R_GIS 04

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

Novembre 2023

LIÈGE université
Gembloux
Agro-Bio Tech

TABLE DES MATIERES

1. Préambule

- Le présent document a été développé par l'Axe de Gestion des Ressources forestières de Gembloux Agro-Bio Tech – Université de Liège.
- Ce document a été écrit et vérifié par les auteurs. Cependant, il est possible que des erreurs subsistent et les éventuelles remarques et corrections sont toujours les bienvenues.
- La responsabilité de l'ULiège-GxABT et des auteurs ne peut, en aucune manière, être engagée en cas de litige ou dommage lié à l'utilisation de ce document.

Auteur

- Philippe Lejeune (p.lejeune@uliege.be)

Licence de ce document

2. 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).
- Les manipulations sont présentées au travers de l'environnement de travail RStudio. Elles sont réalisées avec la **version 4.0.3 de R** et la **version 4.0.1 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.

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 et la définition des répertoires de travail.

```
# 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)

# clear memory
rm(list=ls())

# chemins vers les répertoires
path0="C:/PL/01_COURS/tthr_2022/R_GIS_04"
path_in = paste0(path0,"/input")
path_out = paste0(path0,"/output")
```

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"))
las1 = lidR::readALSLAS(paste0(path_in,"/laz/13.laz"))


# Quelques infos sur le fichier .laz
las1
```

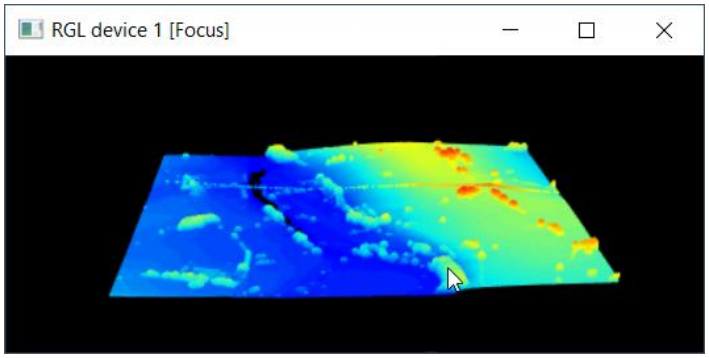
```
> las1
class         : LAS (v1.2 format 1)
memory        : 122.7 Mb
extent        : 214000, 215000, 45000, 45999.99 (xmin, xmax, ymin, ymax)
coord. ref.   : NA
area          : 0.99 kunits²
points        : 1.61 million points
density       : 1.63 points/units²
density       : 1.41 pulses/units²
```

- 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**).
- Avant d'aller plus loin, il est utile de définir explicitement le CRS du fichier .laz.

```
# Définir le CRS de l'objet las
st_crs(las1)
st_crs(las1)=31370
```

- 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)
```

```
> str(las1)
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 ...


  ..@ header    :Formal class 'LASheader' [package "lidR"] with 2 slots
  .. .. ..@ PHB:List of 29
  .. .. .. ..$ File Signature                 : chr "LASF"
  .. .. .. ..$ File Source ID                 : int 0
  .. .. .. ..$ Global Encoding                :List of 6
  .. .. .. .. ..$ GPS Time Type               : logi TRUE
  ..@ 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
  .. .. ..@ projargs: 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 objet ***sp*** (SpatialPointDataframe) contenant un grand nombre de points et possédant des attributs décrivant les propriétés des points LiDAR.

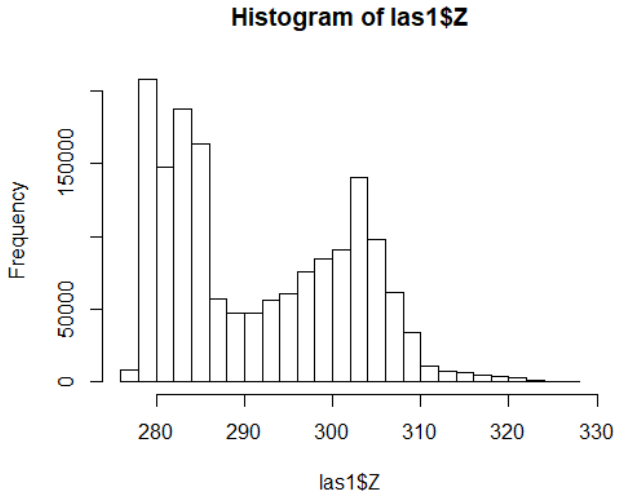
- Le dataframe contient notamment leurs coordonnées X, Y et Z, ainsi qu'un temps GPS associé à l'impulsion laser, ou encore un code de classification (points sol, végétation…) fourni par le producteur. La signification des codes de classification des points LiDAR est décrite en annexe 1.

```
# 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)
```

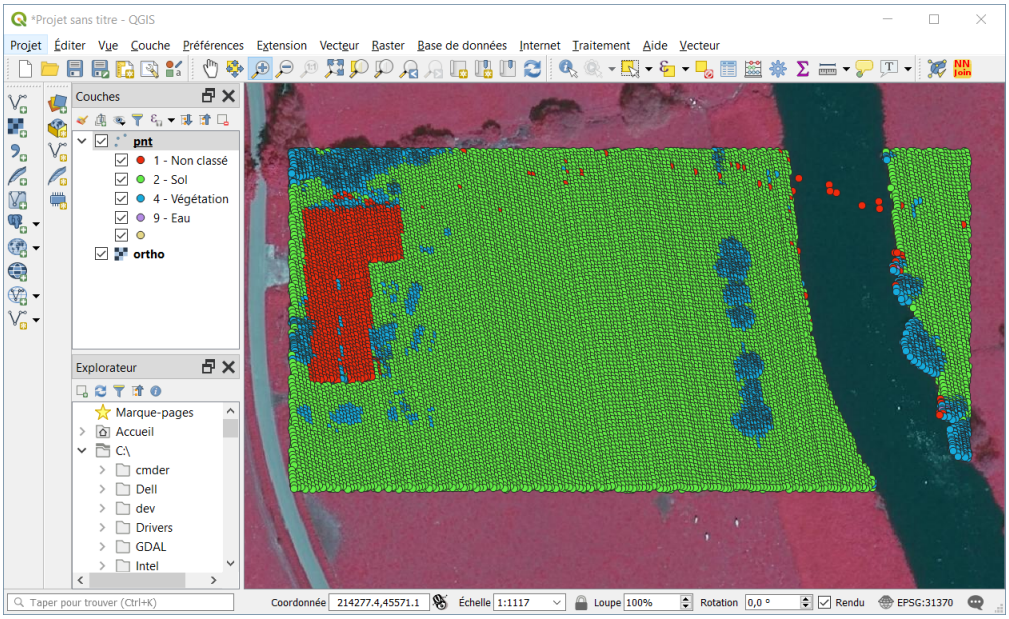


- La fonction ***clip_rectangle()*** est utilisée pour extraire des parties du nuage de points délimitées selon une emprise rectangulaire (x_{min}, y_{min}, x_{max}, y_{max}).
- Le nouvel objet las est ensuite transformé en objet sf avec la fonction ***st_as_sf()***. L'objet sf peut ensuite être exporté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

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

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

f_out=paste0(path_out,"/pnt.shp")
st_write(pnt,f_out,overwrite=TRUE)
```

- Observer le contenu du shapefile **pnt.shp** dans QGIS.
- Afficher la couche **ortho.vrt** en arrière-plan. Elle se trouve dans le répertoire **\ortho**.
- Appliquer au fichier **pnt.shp** une symbologie de type « catégorisé » en considérant le champ [Classification] correspondant à la classification des points.
- **Remarque** : les formats vectoriels classiquement utilisés dans les SIG ne sont pas du tout adaptés pour la manipulation de nuage de points LiDAR, à moins de ne concerner que de très petites surfaces comme c'est le cas ici.
- 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.



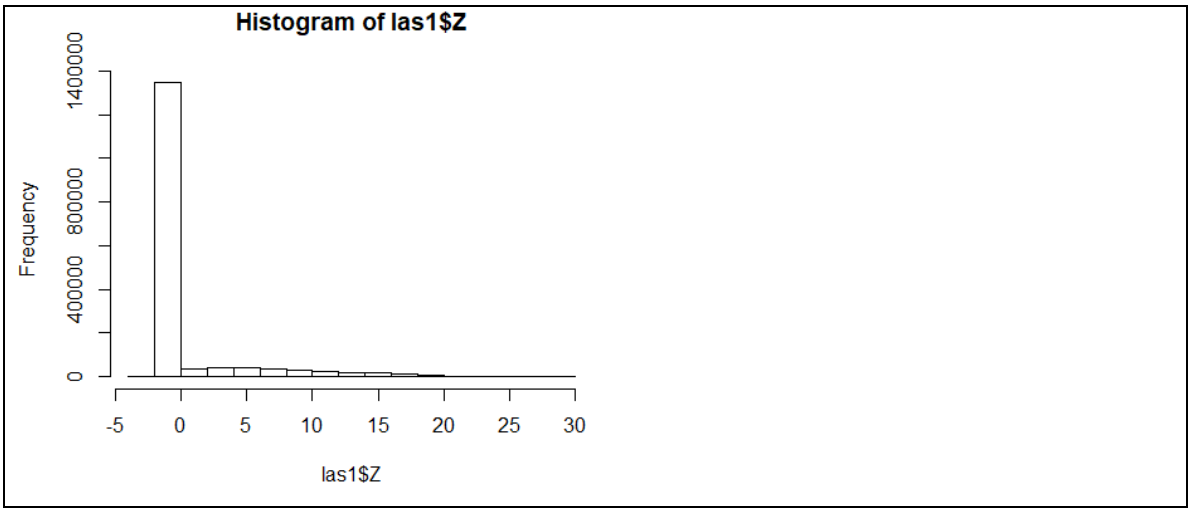
3.3. Lasnormalize : normalisation d'un nuage de points

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



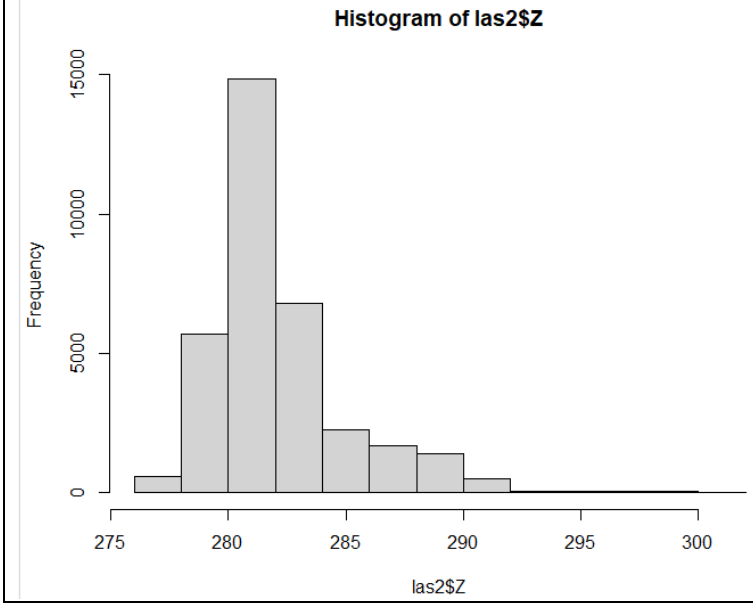
- L'opération de normalisation remplace l'altitude par la hauteur dans le champ las1@data$Z. L'information relative à l'altitude n'est cependant pas complètement supprimée. Elle est recopiée dans un nouveau champ baptisé las1@data$Zref.

```
# L'altitude est conservée dans le champ Zref
names(las1@data)
```

```
>   names(las1@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)
hist(las2$Z)
```

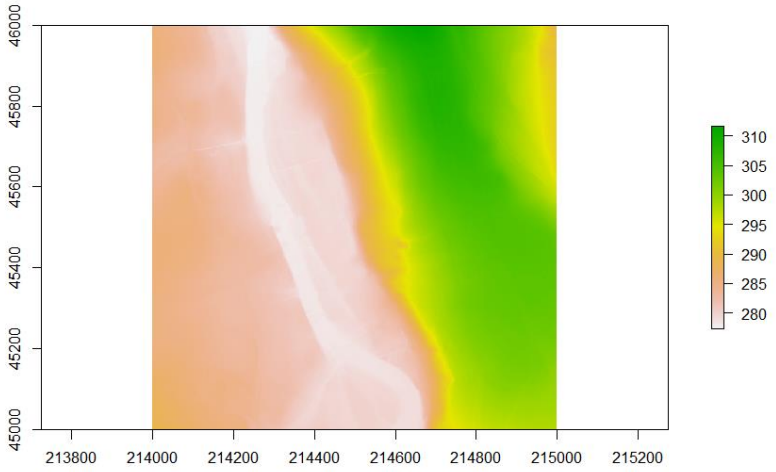


3.4. Grid_terrain : création d'un DEM (Digital Elevation Model)

- Les Modèles Numériques de Terrain (ou Digital Elevation Model) sont sans doute un des principaux produits dérivés des nuages de points LiDAR.
- La fonction ***grid_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. Grid_terrain : construction d'un MNT ------------

# Utiliser des données non normalisées (sinon z=0 pour les points sol)
dem=grid_terrain(las2, 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.

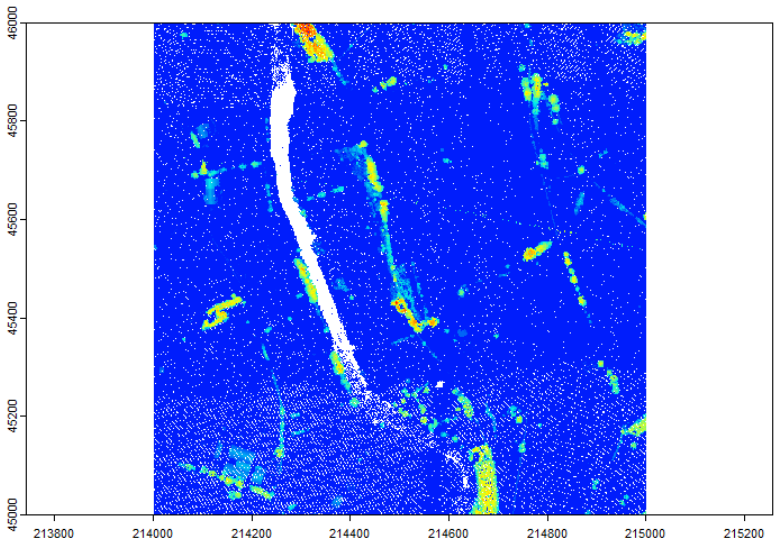


- **Remarque importante** : la construction d'un MNT ne peut s'envisager sur un nuage de points normalisé, puisque dans ce dernier, les points « sols » ont une hauteur nulle.

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

- 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.
- La fonction ***rasterize_canopy()*** prend en charge cette transformation. 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.
- 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,res=1,p2r())
plot(chm1, col = height.colors(50))
```

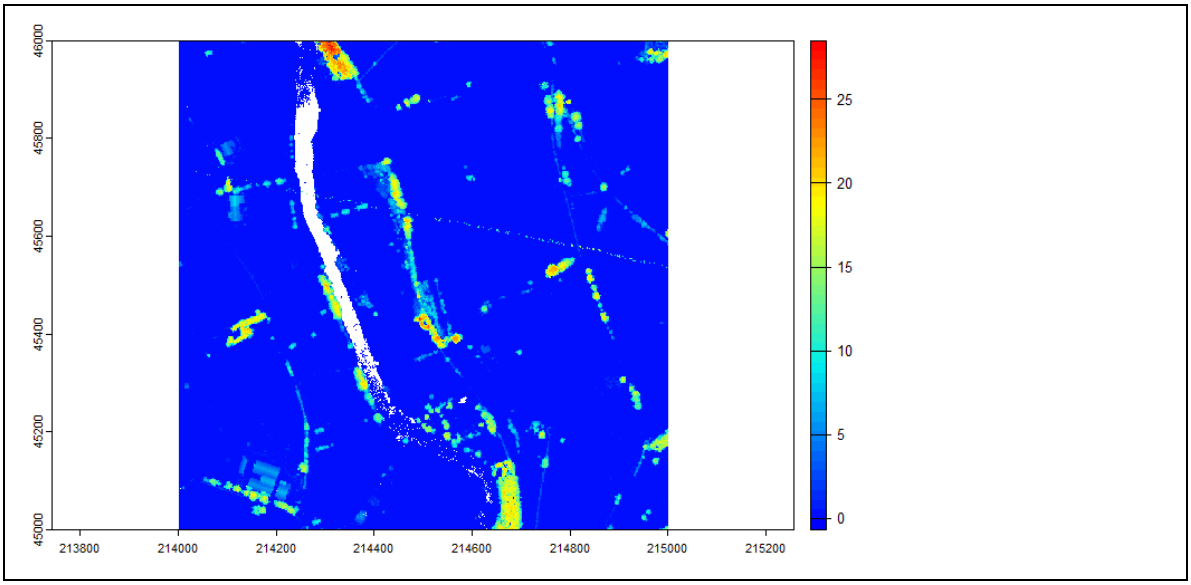


- **Remarque** : tous les produits rasters générés par lidR le sont sous forme d'objets **SpatRaster**.

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

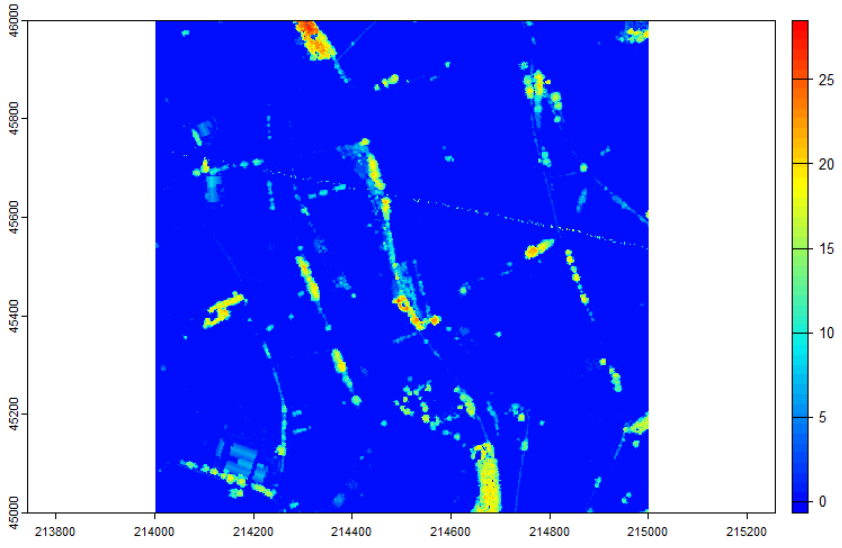
- La principale faiblesse de l'algorithme ***p2r()*** tient au fait qu'en l'absence de point LiDAR dans un pixel, une valeur « NODATA » est attribuée à celui-ci. Pour atténuer ce phénomène, on utilise l'option ***subcircle()***. Celle-ci remplace chaque point LiDAR par un groupe de 8 points de même altitude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

```
# option "subcircle" : remplace chaque point par 8 points répartis dans 1 cercle
chm1 = rasterize_canopy(las1,res=1,p2r(subcircle=0.3))
plot(chm1, col = height.colors(50))
```



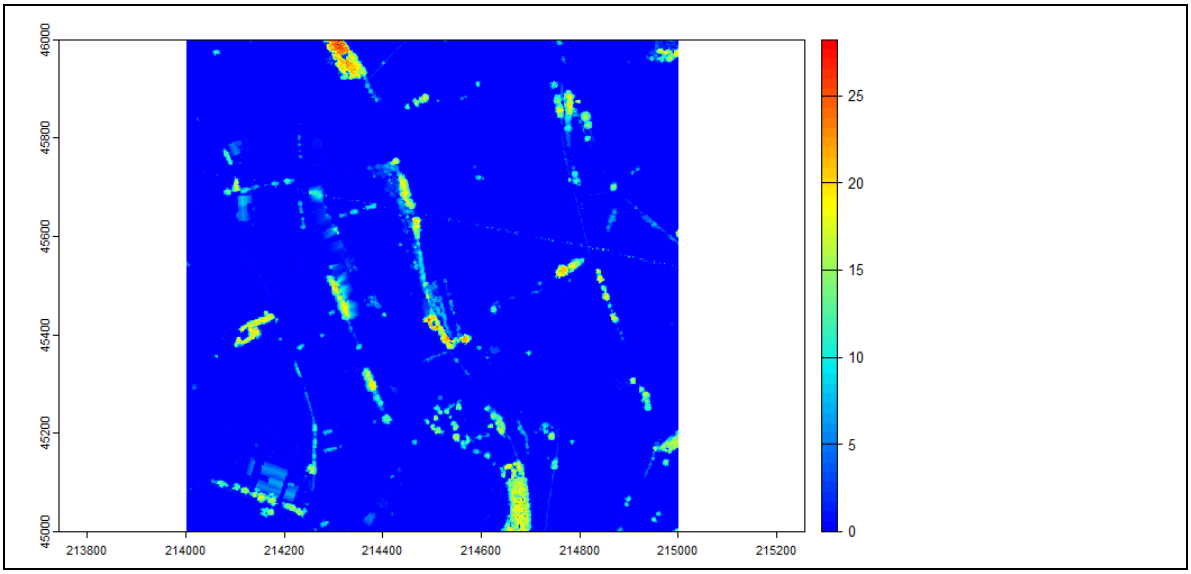
- Malgré l'utilisation de cette option, il subsiste de nombreux pixels « NODATA » dans le CHM produit. Ceux-ci se trouvent sur le cours d'eau qui traverse la tuile. S'agissant d'un CHM, il est conseillé de leur attribuer une valeur nulle (figure ci-dessous).

```
# Transformer les nodata -> 0
chm1=ifel(is.na(chm1),0,chm1)
plot(chm1, col = height.colors(50))
```



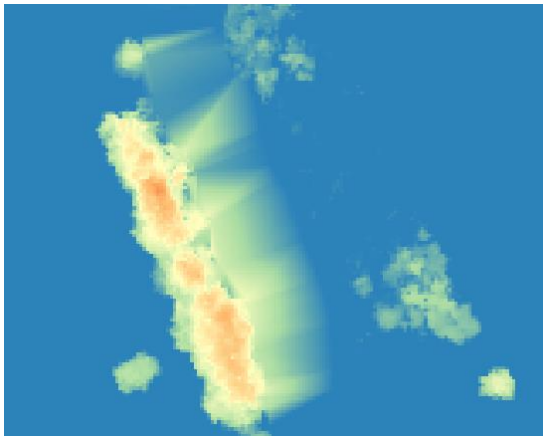
- L'algorithme ***dsmtin()*** génère un réseau TIN (*Triangulated Irregular Network*) sur le nuage de points « premiers retours » avant d'interpoler les valeurs de hauteur pour chaque pixel.

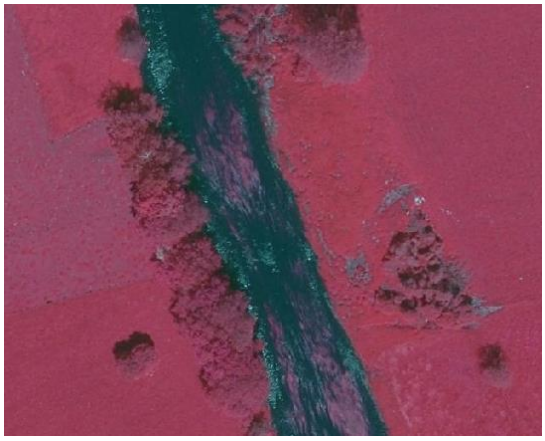
```
# Algorithm dsmtin : triangulation TIN sur les premiers retours
chm2 = rasterize_canopy(las1,res=1,dsmtin())
plot(chm2, col = height.colors(50))
```



- Le résultat ne présente plus de pixels vides. Par contre, 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.

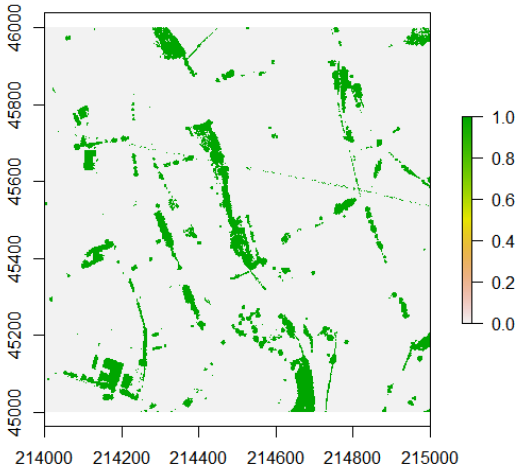
```
# exporter chm2 en fichier .tif pour l'afficher dans QGIS
setwd(path_output)
writeRaster(chm2,"chm2.tif",overwrite=TRUE)
```





- lidR dispose d'un troisième algorithme pour générer un CHM. Il s'agit de ***pitfree()***, qui est une version améliorée de ***dsmtin()***.
- **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)
```

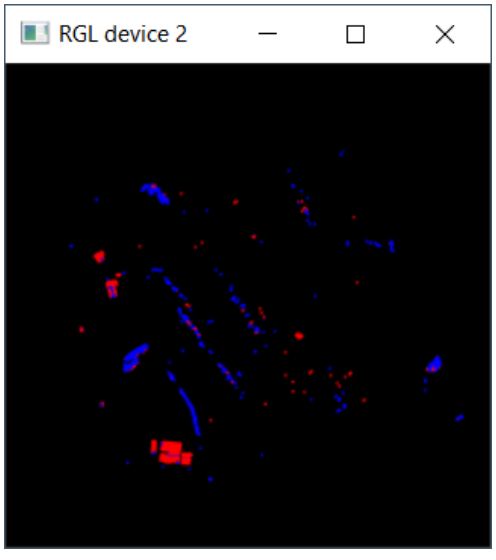


- **Remarque** : en présence de cultures (éléments non ligneux) susceptibles de dépasser la valeur seuil, il est nécessaire de relever celle-ci.

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,NumberOfReturns==1,Z>3,Z<30)
plot(las2)
```



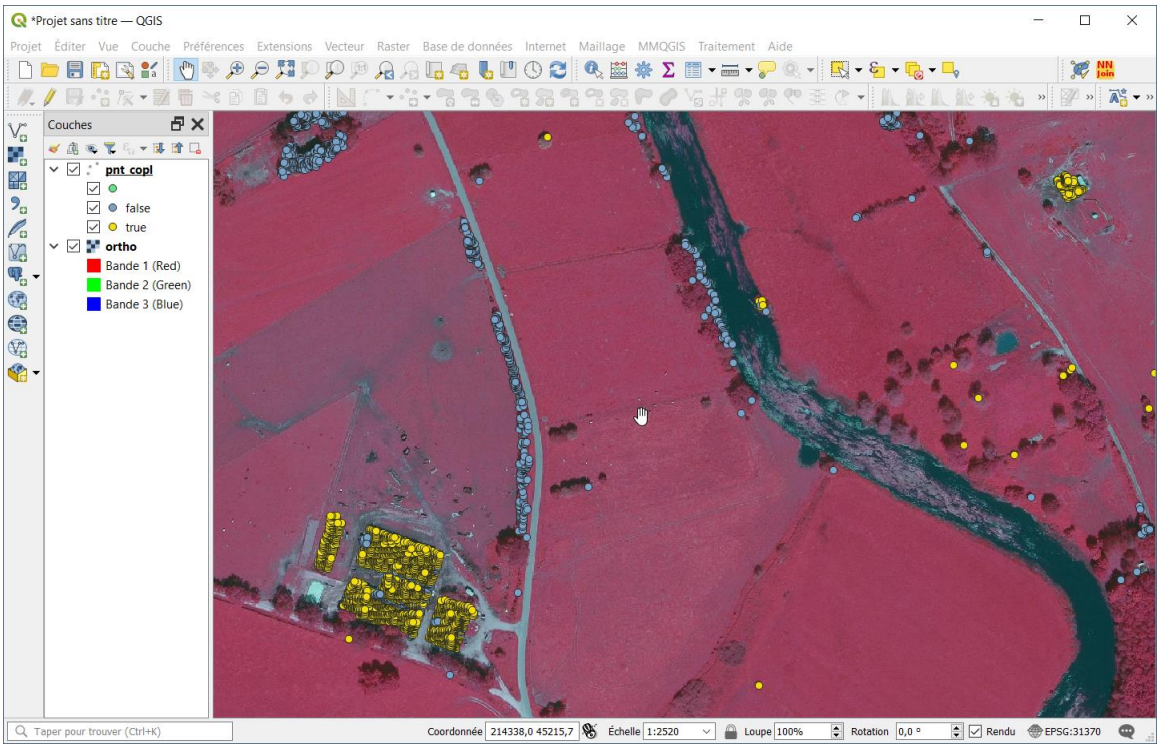
- La fonction ***segment_shapes()***, utilisée avec l'algorithme ***shp_plane()***, teste la présence d'une organisation planaire des points dans le voisinage d'un point donné. Le paramètre k = 30 désigne le nombre de points considéré pour l'ajustement d'un plan. Si la structure planaire est détectée,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

FALSE  TRUE
 7856 26111
```

- 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_copl.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.



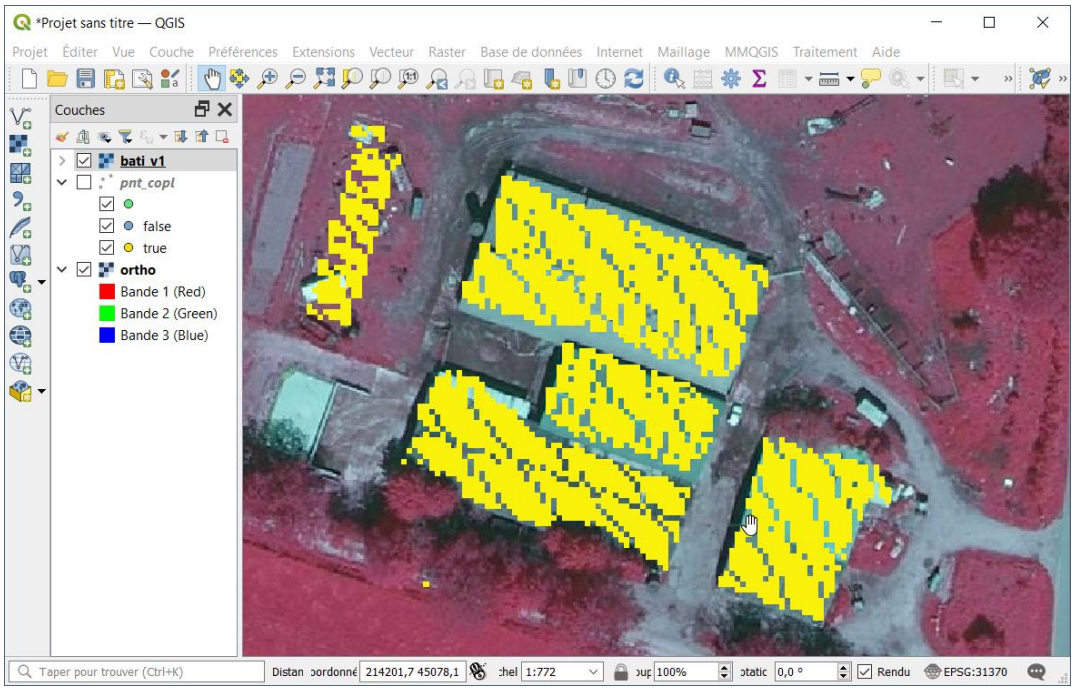
3.7. Création d'un masque « bâtiments »

- 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 :

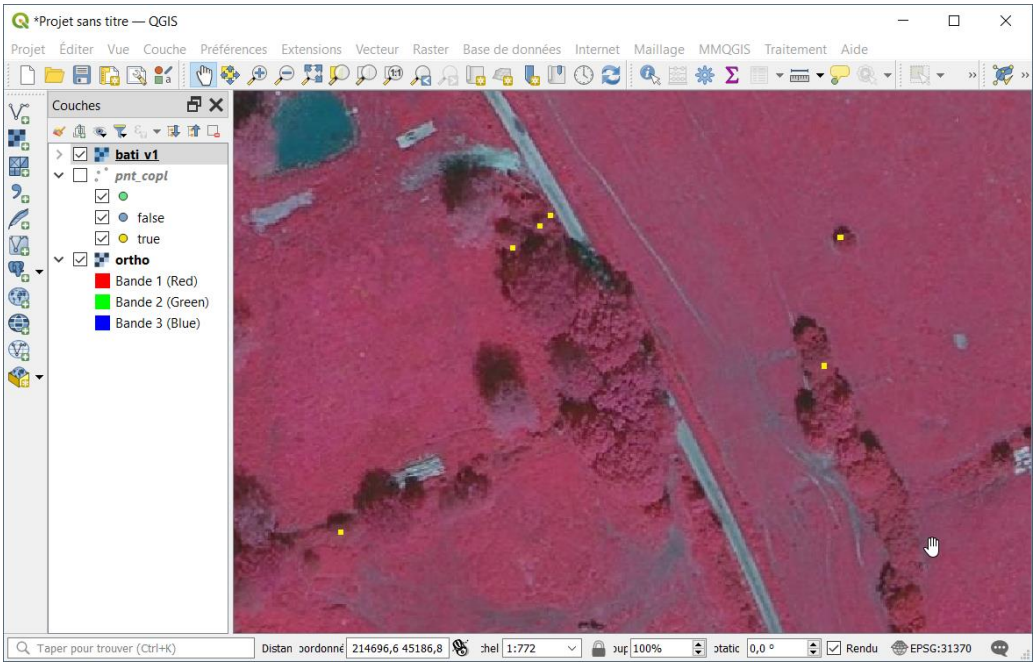
- Générer un raster qui calcule la densité de points ;
- Sélectionner les pixels dont la densité est > 0 (raster binaire) ;
- Appliquer un filtre « majorité » pour supprimer une partie des faux positifs.

```
# 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'1 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.



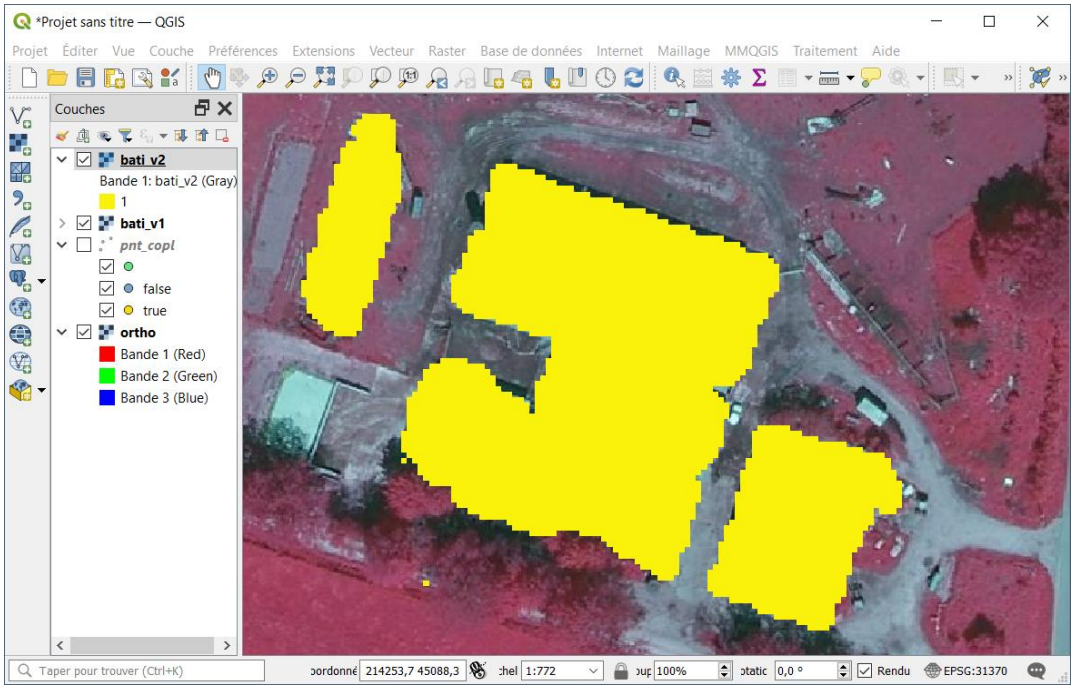
- Pour corriger ces problèmes, on va utiliser successivement une fonction de tamisage (élimination des petits groupes de pixels) et un filtre morphologique (amélioration de la morphologie des groupes de pixels).
- La fonction de tamisage est mise en œuvre via l'extension qgisprocess et l'algorithme **«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

```
# Création d'1 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 est de type « dilatation ». La fonction ***dilate()*** va ajouter deux couches de pixels sur chaque groupe de pixels (kernel de 5 x 5). Cette fonction est issue de la librairie **mmand**.

```
# Filtre morphologique

bati_v2=rast(f_bati_v2)
m=as.matrix(bati_v2,wide=T)
k <- shapeKernel(c(5,5), type="disc")
k
m=dilate(m,k)
bati_v2 <- setValues(bati_v2, m)
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)
```

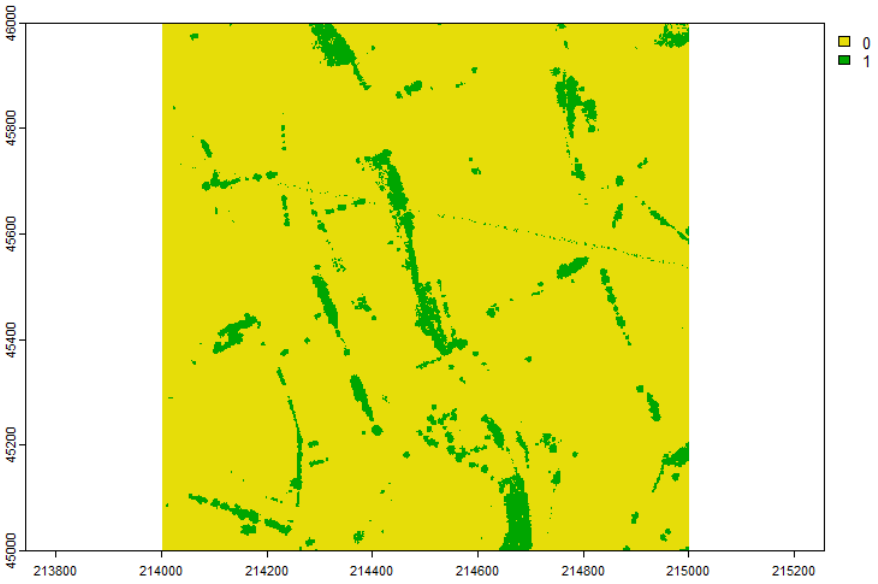
```
> st_bbox(bati)
  xmin   ymin   xmax   ymax
214000  45036 214972  46000
> st_bbox(chm)
  xmin   ymin   xmax   ymax
214000  45000 215000  46000
```

- 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)
```

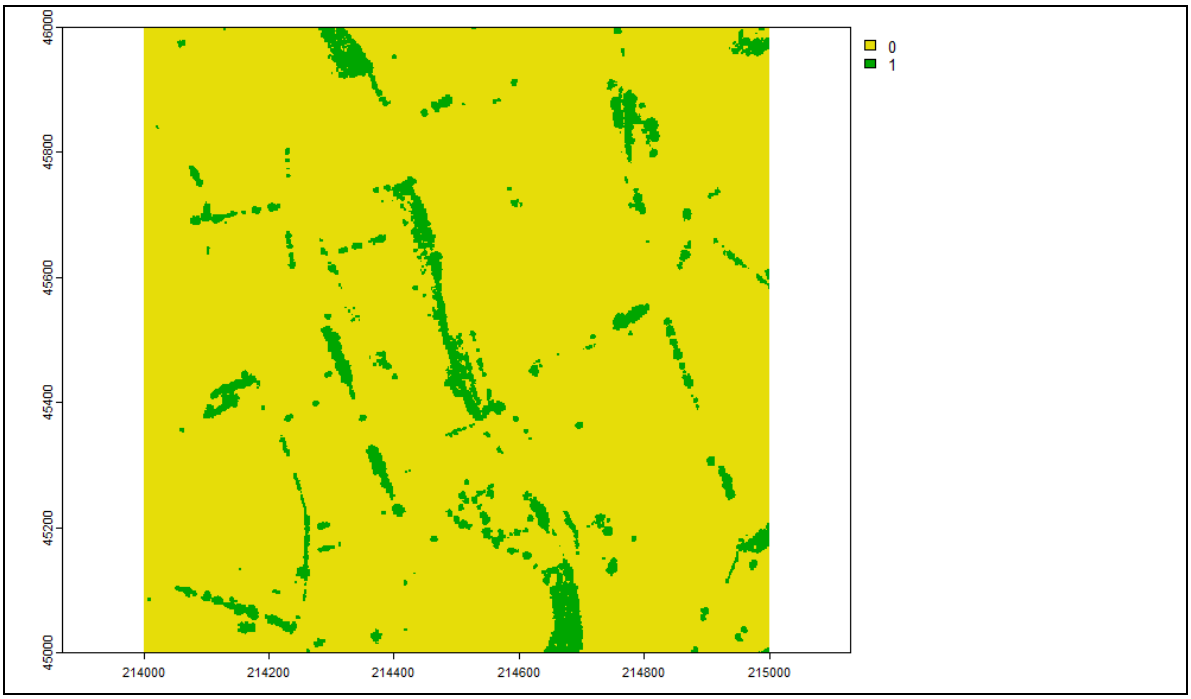
- Les deux couches peuvent maintenant être combinées pour faire ressortir les éléments arborés présents dans la zone d'étude. Ceux-ci seront définis comme suit : éléments dont la hauteur est supérieure ou égale à 3 m et qui ne sont pas des bâtiments.

```
# 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 « ouverture » (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,wide=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

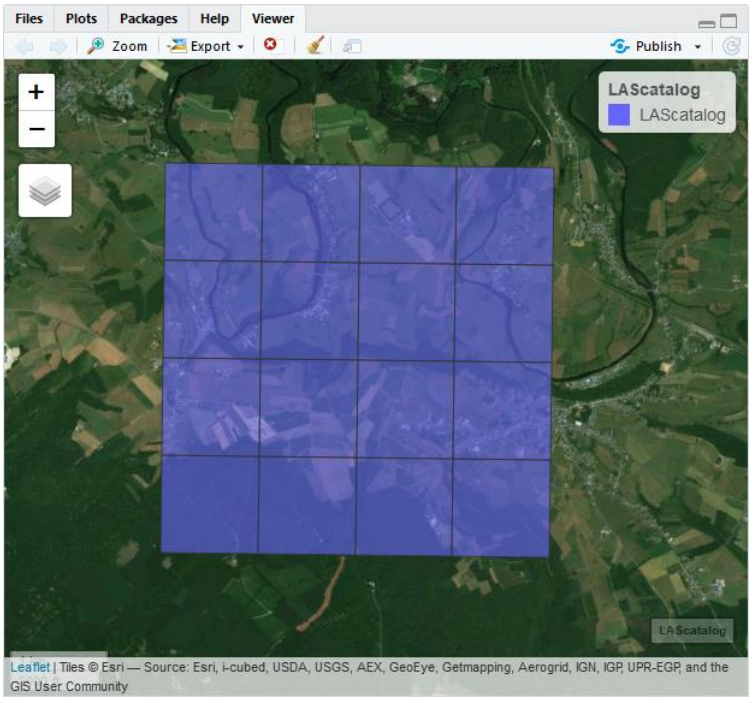
- Les catalogues sont des outils particulièrement efficaces pour gérer et traiter simultanément un ensemble de fichiers .las, même si celui-ci contient plusieurs milliers de fichiers.
- L'intérêt principal des catalogues est de réduire le temps d'accès aux données, qui constitue la partie la plus importante 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)
st_crs(cat1)=31370

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 représentant
# les tuiles d'un catalogue
tuiles=st_as_sf(cat1)
names(tuiles)
plot(tuiles$geometry)
f_out=paste0(path_out,"/tuiles.shp")
st_write(tiles,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)
```

```
> 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.

```
# 3.9.2 indexation d'un catalogue ----------------
# (optimise l'accès aux données du catalogue)
lidR:::catalog_laxindex(cat1)
```

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

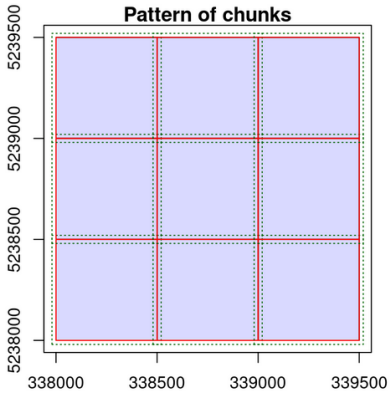
- Avant de créer le CHM, il convient de normaliser les nuages de points pour transformer les altitudes en hauteurs.
- Préalablement, on a créé un répertoire qui va recevoir les nouveaux fichiers .laz.

```
# 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
f_dtm=paste0(path_in,"/mnt_florenv.tif")
dtm=rast(f_dtm)
t1=Sys.time()
cat2=lidR::normalize_height(cat1, algorithm=tin(),dtm=dtm)
t2=Sys.time()
(t2-t1)
> (t2-t1)
Time difference of 3.027933 mins
```

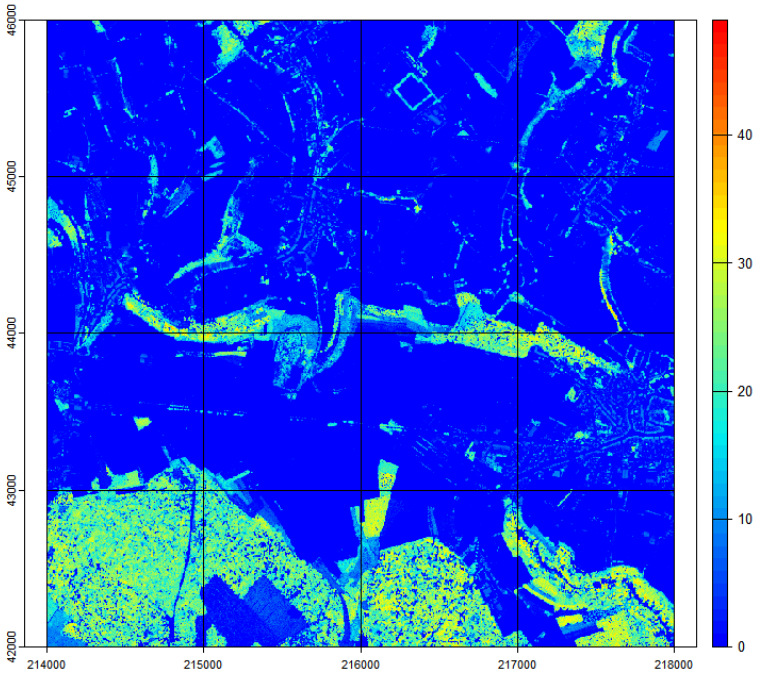
- **Remarque** : lors de l'exécution d'une fonction sur un catalogue, la fenêtre « Viewer » affiche la progression du traitement sur le catalogue. Les « chunks » sont progressivement coloriés selon leur statut (vert : traitement terminé, bleu : traitement en cours…).

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_florenv")
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(0,2,5,10,15), 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.

3.9.4. Application d'une fonction lidR à un catalogue : détection des arbres

- Dans l'exemple qui suit, le moteur de traitement des catalogues .las est utilisé via la fonction ***catalog_map()***. L'intérêt de cette dernière est de pouvoir appliquer une fonction construite sur mesure et de préciser certaines options pour le déroulement du traitement.
- La fonction ***detection_arbre()*** qui est utilisée ici permet de générer une couche de points correspondant aux arbres présents dans la scène couverte par le catalogue. Elle comporte une seule instruction qui applique la fonction ***locate_trees()*** à chaque tuile du catalogue.
- La localisation des arbres avec ***locate_trees()*** utilise une fonction **lmf** (*local maxima filter*). Celle-ci comporte deux paramètres : ws (*window size*) et hmin (*height threshold*).
- Au préalable, certaines options du catalogue ont été précisées : la prise en compte d'un buffer (« opt_chunk_buffer(cat2)=10 »), le nom du fichier de sortie reçoit la valeur « » pour éviter de générer une série de fichiers correspondant aux tuiles. Les options « opt_select(cat2) = « xyz » » et « opt_filter(cat2) = « -keep_first » » ont été ajoutées : elles limitent la lecture des données « xyz » pour les premiers retours. Cela permet de diminuer le volume de données lues et donc de diminuer le temps de traitement. Cette limitation est compatible avec les données nécessaires au bon fonctionnement de la fonction ***locate_trees()***.

- L'utilisation de la fonction ***catalog_map()*** permet de spécifier d'autres options relatives au fonctionnement du moteur de traitement des catalogues en utilisant l'option « .opt ». Dans l'exemple qui suit, deux options sont spécifiées : « need_buffer=TRUE » et « automerge=TRUE ». La première s'assure qu'un buffer a bien été défini dans les options de fonctionnement du catalogue. La seconde fusionne les résultats produits issus des différentes tuiles en un objet **sf** unique.

```
# 3.9.4 Détection des arbres sur l'ensemble du catalogue --------------
# On détecte seulement les arbres de plus de 15 m

cat2=catalog(path_laz_norm) # réinitialiser le catalogue
opt_chunk_buffer(cat2)=10
opt_output_files(cat2) = ""
opt_select(cat2) <- "xyz" # Read only the coordinates.
opt_filter(cat2) <- "-keep_first" # Read only first returns.

detection_arbre = function(las, ws, hmin)
{
  ttops <- locate_trees(las, lmf(ws,hmin))
  return(ttops)
}

opt    <- list(need_buffer = TRUE, automerge=TRUE)
t1=Sys.time()
output <- catalog_map(cat2, detection_arbre,  ws=6,hmin = 15, .options = opt)
t2=Sys.time()
(t2-t1)

nrow(output)

# Sauvegarde du résultat
f_out=paste0(path_out,'/arbres_sup15m.gpkg')
st_write(output,dsn=f_out,layer="arbres")
```

```
> (t2-t1)
Time difference of 2.880464 mins
```

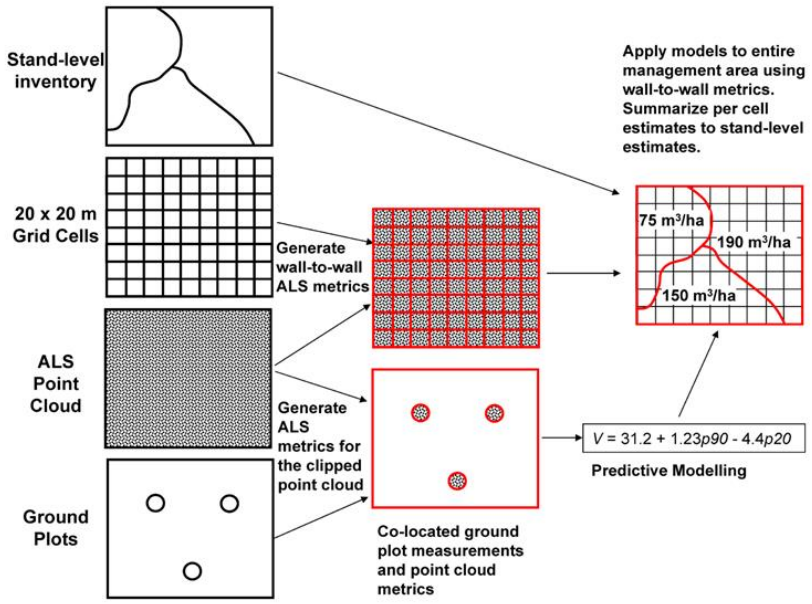
```
nrow(output)
```

```
> nrow(output)
[1] 48888
```

```
# Sauvegarde du résultat
f_out=paste0(path_out,'/arbres_sup15m.gpkg')
st_write(output,dsn=f_out,layer="arbres")
```

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.

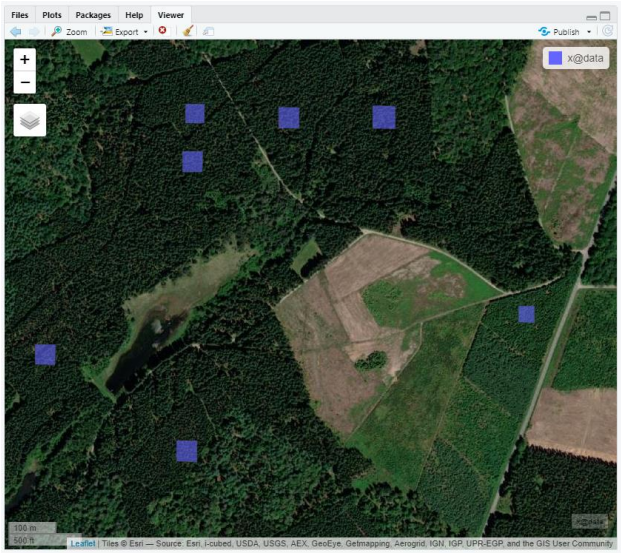
3.10.1. Lire les données d'entrée

```
# 3.10.1. Lire les données d'entrée ---------
f_pl=paste0(path_in,"/plots.shp")
plots=st_read(f_pl,stringsAsFactors = F)
head(plots)
```

```
> head(plots)
Simple feature collection with 6 features and 8 fields
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: 214744.6 ymin: 48038.9 xmax: 218063.1 ymax: 50763.33
Projected CRS: Belge 1972 / Belgian Lambert 72
     id rayon slope surf   hdom  gha   vha  nha                          geometry
1 35_15  13.9     7  610 36.100 58.8 885.0  311 POLYGON ((218063.1 50749.44...
2 33_15  12.0     3  454 37.625 61.0 956.2  330 POLYGON ((217862.7 50745.54...
3 31_15   9.9     4  310 30.625 52.0 693.1  516 POLYGON ((217664.4 50751.94...
```

- Une série de fichiers .laz a été générée au départ d'un catalogue complet couvrant la zone d'intérêt. 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 assemblés sous la forme d'un catalogue.

```
path_laz_plots=paste0(path_in,"/laz_plots")
cat1=catalog(path_laz_plots)

plot(cat1, mapview = TRUE, map.type = "Esri.WorldImagery")
```



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 métrics 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"    "pzabovezmean"
[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 noms des 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*).

```
m=as.matrix(select(df,hdom,zq70:zq95))
cor(m)
> cor(m)
          hdom      zq70      zq75      zq80      zq85      zq90      zq95
hdom 1.0000000 0.9937928 0.9940322 0.9943690 0.9948285 0.9950259 0.9947378
zq70 0.9937928 1.0000000 0.9999553 0.9998398 0.9996586 0.9992737 0.9983888
zq75 0.9940322 0.9999553 1.0000000 0.9999485 0.9998195 0.9995236 0.9987840
zq80 0.9943690 0.9998398 0.9999485 1.0000000 0.9999512 0.9997437 0.9991410
zq85 0.9948285 0.9996586 0.9998195 0.9999512 1.0000000 0.9999037 0.9994303
zq90 0.9950259 0.9992737 0.9995236 0.9997437 0.9999037 1.0000000 0.9997624
zq95 0.9947378 0.9983888 0.9987840 0.9991410 0.9994303 0.9997624 1.0000000
```

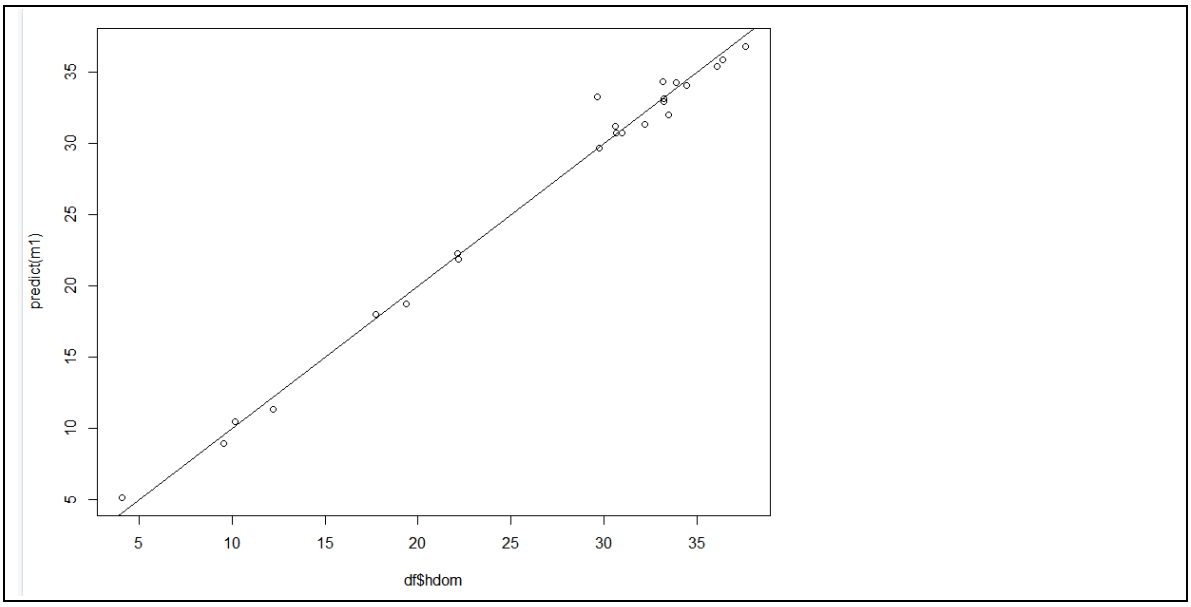
- Le modèle qui est proposé utilise le 95ème percentile des hauteurs des points LiDAR situés au sein des placettes d'inventaire.

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)
plot(df$hdom, predict(m1))
abline(0,1)
```

```
> summary(m1)

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

Residuals:
    Min      1Q  Median      3Q     Max
-3.6541 -0.3050  0.2303  0.5892  1.4835

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.48216    0.60550   2.448   0.0232 *
zq95         1.05110    0.02362  44.493   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.037 on 21 degrees of freedom
Multiple R-squared:  0.9895,    Adjusted R-squared:  0.989
F-statistic:  1980 on 1 and 21 DF,  p-value: < 2.2e-16
```

- On constate que la pente du modèle linéaire est très proche de 1 (1,05 avec une erreur standard de 0,024), ce qui traduit la « quasi proportionnalité » entre les deux variables. L'ordonnée à l'origine est significativement différente de 0 (1,48 m). Cela signifie que le signal LiDAR va, en moyenne, sous-estimer la hauteur des arbres dominants de la placette de 1,5 m.



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.

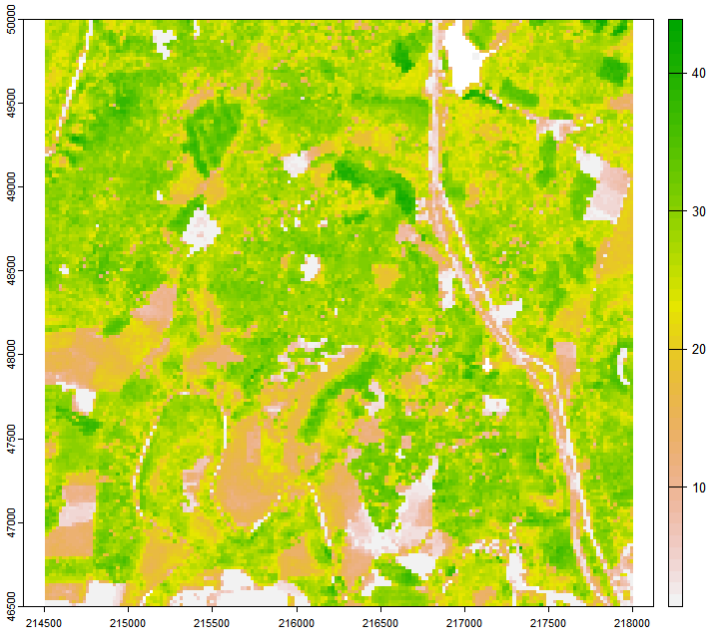
```
# 3.10.4 Appliquer le modèle de prédiction en plein
f_test=paste0(path0,"/input/laz_test/test.laz")
las_test=readLAS(f_test)
st_crs(las_test)=31370

# 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 95ème 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"
```

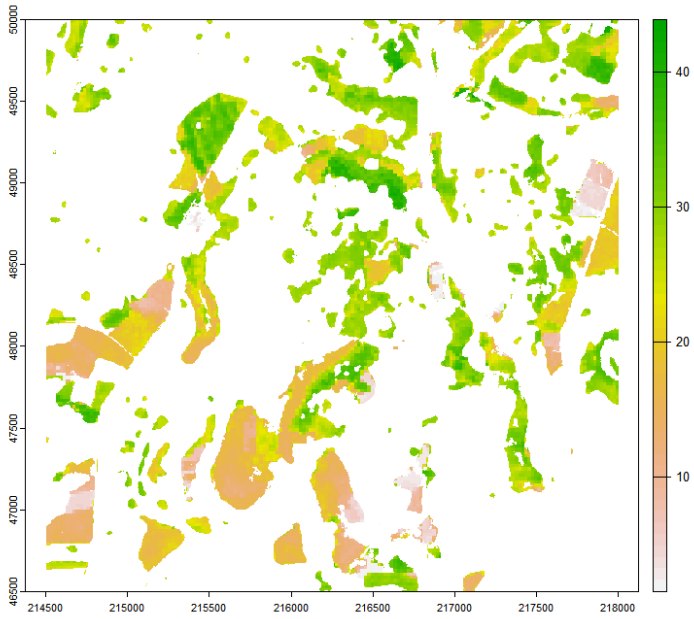
- La prédiction de la hauteur dominante s'effectue très simplement avec la fonction ***terra::predict()*** qui associe un objet possédant une méthode « predict » (ici le modèle m1) avec un objet SpatRaster.

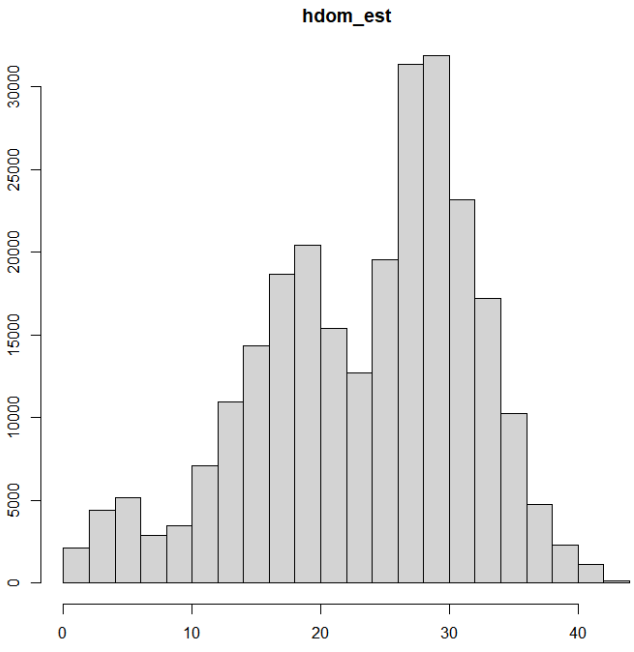
```
# Prédiction de Hdom en plein
hdom <- terra::predict(w2w, m1)
names(hdom)="hdom_est"
plot(hdom)
```



- 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.

```r
# 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)
```

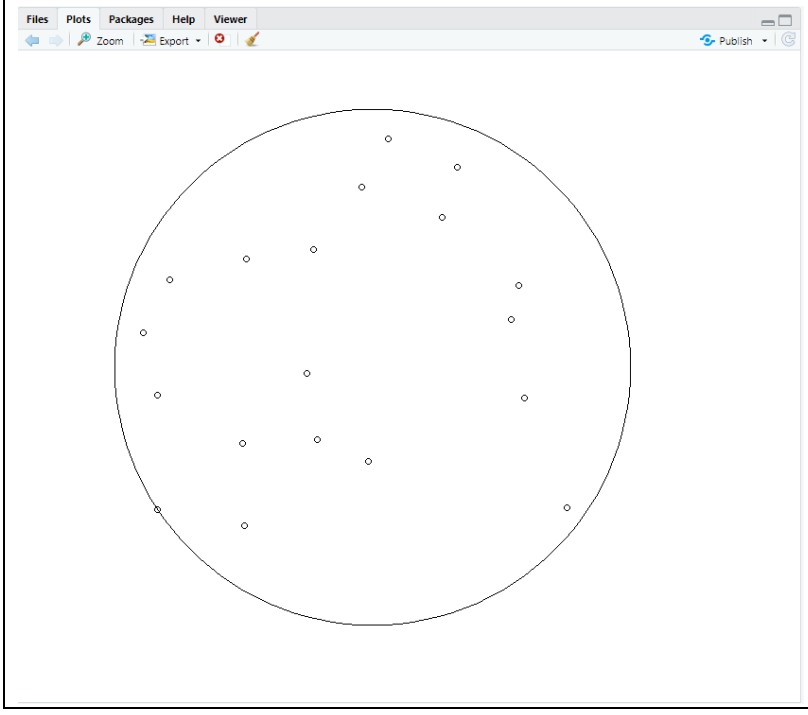




3.11. Segmentation des cimes d'arbre

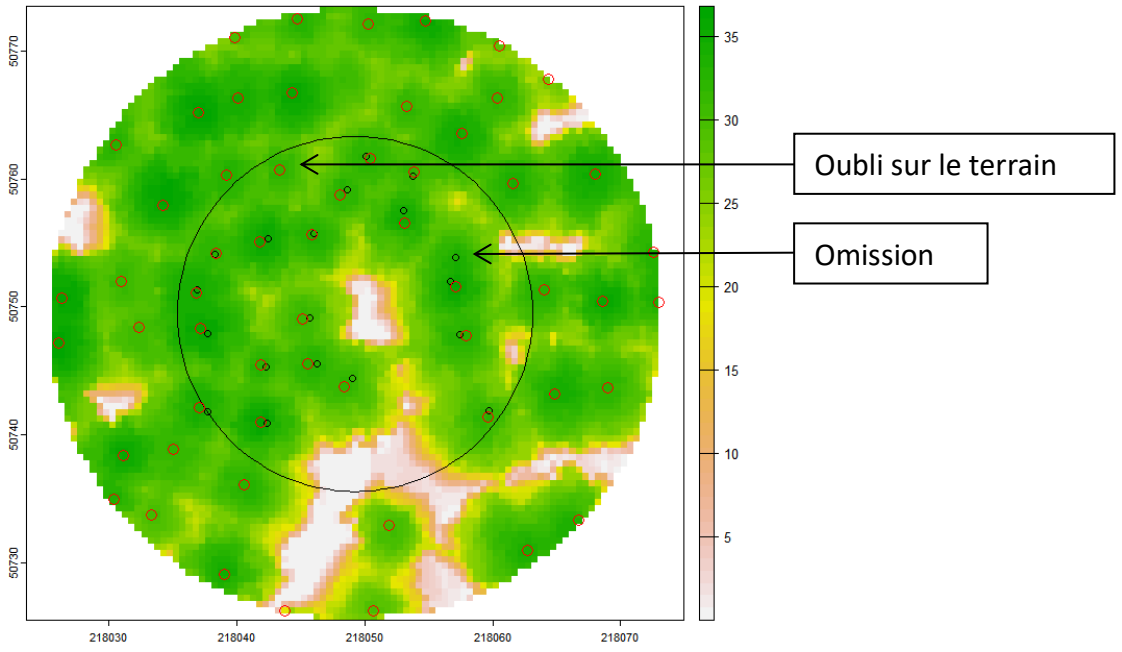
- Nous avons abordé au §3.9.4 la détection d'arbres dont le résultat prend la forme d'une couche de points. Il existe également des algorithmes dits *de segmentation*, qui tentent de délimiter les cimes d'arbres.
- Nous illustrons ceux-ci sur les arbres d'une placette extraite de l'inventaire utilisé au paragraphe précédent.
- Le shapefile **trees.shp** contient la localisation des arbres au sein de placettes contenues dans **plots.shp**.

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

i=1
plot0=plots[i,]

# Lire le fichier arbre
f_tr=paste0(path_in,"/trees.shp")
trees=st_read(f_tr,stringsAsFactors = F)
trees=trees[trees$id_plot==plot0$id,]

plot(plot0$geometry)
plot(trees$geometry,add=T)
```



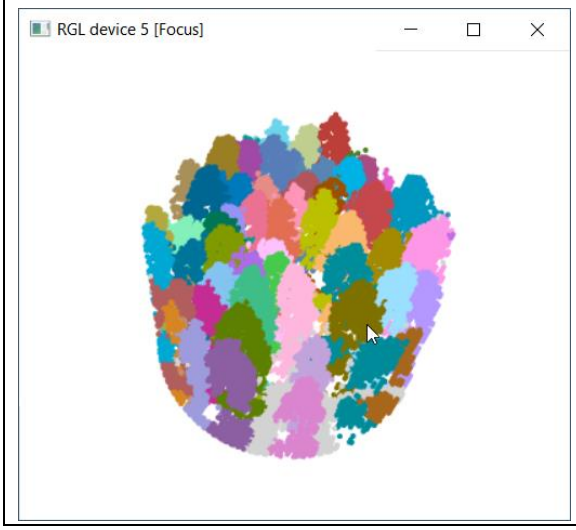
- La localisation des arbres et la segmentation des cimes utilisent une fonction **lmf** (*local maxima filter*). Celle-ci comporte deux paramètres : ws (*window size*) et hmin (*height threshold*). La taille de la fenêtre de recherche des maxima locaux peut être fixe ou variable. Dans l'exemple qui suit, le paramètre ws est défini en considérant une fonction qui dépend de la hauteur locale de la canopée (ws variable).

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

# Fonction pour adapter la fenêtre ws à la hauteur
f <- function(x) { x * 0.05 + 2}
ws_args <- list(x = "Z")

# Localisation des arbres
ml1=locate_trees(las1,  lmf(hmin=12,f,ws_args=ws_args,shape="circular"))

# Création du chm
chm1=lidR::rasterize_canopy(las1,res=0.5,
                  pitfree(thresholds = c(0, 2, 5, 10, 15),
                  max_edge = c(0, 1), subcircle = 0.25))

# Affichage des couches
plot(chm1)
plot(plot0$geometry,add=T)
plot(trees$geometry,add=T)
plot(ml1$geometry,add=T,col="red",cex=1.5)
```



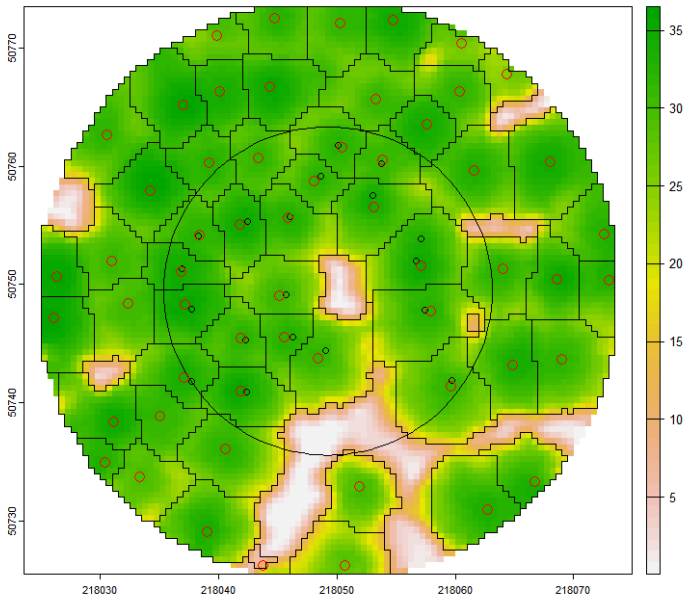
- On constate que la détection des arbres a relativement bien fonctionné. Seul un arbre observé sur le terrain a été omis par l'algorithme. Par ailleurs, un arbre qui a été détecté dans le nuage de points a été oublié lors des mesures de terrain.
- La segmentation proprement dite s'effectue avec la fonction ***segment_trees()*** à laquelle doit être associé un algorithme. Plusieurs algorithmes sont proposés pour effectuer cette segmentation. Nous en présentons deux dans les exemples qui suivent : ***dalponte2016()*** et ***silva2016()***.
- La fonction ***segment_trees()*** va attribuer un identifiant d'arbre (treeID) à chaque point du nuage.

```
# Segmentation du nuage de points --------------

# Algo dalponte2016
algo = dalponte2016(chm1, ml1 ,th_tree=hmin)
las1 = segment_trees(las1, algo)
plot(las1, bg = "white", size = 4, color = "treeID")
```



- Si l'on souhaite polygoniser les segments de cimes, on utilise le même algorithme que l'on associe au CHM et à la couche de localisation des arbres.

```
# Polygonisation des segments -------------
seg1 = dalponte2016 (chm1,ml1)()
nrow(seg1)
seg1=as.polygons(seg1,dissolve=T)

# Afficher le résultat
plot(chm1)
plot(plot0$geometry,add=T)
plot(trees$geometry,add=T)
plot(ml1$geometry,add=T,col="red",cex=1.5)
plot(seg1,add=T)
```



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)
- `pzabovezmean` : 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, ...)

$$d_i = \int_{z_{min}}^{i\frac{z_{max}}{10}} f(z)dz$$

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 1^{st} returns
- `ip2nd` : percentage of intensity returned by 2^{nd} returns
- `ip3rd` : percentage of intensity returned by 3^{rd} returns
- `ipxth` : percentage of intensity returned by x^{th} returns
- `pxth` : percentage x^{th} returns
- `pground` : percentage of returns classified as "ground"