
R_GIS 04



Introduction aux traitements de
données de LiDAR aérien avec le
package lidR

Septembre 2026



TABLE DES MATIERES

1. PRÉAMBULE	ERREUR ! SIGNET NON DEFINI.
AUTEUR	ERREUR ! SIGNET NON DEFINI.
LICENCE DE CE DOCUMENT	ERREUR ! SIGNET NON DEFINI.
2. INTRODUCTION.....	1
3. EXEMPLES D'UTILISATION DES FONCTIONNALITÉS DE LIDR.....	1
3.1. PRÉPARATION DE L'ENVIRONNEMENT DE TRAVAIL	2
3.2. CHARGEMENT ET VISUALISATION D'UN FICHIER .LAS (.LAZ)	2
3.3. NORMALISATION D'UN NUAGE DE POINTS	6
3.4. CRÉATION D'UN DEM (DIGITAL ELEVATION MODEL).....	8
3.5. CRÉATION D'UN MNC (MODÈLE NUMÉRIQUE DE CANOPÉE) OU CHM (CANOPY HEIGHT MODEL)	9
3.6. LASDETECTSHAPE : DÉTECTION DE FORMES PLANAIRES (BÂTIMENTS)	12
3.7. CRÉATION D'UN MASQUE « BÂTIMENTS ».....	13
3.8. CRÉATION D'UN MASQUE POUR LES ÉLÉMENTS ARBORÉS	16
3.9. LES CATALOGUES DE DONNÉES LIDAR.....	18
3.9.1. <i>Création d'un catalogue</i>	<i>18</i>
3.9.2. <i>Vérification et indexation d'un catalogue</i>	<i>19</i>
3.9.3. <i>Application d'une fonction lidR à un catalogue : création d'un CHM.....</i>	<i>19</i>
3.10. MODÈLE DENDROMÉTRIQUE POUR DES FORÊTS RÉSINEUSES.....	23
3.10.1. <i>Lire les données d'entrée</i>	<i>24</i>
3.10.2. <i>Extraire les metrics LiDAR sur les placettes d'inventaire</i>	<i>25</i>
3.10.3. <i>Ajuster un modèle de prédiction de la hauteur dominante</i>	<i>27</i>
3.10.4. <i>Appliquer le modèle en plein</i>	<i>28</i>
3.11. SEGMENTATION DES CIMES D'ARBRES	31

1. Introduction

- L'objectif de cet exercice est de présenter les principales fonctionnalités du package lidR dédié au traitement et à l'analyse de données de LiDAR aérien (<https://github.com/Jean-Romain/lidR>).
- lidR est un package R open source destiné à la recherche et au développement (R&D), qui permet de manipuler facilement des données LiDAR à l'aide du langage de programmation R. Il est largement utilisé dans le monde depuis 10 ans et a été cité dans plus de 1 000 publications scientifiques.
- Les manipulations sont présentées au travers de l'environnement de travail RStudio. Elles sont réalisées avec la **version 4.4.3 de R** et la **version 4.2.1 de lidR**.
- Le présent document décrit le déroulé de l'exercice, en le structurant en paragraphes. Dans chacun de ceux-ci sont présentés les différents concepts qui sont illustrés par des extraits du script de référence et des résultats obtenus par l'exécution de ces derniers.
- Le package lidR est doté d'un guide d'utilisateur très complet accessible avec ce lien : <https://r-lidar.github.io/lidRbook/index.html>.

2. Installation de lidR

- Le package lidR présente des dépendances vis-à-vis du package rlas. De ce fait, leur installation s'effectue depuis un dépôt spécifique.

```
# Installation de rlas et lidR depuis 1 dépôt spécifique
if(F){
  install.packages(
    c("rlas", "lidR"),
    repos = c(
      "https://r-lidar.r-universe.dev",
      "https://cloud.r-project.org"
    )
  )
}

library(lidR)
library(rlas)
# version de R et de lidR
R.version$version.string
packageVersion("lidR")
packageVersion("rlas")
> R.version$version.string
[1] "R version 4.6.1 (2026-06-24 ucrt)"
> packageVersion("lidR")
[1] '4.3.3'
> packageVersion("rlas")
[1] '1.9.5'
```

- Les exemples présentés dans ce tutoriel ont été testés avec les versions 4.6.1 pour R, 4.3.3 de lidR et 1.9.5 de rlas.

3. Exemples d'utilisation des fonctionnalités de lidR

3.1. Préparation de l'environnement de travail

- Cette partie du script concerne le chargement des librairies.

```
# 3.1. Préparation de l'environnement de travail -----  
# chargement des librairies  
library(sf)  
library(lidR)  
library(dplyr)  
library(mmand)  
library(mapview)  
library(qgisprocess)  
library(terra)  
library(rgl)  
# clear memory  
rm(list=ls())
```

- Les chemins vers les répertoires (« données sources » et « résultats ») sont ensuite définis. L'existence de ces répertoires est testée avec la fonction `dir.exists()`.
- Remarque : le contenu de la variable `path0` doit être adapté en fonction de l'endroit où le jeu de données a été installé.

```
# Chemins vers les répertoires  
path0="E:/3_tthr/exo_dossiers/R_GIS_04"  
path_in=paste0(path0,"/input") # contient les données d'entrée  
path_out=paste0(path0,"/output") # sauvegarde des résultats  
# Vérifier si les chemins sont ok  
dir.exists(path_in)  
dir.exists(path_out)  
  
> dir.exists(path_in)  
[1] TRUE  
> dir.exists(path_out)  
[1] TRUE
```

3.2. Chargement et visualisation d'un fichier .las (.laz)

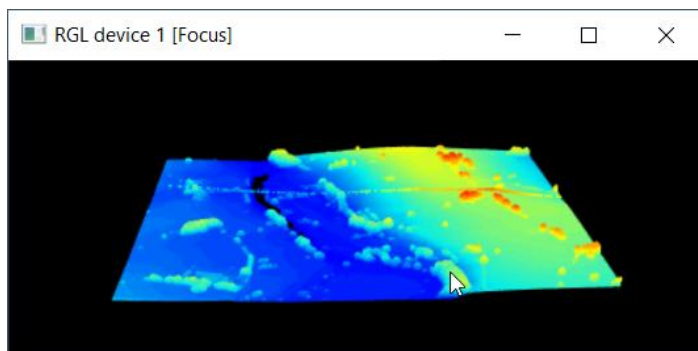
- La fonction **`readLAS()`** est utilisée pour charger un fichier .las (ou .laz) dans un objet las.

```
# 3.2. lecture et exploration d'un fichier .LAS -----  
  
las1 = readLAS(paste0(path_in,"/laz/13.laz"))  
  
# Quelques infos sur le fichier .laz  
las1
```

```
> # Quelques infos sur le fichier .laz
> las1
class      : LAS (v1.2 format 1)
memory     : 98.1 Mb
extent     : 214000, 215000, 45000, 45999.99 (xmin, xmax, ymin, ymax)
coord. ref.: BD72 / Belgian Lambert 72
area       : 0.98 km2
points     : 1.61 million points
type       : airborne
density    : 1.64 points/m2
density    : 1.42 pulses/m2
```

- Les informations de base qui décrivent le fichier .las (ou .laz) concernent son emprise spatiale (extent), le système de coordonnées, le nombre de points, ainsi que la densité de points et de pulsations.
- Comme on peut le constater, les fichiers .las (ou .laz) contiennent des volumes de données très importants (1,61 millions de points dans le cas du fichier **13.laz**).
- La fonction **plot()** est utilisée pour afficher le nuage de points dans une fenêtre graphique indépendante de l'environnement RStudio.

```
# Afficher le nuage de points
plot(las1)
```



- La structure d'un objet **las** peut être « analysée » avec la fonction **str()**.

```
# Structure d'un objet las
str(las1, 3)
```

```

> str(las1,3)
Formal class 'LAS' [package "lidR"] with 4 slots
..@ data :Classes 'data.table' and 'data.frame': 1607721 obs. of 16 variables:
.. ..$ X : num [1:1607721] 214000 214000 214000 214000 214000 ...
.. ..$ Y : num [1:1607721] 45842 45853 45854 45844 45845 ...
.. ..$ Z : num [1:1607721] 285 285 285 285 285 ...
.. ..$ gpstime : num [1:1607721] 76711310 76711310 76711310 76711310 76711310 ...
.. ..$ Intensity : int [1:1607721] 431 447 374 419 392 399 425 403 479 440 ...
.. ..$ ReturnNumber : int [1:1607721] 1 1 1 1 1 1 1 1 1 1 ...
.. ..$ NumberOfReturns : int [1:1607721] 1 1 1 1 1 1 1 1 1 1 ...
.. ..$ ScanDirectionFlag: int [1:1607721] 0 0 0 0 0 0 0 0 0 0 ...
.. ..$ EdgeOfFlightline : int [1:1607721] 0 0 1 0 0 0 0 0 0 0 ...
.. ..$ Classification : int [1:1607721] 2 2 2 2 2 2 2 2 2 2 ...
.. ..$ Synthetic_flag : logi [1:1607721] FALSE FALSE FALSE FALSE FALSE FALSE ...
.. ..$ Keypoint_flag : logi [1:1607721] FALSE FALSE FALSE FALSE FALSE FALSE ...
.. ..$ Withheld_flag : logi [1:1607721] FALSE FALSE FALSE FALSE FALSE FALSE ...
.. ..$ ScanAngleRank : int [1:1607721] -30 -30 -30 -30 -30 -30 -30 -30 -29 -29 ...
.. ..$ UserData : int [1:1607721] 3 3 3 3 3 3 3 3 3 3 ...
.. ..$ PointSourceID : int [1:1607721] 5026 5026 5026 5026 5026 5026 5026 5026 5026 5026
...
.. ..- attr(*, ".internal.selfref")=<externalptr>
..@ header:Formal class 'LASheader' [package "lidR"] with 3 slots
..@ crs :List of 2
.. ..$ input: chr "EPSG:31370"
.. ..$ wkt : chr "PROJCRS[\"BD72 / Belgian Lambert 72\",\n BASEGEOGCRS[\"BD72\", \n
DATUM[\"Reseau National Belge 1972\"]| __truncated__
.. ..- attr(*, \"class\")= chr \"crs\"
..@ index :List of 2
.. ..$ sensor: int 1
.. ..$ index : int 0
..
..@ bbox : num [1:2, 1:2] 214000 45000 215000 46000
.. ..- attr(*, \"dimnames\")=List of 2
.. .. ..$ : chr [1:2] \"x\" \"y\"
.. .. ..$ : chr [1:2] \"min\" \"max\"
..@ proj4string:Formal class 'CRS' [package \"sp\"] with 1 slot
.. ..- attr(*, \"proj4args\")= chr NA

```

- Un objet las est constitué de 4 slots :

- @data : dataframe décrivant les points LiDAR ;
- @header : informations techniques relatives au fichier ;
- @bbox : emprise de la scène couverte par le fichier. Il s'agit généralement de tuiles carrées ;
- @proj4string : système de coordonnées dans lequel sont définis les points du nuage.

Formellement, un objet **las**, c'est un dataframe contenant un grand nombre de points et possédant des attributs décrivant les propriétés de ces points.

- Le dataframe contient notamment leurs coordonnées X, Y et Z, ainsi qu'un temps GPS associé à l'impulsion laser, ou encore un code de classification (points sol, végétation...) fourni par le producteur. La signification des codes de classification des points LiDAR est décrite en annexe 1.

```

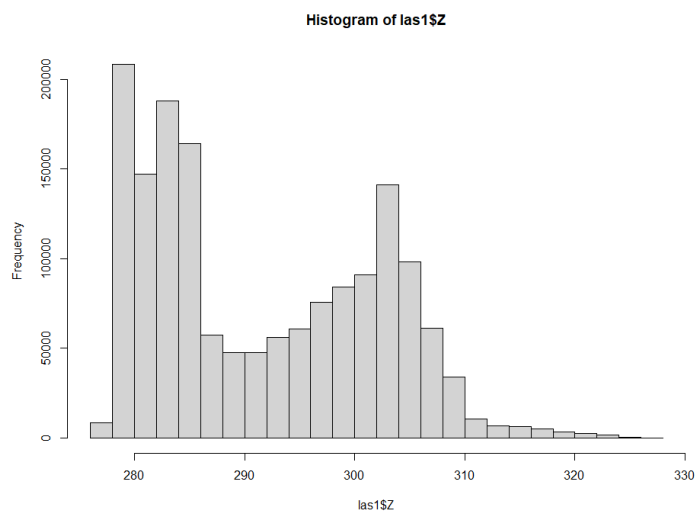
# Contenu du slot @data
head(las1@data)

```

```
> head(las1@data)
      X      Y      Z gpstime Intensity ReturnNumber NumberOfReturns
1: 215938.8 44000.03 359.43 76710772      395           2             2
2: 215938.1 44000.03 358.46 76710772      416           2             2
3: 215925.1 44000.03 357.33 76710772       21           1             3
4: 215922.8 44000.00 350.64 76710772      324           3             3
5: 215920.7 44000.00 358.27 76710772       33           1             4
6: 215917.6 44000.02 355.02 76710772       69           3             4
      ScanDirectionFlag EdgeOfFlightline Classification Synthetic_flag Keypoint_flag
1:                   0                   0              4          FALSE          FALSE
2:                   0                   0              4          FALSE          FALSE
3:                   0                   0              4          FALSE          FALSE
```

- Les données d'altitude peuvent être analysées sous la forme d'un histogramme.

```
# Z : altitude des points
hist(las1$Z)
```



- La fonction **clip_rectangle()** est utilisée pour extraire des parties du nuage de points délimitées selon une emprise rectangulaire (x_{min} , y_{min} , x_{max} , y_{max}).
- Le nouvel objet las est ensuite transformé en objet sf avec la fonction **st_as_sf()**. L'objet sf peut ensuite être exporté sous forme de geopackage.

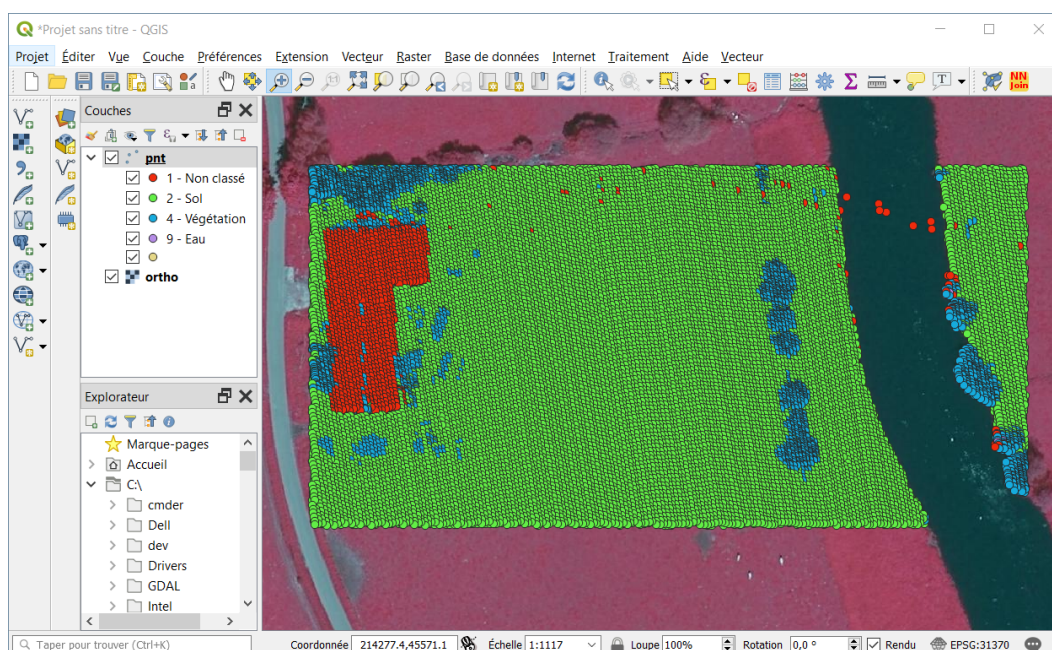
```
# Extraire une partie du nuage de points
# puis convertir en geopackage

las2=clip_rectangle(las1,214100,45600,214300,45700)
pnt=st_as_sf(las2)
nrow(pnt)
plot(pnt$geometry)

f_out=paste0(path_out,"/pnt.gpkg")
st_write(pnt,f_out,overwrite=TRUE,delete_layer = T)
```

- Remarque importante :** les formats vectoriels classiquement utilisés dans les SIG ne sont pas du tout adaptés pour la manipulation de nuages de points LiDAR, à moins de ne concerner que de très petites quantités de points comme c'est le cas ici. Il n'est donc pas recommandé de convertir les nuages de points lidar en couche vectorielle (format shp ou gpkg).

- Observer le contenu du shapefile **pnt.gpkg** dans QGIS.
- Afficher la couche **ortho.vrt** en arrière-plan. Elle se trouve dans le répertoire **\ortho**.
- Appliquer au fichier **pnt.gpkg** une symbologie de type « catégorisé » en considérant le champ [Classification] correspondant à la classification des points. Utiliser le fichier de style **las_classification.qml**.
- Un élément intéressant à noter en observant la superposition de la couche de points et de l'orthoimage est l'absence d'écho LiDAR au niveau du cours d'eau.
- Un autre enseignement de l'analyse visuelle de cet extrait du nuage de points est que la classification proposée par le producteur de la couverture LiDAR est loin d'être parfaite.



3.3. Normalisation d'un nuage de points

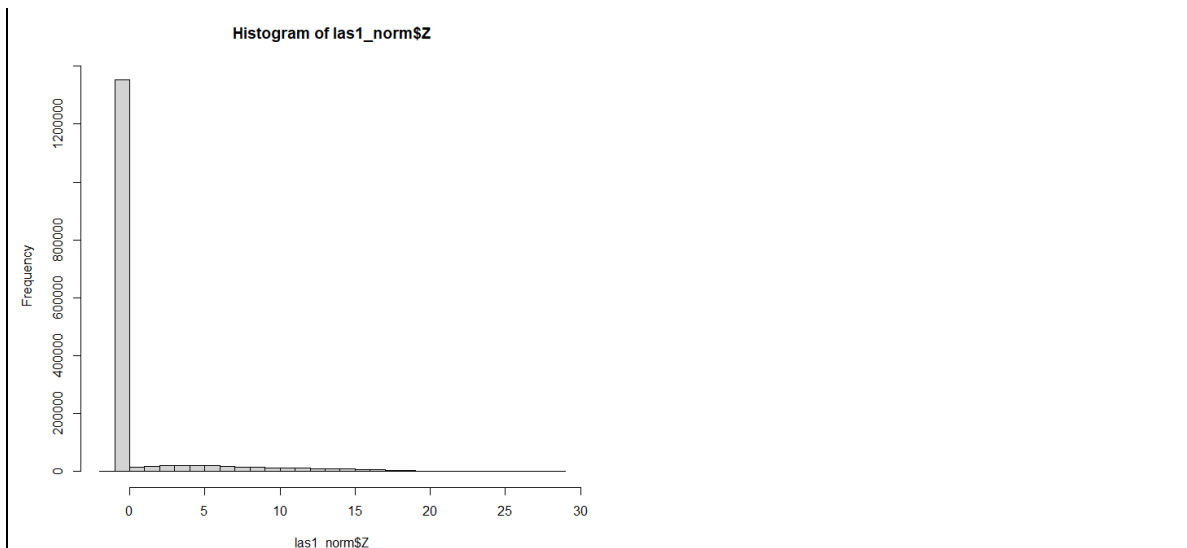
- La normalisation d'un nuage de points consiste à déterminer la hauteur de chaque point par rapport au niveau du sol et à stocker cette information dans le champ [Z] du slot @data.

La donnée « altitude » est quant à elle sauvegardée dans un nouveau champ baptisé [Zref].

- La fonction **normalize_height()**, qui prend en charge cette transformation, nécessite de définir l'algorithme qui calcule le niveau du sol. L'exemple qui suit utilise l'algorithme **tin()**. Ce dernier réalise une triangulation sur les points de la classe « sol » pour déterminer l'altitude du sol et, par différence, la hauteur de chaque point au-dessus du sol.

```
# 3.3. Normalisation du nuage de points -----

las1_norm=normalize_height(las1,tin())
hist(las1_norm$Z)
```



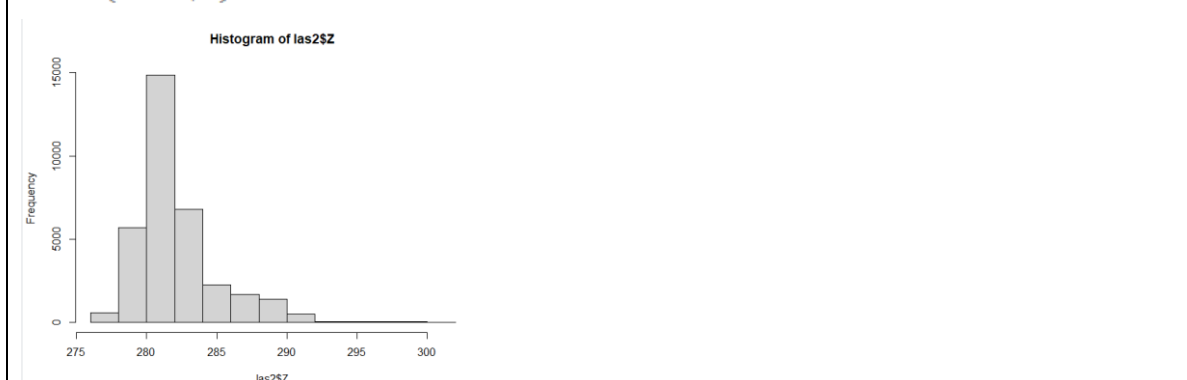
- L'opération de normalisation remplace l'altitude par la hauteur dans le champ las1@data\$Z. L'information relative à l'altitude n'est cependant pas complètement supprimée. Elle est recopiée dans un nouveau champ baptisé las1@data\$Zref.

```
# L'altitude est conservée dans le champ Zref
names(las1_norm@data)
```

```
> names(las1_norm@data)
[1] "X"           "Y"           "Z"           "gpstime"
[5] "Intensity"   "ReturnNumber" "NumberOfReturns" "ScanDirectionFlag"
[9] "EdgeOfFlightline" "Classification" "Synthetic_flag" "Keypoint_flag"
[13] "Withheld_flag" "ScanAngleRank" "UserData"      "PointSourceID"
[17] "Zref"
```

- La fonction **unnormalize_height()** réalise l'opération inverse : elle recopie le champ data\$Zref dans le champ data\$Z.

```
# Unnormalize : opération inverse
las2=unnormalize_height(las1_norm)
hist(las2$Z)
```

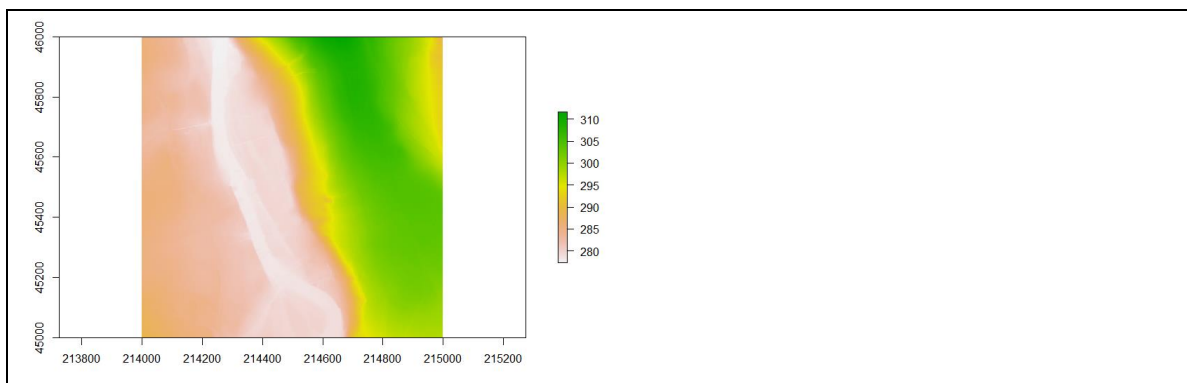


3.4. Création d'un DEM (Digital Elevation Model)

- Les Modèles Numériques de Terrain (ou Digital Elevation Model) sont sans doute un des principaux produits dérivés des nuages de points LiDAR.
- La fonction ***rasterize_terrain()*** réalise une interpolation des points classés « sols » pour générer un MNT sous forme de couche raster. Les paramètres à définir concernent la résolution du raster ainsi que l'algorithme utilisé pour l'interpolation. Dans l'exemple qui suit, c'est l'algorithme ***knnidw()*** qui est utilisé. Celui-ci applique une interpolation de type IDW (*inverse-distance weighting*) sur les plus proches voisins du pixel considéré.

```
# 3.4. Rasterize_terrain : construction d'un MNT -----  
  
# Utiliser des données non normalisées (sinon z=0 pour les points sol)  
dem=rasterize_terrain(las1, res = 1, knnidw(), keep_lowest = FALSE)  
plot(dem)
```

- **Remarque** : l'option **keep_lowest** concerne le fait de conserver ou pas les valeurs d'altitude qui sont inférieures aux valeurs interpolées. Dans le cas présent, cette option est rejetée et seules les valeurs interpolées sont prises en compte.



- **Remarque importante** : la construction d'un MNT ne peut s'appliquer à un nuage de points normalisé, puisque dans ce dernier, les points « sols » ont une hauteur nulle.

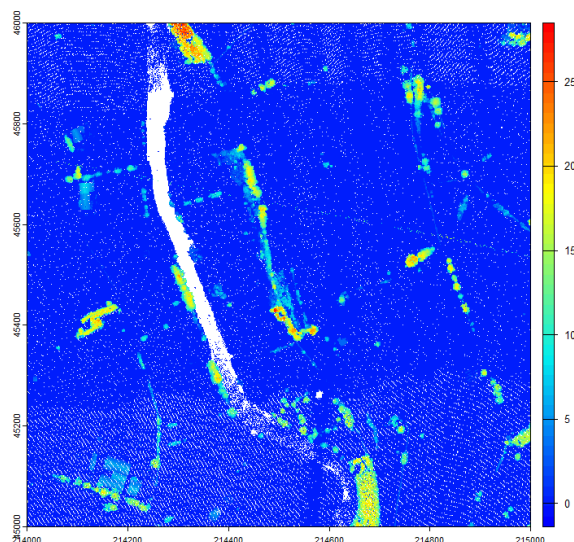
3.5. Création d'un MNC (Modèle Numérique de Canopée) ou CHM (Canopy Height Model)

Les paragraphes 3.5 à 3.8 expliquent comment créer un Modèle Numérique de Canopée et comment exploiter ce dernier pour générer une couche mettant en évidence la présence d'éléments arborés dans le paysage.

- Une autre couche dérivée du nuage de points LiDAR qui intéresse particulièrement le forestier est le **Modèle Numérique de Canopée (MNC)** ou **Canopy Height Model (CHM)** qui est une représentation raster continue de la hauteur des éléments présents à la surface du sol.
- Le MNC est généré avec la fonction `rasterize_canopy()`. Les paramètres à définir concernent la résolution du raster de sortie, ainsi que l'algorithme utilisé pour définir la hauteur du couvert au sein de chaque pixel.
- **Remarque importante** : la fonction `rasterize_canopy()` peut être appliquée à un nuage de point normalisé ou non normalisé. Selon le cas, le résultat prendra la forme d'un MNC (hauteur) ou d'un MNS (altitude).
- L'algorithme `p2r()` est le plus simple et le plus rapide. Il attribue à chaque pixel l'altitude (ou la hauteur) du point le plus élevé au sein du pixel.

```
# 3.5. Création d'un Modèle Numérique de Canopée (CHM) --
```

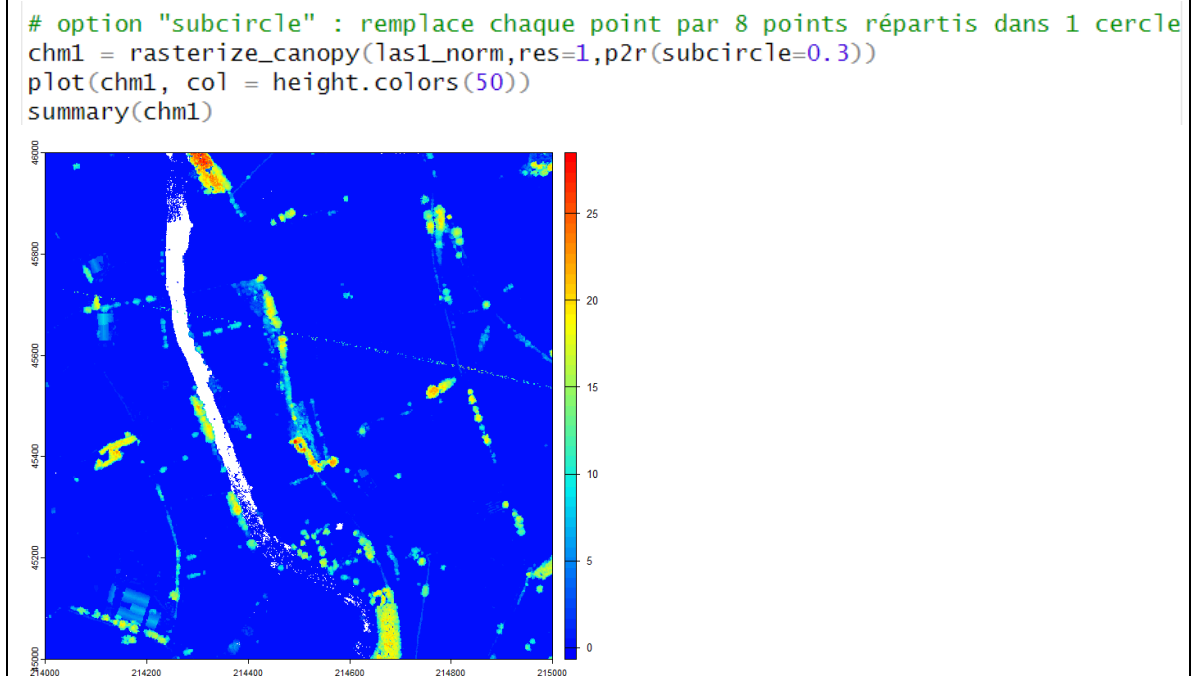
```
# algorithm p2r : points to raster
# considère le z maximum par pixel
chm1 = rasterize_canopy(las1_norm,res=1,p2r())
plot(chm1, col = height.colors(50))
```



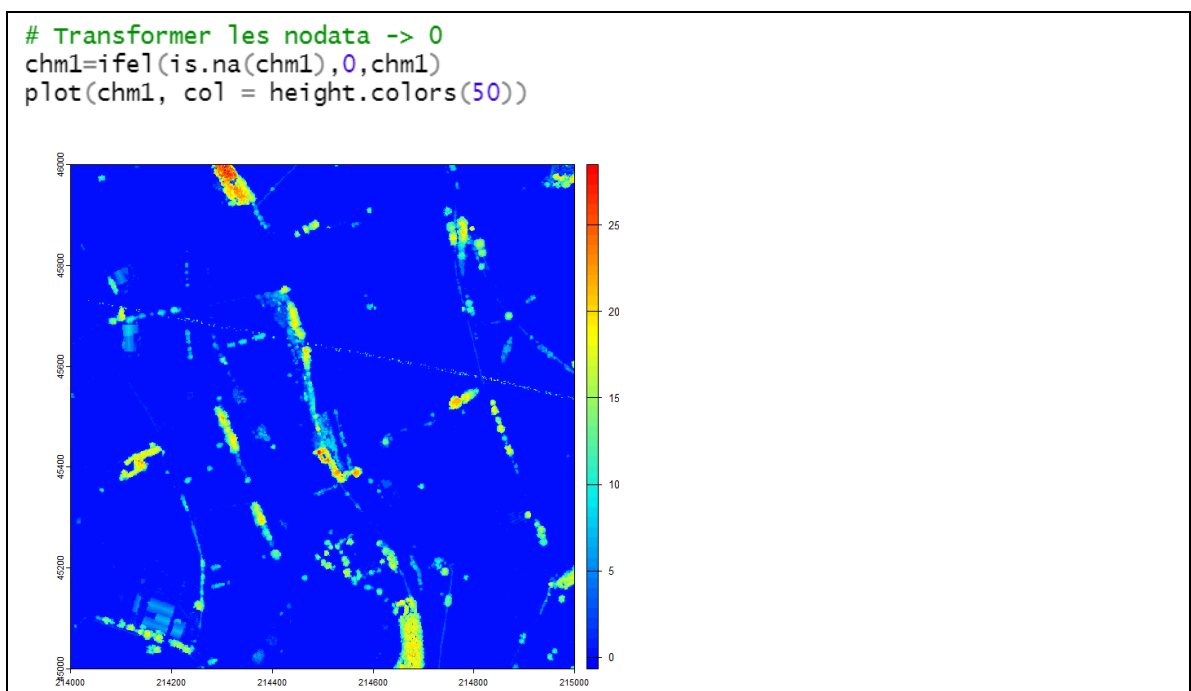
- **Remarque** : tous les produits rasters générés par lidR le sont sous forme d'objets **SpatRaster**.

```
> class(chm1)
[1] "SpatRaster"
attr(,"package")
[1] "terra"
```

- La principale faiblesse de l'algorithme **p2r()** tient au fait qu'en l'absence de point LiDAR dans un pixel, une valeur « NODATA » est attribuée à celui-ci. Pour atténuer ce phénomène, on utilise l'option **subcircle()**. Celle-ci remplace chaque point LiDAR par un groupe de 8 points de même altitude distribués au sein d'un cercle de rayon donné (30 cm dans l'exemple ci-dessous).

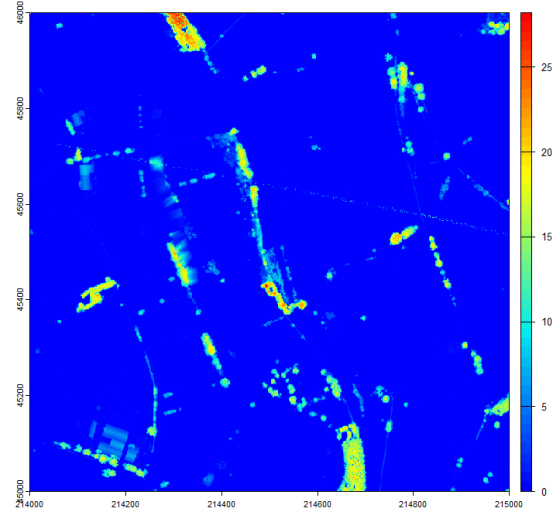


- Malgré l'utilisation de cette option, il subsiste de nombreux pixels « NODATA » dans le CHM produit. Ceux-ci se trouvent sur le cours d'eau qui traverse la tuile. S'agissant d'un CHM, il est conseillé de leur attribuer une valeur nulle (figure ci-dessous).



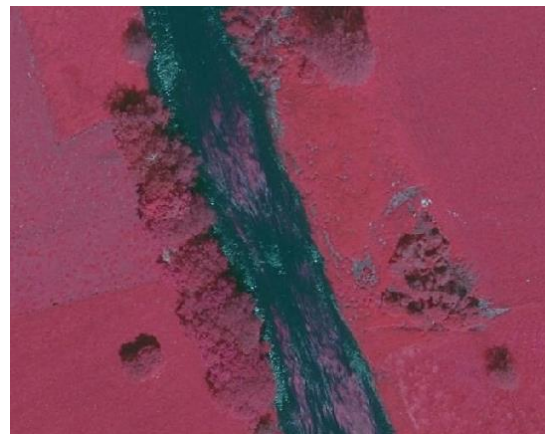
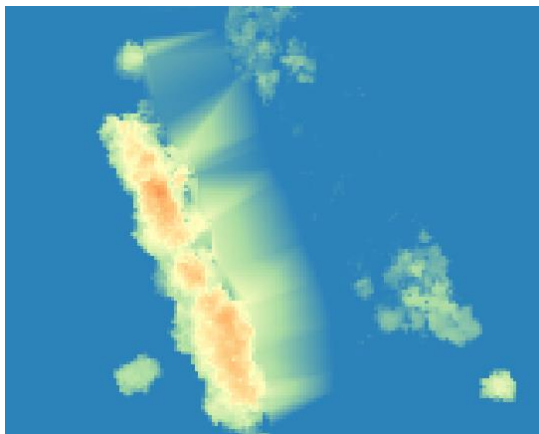
- L'algorithme **dsmtin()** génère un réseau TIN (*Triangulated Irregular Network*) sur le nuage de points « premiers retours » avant d'interpoler les valeurs de hauteur pour chaque pixel.

```
# Algorithme dsmtin : triangulation TIN sur les premiers retours
chm2 = rasterize_canopy(las1_norm, res=1, dsmtin())
plot(chm2, col = height.colors(50))
```



- Le résultat ne présente plus de pixels vides. En l'absence de points sur des surfaces importantes, l'interpolation réalisée sur les facettes triangulaires du TIN peut produire des artéfacts. C'est notamment le cas lorsque des arbres sont situés le long d'un cours d'eau.

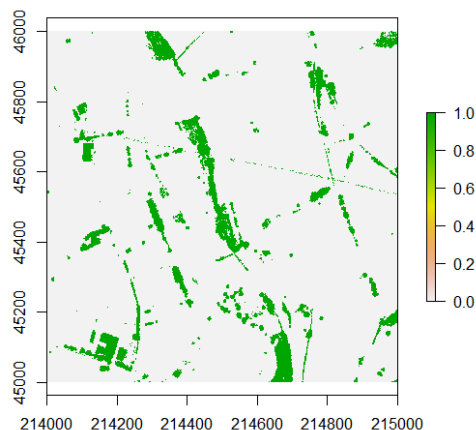
```
# exporter chm2 en fichier .tif pour l'afficher dans QGIS
setwd(path_output)
writeRaster(chm2, "chm2.tif", overwrite=TRUE)
```



- lidR dispose d'un troisième algorithme pour générer un CHM. Il s'agit de **pitfree()**, qui est une version améliorée de **dsmtin()**. L'algorithme **pitfree()** sera abordé plus loin dans ce tutoriel.
- Remarque** : le tutoriel « Rasterizing perfect canopy height models » (<https://github.com/Jean-Romain/lidR/wiki/Rasterizing-perfect-canopy-height-models>) présente en détails les différentes approches conduisant à la création d'un CHM.
- Pour la suite de l'exercice, nous allons considérer le CHM produit avec l'algorithme **p2r()**.

- Dans la perspective d'une cartographie des éléments ligneux présents dans le paysage, on applique généralement un seuillage au CHM produit afin de créer un « masque » identifiant les éléments présentant une hauteur supérieure à une valeur seuil. Dans la suite de l'exercice, nous allons considérer une valeur seuil de 2 m.

```
# seuiller le chm pour récupérer les éléments hors sol
chm_gt2=chm>2
plot(chm_gt2)
```

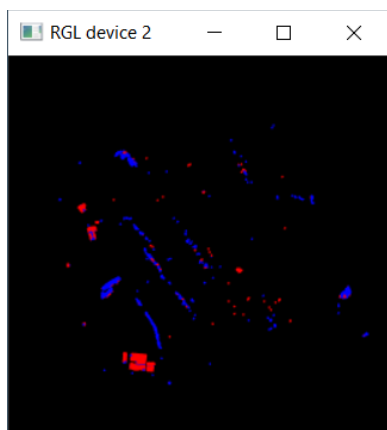


- **Remarque** : en présence de cultures (éléments non ligneux) susceptibles de dépasser la valeur seuil, il est nécessaire de relever celle-ci.

3.6. Lasdetectshape : détection de formes planaires (bâtiments)

- Dans la perspective d'une détection des points situés sur des bâtiments, un premier filtre est appliqué : il sélectionne les points correspondant à une impulsion LiDAR qui n'a produit qu'un seul retour et dont la hauteur est comprise entre 3 m et 30 m. On fait l'hypothèse que la hauteur des bâtiments présents dans la scène est comprise entre ces deux valeurs.

```
# 3.6. Détection d'une structure planaire (bâti) -----
# Présélection des points (1 seul retour, 3 m < hauteur < 30 m)
las2=filter_poi(las1_norm,NumberOfReturns==1,Z>3,Z<30)
plot(las2)
```



- La fonction **`segment_shapes()`**, utilisée avec l'algorithme **`shp_plane()`**, teste la présence d'une organisation planaire des points dans le voisinage d'un point donné. Le paramètre `k = 30` désigne le nombre de points considéré pour l'ajustement d'un plan. Si la structure planaire est détectée, le point reçoit la valeur « TRUE » dans l'attribut « Coplanar ».

```
# Extraction des points organisés en surfaces planes
las2 <- segment_shapes(las2, shp_plane(k = 30), attribute = "Coplanar")
table(las2$Coplanar)

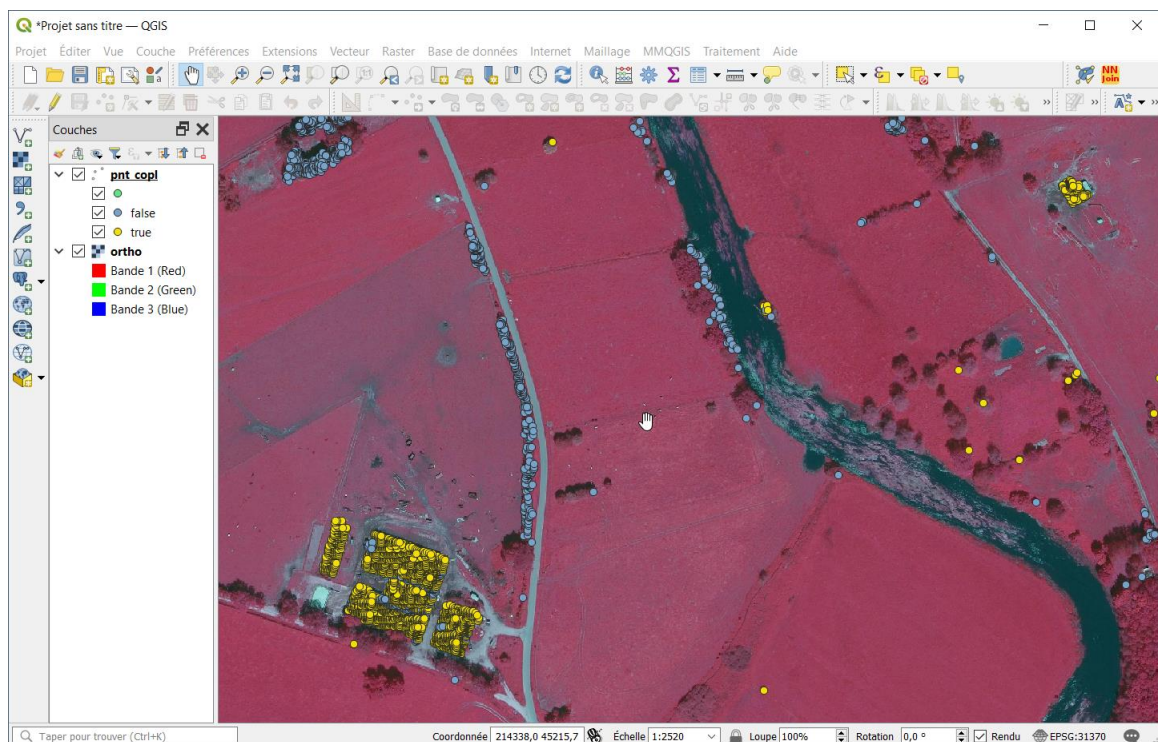
> table(las2$Coplanar)

FALSE  TRUE
 2970  5445
```

- Pour visualiser le résultat de ce traitement, le plus simple est d'exporter le nuage de points sous forme d'un geopackage et de l'afficher dans QGIS.

```
# Export des points de las2 dans un geopackage
sp2=st_as_sf(as.spatial(las2))
st_write(sp2,paste0(path_out,"/pnt_cop1.gpkg"),delete_layer=T)
```

- Lorsqu'on visualise les points dans QGIS, on constate que les bâtiments sont correctement détectés, mais que des faux positifs ont été générés. Il s'agit le plus souvent de points isolés ou en petits groupes.



3.7. Création d'un masque « bâtiments »

- Les points qui viennent d'être identifiés comme situés sur des bâtiments (Coplanar = TRUE) peuvent ensuite être convertis en couche raster représentant les surfaces bâties.

- Ce processus est réalisé en plusieurs étapes :
 - Générer un raster qui calcule la densité de points ;
 - Sélectionner les pixels dont la densité est > 0 (raster binaire) ;
 - Appliquer un filtre « majorité » pour supprimer une partie des faux positifs.

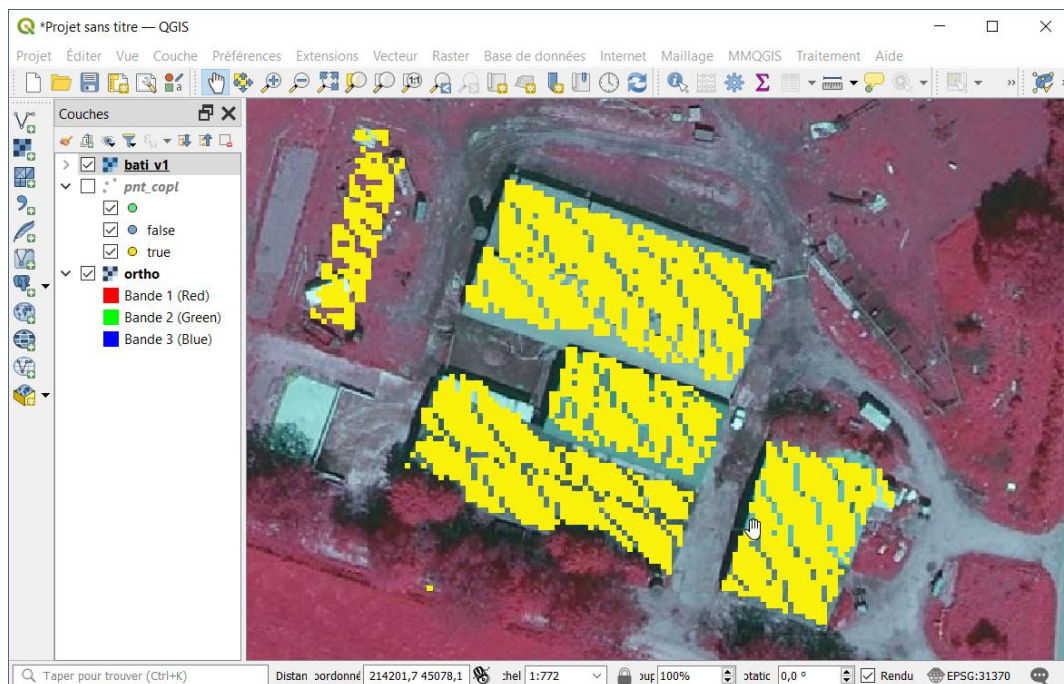
```
# 3.7 création d'un masque pour les surfaces baties -----

# sélection des points "coplanaires"
las2=filter_poi(las2, Coplanar==TRUE)

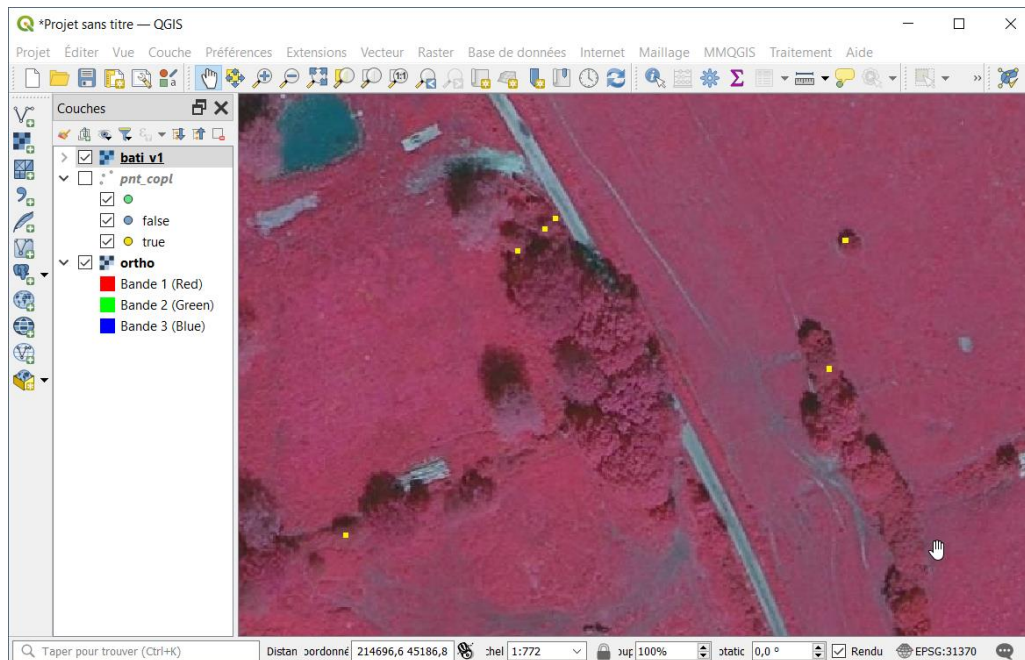
# Conversion du nuage de points en raster de densité de points
bati=rasterize_density(las2,res=1)
plot(bati)

# Création d'un raster "bati" binaire
f_bati_v1=paste0(path_out, "/bati_v1.tif")
bati_v1=ifel(bati>0,1,0,filename=f_bati_v1,overwrite=T)
plot(bati_v1)
```

- Le résultat peut être visualisé dans QGIS. On constate qu'avec une résolution de 1 m, certains pixels ne contiennent pas de points. En outre, les bords des bâtiments ne sont pas repris dans le masque.



- Les faux positifs qui avaient été identifiés précédemment sont également présents dans cette première version du masque.



- Pour corriger ces problèmes, on va utiliser successivement une fonction de tamisage (élimination des petits groupes de pixels) et un filtre morphologique (amélioration de la morphologie des groupes de pixels).
- La fonction de tamisage est mise en œuvre via l'extension qgisprocess et l'algorithme « **gdal:sieve** ». Un seuil de 6 pixels est utilisé.

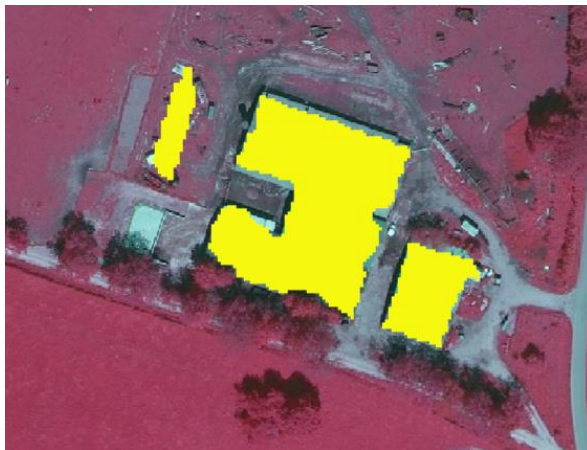
```
# Création d'un raster "bati" binaire
f_bati_v1=paste0(path_out,"/bati_v1.tif")
bati_v1=ifel(bati>0,1,0,filename=f_bati_v1,overwrite=T)
plot(bati_v1)

# Tamisage du raster (suppression des faux positifs)
f_bati_v2=paste0(path_out,"/bati_v2.tif")
algo = "gdal:sieve"
result = qgis_run_algorithm(
  algo,
  INPUT = f_bati_v1,
  THRESHOLD = 6,
  OUTPUT = f_bati_v2,
  .quiet = T
)
```

- Le filtre morphologique utilisé est de type « closing » (application successive d'une dilatation puis d'une érosion). Cette fonction est issue de la librairie **mmand**.

```
# Filtre morphologique
m=as.matrix(bati_v2,wide=T)
k <- shapeKernel(c(5,5), type="disc")
k
m=mmand::closing(m,k)
bati_v2 <- setValues(bati_v2, m)
plot(bati_v2)
writeRaster(bati_v2,file=f_bati_v2,overwrite=TRUE)
```

- La figure qui suit présente le résultat obtenu après ces deux étapes.



3.8. Création d'un masque pour les éléments arborés

- La création d'un masque décrivant l'emprise des éléments arborés peut s'obtenir en combinant le CHM et le masque des bâtiments. Avant de les combiner, il faut s'assurer qu'ils présentent la même emprise spatiale.

```
# 3.8 Création d'un masque pour les éléments arborés
# combinaison du chm et du masque "bati"

f_bati=paste0(path_out, "/bati_v2.tif")
bati=rast(f_bati_v2)
f_chm=paste0(path_out, "/chm1.tif")
chm=rast(f_chm)

# Comparer les emprises spatiales des 2 couches
st_bbox(bati)
st_bbox(chm)

> st_bbox(bati)
  xmin ymin xmax ymax
214000 45036 214972 46000
> st_bbox(chm)
  xmin ymin xmax ymax
214000 45000 215000 46000
```

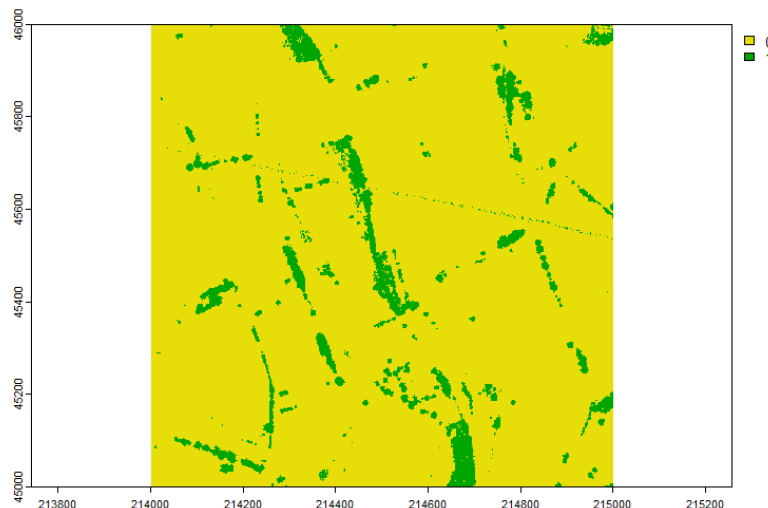
- On constate que l'emprise de la couche **bati** est moindre que celle de **chm**.
- La fonction **extend()** permet d'étendre l'emprise d'une couche pour qu'elle corresponde à celle d'une autre couche, les nouveaux pixels recevant des valeurs « nodata ». Ces dernières peuvent ensuite être converties en « 0 » avec la fonction **ifel()**.

```
# Etendre le raster bati (compléter avec des "0")
bati=extend(bati,chm)
bati=ifel(is.na(bati),0,bati)
summary(bati)

st_bbox(bati)
st_bbox(chm)
```

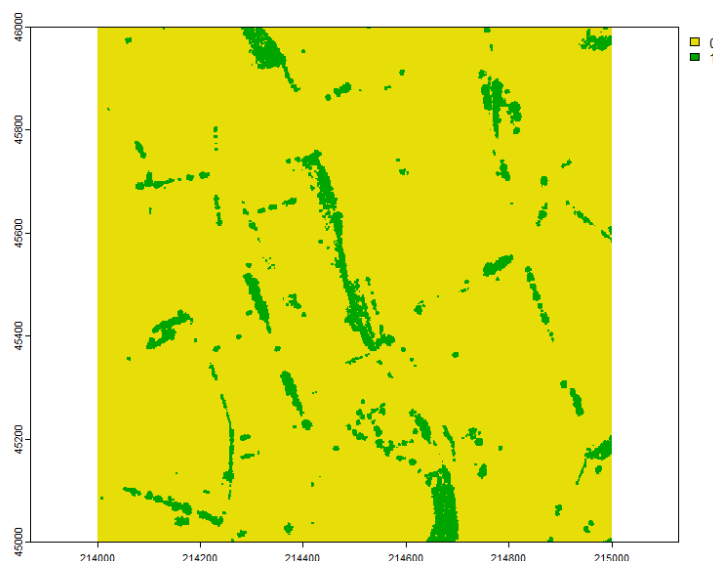
- Les deux couches peuvent maintenant être combinées pour faire ressortir les éléments arborés présents dans la zone d'étude. Ceux-ci seront définis comme suit : éléments dont la hauteur est supérieure ou égale à 3 m et qui ne sont pas des bâtiments.

```
# Combiner les 2 rasters en seuillant le chm à 3 m
f_trees_v1=paste0(path_out, "/trees_v1.tif")
trees_v1=ifel(chm>=3 & bati==0,1,0,filename=f_trees_v1,overwrite=T)
plot(trees_v1)
```



- L'application d'un filtre morphologique de type « opening » (érosion suivie d'une dilatation) permet de nettoyer la couche en supprimant les plus petits objets, correspondant notamment aux artéfacts liés aux lignes électriques.

```
# Appliquer 1 filtre morphologique sur le résultat
m=as.matrix(trees_v1,width=T)
k <- shapeKernel(c(3,3), type="disc")
m=opening(m,k)
trees_v2 <- setValues(trees_v1, m)
plot(trees_v2)
writeRaster(trees_v2,paste0(path_out, "/trees_v2.tif"),overwrite=TRUE)
```



3.9. Les catalogues de données LiDAR

3.9.1. Création d'un catalogue

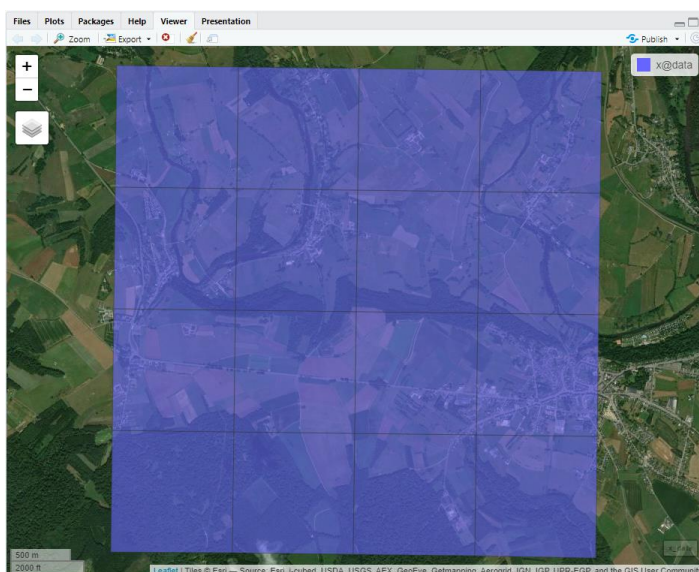
- Les catalogues sont des outils particulièrement efficaces pour gérer et traiter simultanément un ensemble de fichiers .las, même si celui-ci contient plusieurs milliers de fichiers.
- L'intérêt principal des catalogues est de réduire le temps d'accès aux données, qui constitue la partie la plus importante du temps de traitement, loin devant le temps de calcul.
- La création d'un catalogue est donc généralement la première étape à entreprendre lorsque l'on souhaite exploiter une couverture LiDAR constituée d'un grand nombre de fichiers. Elle s'opère avec la fonction **catalog()**, en renseignant tout simplement le nom du répertoire dans lequel sont rangés les fichiers .las ou .laz.

```
# 3.9.1. Création d'un catalogue --
path_laz=paste0(path_in,"/laz")
cat1=catalog(path_laz)
cat1

> cat1
class      : LAScatalog (v1.2 format 1)
extent     : 214000, 218000, 42000, 45999.99 (xmin, xmax, ymin, ymax)
coord. ref.: Belge 1972 / Belgian Lambert 72
area       : 16 km²
points     : 40.37 million points
density    : 2.5 points/m²
density    : 1.4 pulses/m²
num. files : 16
```

- Le catalogue **cat1** rassemble les données des 16 fichiers .laz contenus dans le répertoire \laz des données de l'exercice. Ces 16 fichiers totalisent plus de 40 millions de points LiDAR.
- Il est possible d'afficher un catalogue avec un fond de carte en arrière-plan.

```
# Affichage du catalogue avec 1 couche de référence en arrière-plan
plot(cat1, mapview = TRUE, map.type = "Esri.WorldImagery")
```



- La fonction **st_as_sf()**, appliquée à un catalogue, génère un objet **sf** contenant des polygones correspondant aux tuiles de ce catalogue.

```
# Création d'une couche de polygones correspondant
# aux tuiles d'un catalogue
tuiles=st_as_sf(cat1)
names(tuiles)
plot(tuiles$geometry)
f_out=paste0(path_out,"/tuiles.gpkg")
st_write(tuiles,f_out,delete_layer=T)
```

3.9.2. Vérification et indexation d'un catalogue

- La fonction **las_check()** est utilisée pour contrôler les fichiers constitutifs du catalogue.

```
# Vérification du catalogue
las_check(cat1)

Checking headers consistency
- Checking file version consistency... ✓
- Checking scale consistency... ✓
- Checking offset consistency... ✓
- Checking point type consistency... ✓
- Checking VLR consistency... ✓
- Checking CRS consistency... ✓
Checking the headers
- Checking scale factor validity... ✓
- Checking Point Data Format ID validity... ✓
Checking preprocessing already done
- Checking negative outliers... ✓
- Checking normalization... no
Checking the geometry
- Checking overlapping tiles... ✓
- Checking point indexation... yes
```

- La fonction **catalog_laxindex()** est utilisée pour générer les fichiers d'indexation des différentes tuiles d'un catalogue. Ces fichiers d'indexation possèdent l'extension **.lax**. Ces fichiers d'indexation accélèrent le temps d'accès aux tuiles d'un catalogue. Il est donc recommandé de les créer lorsqu'ils n'existent pas.
- Remarque : le catalogue cat1 a déjà été indexé. Il n'est pas nécessaire de répéter l'opération.

```
# Indexation du catalogue (optimise l'accès aux données)
# lidR::catalog_laxindex(cat1) # l'indexation est déjà réalisée
```

3.9.3. Application d'une fonction lidR à un catalogue : création d'un CHM

- La plupart des fonctions de calcul applicables à un objet **.las** le sont également à un catalogue.
- Dans l'exemple qui suit, on procède à la création d'un CHM sur l'ensemble du catalogue en utilisant la fonction **rasterize_canopy()** qui avait été utilisée au § 3.5 pour générer un CHM sur une tuile.

3.9.3.1. Création du catalogue de points normalisés

- Avant de créer le CHM, il convient de normaliser les nuages de points pour transformer les altitudes en hauteurs.
- Préalablement, on a créé un répertoire qui va recevoir les nouveaux fichiers .laz.

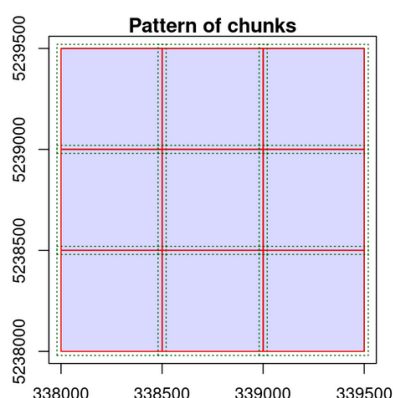
```
# 3.9.3.1 Créer 1 catalogue avec des nuages de points normalisés ----

# Répertoire pour les laz normalisés
path_norm = paste0(path_out, "/laz_norm")
if(!dir.exists(path_norm)){
  dir.create(path_norm)
}
```

- Ensuite, on adapte les options du catalogue à la normalisation des nuages de points qui va être réalisée. Ces options vont concerner principalement l'accès aux données du catalogue ou la manière de sauvegarder les fichiers de résultats.

```
# Adapter les options du catalogue
opt_progress(cat1)=TRUE # affiche la progression du traitement
opt_chunk_buffer(cat1)=10 # utilise un buffer de 10 m pour traiter chaque tuile
path_norm=paste0(path_out, "/laz_norm")
opt_output_files(cat1) = paste0(path_norm, "{ORIGINALFILENAME}") # nom des outputs
opt_laz_compression(cat1)=TRUE
```

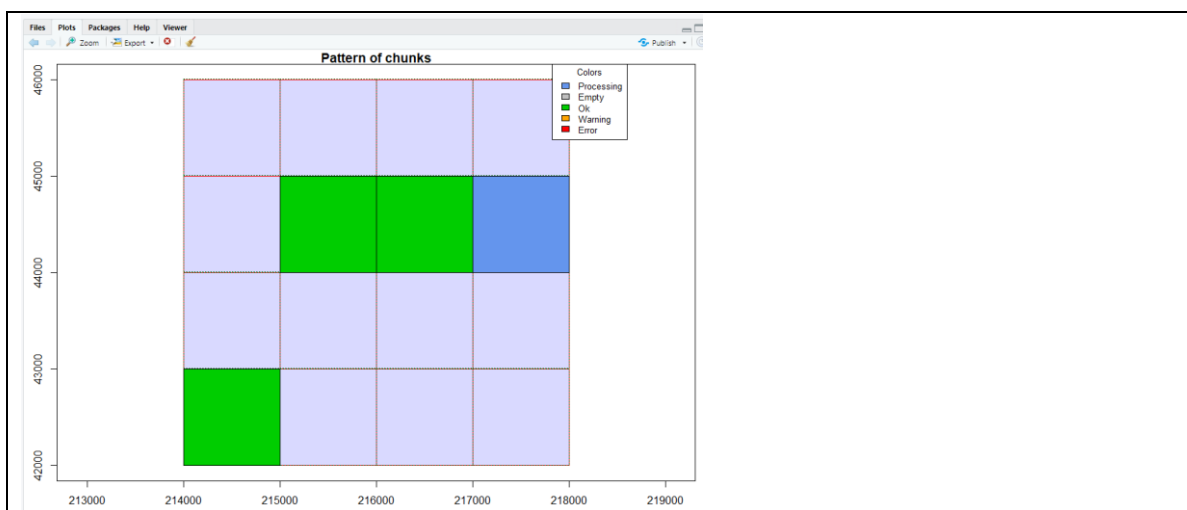
- L'option « `opt_progress(cat1)=TRUE` » permet l'affichage d'une barre de progression dans la console de RStudio.
- L'option « `opt_chunk_buffer(cat1)=10` » signifie que lors du traitement d'une tuile, lidR prend également en compte les points des tuiles voisines qui sont situées dans un buffer de 10 m autour de la tuile en cours de traitement. Cette option est particulièrement importante lorsque les algorithmes utilisés réalisent des interpolations sur des ensembles de points. Le buffer évite d'avoir des effets de bords indésirables. Le terme « chunk » (« morceaux ») désigne les parties du catalogue soumises à un traitement.
- La figure suivante illustre le découpage en chunks (« morceaux ») avec un buffer de 20 m pour des tuiles de 500 m de côté. Les limites des tuiles sont représentées en rouge et les limites des chunks en vert pointillé.



- L'option « `opt_output_files(cat1)` » définit la manière avec laquelle les fichiers de sortie sont gérés. Dans l'exemple présent, ils sont sauvegardés dans le répertoire `/path_norm` et le nom des fichiers de sortie est le même que celui des tuiles (`ORIGINALFILENAME`).
- Enfin, l'option « `opt_laz_compression(cat1)=TRUE` » fait en sorte que les fichiers de sortie sont produits au format `.laz` au lieu de `.las`.
- La normalisation du nuage de points est finalement opérée avec la fonction **`normalize_height()`**. Dans l'exemple qui est présenté ci-dessous, la hauteur des points n'est pas calculée par rapport aux « points sol » comme dans le § 3.5, mais bien par rapport aux pixels d'une couche raster représentant le MNT.

```
# Normaliser le catalogue
t1=Sys.time()
cat2=lidR::normalize_height(cat1, algorithm=tin())
t2=Sys.time()
(t2-t1)
> (t2-t1)
Time difference of 3.027933 mins
```

- **Remarque** : lors de l'exécution d'une fonction sur un catalogue, la fenêtre « Viewer » affiche la progression du traitement sur le catalogue. Les « chunks » sont progressivement coloriés selon leur statut (vert : traitement terminé, bleu : traitement en cours...).



3.9.3.2. Création d'un CHM

- Une fois le catalogue de nuages de points normalisés disponible, celui-ci peut être utilisé pour générer un CHM, avec la fonction **`rasterize_canopy()`**.

```
# 3.9.3.2 Créer 1 CHM sur l'ensemble du catalogue ----

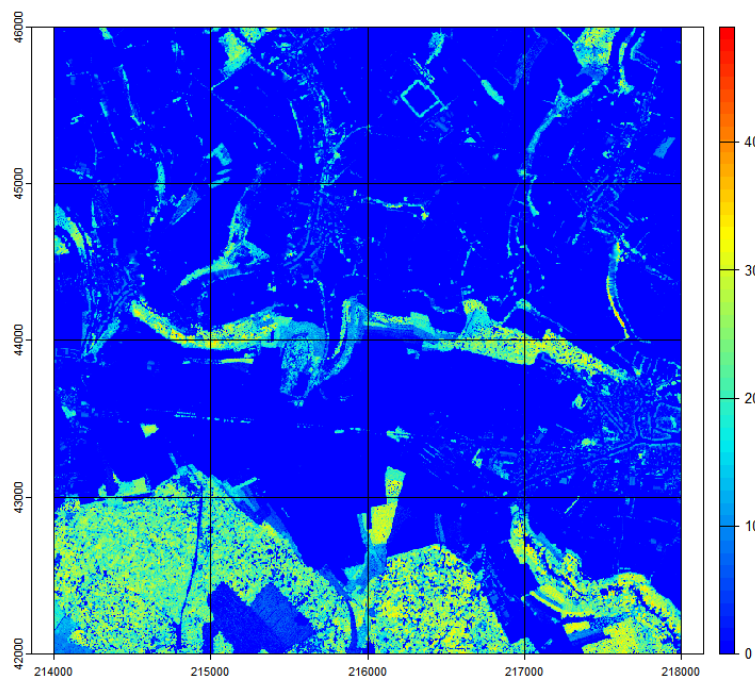
# Répertoire pour les tuiles du CHM
path_chm = paste0(path_out, "/chm_florenville")
if(!dir.exists(path_chm)){
  dir.create(path_chm)
}

# Adapter les options du catalogue
opt_progress(cat2)=TRUE
opt_chunk_buffer(cat2)=10
opt_output_files(cat2) = paste0(path_chm, "/{ORIGINALFILENAME}")

# Créer le CHM
t1=Sys.time()
chm=lidR::rasterize_canopy(cat2,res=1,pitfree(c(0,2,5,10), c(0, 1.5)))
t2=Sys.time()
(t2-t1)

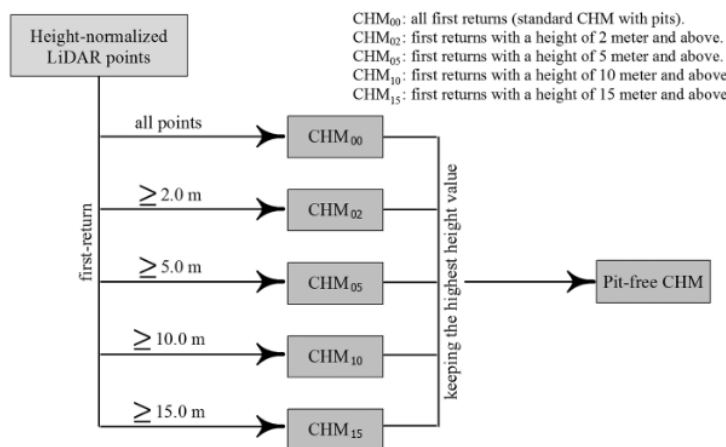
> (t2-t1)
Time difference of 6.058469 mins

plot(chm, col = height.colors(50))
plot(tuiles$geometry,add=T)
```



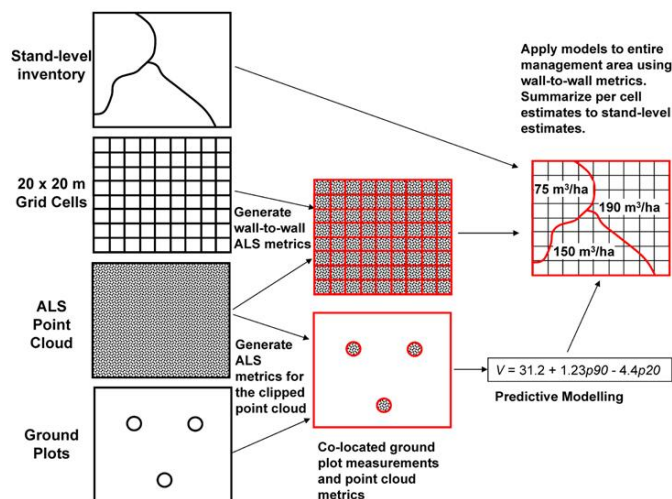
- **Remarque** : le répertoire dans lequel ont été sauvegardées les tuiles du CHM contient également un fichier **rasterize_canopy.vrt** reprenant l'ensemble des fichiers .tif sous forme d'une mosaïque. Il peut être utilisé dans QGIS pour afficher l'ensemble du CHM.
- Dans l'exemple qui vient d'être présenté, nous avons utilisé l'algorithme « pitfree ». La figure ci-dessous présente le principe de fonctionnement de celui-ci. Il génère une série de chm basé sur des seuils de hauteur croissant (ici 2, 5, 10 et 15 m). Ces différents chm sont ensuite assemblés en conservant, pour chaque pixel, la valeur la plus élevée. Cette manière de procéder atténue la

présence de trous (pits) dans les houppiers des arbres. L'article décrivant cette approche est présent dans le répertoire \documents (pitfree.pdf).



3.10. Modèle dendrométrique pour des forêts résineuses

- L'intérêt du LiDAR aérien pour le gestionnaire forestier réside notamment dans le fait que certaines variables dendrométriques peuvent être assez facilement prédites au départ de variables (« metrics ») dérivées de données de LiDAR aérien. L'approche la plus utilisée pour réaliser ces prédictions est l'approche ABA (*Area Based Approach*). Elle est représentée de manière schématique dans la figure qui suit (source : White et al., 2013, <https://pubs.cif-ifc.org/doi/10.5558/tfc2013-132>).



- Dans la suite, nous présentons un exemple simple de ce type de modèle qui vise à estimer la hauteur dominante dans les peuplements d'épicéa et de douglas dans la région des Epioux.
- Les données de base sont issues d'un inventaire de terrain qui a permis d'estimer différentes variables dendrométriques au sein de 23 placettes. Les données relatives à ces placettes sont rassemblées dans le shapefile **plots.shp**. Le champ [hdom] contient les estimations de hauteur dominante (en m), dérivées de la mesure des plus gros arbres de chaque placette.

3.10.1. Lire les données d'entrée

- Le fichier **plots.gpkg** contient les localisations ainsi que la description de 18 placettes d'inventaire forestier d'aménagement.

```
# 3.10.1. Lire les données d'entrée -----

# placettes d'inventaire (IFA)
f_pl=paste0(path_in,"/plots.gpkg")
plots=st_read(f_pl)
names(plots)

> names(plots)
[1] "id"      "rayon"  "slope"  "surf"   "hdom"   "gha"    "vha"    "nha"    "geom"
```

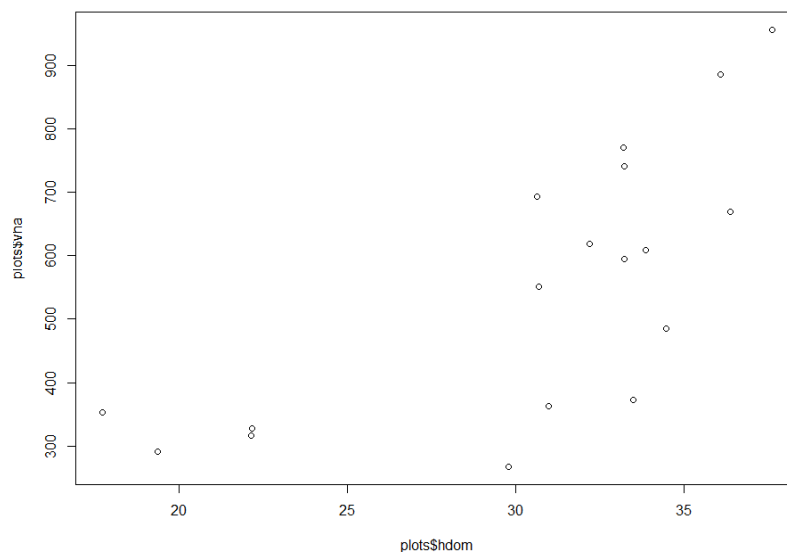
- Parmi les attributs de cette couche, on retrouve le rayon et la surface des placettes, ainsi que les principales variables dendrométriques dérivées des mesures de terrain : hauteur dominante, volume, surface terrière et nombre de tiges par ha.
- Les hauteurs dominantes varient entre 18 et

```
summary(plots)
plot(plots$hdom,plots$vha) # Hauteur dominante vs Volume par ha

> summary(plots)
```

id	rayon	slope	surf	hdom
Length:18	Min. : 6.00	Min. : 3.000	Min. : 111.0	Min. :17.73
Class :character	1st Qu.:10.28	1st Qu.: 4.000	1st Qu.: 334.8	1st Qu.:29.99
Mode :character	Median :12.15	Median : 4.000	Median : 472.0	Median :32.70
	Mean :11.97	Mean : 4.944	Mean : 480.7	Mean :30.40
	3rd Qu.:13.72	3rd Qu.: 5.000	3rd Qu.: 593.0	3rd Qu.:33.78
	Max. :17.90	Max. :14.000	Max. :1003.0	Max. :37.62

gha	vha	nha	geom
Min. :22.70	Min. :267.2	Min. : 199.0	POLYGON :18
1st Qu.:32.83	1st Qu.:355.4	1st Qu.: 274.8	epsg:31370 : 0
Median :42.30	Median :572.7	Median : 328.5	+proj=lcc ...: 0
Mean :41.74	Mean :548.0	Mean : 413.1	
3rd Qu.:47.75	3rd Qu.:687.2	3rd Qu.: 493.5	
Max. :61.00	Max. :956.2	Max. :1057.0	



- Une série de fichiers .laz a été générée au départ d'un catalogue complet couvrant la zone d'étude. Ces fichiers .laz ont été découpés à l'aide de buffers de 10 m autour de chaque placette. Cette étape préliminaire avait pour seul objectif de limiter le volume des données à manipuler.
- Ces fichiers .laz se trouvent dans le répertoire /laz_plots. Même s'ils ne sont pas organisés en tuiles jointives, ils peuvent être exploités sous la forme d'un catalogue.



3.10.2. Extraire les metrics LiDAR sur les placettes d'inventaire

- La fonction **plot_metrics()** est spécifiquement dédiée au calcul de metrics standards au sein de surfaces correspondant à des placettes d'inventaires. Le format de sortie de cette fonction est un dataframe rassemblant les metrics ainsi que les attributs de la couche cartographique contenant les limites des placettes. Dans le cas présent, nous limitons le calcul des metrics standards associés à la hauteur en utilisant l'option « .stdmetrics_z ».

```
# 3.10.2 Extraire les métriques de base sur les placettes
df <- plot_metrics(cat1, .stdmetrics_z, plots)
names(df)
```

> names(df)				
[1] "id"	"rayon"	"slope"	"surf"	"hdom"
[6] "gha"	"vha"	"nha"	"zmax"	"zmean"
[11] "zsd"	"zskew"	"zkurt"	"zentropy"	"pzabovemean"
[16] "pzabove2"	"zq5"	"zq10"	"zq15"	"zq20"
[21] "zq25"	"zq30"	"zq35"	"zq40"	"zq45"
[26] "zq50"	"zq55"	"zq60"	"zq65"	"zq70"
[31] "zq75"	"zq80"	"zq85"	"zq90"	"zq95"
[36] "zpcum1"	"zpcum2"	"zpcum3"	"zpcum4"	"zpcum5"
[41] "zpcum6"	"zpcum7"	"zpcum8"	"zpcum9"	"geometry"

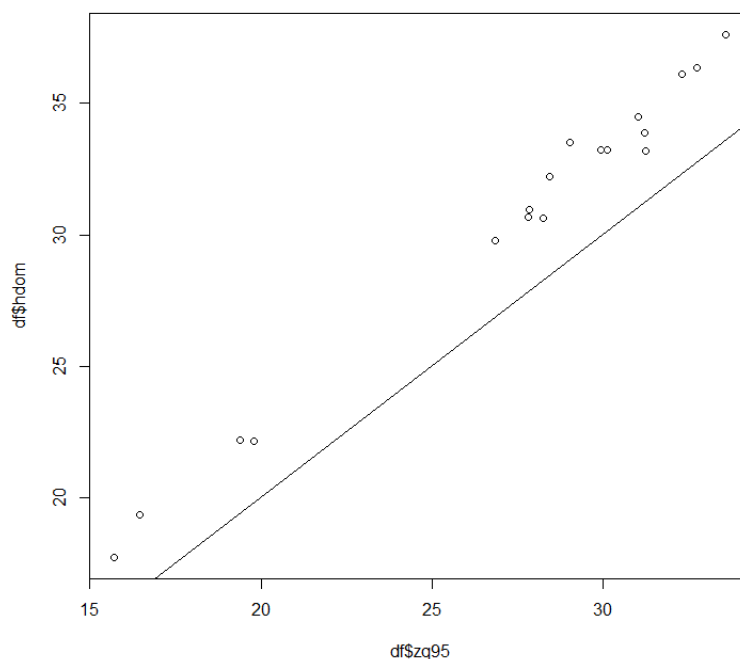
- Les principaux metrics sont décrits en **annexe 2**. Dans la littérature, la prédiction de la hauteur dominante est souvent réalisée au départ de quantiles de hauteur (zq_x).

```
df= as.data.frame(plots2)
m=as.matrix(select(df, hdom, zq85:zq95, zmax))
cor(m)
> cor(m)
```

	hdom	zq85	zq90	zq95	zmax
hdom	1.0000000	0.9936131	0.9943133	0.9950178	0.9903329
zq85	0.9936131	1.0000000	0.9996835	0.9984323	0.9841977
zq90	0.9943133	0.9996835	1.0000000	0.9994354	0.9870374
zq95	0.9950178	0.9984323	0.9994354	1.0000000	0.9910582
zmax	0.9903329	0.9841977	0.9870374	0.9910582	1.0000000

- On propose d'estimer la hauteur dominante en fonction du 95^{ème} percentile (zq95). La relation graphique entre la hauteur dominante et zq95 est linéaire. Elle présente un biais dans la mesure où le zq95 est systématiquement inférieur à hdom. Cela s'explique par le fait que les impulsions lidar ne parviennent généralement pas à être réfléchies par les apex des arbres.

```
plot(df$zq95, df$hdom)
abline(0,1)
```



3.10.3. Ajuster un modèle de prédiction de la hauteur dominante

```
# 3.10.3 Ajuster un modèle de prédiction de la Hauteur dominante
m1 <- lm(hdom ~ zq95, data = df)
summary(m1)
> summary(m1)

Call:
lm(formula = hdom ~ zq95, data = df)

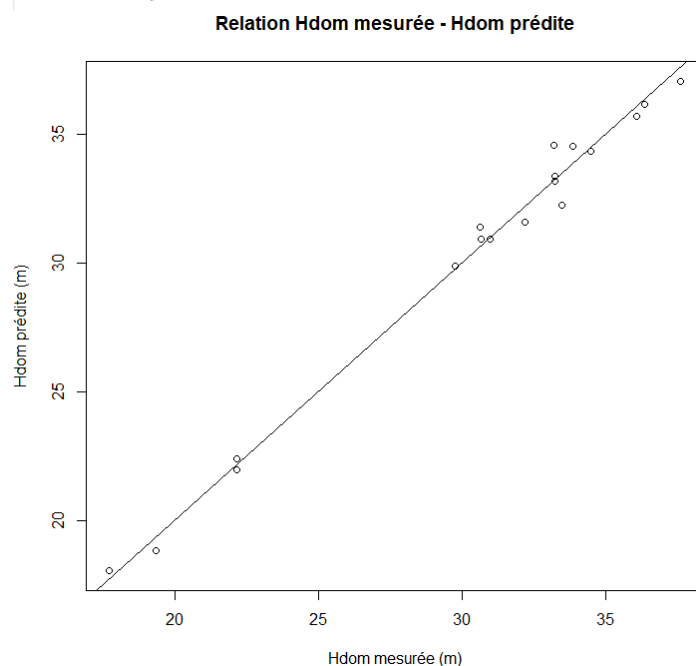
Residuals:
    Min       1Q   Median       3Q      Max
-1.36318 -0.26170  0.02727  0.36300  1.26266

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.3855     0.7411   1.869   0.08 .
zq95          1.0620     0.0266  39.922 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6131 on 16 degrees of freedom
Multiple R-squared:  0.9901,    Adjusted R-squared:  0.9894
F-statistic: 1594 on 1 and 16 DF,  p-value: < 2.2e-16
```

- On constate que la pente du modèle linéaire est très proche de 1 (1,06 avec une erreur standard de 0,027), ce qui traduit la « quasi proportionnalité » entre les deux variables. L'ordonnée à l'origine est significativement différente de 0 (1,39 m). Cela signifie que le signal LiDAR va, en moyenne, sous-estimer la hauteur des arbres dominants de la placette de 1,5 m.

```
# graphique valeurs mesurées - valeurs prédites
plot(df$hdom, predict(m1), main="Relation Hdom mesurée - Hdom prédite",
     xlab="Hdom mesurée (m)", ylab="Hdom prédite (m)")
abline(0,1)
```



3.10.4. Appliquer le modèle en plein

- Ce modèle peut ensuite être utilisé pour réaliser des prédictions « en plein », c'est-à-dire sur l'ensemble des peuplements résineux de la zone d'étude. Dans l'exemple qui suit, la prédiction est réalisée sur une tuile de 1km x 1km située dans la région de Florenville (domaine des Epioux), disponible dans le répertoire /laz_test.
- Pour appliquer le modèle, il faut disposer des estimations de la variable explicative au sein d'une couche raster dont la résolution doit être du même ordre de grandeur que les placettes d'inventaires. Cela garantit que le metric (ici le h95) est estimé dans les mêmes conditions que pour les données ayant servi à ajuster le modèle.
- Pour définir cette résolution, nous considérons la surface moyenne des placettes circulaires (en m²), qui est convertie en taille de pixel (en m). Celle-ci est fixée à 20 m.

```
# 3.10.4 Appliquer le modèle de prédiction en plein
f_test=paste0(path0,"/input/laz_test/test.laz")
las_test=readLAS(f_test)
st_crs(las_test)=31370

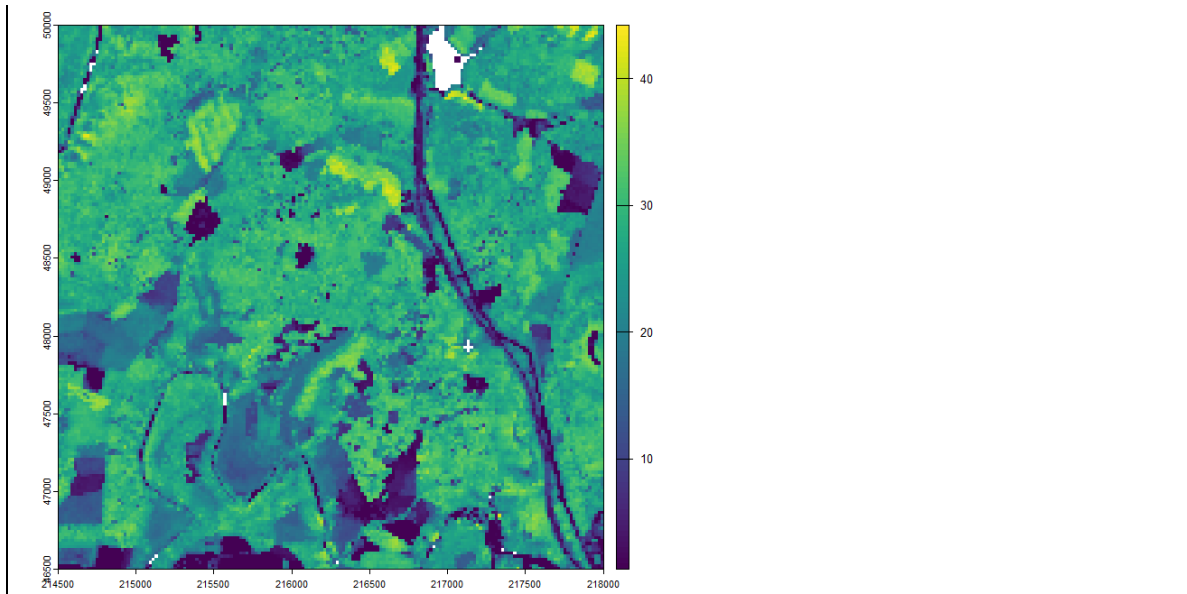
# Taille des placettes de l'inventaire
hist(plots$surf)
(mean(plots$surf))^0.5
> (mean(plots$surf))^0.5
[1] 20.38328
```

- Le calcul des metrics standards au sein d'une couche raster est réalisé avec la fonction **pixel_metrics()**, en précisant la résolution de sortie (ici 20 m) et la liste des metrics à calculer.
- Pour limiter le temps de calcul, nous remplaçons la liste « .stdmetrics_z », par le 95^{ème} percentile de hauteur. Cela permet de raccourcir quelque peu le temps de calcul.

```
# Calcul de zq95 en plein avec 1 résolution de 20 m
###w2w = pixel_metrics(las_test, .stdmetrics_z, res = 20,
###                               pkg = "terra")
w2w = pixel_metrics(las_test, quantile(Z,probs=0.95),
                    res = 20, pkg = "terra")
names(w2w)="zq95"
```

- La prédiction de la hauteur dominante s'effectue très simplement avec la fonction **terra::predict()** qui associe un objet possédant une méthode « predict » (ici le modèle m1) avec un objet SpatRaster.

```
# Prédiction de Hdom en plein
hdom <- terra::predict(w2w, m1)
names(hdom)="hdom_est"
plot(hdom)
```



- La couche produite couvre l'ensemble de la zone d'étude. Pour qu'elle soit pertinente, et correctement interprétable, il convient de la combiner avec une couche qui délimite les peuplements forestiers résineux (épicéa et douglas). Nous utilisons pour cela la couche contenue dans le fichier **type_foret.tif**. Les peuplements constitués de douglas et d'épicéa sont désignés respectivement par les codes numérique 4 et 5. Cette couche est produite à une résolution de 2,5 m.
- Dans les lignes de code qui suivent, un masque correspondant aux deux essences est généré. Il est baptisé « **resineux** ». Ce masque est ensuite combiné à la couche **hdom** en rééchantillonnant cette dernière à 2,5 m de résolution.
- La couche finale est sauvegardée dans un fichier .tif.

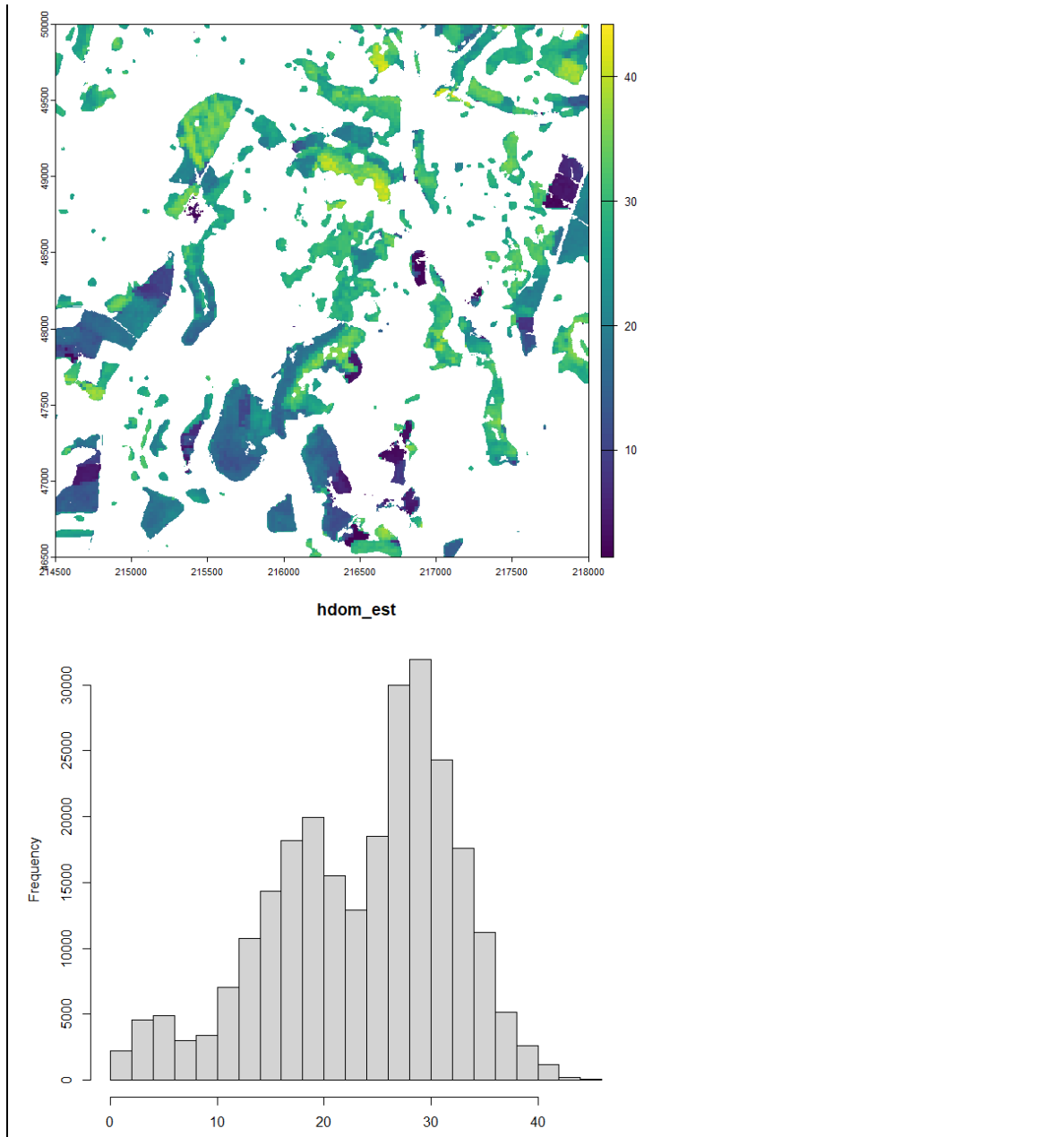
```
# Masquer avec 1 carte des types de peuplements résineux
f_in=paste0(path_in,"/type_foret.tif")
foret=rast(f_in)
resineux=ifel(foret==4|foret==5,1,NA) # (4:douglas, 5:épicéa)

hdom=terra::resample(hdom,resineux,method="near")

hdom=hdom*resineux

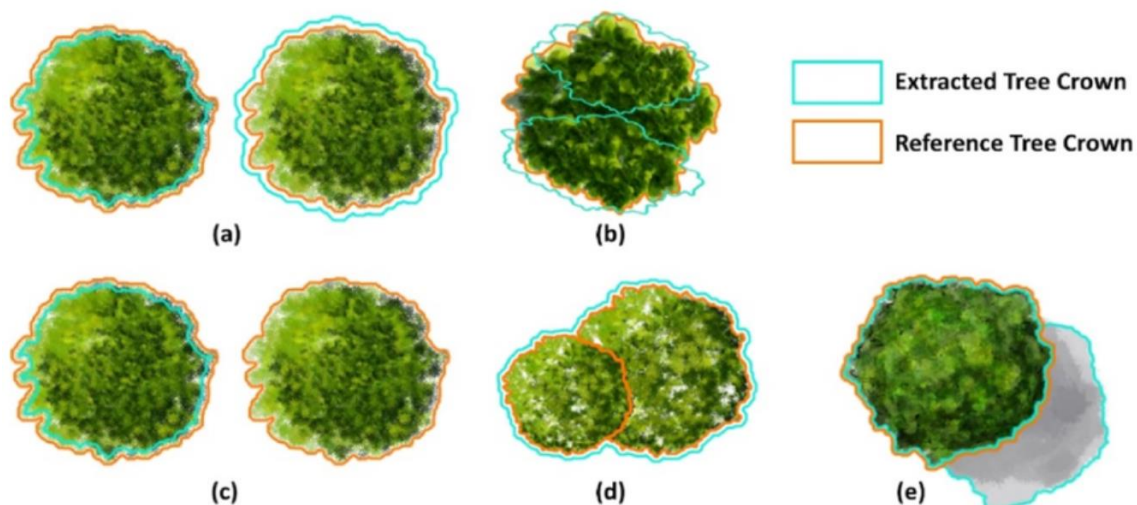
plot(hdom)
hist(hdom)

# Sauvegarder la couche finale
f_out=paste0(path_out,"/hdom_resineux.tif")
writeRaster(hdom,f_out)
```



3.11. Segmentation des cimes d'arbres

- Dans les applications forestières du lidar aérien, les algorithmes dits *de segmentation* ont pour objectif de délimiter les cimes d'arbres. Cette tâche est à la base de l'approche ITC (Individual Tree Crown).
- En apparence séduisante, cette approche conduit rarement à des résultats complètement satisfaisants. En présence de structure complexe, elle peut conduire à de la sous-segmentation (plusieurs arbres englobés dans 1 seul segment) ou de la sur-segmentation (1 arbre découpé en plusieurs segments). La figure suivante illustre ces 2 résultats (<https://doi.org/10.1016/j.jag.2022.103011>).



Possible outcomes of comparing reference tree crowns and extracted tree crowns. (a) Matched. (b). Over Segmentation. (c). Missed. (d) Under Segmentation. (e) Wrong.

- Nous illustrons ici le principe de segmentation sur les arbres d'une placette extraite de l'inventaire utilisé au paragraphe précédent.
- Le shapefile **arbres.gpkg** contient la localisation des arbres au sein de placettes contenues dans **plots.gpkg**.

```
# 3.11. Délimitation des cimes d'arbres -----

# Lire le fichier arbre
f_arbre=paste0(path_in,"/arbres.gpkg")
arbre=st_read(f_arbre)
names(arbre)

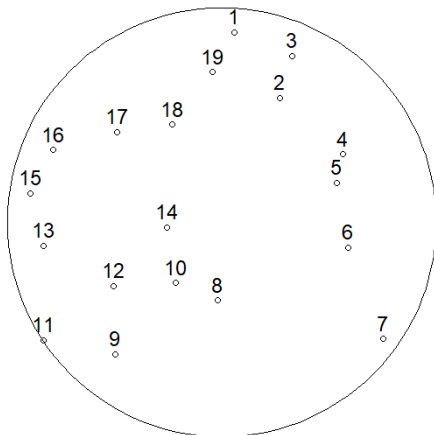
> names(arbre)
[1] "id_plot" "numarbre" "ess"      "c150"    "htot"    "geom"
> |
```

- Chaque arbre est caractérisé par son numéro d'inventaire, son essence, sa circonférence et sa hauteur totale. Cette dernière n'est cependant mesurée que pour quelques arbres dominants au sein de chaque placette.

```
# sélection des arbres de la première placette
plot0=plots[1,]
arbre=arbre[arbre$id_plot==plot0$id,]
head(arbre)

> head(arbre)
Simple feature collection with 6 features and 6 fields
Geometry type: POINT
Dimension: XY
Bounding box: xmin: 218042.4 ymin: 50740.89 xmax: 218059.7 ymax: 50751.99
Projected CRS: BD72 / Belgian Lambert 72
  id_plot numarbre ess c150 htot      geom      x
1  35_15         5  41  171 NA POINT (218056.7 50751.99) 218056.7
2  35_15         6  41  148 NA POINT (218057.4 50747.79) 218057.4
3  35_15         7  41  191 NA POINT (218059.7 50741.88) 218059.7
4  35_15         8  41  183 NA POINT (218049 50744.39) 218049.0
5  35_15         9  41  177 NA POINT (218042.4 50740.89) 218042.4
6  35_15        10  41  131 NA POINT (218046.3 50745.52) 218046.3

# affichage de la placette et des arbres
plot(plot0$geom)
plot(arbre$geom,add=T)
arbre$x=st_coordinates(arbre)[,1]
text(st_coordinates(arbre)[,1], st_coordinates(arbre)[,2],
     labels = arbre$numarbre, pos = 3, cex = 1.5)
```

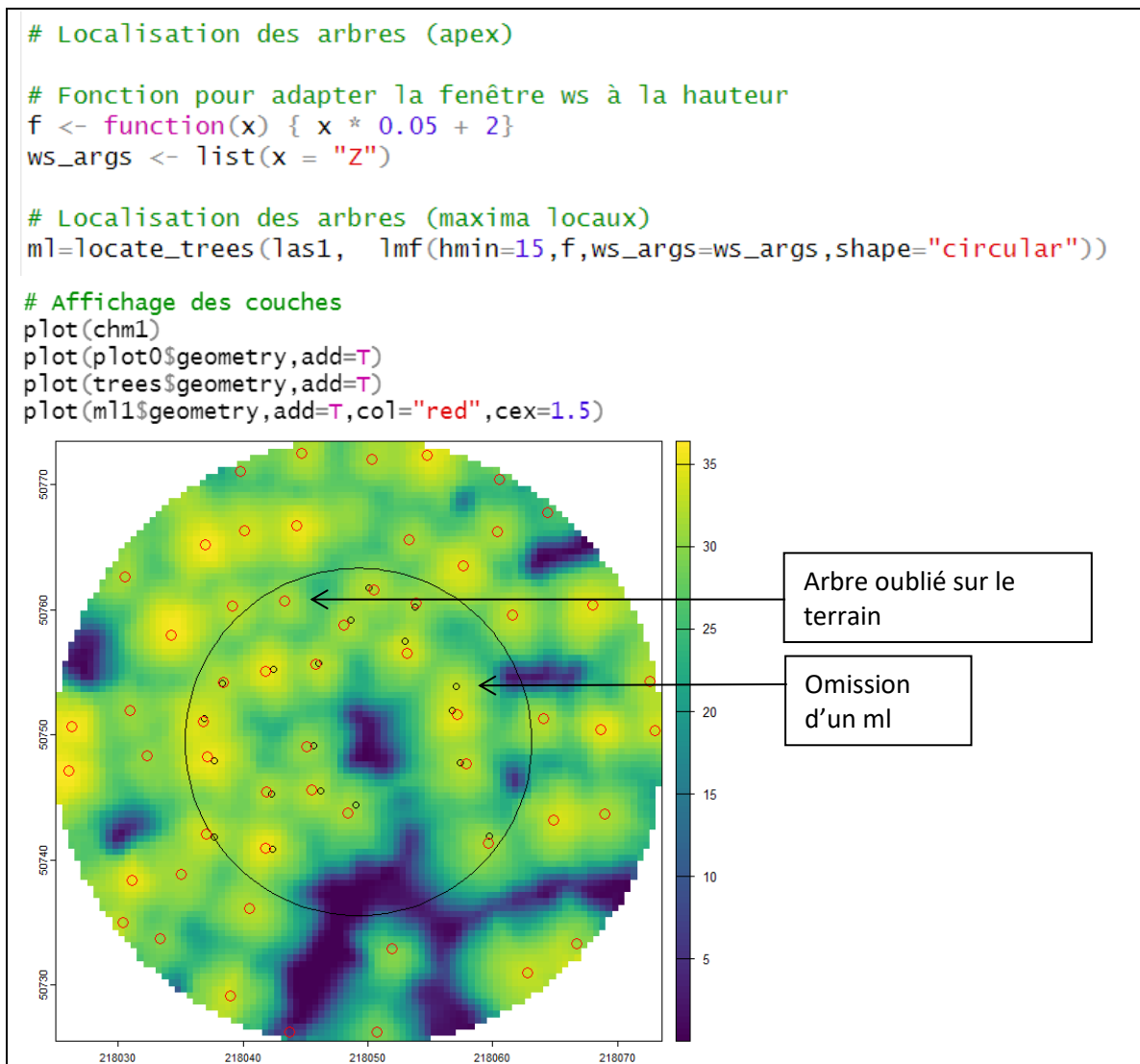


- Les données lidar relatives à la placette sont chargées dans l'objet las1. Elles sont utilisées pour générer 1 chm avec 1 résolution de 50 cm.
- Le chm est construit avec l'algorithme « pitfree ». Un filtre de lissage lui est ensuite appliqué. Cette étape est recommandée en amont du processus de segmentation pour éviter le phénomène de sur-segmentation.

```
# Lecture des données las
f_las=paste0(path_in,"/laz_plots/",plot0$id,".laz")
las1=readLAS(f_las)

# création d'un chm avec l'algorithme pitfree
chm=lidR::rasterize_canopy(las1,res=0.5,
                           pitfree(c(0,2,5,10,15), c(0, 1.5)))
chm = terra::focal(chm, w = 3, fun = mean, na.rm = TRUE)
```

- La segmentation proprement dite est précédée d'une étape de la localisation des apex des arbres. Cette localisation est basée sur la fonction **locate_tress()**. Celle-ci utilise l'algorithme **lmf()** (*local maxima filter*) qui va identifier les maxima locaux.
- L'algorithme **lmf()** comporte deux paramètres : **ws** (*window size*) et **hmin** (*height threshold*). La taille de la fenêtre de recherche des maxima locaux peut être fixe ou variable. Dans l'exemple qui suit, le paramètre **ws** est défini en considérant une fonction qui dépend de la hauteur locale de la canopée (**ws** variable).



- On constate que la détection des arbres a relativement bien fonctionné. Seul un arbre observé sur le terrain a été omis par l'algorithme. Par ailleurs, un arbre qui a été détecté dans le nuage de points a été oublié lors des mesures de terrain.
- La segmentation proprement dite s'effectue avec la fonction **segment_trees()** à laquelle doit être associé un algorithme. Plusieurs algorithmes sont proposés pour effectuer cette segmentation. Dans l'exemple qui suit, nous utilisons l'algorithme **dalponte2016()**. Celui-ci fonctionne au départ du CHM.

- La fonction **`segment_trees()`** va attribuer aux points du nuage de points l'identifiant du maxima local (treeID) le plus proche. Seuls les points répondant à certains critères sont pris en compte. Ces critères sont décrits dans l'aide en ligne fournie par lidR.

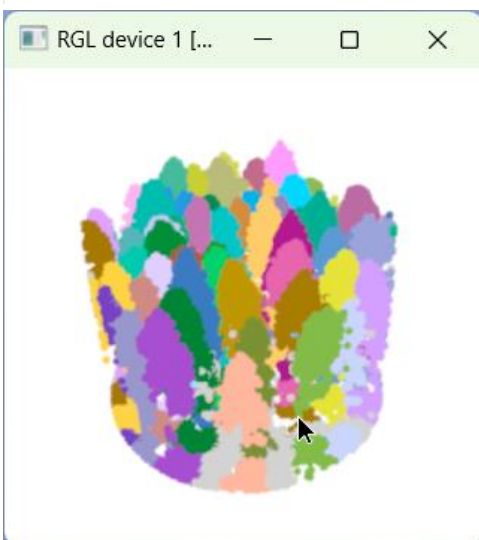
```
dalponte2016(  
  chm,  
  treetops,  
  th_tree = 2,  
  th_seed = 0.45,  
  th_cr = 0.55,  
  max_cr = 10,  
  ID = "treeID"  
)
```

Arguments

chm	'RasterLayer', 'SpatRaster' or 'stars'. Canopy height model. Can be computed with rasterize_canopy or read from an external file.
treetops	'SpatialPoints*' or 'sf/sfc_POINT' with 2D or 3D coordinates. Can be computed with locate_trees or read from an external file
th_tree	numeric. Threshold below which a pixel cannot be a tree. Default is 2.
th_seed	numeric. Growing threshold 1. See reference in Dalponte et al. 2016. A pixel is added to a region if its height is greater than the tree height multiplied by this value. It should be between 0 and 1. Default is 0.45.
th_cr	numeric. Growing threshold 2. See reference in Dalponte et al. 2016. A pixel is added to a region if its height is greater than the current mean height of the region multiplied by this value. It should be between 0 and 1. Default is 0.55.
max_cr	numeric. Maximum value of the crown diameter of a detected tree (in pixels). Default is 10.

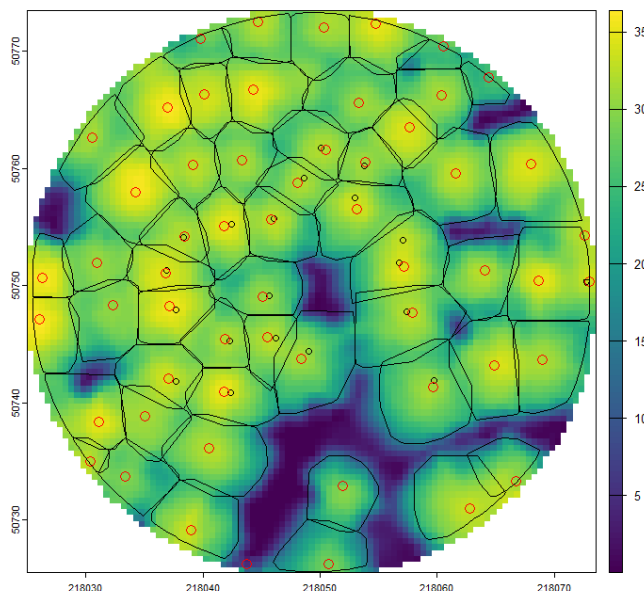
- Les paramètres qui ne sont pas précisés explicitement dans l'appel de la fonction conserve leur valeur par défaut.

```
# Segmentation du nuage de points avec l'algo dalponte2016 ---  
algo = dalponte2016(chm, ml ,th_tree=12)  
las1 = segment_trees(las1, algo)  
  
plot(las1,bg="white", size=4, color="treeID")
```



- L'étape suivante consiste à convertir le nuage de points segmenté en une couche de polygones. Ceux-ci constituent 1 représentation 2D des segments de nuages de points (objet 3D). Cette conversion est prise en charge par la fonction `crown_metrics()`. L'option « geom » précise la nature de la géométrie « output ». Dans l'exemple qui suit, cette représentation est de type « polygone convexe ». Les 3 autres options sont « concave » (polygones concaves), « point » (équivalent aux ml) ou bbox (bounding box du nuage de points).
- La couche segm1 contient la valeur de la hauteur du ml, ainsi que le nombre de points faisant partie du segment et la surface du polygone convexe décrivant le segment.

```
# Conversion du nuage de points segmenté en couche de polygones
segm1=lidR::crown_metrics(las1,.stdtreemetrics,geom="convex")
plot(chm)
plot(arbre$geom,add=T)
plot(ml$geometry,add=T,col="red",cex=1.5)
plot(segml$geometry,add=T)
```



```
head(segml)
```

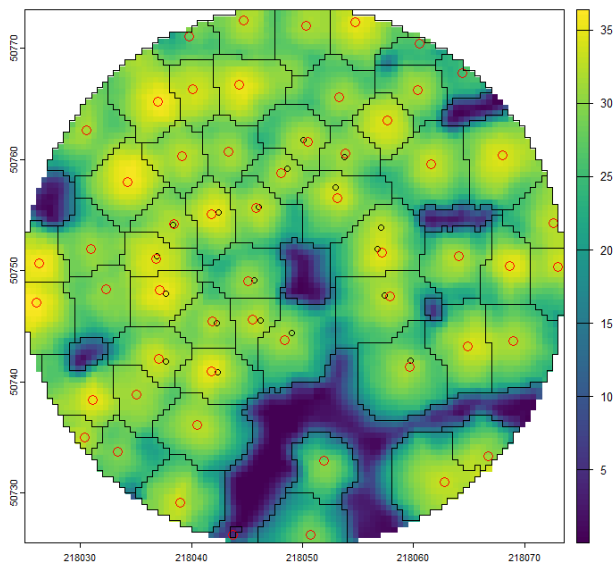
```
> head(segml)
Simple feature collection with 6 features and 4 fields
Geometry type: POLYGON
Dimension: XY
Bounding box: xmin: 218058 ymin: 50728.29 xmax: 218073 ymax: 50755
Projected CRS: BD72 / Belgian Lambert 72
  treeID   Z npoints convhull_area geometry
1      1 33.88   1045    19.313 POLYGON ((218067.8 50734.38...
2      2 34.32   1485    39.925 POLYGON ((218064.3 50730.9,...
3      3 34.22    14    0.156 POLYGON ((218072.8 50750.08...
4      4 32.61   1835   41.656 POLYGON ((218072.1 50742.54...
5      6 34.55   1581   42.379 POLYGON ((218067 50742.9, 2...
6      7 36.01   1816   51.522 POLYGON ((218073 50749.41, ...
```

- Il est également possible de générer 1 couche de polygones décrivant les segments directement au départ du chm.

- Le résultat est assez similaire, mais aussi abouti que dans le premier cas. Il faut en effet établir le lien entre les segments et les maxima locaux pour récupérer la hauteur maximale.

```
# Segmentation du chm avec l'algo dalponte2016
segm2=dalponte2016(chm,m1)()
class(segm2)
segm2=st_as_sf(as.polygons(segm2))

plot(chm)
plot(arbre$geom,add=T)
plot(m1$geometry,add=T,col="red",cex=1.5)
plot(segm2$geometry,add=T)
```



```
head(segm2)
> head(segm2)
Simple feature collection with 6 features and 1 field
Geometry type: POLYGON
Dimension: XY
Bounding box: xmin: 218058 ymin: 50727.5 xmax: 218073.5 ymax: 50758.5
Projected CRS: BD72 / Belgian Lambert 72
  focal_mean geometry
1 1 POLYGON ((218065 50737, 218...
2 2 POLYGON ((218059.5 50735.5,...
3 3 POLYGON ((218070.5 50752, 2...
4 4 POLYGON ((218066.5 50747, 2...
5 5 POLYGON ((218071.5 50758.5,...
6 6 POLYGON ((218064.5 50747.5,...
```

- Une autre solution consiste à utiliser la fonction **extract()** pour extraire certaines informations relatives au chm à l'intérieur des polygones.

```
# hauteur moyenne
df=terra::extract(chm,vect(segm2),fun="mean",na.rm=T)
segm2$h moy=df[,2]
# q95
df=terra::extract(chm,vect(segm2),
                  fun = function(x) quantile(x, probs = 0.95, na.rm = TRUE))
segm2$hq95=df[,2]
# surface
segm2$surf=as.numeric(st_area(segm2))
segm2
```

Annexe 1 – Signification des codes de classification des points LiDAR

Table 3. ASPRS Standard Lidar Point Classes (Point Data Record Formats 6-10)

Classification Value	Meaning
0	Created, never classified
1	Unclassified ⁵
2	Ground
3	Low Vegetation
4	Medium Vegetation
5	High Vegetation
6	Building
7	Low Point (noise)
8	Reserved
9	Water
10	Rail
11	Road Surface
12	Reserved
13	Wire – Guard (Shield)
14	Wire – Conductor (Phase)
15	Transmission Tower
16	Wire-structure Connector (e.g., Insulator)
17	Bridge Deck
18	High Noise
19-63	Reserved
64-255	User definable

Annexe 2 – Liste de metrics standards de LidR (<https://github.com/r-lidar/lidR/wiki/stdmetrics>)

Standard metrics list

- `n` : number of points
- `area` : approximative actual area of a raster (should be close to the square of the resolution but not on the edge)

$$(X_{max} - X_{min}) \times (Y_{max} - Y_{min})$$

- `angle` : average absolute scan angle
- `zmax` : maximum height
- `zmean` : mean height
- `zsd` : standard deviation of height distribution
- `zskew` : skewness of height distribution
- `zkurt` : kurtosis of height distribution
- `zentropy` : entropy of height distribution (see function entropy)
- `pzabovemean` : percentage of returns above `zmean`
- `pzabovex` : percentage of returns above x.
- `zqx` : x^{th} percentile (quantile) of height distribution
- `zpcumx` : cumulative percentage of return in the i^{th} layer according to Wood et al. 2008 (see metrics named $d_1, d_2, \dots$)

$$d_i = \int_{z_{min}}^{z_{max}} f(z) dz$$

with $f(z)$ the probability distribution of elevations. z_{min} is the lower bound and was hard coded to 0 in lidR < 3.1.3 and is now a parameter in from 3.1.3. z_{max} is the elevation of the highest point

- `itot` : sum of intensities for each return
- `imax` : maximum intensity
- `imean` : mean intensity
- `isd` : standard deviation of intensity
- `iskew` : skewness of intensity distribution
- `ikurt` : kurtosis of intensity distribution
- `ipground` : percentage of intensity returned by points classified as "ground"
- `ipcumzqx` : percentage of intensity returned below the k^{th} percentile of height
- `ip1st` : percentage of intensity returned by 1st returns
- `ip2nd` : percentage of intensity returned by 2nd returns
- `ip3rd` : percentage of intensity returned by 3rd returns
- `ipxth` : percentage of intensity returned by x^{th} returns
- `pxth` : percentage x^{th} returns
- `pground` : percentage of returns classified as "ground"

4. Résumé des fonctions lidR

Fonctions	Descriptif	Réf
<i>readLAS()</i> <i>clip_rectangle()</i>	Lecture d'un fichier las Découpe 1 objet las selon un emprise (xmin, ymin, xmax,ymax)	3.2
<i>normalize_height()</i> <i>unnormalize()</i>	Normalise 1 nuage de points ($Z \rightarrow h$) Annule la normalisation ($h \rightarrow Z$) du nuage de points	3.3
<i>grid_terrain()</i>	Création d'un MNT au départ des points « sol »	3.4
<i>rasterize...</i> <i>... algo=***</i>	Création de raster au départ d'un nuage de points préciser l'algorithme utilisé pour cette transformation	
<i>rasterize_terrain</i>	Création d'un MNT (nuage de point non normalisé)	3.4
<i>rasterize_canopy</i>	Création d'un MNC (nuage de point normalisé) ou d'un MNS (nuage de points non normalisé)	3.5
<i>rasterize_density</i>	Création d'un raster décrivant la densité de points	3.7
<i>filter_poi()</i> <i>segment_shapes()</i>	Sélectionne les points du nuage qui répondent à certains critères (<i>poi = point of interest</i>) Segmentation du nuage de points pour détecter des structures planaires	3.5
<i>catalog()</i> <i>las_check()</i> <i>opt_***</i>	Création d'un catalogue de données lidar Vérification d'un catalogue points Options pour gérer les fonctions qui vont utiliser le catalogue	3.9.1 3.9.2 3.9.3
<i>opt_progress()</i> <i>opt_chunk_buffer()</i> <i>opt_output()</i>	Affiche la progression du traitement Applique un buffer sur les tuiles (prise en compte des points voisins) Défini la manière de nommer les fichiers de sortie	
<i>***_metrics()</i> <i>plot_metrics()</i> <i>pixel_metrics()</i> <i>crown_metrics()</i>	Fonctions de calcul des metrics par rapport à différents éléments spatiaux (placettes, pixels, cime,...) Calcul de metrics au niveau de placettes d'inventaire détecter des structures planaires Calcul des metrics au sein de pixels d'une grille raster Calcul de metrics sur des cimes segmentées (par <i>segment_trees()</i>)	3.10 3.10.2 3.10.4 3.10.4
<i>locate_trees()</i> <i>segment_trees()</i> <i>dalponte2016()</i>	Localisation des apex (sommets des arbres considérés comme maxima locaux) Segmentation des cimes d'arbres correspondant aux maxima locaux Algorithme de segmentation du nuage de points utilisé par <i>segment_trees()</i>	3.11 3.10.4 3.10.4