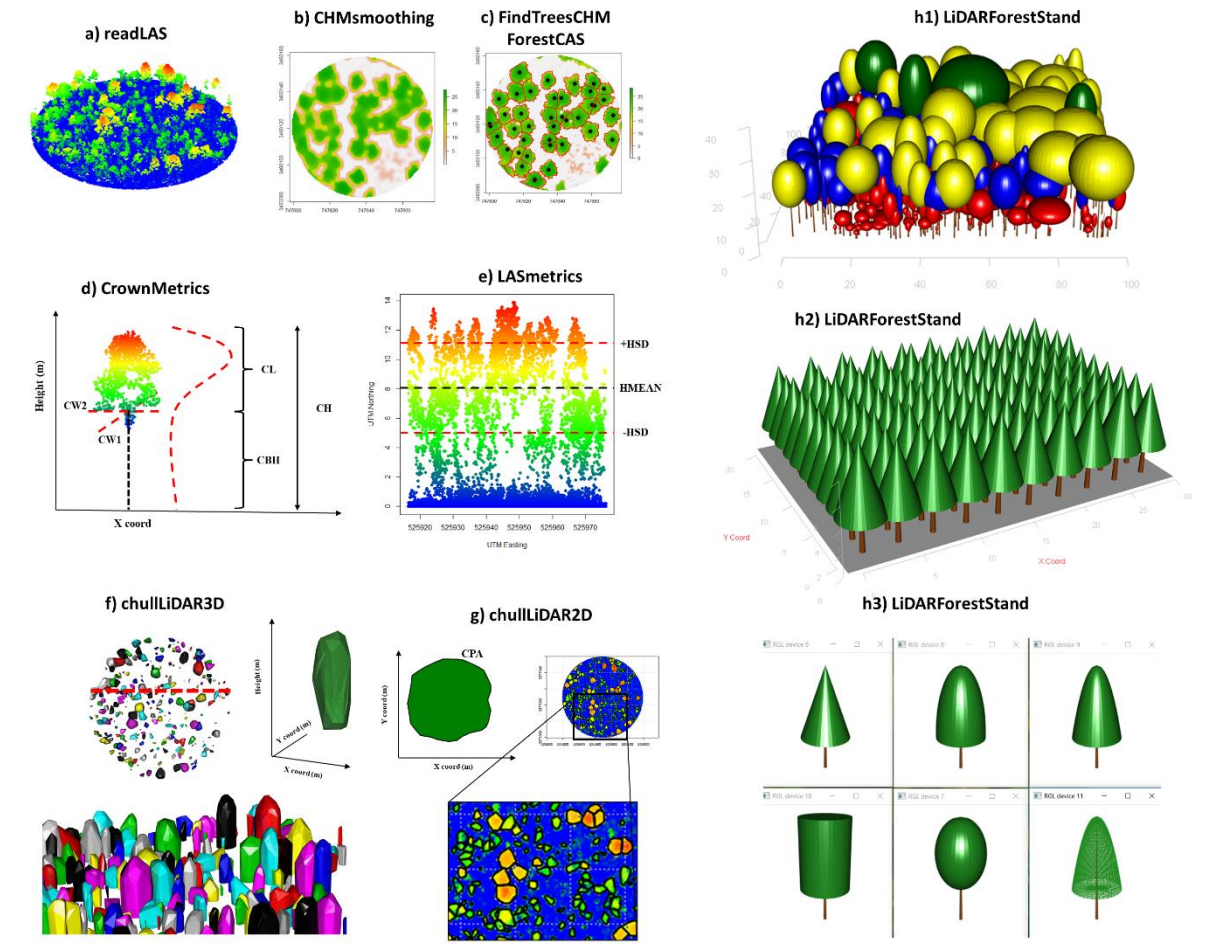


Figure A1. Figure A1. rLiDAR: An R package for reading, processing and visualizing lidar data

3.2 chm Lidar-derived Canopy Height Model - (CHM)

Description

Lidar-derived Canopy Height Model - (CHM)

Usage

data(chm)

Format

The format is:'RasterLayer'

Details

The lidar-derived-CHM provided as an example dataset is from a longleaf pine forest at Eglin AFB, USA.

Source

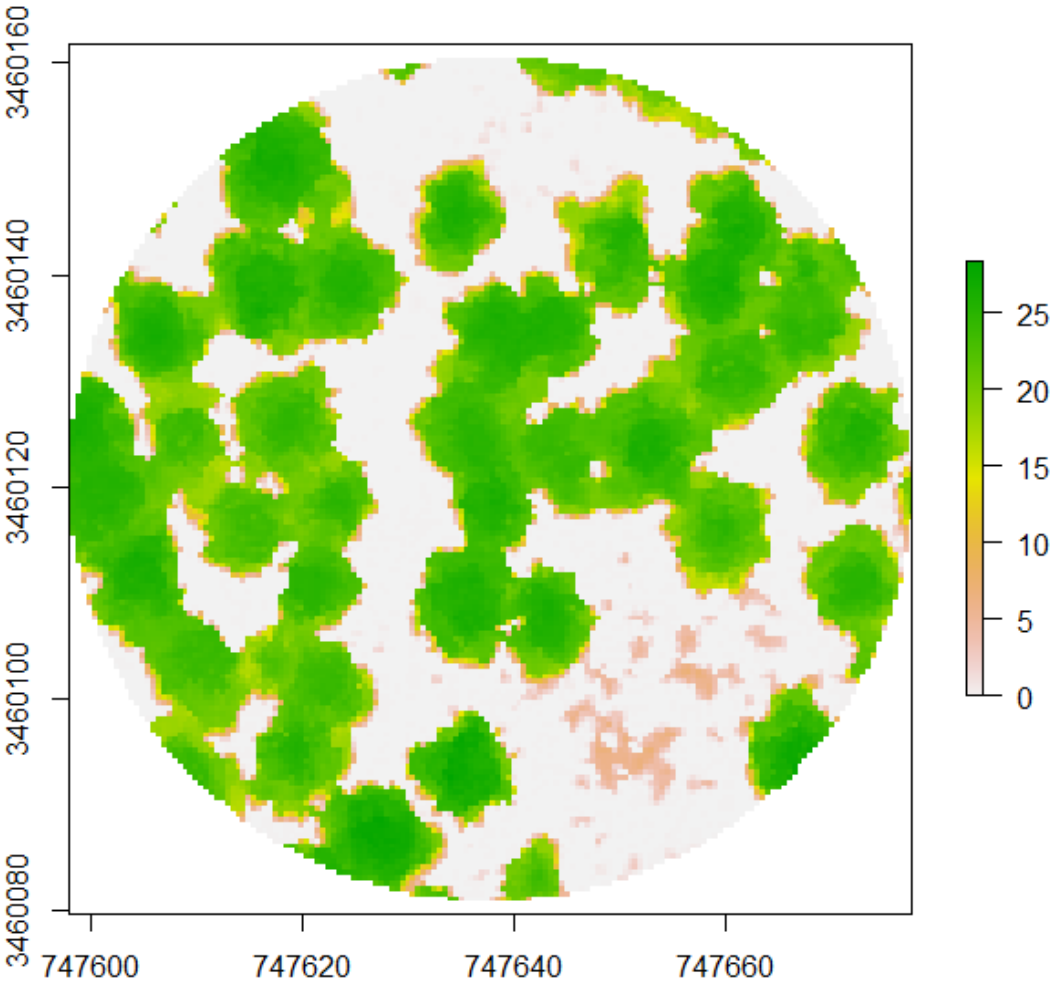
The CHM was generated using Lastools software. The lidar data collection was funded by a grant (11-2-1-11) from the Joint Fire Science Program: Data set for fuels, fire behavior, smoke, and fire effects model development and evaluation-the RxCADRE project. R. Ottmar, PI; multiple Co-Is."

References

USDA Forest Service. Rocky Mountain Research Station - RMRS - Moscow, Idaho, USA.

Examples

```
data(chm)
plot(chm)
```


3460160
3460140
3460120
3460100
3460080
747600
747620
747640
747660
25
20
15
10
5
0

3.3 CHMsmoothing Lidar-derived Canopy Height Model (CHM) smoothing

Description

Lidar-derived Canopy Height Model (CHM) smoothing is used to eliminate spurious local maxima caused by tree branches.

Usage

CHMsmoothing(chm, filter, ws, sigma)

Arguments

chm	A lidar-derived Canopy Height Model (CHM) RasterLayer or SpatialGrid- DataFrame file.
filter	Filter type: mean, median, maximum or Gaussian. Default is mean.
ws	The dimension of a window size, e.g. 3,5, 7 and so on. Default is 5.
sigma	Used only when filter parameter is equal to Gaussian, e.g. 0.5, 1.0, 1.5 and so on. Default is 0.67.

Value

Returns a CHM-smoothed raster.

Author(s)

Carlos Alberto Silva.

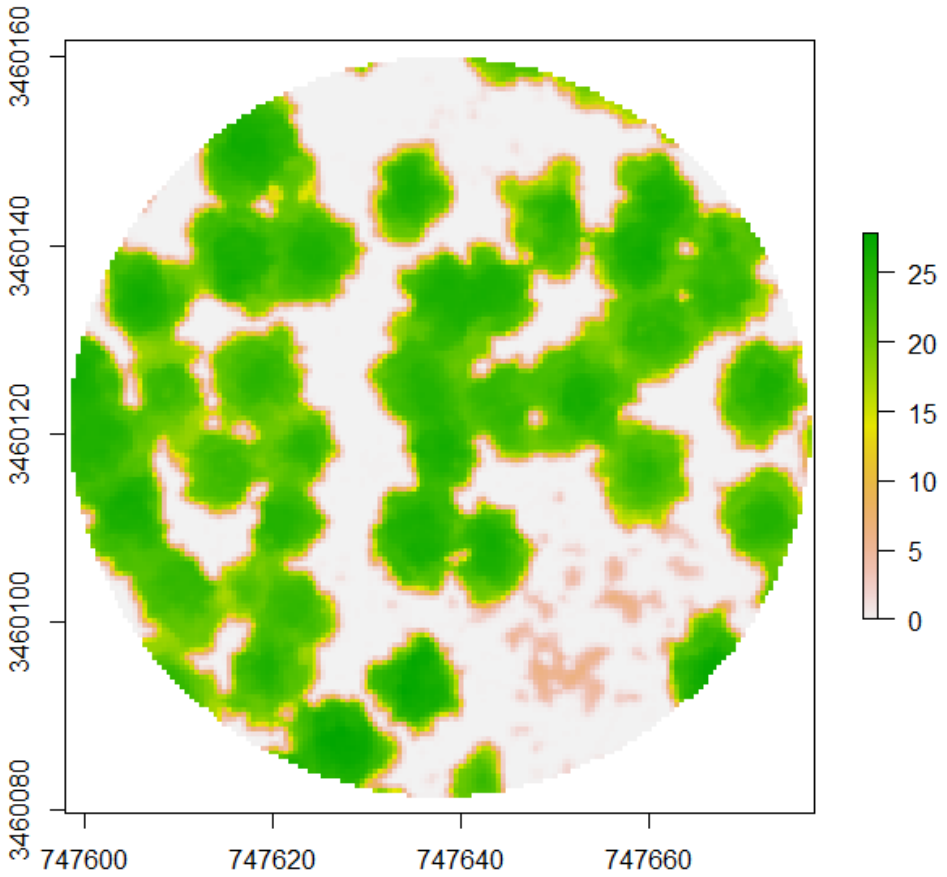
See Also

focal in the *raster* package.

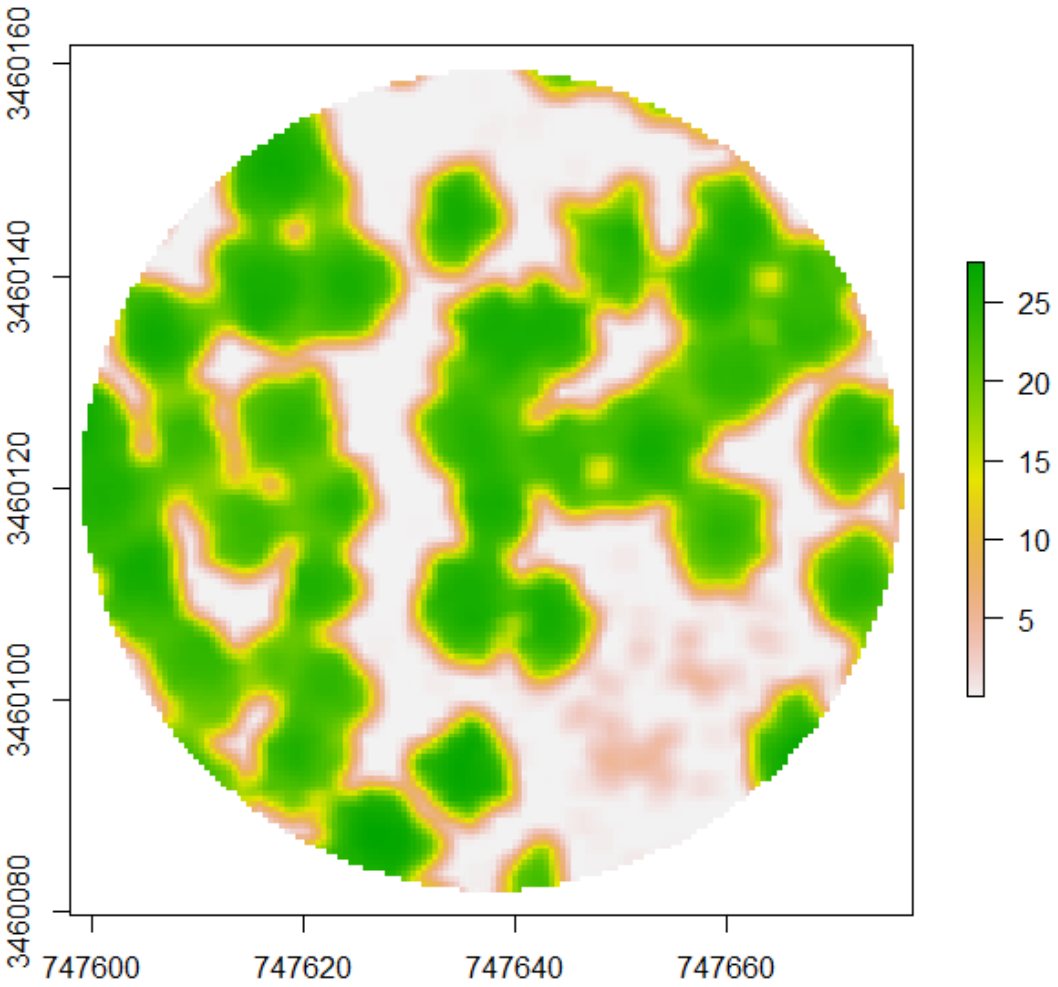
Examples

```
#============================================================#
# Importing the lidar-derived CHM file
data(chm) # or set a CHM. e.g. chm<-raster("CHM_stand.asc")
```

```
#=================================================================#
# Example 01: Smoothing the CHM using a Gaussian filter
#=================================================================#
# Set the ws:
ws<-3 # dimension 3x3


# Set the filter type
filter<-"Gaussian"


# Set the sigma
value sigma<-0.6


# Smoothing CHM
sCHM<-CHMsmoothing(chm, filter, ws, sigma)
plot(sCHM)
```



```
#=================================================================#
# Example 02: Smoothing the CHM using a mean filter
#=================================================================#
# Set the ws:
ws<-5 # dimension 5x5


# Set the filter type
filter<-"mean"
# Smoothing and plotting lidar-derived CHM
sCHM<-CHMsmoothing(chm, filter, ws)
```



3.4 chullLiDAR2D 2D Convex hull of individual tree lidar-derived point cloud

Description

Compute and plot the 2D convex hull of individual tree lidar-derived point cloud

Usage

chullLiDAR2D(xyid)

Arguments

xyid	A 3-column matrix with the x, y coordinates and points id of the lidar point cloud.

Value

Returns A list with components "chullPolygon" and "chullArea", giving the polygon and area of the convex hull.

Author(s)

Carlos Alberto Silva

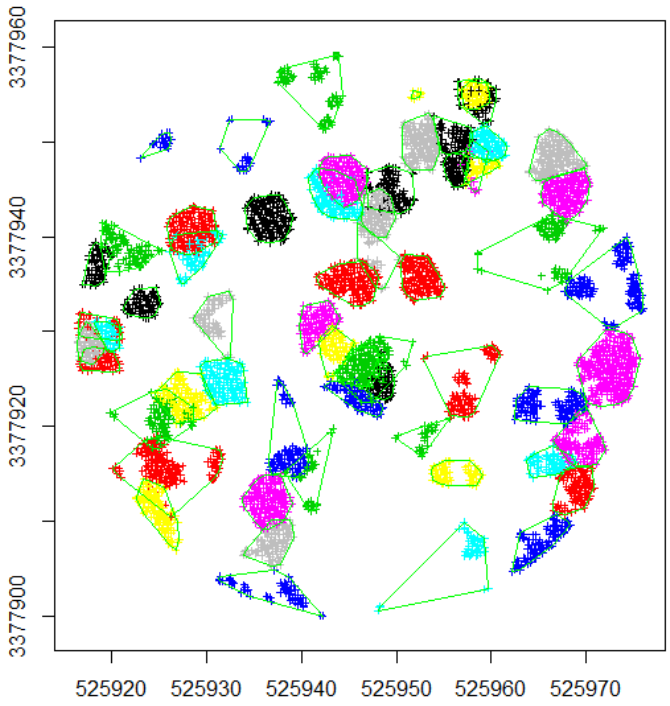
References

grDevices package,see chull.

Examples

```
# Importing LAS file:
LASfile <- system.file("extdata", "LASexample1.las", package="rLiDAR")


# Reading LAS file
LAS<-readLAS(LASfile,short=TRUE)


# Height subsetting the data
xyz<-subset(LAS[,1:3],LAS[,3] >= 1.37)
# Getting lidar clusters
set.seed(1)
clLAS<-kmeans(xyz, 60)
```

```
# Set the points id
id<-as.factor(clLAS$cluster)


# Set the xyid input
xyid<-cbind(xyz[,1:2],id)


# Compute the lidar convex hull of the
clusters chullTrees<-chullLiDAR2D(xyid)


# Plotting the lidar convex hull
library(sp)
plot(SpatialPoints(xyid[,1:2]),cex=0.5,col=xyid[,3], axes=T)
plot(chullTrees$chullPolygon,add=TRUE, border="green")


# Get the ground-projected area of lidar convex
hull chullList<-chullTrees$chullArea
summary(chullList) # summary
```



3.5 chullLiDAR3D 3D convex hull of the individual tree lidar-derived point cloud

Description

Compute and plot the 3D convex hull (and its surface area and volume) of the individual tree lidar-derived point cloud.

Usage

chullLiDAR3D(xyzid,plotit=TRUE,col="forestgreen",alpha=0.8)

Arguments

xyzid	A matrix with four columns (xyz coordinates and tree id).
plotit	Logical. If FALSE, returns only volume and surface area.
col	A vector or a caracter of the convex hull color.
alpha	A vector or a caracter of the convex hull transparency (0-1).

Value

A list with components 'crownvolume' and 'crownsurface', giving the volume and surface of the convex hull.

Author(s)

Carlos Alberto Silva. Uses code by Remko Duursma (*YplantQMC* package,see "crownhull").

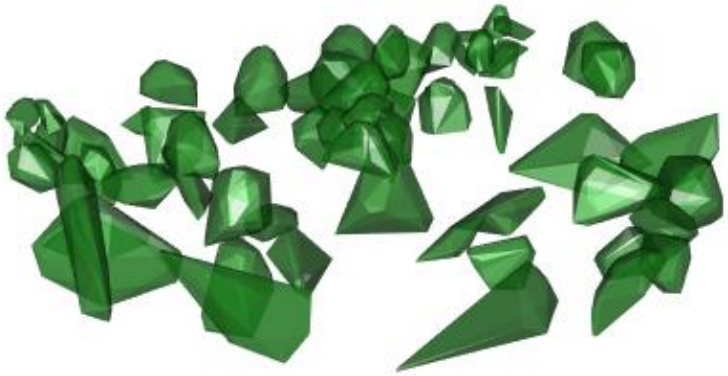
References

www.qhull.organd *geometry* package (see convhulln).

Examples

```
# Importing LAS file:
LASfile <- system.file("extdata", "LASexample1.las", package="rLiDAR")

# Reading LAS file
LAS<-readLAS(LASfile,short=TRUE)
```

```
# Setring the xyz coordinates and subsetting the
data xyz<-subset(LAS[,1:3],LAS[,3] >= 1.37)


# Finding clusters
clLAS<-kmeans(xyz,
32)


# Set the id vector
id<-as.factor(clLAS$cluster)


#============================================================#
# Example 01
#============================================================#
# Set the alpha
alpha<-0.6

# Set the plotCAS parameter
plotit=TRUE

# Set the convex hull color
col="forestgreen"

# Combining xyz and id
xyzid<-cbind(xyz,id)

# Get the volume and surface area
install.packages(rgl)
require(rgl)
open3d()

volumeList<-chullLiDAR3D(xyzid=xyzid, plotit=plotit, col=col,alpha=alpha)
```



summary(volumeList) # summary

output:

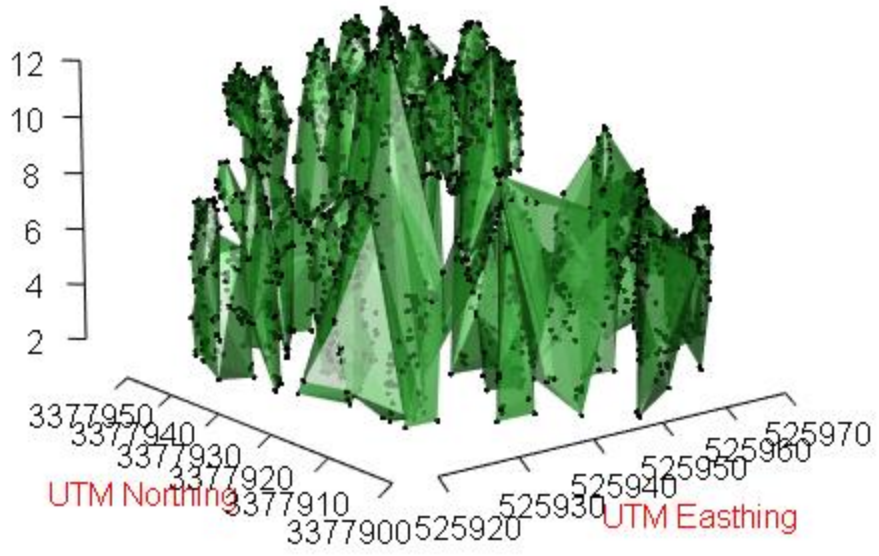
```
     Tree        crownvolume        crownsurface
 Min.   : 1.00   Min.   :  0.6191   Min.   :  5.48
 1st Qu.:15.75   1st Qu.: 22.5697   1st Qu.: 46.80
 Median :30.50   Median : 43.2469   Median : 72.21
 Mean   :30.50   Mean   : 51.4924   Mean   : 76.48
 3rd Qu.:45.25   3rd Qu.: 64.1821   3rd Qu.: 95.33
 Max.   :60.00   Max.   :186.2450   Max.   :195.49
```

plot3d(xyzid[,1:3], add=TRUE)

axes3d(c("x-", "y-", "z-+"))

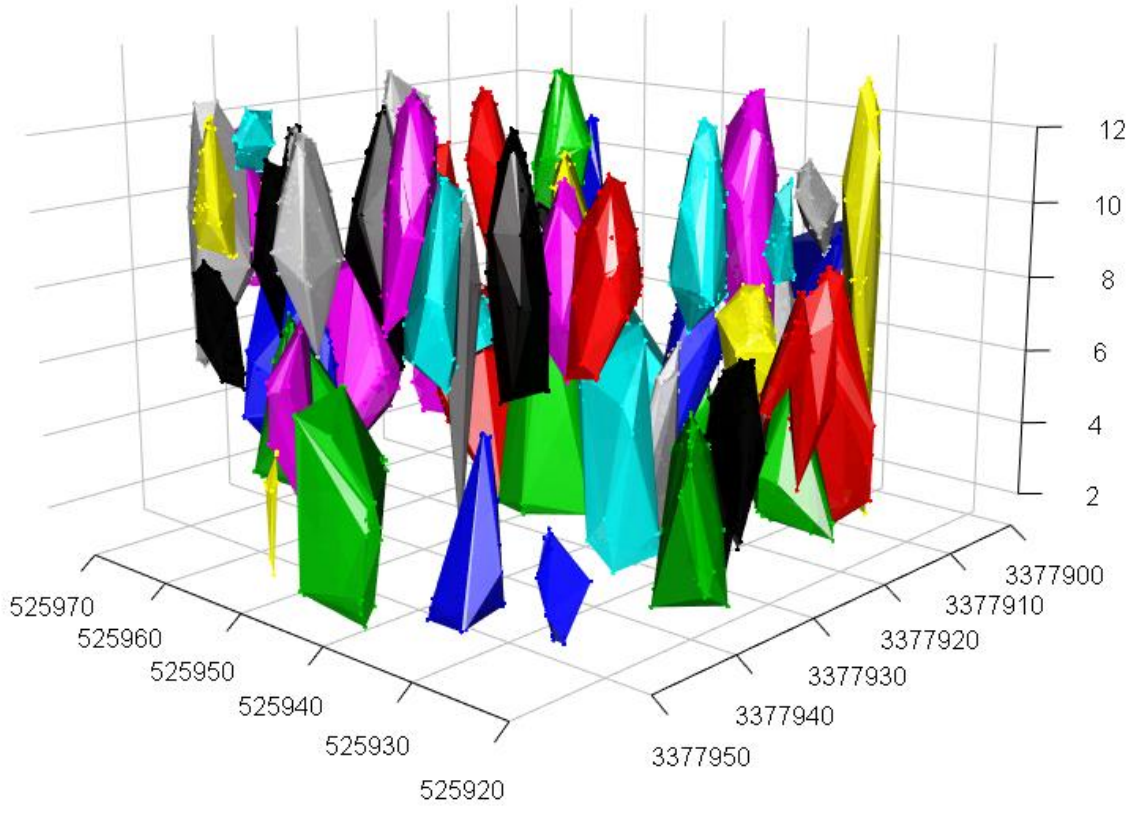
title3d(xlab = "UTM Easthing", ylab = "UTM Northing",zlab = "Height", col="red")

aspect3d(1,1,0.7) # scale




```
#============================================================#
# Example 02
#============================================================#
# Set the alpha
alpha<-0.85

# Set the plotCAS parameter
plotit=TRUE

# Set the convex hull color
col=levels(factor(id))
# Combining xyz and id
xyzid<-cbind(xyz,id)
# Get the volume and surface area
 open3d()
```

```
volumeList<-chullLiDAR3D(xyzid=xyzid, plotit=plotit, col=col,alpha=alpha)
summary(volumeList)


# Add other plot parameters
plot3d(xyzid[,1:3], col=xyzid[,4], add=TRUE)
# add the 3D point cloud
axes3d(c("x+", "y-", "z-"))
# axes
grid3d(side=c("x+", "y-", "z"), col="gray") # grid
title3d(xlab = "UTM Easthing", ylab = "UTM Northing",zlab = "Height", col="red")
aspect3d(1,1,0.7) # scale
```



3.6 CrownMetrics Lidar-derived individual tree crown metrics

Description

Compute individual tree crown metrics from lidar data

Usage

```
CrownMetrics(xyziId)
```

Arguments

xyziId	A 5-column matrix with the x, y, z coordinates, intensity and the tree id classifi- cation for the lidar point cloud.

Details

List of the individual tree crown metrics:

- TotalReturns: Total number of returns
- ETOP - UTM Easting coordinate of the tree top
- NTOP - UTM Northing coordinate of the tree top
- EMIN - Minimum UTM Easting coordinate
- NMIN - Minimum UTM Northing coordinate
- EMAX - Maximum UTM Easting coordinate
- NMAX - Maxmium UTM Northing coordinate
- EWIDTH - Tree crown width 01
- NWIDTH - Tree crown width 02
- HMAX - Maximum Height
- HMEAN - Mean height
- HSD - Standard deviation of height
- HCV - Coefficient of variation of height
- HMOD - Mode of height
- H5TH - 5th percentile of height
- H10TH - 10th percentile of height
- H20TH - 20th percentile of height
- H25TH - 25th percentile of height
- H30TH - 30th percentile of height
- H40TH - 40th percentile of height

- H50TH - 50th percentile of height
- H60TH - 60th percentile of height
- H70TH - 70th percentile of height
- H75TH - 75th percentile of height
- H80TH - 80th percentile of height
- H90TH - 90th percentile of height
- H95TH - 95th percentile of height
- H99TH - 99th percentile of height
- IMAX - Maximum intensity
- IMEAN - Mean intensity
- ISD - Standard deviation of intensity
- ICV - Coefficient of variation of intensity
- IMOD - Mode of intensity
- I5TH - 5th percentile of intensity
- I10TH - 10th percentile of intensity
- I20TH - 20th percentile of intensity
- I25TH - 25th percentile of intensity
- I30TH - 30th percentile of intensity
- I40TH - 40th percentile of intensity
- I50TH - 50th percentile of intensity
- I60TH - 60th percentile of intensity
- I70TH - 70th percentile of intensity
- I75TH - 75th percentile of intensity
- I80TH - 80th percentile of intensity
- I90TH - 90th percentile of intensity
- I95TH - 95th percentile of intensity
- I99TH - 99th percentile of intensity

Value

Returns A matrix of the lidar-based metrics for the individual tree detected.

Author(s)

Carlos Alberto Silva

Examples

```
#=================================================================
#
# Individual tree detection using K-means cluster
#=================================================================#
# Importing LAS file:
LASfile <- system.file("extdata", "LASexample1.las", package="rLiDAR")


# Reading LAS file
LAS<-readLAS(LASfile,short=TRUE)


# Setting the xyz coordinates and subsetting the data
xyzi<-subset(LAS[,1:4],LAS[,3] >= 1.37)

# Finding clusters (trees)
clLAS<-kmeans(xyzi[,1:2], 32)

# Set the tree id vector
Id<-as.factor(clLAS$cluster)
# Combining xyzi and tree id
xyziId<-cbind(xyzi,Id)


#=================================================================#
# Computing individual tree lidar metrics
#=================================================================#
TreesMetrics<-CrownMetrics(xyziId)
head(TreesMetrics)
```

3.7 FindTreesCHM Individual tree detection whitin the lidar-derived Canopy Height Model (CHM)

Description

Detects and computes the location and height of individual trees within the lidar-derived Canopy Height Model (CHM). The algorithm implemented in this function is local maximum with a fixed window size.

Usage

```
FindTreesCHM(chm,fws,minht)
```

Arguments

chm	A lidar-derived Canopy Height Model (CHM) raster file.
fws	A single dimension (in raster grid cell units) of fixed square window size, e.g. 3, 5, 7 and so on. Default is 5.
minht	Height threshold. Detect individual trees above specified height threshold, e.g. 1.37, 2.0, 3.5 m and so on. Default is 1.37 m.

Value

Returns A matrix with four columns (tree id, xy coordinates, and height).

Author(s)

Carlos Alberto Silva

Examples

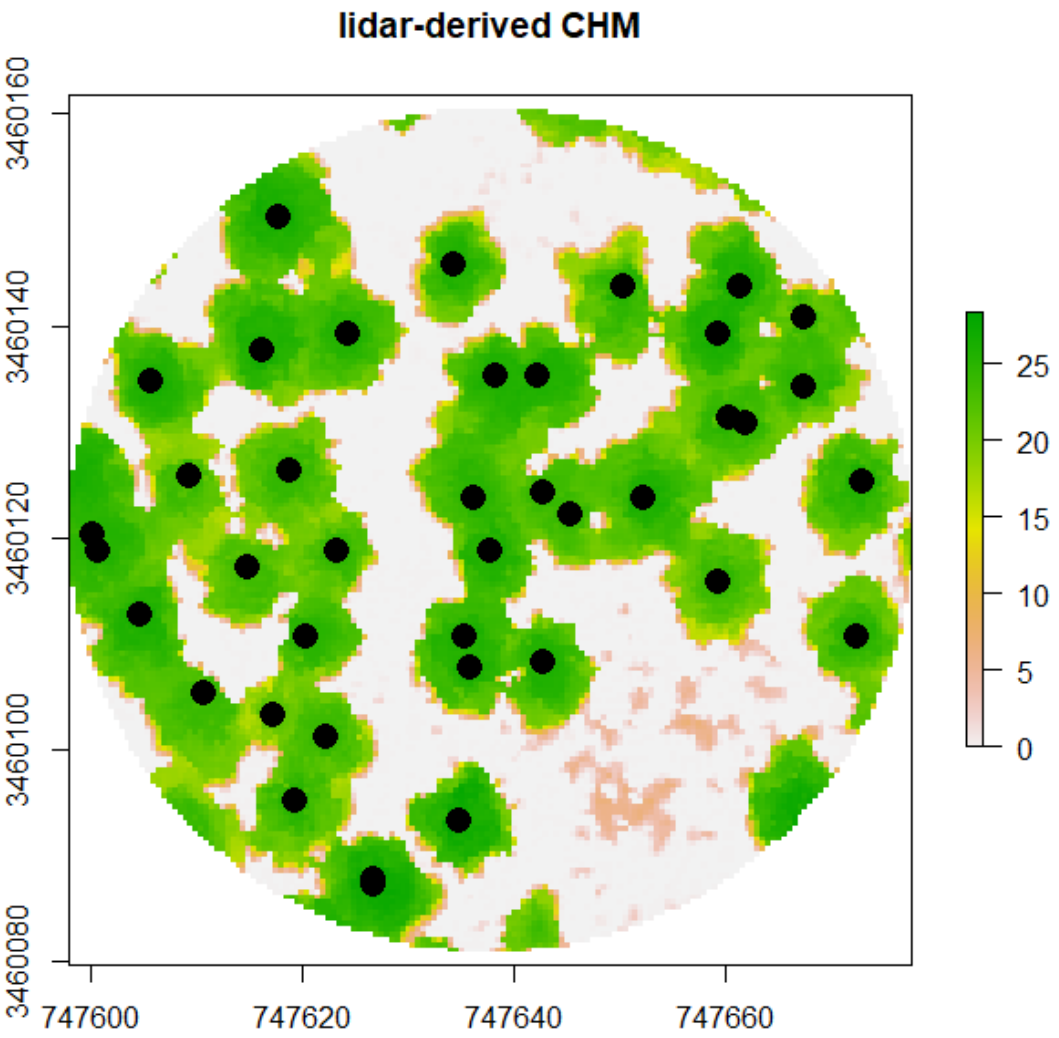
```
# Importing the lidar-derived CHM raster file
data(chm) # or set a CHM. e.g. chm<-raster("CHM_stand.asc")

# Smoothing CHM
schm<-CHMsmoothing(chm, "mean", 5)

# Setting the fws:
```

```
fws<-5 # dimention 5x5

# Setting the specified height above ground for detectionbreak
minht<-8.0
# Getting the individual tree detection list
treeList<-FindTreesCHM(schm, fws, minht)
summary(treeList)
```

output

```
       x                y              height     
 Min.   :747600   Min.   :3460087   Min.   :22.04  
 1st Qu.:747619   1st Qu.:3460111   1st Qu.:24.44  
 Median :747635   Median :3460123   Median :25.41  
 Mean   :747635   Mean   :3460121   Mean   :25.30  
 3rd Qu.:747651   3rd Qu.:3460135   3rd Qu.:26.02  
 Max.   :747673   Max.   :3460150   Max.   :27.50  
```

```
# Plotting the individual tree location on the CHM library(raster)
plot(chm, main="lidar-derived CHM")
library(sp)
XY<-SpatialPoints(treeList[,1:2]) # Spatial points
plot(XY, add=TRUE, col="black", pch=16,cex=2)   # plotthing tree location
```


lidar-derived CHM
3460160
3460140
3460120
3460100
3460080
747600
747620
747640
747660
25
20
15
10
5
0

3.8 ForestCAS Individual trees crown deliniation from lidar-derived Canopy Height Model (CHM)

Description

Delineate and compute ground-projected area of individual tree crowns detected from lidar- derived CHM

Usage

ForestCAS(chm, loc, maxcrown, exclusion)

Arguments

chm	A lidar-derived Canopy Height Model (CHM) RasterLayer or SpatialGrid- DataFrame file.
loc	A matrix or dataframe with three columns (tree xy coordinates and height).
maxcrown	A single value of the maximum individual tree crown radius expected. Default 10.0 m.
exclusion	A single value from 0 to 1 that represents the

Value

Returns a list that contains the individual tree canopy boundary polygons and the 4-column ma- trix with the tree xy coordinates, heights and ground-projected canopy area (with units of square meters).

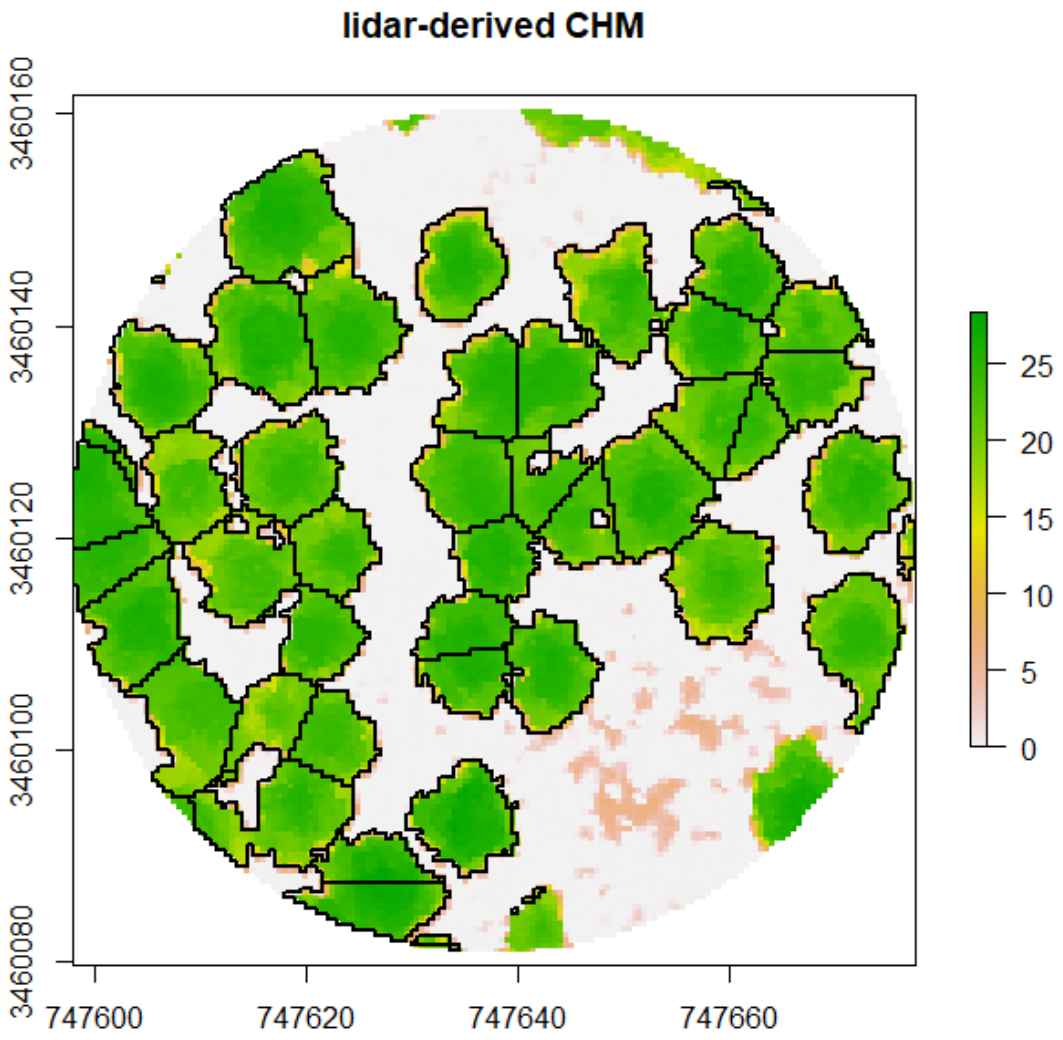
Author(s)

Carlos Alberto Silva

Examples

```
## Not run:

# Import the lidar-derived CHM file
data(chm) # or set a CHM. e.g. chm<-raster("CHM_stand.asc")
```

```
# Set the loc parameter
sCHM<-CHMsmoothing(chm, filter="mean", ws=5) # smoothing CHM
loc<-FindTreesCHM(sCHM, fws=5, minht=8)    # or import a tree list


# Set the maxcrown parameter
maxcrown=10.0

# Set the exclusion parameter
exclusion=0.3 # 30%


# Compute individual tree detection canopy area
canopy<-ForestCAS(chm, loc, maxcrown, exclusion)


#=========================================================#
# Retrieving the boundary for individual tree detection and canopy area calculation
#=========================================================#
boundaryTrees<-canopy[[1]]
# Plotting the individual tree canopy boundary over the CHM
plot(chm, main="lidar-derived CHM")
plot(boundaryTrees, add=T, border="black", lwd=2,bg="transparent") # adding tree canopy
boundary
```




```
#=========================================================#
# Retrieving the list of individual trees detected for canopy area calculation
#=========================================================#
canopyList<-canopy[[2]] # list of ground-projected areas of individual tree canopies
summary(canopyList)          # summary


# Spatial location of the trees library(sp)
XY<-SpatialPoints(canopyList[,1:2])# Spatial points
plot(XY, col="red", add=T, pch=16, cex=2) # adding tree location to the plot
## End(Not run)
```

3.9 LASmetrics Lidar-derived metrics

Description

Compute lidar metrics that describe statistically the Lidar dataset

Usage

LASmetrics(LASfile, minht, above)

Arguments

LASfile	A LAS standard lidar data file
minht	Use only returns above specified height break, e.g. 1.30 m. Default is 1.37 m.
above	Compute covers metrics using specified height break, e.g. 2.5 m. Default is 2 m.

Value

Returns A matrix with the lidar-derived vegetation height and canopy cover metrics (see *cloud- metrics*, in McGaughey, 2014)

Author(s)

Carlos Alberto Silva

See Also

McGaughey, R. 2014. FUSION/LDV: Software for lidar data analysis and visualization. Version 3.41. Seattle, WA: U.S. Department of Agriculture, Forest Service, Pacific Northwest Research Station.

List of the lidar-derived metrics:

- Total all return count
- Total first return count
- Total all return count above *minht*
- Return 1 count above *minht*
- Return 2 count above *minht*
- Return 3 count above *minht*

- Return 5 count above *minht*
- Return 6 count above *minht*
- Return 7 count above *minht*
- Return 8 count above *minht*
- Return 9 count above *minht*
- HMIN - Maximum Height
- HMAX - Maximum Height
- HMEAN - Mean height
- HMOD - Modal height
- HMEDIAN - Median height
- HSD - Standard deviation of heights
- HVAR - Variance of heights
- HCV - Coefficient of variation of heights
- HKUR - Kurtosis of Heights
- HSKE - Skewness of Heights
- H01TH - 01th percentile of height
- H05TH - 05th percentile of height
- H10TH - 10th percentile of height
- H15TH - 15th percentile of height
- H20TH - 20th percentile of height
- H25TH - 25th percentile of height
- H30TH - 30th percentile of height
- H35TH - 35th percentile of height
- H40TH - 40th percentile of height
- H45TH - 45th percentile of height
- H50TH - 50th percentile of height
- H55TH - 55th percentile of height
- H60TH - 60th percentile of height
- H65TH - 65th percentile of height
- H70TH - 70th percentile of height
- H75TH - 75th percentile of height
- H80TH - 80th percentile of height
- H90TH - 90th percentile of height
- H95TH - 95th percentile of height
- H99TH - 99th percentile of height

- CRR - Canopy relief ratio
- IMIN - Minimum intensity
- IMAX - Maximum intensity
- IMEAN - Mean intensity
- IMOD - Modal intensity
- IMEDIAN - Median intensity
- ISD - Standard deviation of intensities
- IVAR - Variance of heights
- ICV - Coefficient of variation of intensities
- IKUR - Kurtosis of intensities
- ISKE - Skewness of intensities
- I01TH - 1th percentile of intensity
- I05TH - 5th percentile of intensity
- I10TH - 10th percentile of intensity
- I15TH - 15th percentile of intensity
- I20TH - 20th percentile of intensity
- I25TH - 25th percentile of intensity
- I30TH - 30th percentile of intensity
- I35TH - 35th percentile of intensity
- I40TH - 40th percentile of intensity
- I45TH - 45th percentile of intensity
- I50TH - 50th percentile of intensity
- I55TH - 55th percentile of intensity
- I60TH - 60th percentile of intensity
- I65TH - 65th percentile of intensity
- I70TH - 70th percentile of intensity
- I75TH - 75th percentile of intensity
- I80TH - 80th percentile of intensity
- I90TH - 90th percentile of intensity
- I95TH - 95th percentile of intensity
- I99TH - 99th percentile of intensity
- Pentage first returns above *above*
- Percentage all returns above *above*

- (All returns above above / Total first returns)*100
- First returns above *above*
- All returns above *above*
- Percentage first returns above mean
- Percentage first returns above mode
- Percentage all returns above mean
- Percentage all returns above mode
- (All returns above mean / Total first returns)*100
- (All returns above mode / Total first returns)* 100
- First returns above mean"
- First returns above mode
- All returns above mean
- All returns above mode

Examples

```
#=================================================================
#
# Example 01: Computing lidar metrics for a single LAS file
#=================================================================
#
# Import the LAS data file
LASfile <- system.file("extdata", "LASexample1.las", package="rLiDAR")


# Set the minht and above parameters
minht<-1.37 # meters or feet
above<-2.00  # meters or feet


# lidar metrics computation
LiDARmetrics<-LASmetrics(LASfile, minht, above)
```

```
#=================================================================
# Example 02: Computing Lidar metrics for multiple LAS files within a folder
#=================================================================
#
# Set folder where LAS source files reside folder=dirname(LASfile)
# Get list of LAS files residing in the folder
LASlist <- list.files(folder, pattern="*.las", full.names=TRUE)


# Set the "minht" and "above" parameters minht<-1.37 # meters or feet
above<-2.00  # meters or feet


# Creat an empty dataframe in whic to store the lidar metrics
getMetrics<-data.frame()


# Set a loop to compute the lidar metrics for ( i in LASlist) {
getMetrics<-rbind(getMetrics, LASmetrics(i, minht, above))}


# Table of the Lidar metrics
LiDARmetrics<-cbind(Files=c(basename(LASlist)), getMetrics) head(LiDARmetrics)
```

3.10 LiDARForestStand 3D stand visualization of lidar-derived individual trees

Description

Draws a 3D scatterplot for individual trees detected from Lidar data.

Usage

```
LiDARForestStand(crownshape = c("cone", "ellipsoid", "halfellipsoid", "paraboloid",
                 "cylinder"), CL = 4, CW = 8, HCB = 10,
                   X = 0, Y = 0, dbh = 0.3, crowncolor = "forestgreen", stemcolor =
                  "chocolate4", resolution="high", mesh=TRUE)
```

Arguments

crownshape	shape of individual tree crown: "cone", "ellipsoid","halfellipsoid", "paraboloid" or "cylinder". Default is "halfellipsoid".
CL	crown length.
CW	crown diameter.
HCB	height at canopy base.
X	x-coordinate.
Y	y-coordinate.
dbh	diameter at breast height (1.73 m).
crowncolor	crown color.
stemcolor	stem color.
resolution	crown resolution: "low", "medium" and "high".
mesh	Logical, if TRUE (default) returns a tree crown mesh model, and if FALSE returns a tree crown line mode.

Value

Returns a 3-D scatterplot of the individual trees as identified automatically from the lidar.

Author(s)

Carlos Alberto Silva and Remko Duursma. Uses code by Remko Duursma (*Maeswrap* package,see "Plotstand").

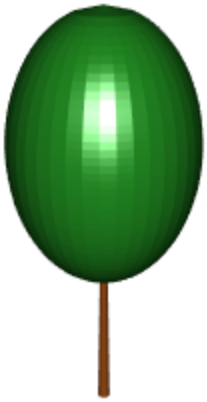
References

http://maespa.github.io/

Examples

```
## Not run:
#============================================================
# EXAMPLE 01: Plotting single trees
#============================================================
#

# cone crown shape

library(rgl)

open3d()

LiDARForestStand(crownshape = "cone", CL = 10, CW =7, HCB = 5, X =0, Y = 0, dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4", resolution="high", mesh=TRUE)
```

```
# ellipsoid crown shape

open3d()

LiDARForestStand(crownshape = "ellipsoid", CL = 10, CW =7,

HCB = 5, X =0, Y = 0, dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4", resolution="high", mesh=TRUE)
```

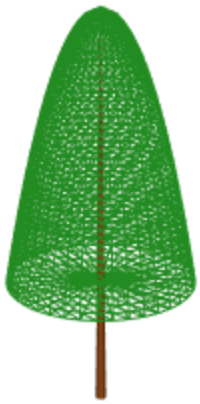


```
# halfellipsoid crown shape
open3d()
LiDARForestStand(crownshape = "halfellipsoid", CL = 10, CW =7,
HCB = 5, X =0, Y = 0, dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4",
resolution="high", mesh=TRUE)
```

```
# paraboloid crown shape
open3d()
LiDARForestStand(crownshape = "paraboloid", CL = 10, CW =7,
HCB = 5, X =0, Y = 0, dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4",
resolution="high", mesh=TRUE)
```

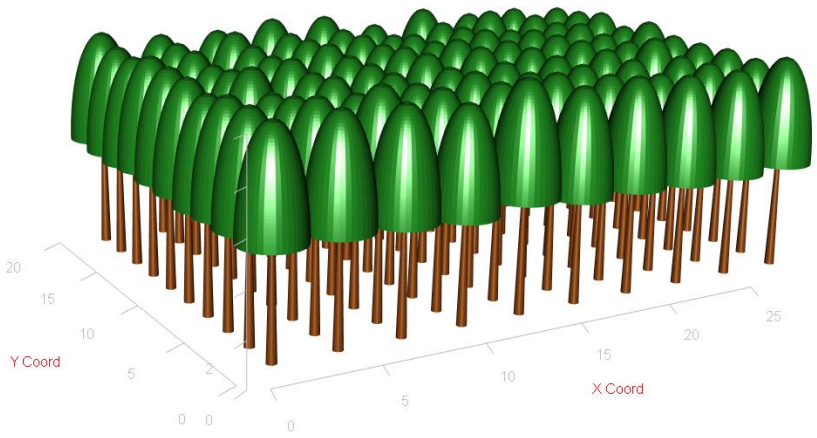
```
# cylinder crown shape
open3d()
LiDARForestStand(crownshape = "cylinder", CL = 10, CW =7,
HCB = 5, X =0, Y = 0, dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4",
resolution="high", mesh=TRUE)
```

```
# Set the shape=FALSE
open3d()
LiDARForestStand(crownshape = "paraboloid", CL = 10, CW =7,
HCB = 5, X =0, Y = 0, dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4",
resolution="high", mesh=FALSE)
```



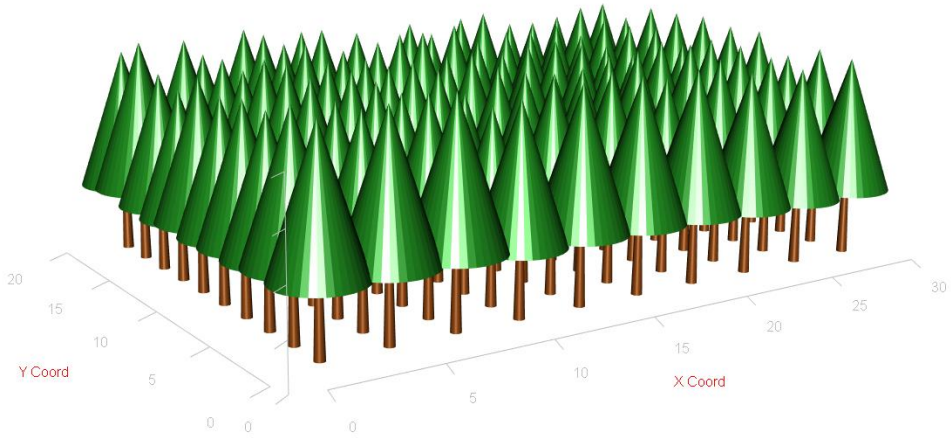
```
#============================================================#
#EXAMPLE 02: Plotting a forest plantation stand in virtual 3-D space
#============================================================#

# Set the dimensions of the displayed forest stand
xlength<-30 # x length
ylength<-20 # y length

# Set the space between trees
sx<-3 # x space length
sy<-2 # y space length

# Tree location grid
XYgrid <- expand.grid(x = seq(1,xlength,sx), y = seq(1,ylength,sy))

# Get the number of trees
Ntrees<-nrow(XYgrid)

# Plot a virtual Eucalyptus forest plantation stand using the halfellipsoid tree crown shape
# Set stand trees parameters
```

```
meanHCB<-5 # mean of the height at canopy base
sdHCB<-0.1 # standard deviation of the height at canopy base
HCB<-rnorm(Ntrees, mean=meanHCB, sd=sdHCB) # height at canopy base
CL<-HCB       # tree crown height
CW<-HCB*0.6 # tree crown diameter

open3d()        # open a rgl window

# Plotting the stand for( i in 1:Ntrees){
LiDARForestStand(crownshape = "halfellipsoid", CL = CL[i], CW = CW[i], HCB =
HCB[i], X = XYgrid[i,1], Y = XYgrid[i,2], dbh = 0.4,
crowncolor = "forestgreen", stemcolor = "chocolate4", resolution="high", mesh=TRUE)
}


# Add other plot parameters
axes3d(c("x-", "x-", "y-", "z"), col="gray")            # axes
title3d(xlab = "X Coord", ylab = " Y Coord", zlab = "Height", col="red") # title
planes3d(0, 0, -1, 0.001, col="gray", alpha=0.7)      # set a terrain plane
```



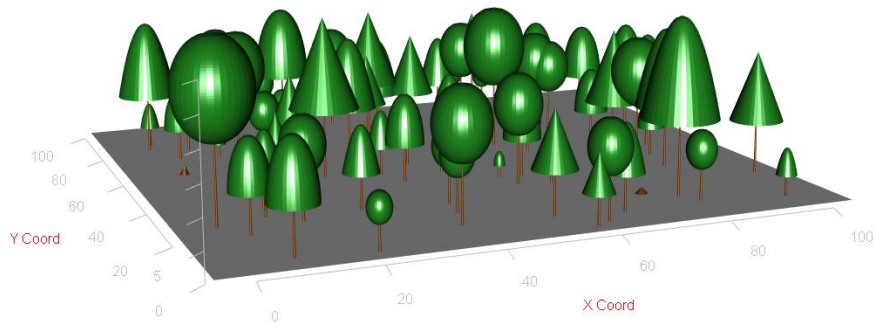

```
# Plotting a virtual single-species forest plantation stand using "cone" tree crown shape #
Set parameters f trees growing within the virtual stand
```

```
meanHCB<-3 # mean of the height at canopy base
sdHCB<-0.1 # standard deviation of the height at canopy base
HCB<-rnorm(Ntrees, mean=meanHCB, sd=sdHCB) # height at canopy base
CL<-HCB*2.0 # tree crown height
CW<-HCB*1.3 # tree crown diameter


open3d() # open a rgl window # Plot stand

for( i in 1:Ntrees){
LiDARForestStand(crownshape = "cone", CL = CL[i], CW = CW[i],
HCB = HCB[i], X = XYgrid[i,1], Y = XYgrid[i,2], dbh = 0.4,
crowncolor = "forestgreen", stemcolor = "chocolate4", resolution="high", mesh=TRUE)
}


# Add other plot parameters
axes3d(c("x-", "x-", "y-", "z"), col="gray")            # axes
title3d(xlab = "X Coord", ylab = " Y Coord", zlab = "Height", col="red") #title
planes3d(0, 0, -1, 0.001, col="gray", alpha=0.7)      # set a terrain plane
```



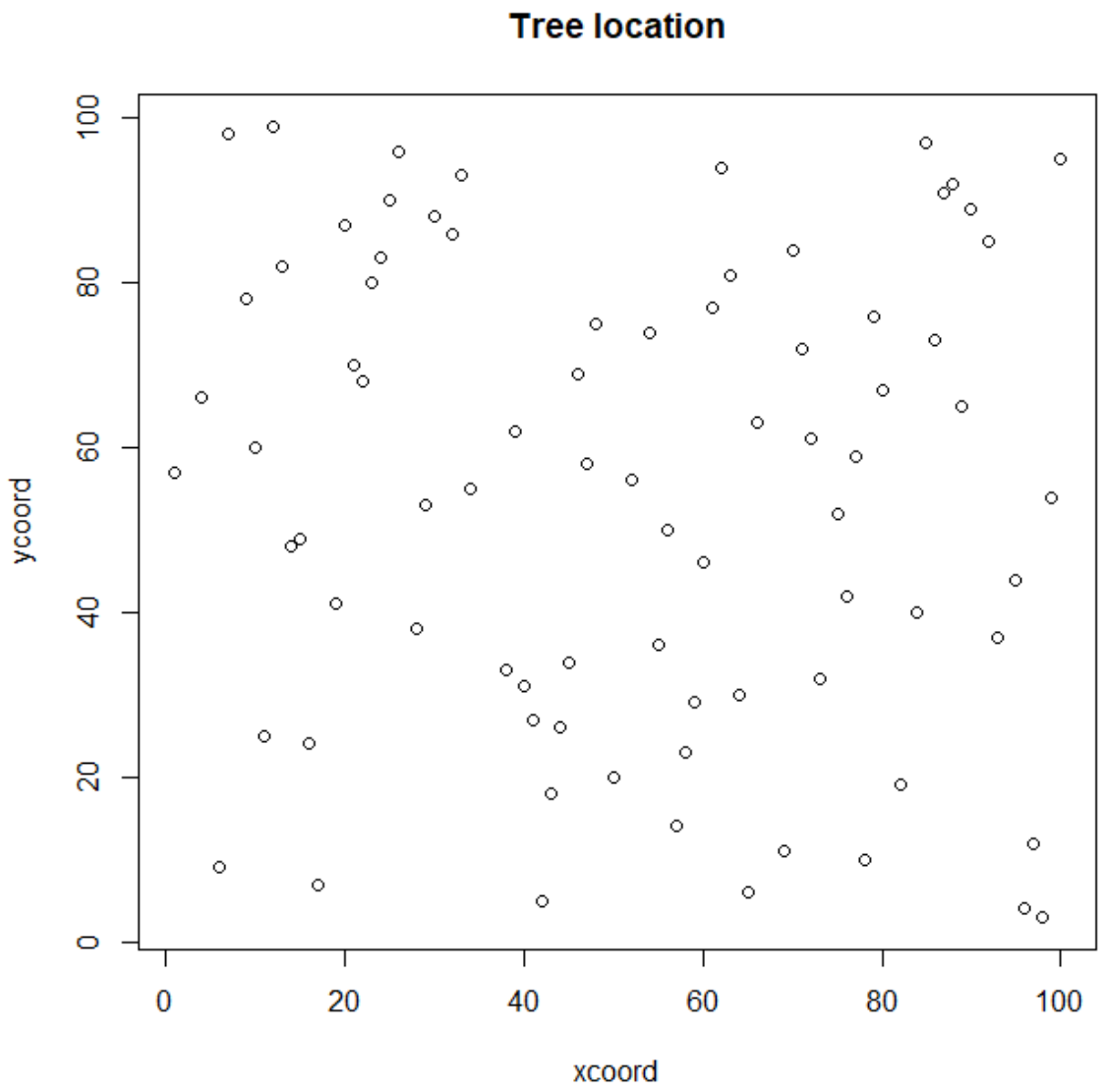
```
#=================================================================#
# EXAMPLE 03: Plotting a virtual mixed forest stand
#=================================================================#

# 01. Plot different trees species in the stand with different crown shapes
# Set the number of trees
Ntrees<-80

# Set the trees locations
xcoord<-sample(1:100, Ntrees) # x coord
ycoord<-sample(1:100, Ntrees) # y coord

# Set a location grid of trees
XYgrid<-cbind(xcoord,ycoord)

# Plot the location of the trees
plot(XYgrid, main="Tree location")

meanHCB<-7 # mean of the height at canopy base
sdHCB<-3      # standard deviation of height at canopy base
HCB<-rnorm(Ntrees, mean=meanHCB, sd=sdHCB) # height at canopy base
crownshape<-sample(c("cone", "ellipsoid","halfellipsoid",
"paraboloid"), Ntrees, replace=TRUE) # tree crown shape
CL<-HCB*1.3 # tree crown height
CW<-HCB      # tree crown diameter

open3d() # open a rgl window # Plot stand

for( i in 1:Ntrees){
  LiDARForestStand(crownshape = crownshape[i], CL = CL[i], CW = CW[i],
```

```
HCB = HCB[i], X = as.numeric(XYgrid[i,1]), Y = as.numeric(XYgrid[i,2]), dbh = 0.4, crowncolor = "forestgreen", stemcolor = "chocolate4", resolution="high", mesh=TRUE)
}

# Add other plot parameters
axes3d(c("x-", "x-", "y-", "z"), col="gray")    # axes
title3d(xlab = "X Coord", ylab = " Y Coord", zlab = "Height", col="red") # title planes3d(0, 0, -1, 0.001, col="gray", alpha=0.7)   # set a terrain plane
planes3d(0, 0, -1, 0.001, col="gray", alpha=0.7)       # set a terrain plane
```



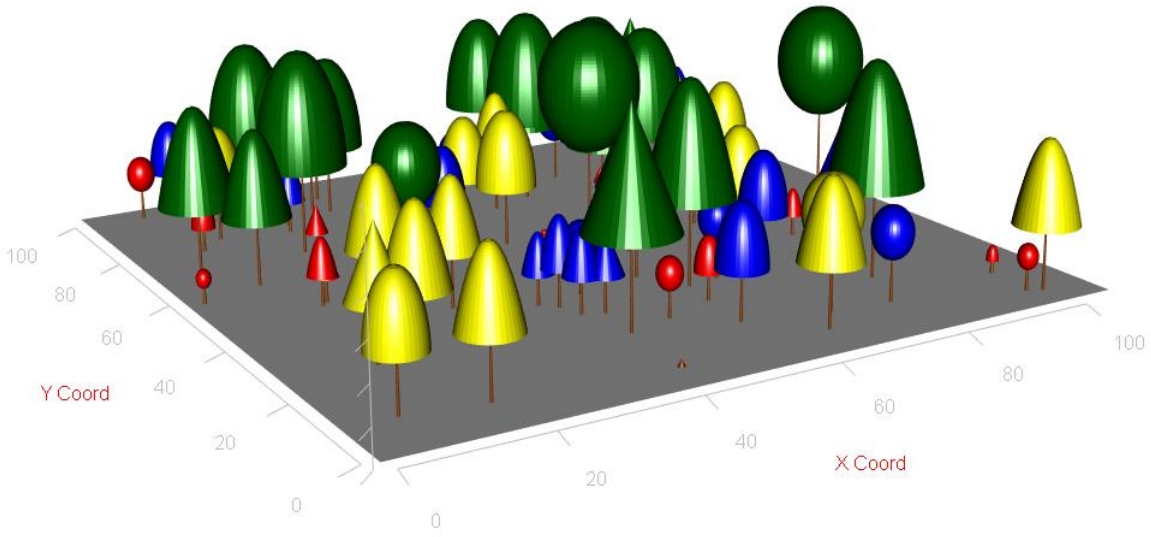

```
# 02. Plot different tree height in the stand using different crown colors # Set the number of trees
Ntrees<-80

# Set the tree locations
xcoord<-sample(1:100, Ntrees) # x coord ycoord<-sample(1:100, Ntrees) # y coord

# Set a location grid of trees
```

```
XYgrid<-cbind(xcoord,ycoord)
# plot the location of the trees
windows()
plot(XYgrid, main="Tree location")
```




```
meanHCB<-7 # mean of the height at canopy base
sdHCB<-3       # standard deviation of the height at canopy base
HCB<-rnorm(Ntrees, mean=meanHCB, sd=sdHCB) # height at canopy base
crownshape<-sample(c("cone", "ellipsoid","halfellipsoid", "paraboloid"), Ntrees,
replace=TRUE)  # tree crown shape
```

```
CL<-HCB*1.3 # tree crown lenght
CW<-HCB      # tree crown width

# Plot tree height based on the HCB quantiles
HCBq<-quantile(HCB) # HCB quantiles
crowncolor<-NA*(1:Ntrees) # set an empty crowncolor vector

# classify trees by HCB quantile
for (i in 1:Ntrees){
  if (HCB[i] <= HCBq[2]) {crowncolor[i]<-"red"}    # group 1
  if (HCB[i] > HCBq[2] & HCB[i] <= HCBq[3] ) {crowncolor[i]<-"blue"} # group 2
  if (HCB[i] > HCBq[3] & HCB[i] <= HCBq[4] ) {crowncolor[i]<-"yellow"} # group 3
  if (HCB[i] >= HCBq[4]) {crowncolor[i]<-"dark green"}    # group 4
}
open3d() # open a rgl window
# Plot stand
for(i in 1:Ntrees){
  LiDARForestStand(crownshape = crownshape[i], CL = CL[i], CW = CW[i],
                   HCB = HCB[i], X = as.numeric(XYgrid[i,1]), Y = as.numeric(XYgrid[i,2]), dbh
= 0.4, crowncolor = crowncolor[i],stemcolor = "chocolate4", resolution="high",
mesh=TRUE)
}

# Add other plot parameters
axes3d(c("x-", "x-", "y-", "z"), col="gray")    # axes
title3d(xlab = "X Coord", ylab = " Y Coord", zlab = "Height", col="red") # title
planes3d(0, 0, -1, 0.001, col="gray", alpha=0.7)       # set a terrain plane
## End(Not run)
```


100
80
60
40
20
0
Y Coord
0
20
40
60
80
100
X Coord

3.11 readLAS Reading lidar data

Description

This function reads and returns values associated with the LAS file format. The LAS file is a public file format for the interchange of lidar 3-dimensional point cloud data (American Society of Photogrammetry and Remote Sensing - ASPRS)

Usage

readLAS(LASfile, short=TRUE)

Arguments

LASfile	A standard LAS data file (ASPRS)
short	Logical, if TRUE it will return only a 5-column matrix with information on the returned point x, y, z locations, point intensity and the number of return within an individual discrete-return system laser pulse.

Value

Returns a matrix listing the information stored in the LAS file.

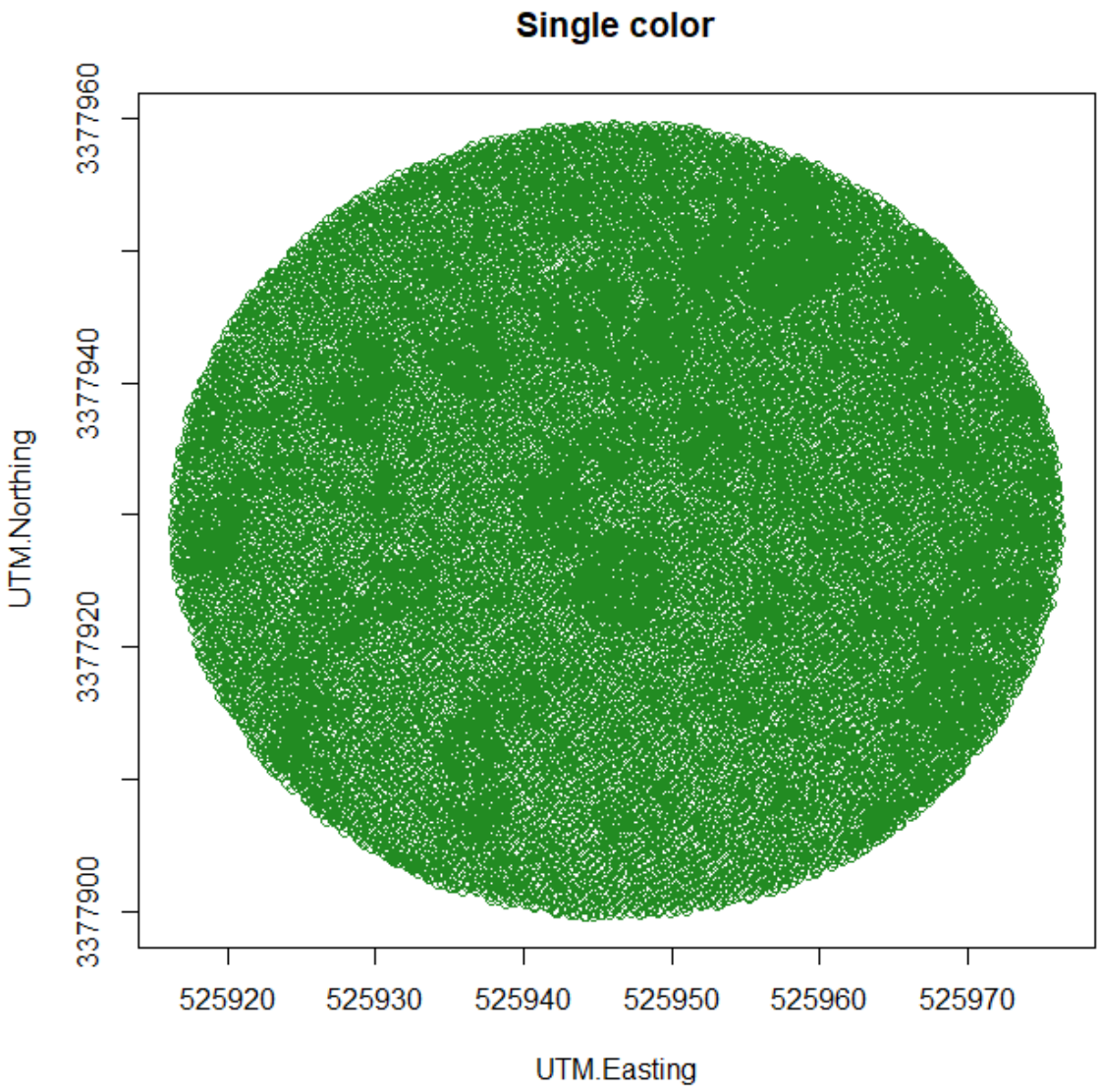
Author(s)

Michael Sumner and Carlos Alberto Silva.

Examples

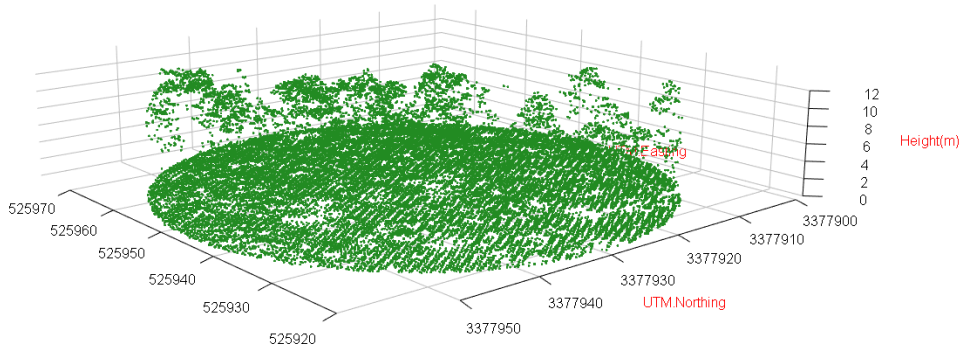
```
#=========================================================#
# Importing LAS file:
#=========================================================#
LASfile <- system.file("extdata", "LASexample1.las", package="rLiDAR")

# Reading LAS file
rLAS<-readLAS(LASfile,short=TRUE)
```

```
# Summary of the LAS file
summary(rLAS)

#=============================================================#
# LAS file visualization:
#=============================================================#

# 01 Set a single color
col<-"forestgreen"
# plot 2D
plot(rLAS[,1],rLAS[,2], col=col,xlab="UTM.Easting", ylab="UTM.Northing",
main="Single color")
```

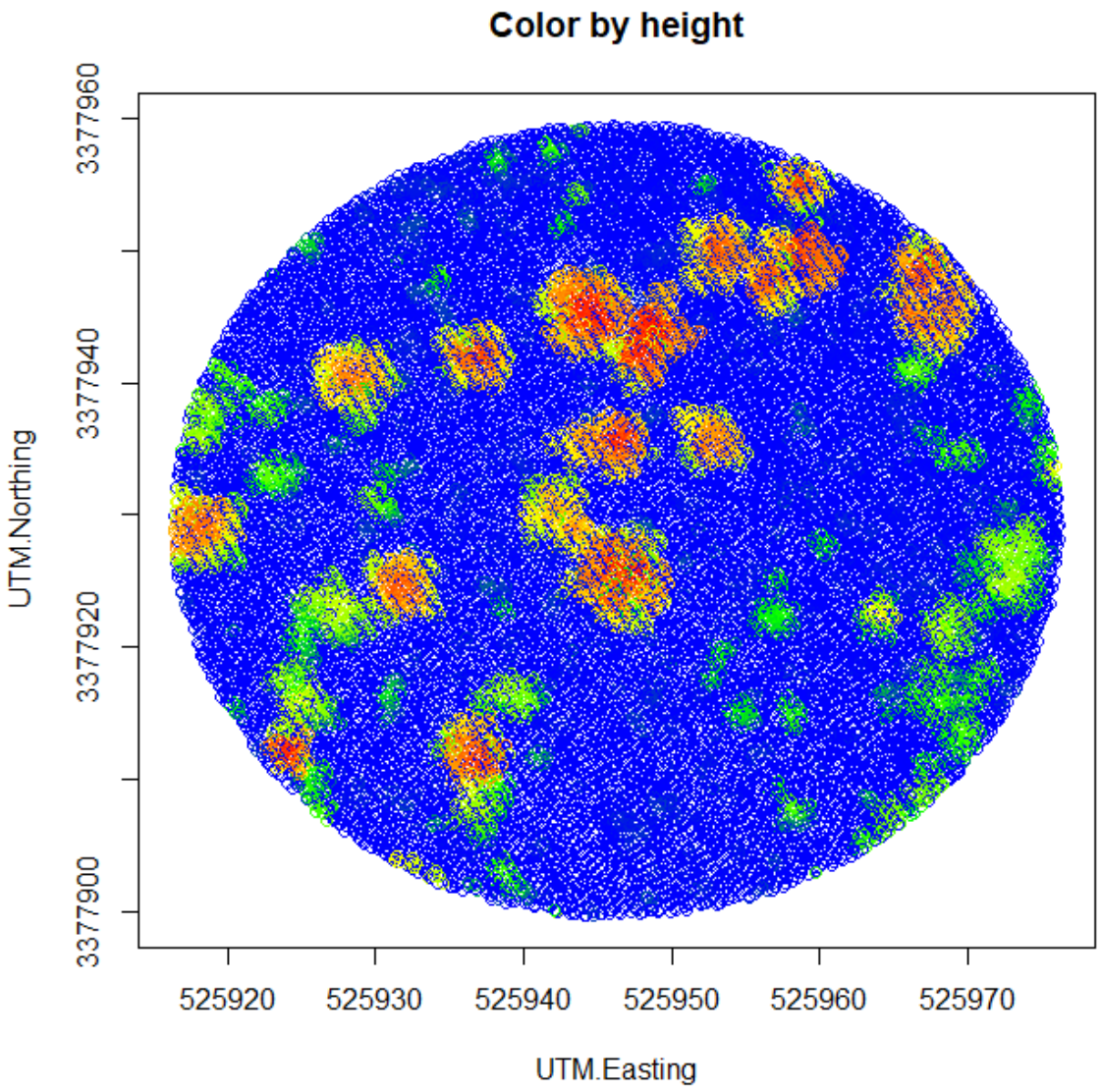



```
# plot 3D library(rgl)
points3d(rLAS[,1:3], col=col, axes=FALSE,xlab="", ylab="", zlab="")
axes3d(c("x+", "y-", "z-"))     # axes
grid3d(side=c('x+', 'y-', 'z'), col="gray") # grid
title3d(xlab = "UTM.Easting", ylab = "UTM.Northing",zlab = "Height(m)", col="red") #
title planes3d(0, 0, -1, 0.001, col="gray", alpha=0.7)# terrain

# 02 Set a color by height
# color ramp
```

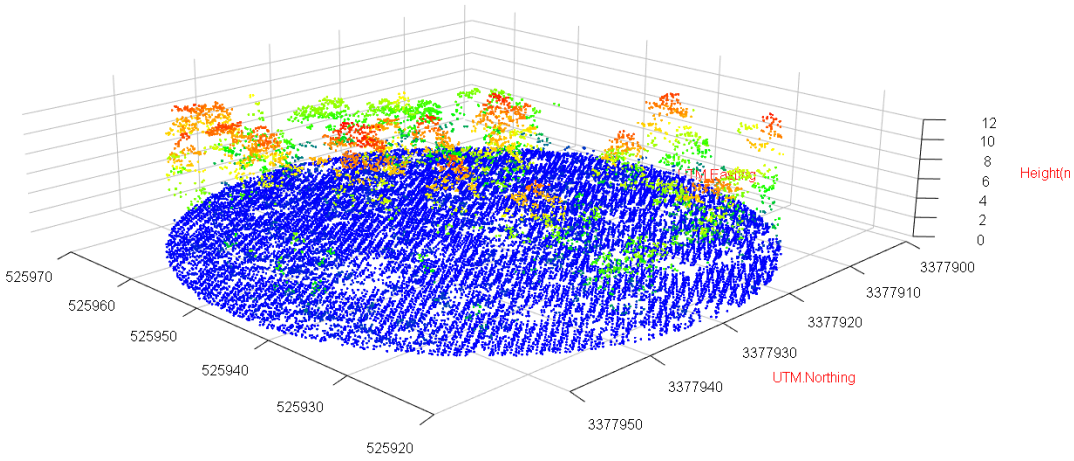
```
myColorRamp <- function(colors, values) {
  v <- (values - min(values))/diff(range(values))
  x <- colorRamp(colors)(v)
  rgb(x[,1], x[,2], x[,3], maxColorValue = 255)
}
```




```
# Color by height
col <- myColorRamp(c("blue","green","yellow","red"),rLAS[,3])

# plot 2D
plot(rLAS[,1], rLAS[,2], col=col, xlab="UTM.Easting", ylab="UTM.Northing",
main="Color by height")
```




```
# plot 3D
points3d(rLAS[,1:3], col=col, axes=FALSE, xlab="", ylab="", zlab="")
axes3d(c("x+", "y-", "z-"))     # axes
grid3d(side=c('x+', 'y-', 'z'), col="gray") # grid
title3d(xlab = "UTM.Easting", ylab = "UTM.Northing",zlab = "Height(m)", col="red") # title
planes3d(0, 0, -1, 0.001, col="gray",alpha=0.7) # terrain
```


12
10
8
6
4
2
0
Height(n
525970
525960
525950
525940
525930
525920
3377900
3377910
3377920
3377930
3377940
3377950
UTM.Northing

References

Silva, C.A., Crookston, N.L., Hudak, A.T., and Vierling,L.A. 2017. rLiDAR: An R package for reading, processing and visualizing lidar (Light Detection and Ranging) data, version 0.1, accessed Oct. 15 2017, < http://cran.r-roject.org/web/packages/rLiDAR/index.html>.

Silva, C.A.; Hudak, A.T.; Vierling, L.A.; Loudermilk, E.L.; O'Brien, J.J.; Hiers, J.K.; Jack, S.B.; Gonzalez-Benecke, C.A.; Lee, H.; Falkowski, M.J.; et al. Imputation of individual longleaf pine forest attributes from field and LiDAR data. Can. J. Remote Sens. 2016, 42, 554–573.

Roussel, J.R, Auty, D., Boissieu., Florian De, and Meador, A. S. 2018. lidR: Airborne LiDAR Data Manipulation and Visualization for Forestry Applications, version 1.4.1, accessed Fev. 15 2018, < https://cran.r-project.org/web/packages/lidR/index.html >.

Plowright, A. ForestTools: Analyzing Remotely Sensed Forest Data, version 0.1.5, accessed Oct. 15 2017, < https://cran.r-project.org/web/packages/ForestTools/index.html>.