R 01

Introduction aux géotraitements dans l'environnement R

Octobre 2020

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.
- Il est largement inspiré de l'ouvrage « Geocomputation with R » (2019) de Robin Lovelace, Jukb Nowosad et Jannes Luenchow (https://geocompr.robinlovelace.net/)
- 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(s)

- Philippe Lejeune (p.lejeune@uliege.be)
- Adrien Michez (adrien.michez@uliege.be)

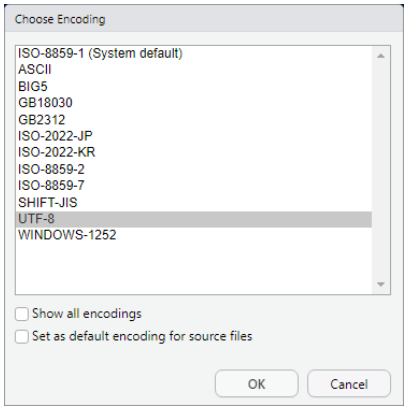
Licence de ce document

2. Introduction

- L'objectif de cet exercice est d'initier à l'utilisation des outils disponibles dans l'environnement R pour le traitement, la gestion et l'analyse de données spatiales, de type vectoriel ou raster.
- Les outils de traitement et d'analyse de données spatiales sont en pleine évolution dans l'environnement R. Anciennement, les opérations de géotraitements impliquaient de recourir à une multitude de packages : **sf** (gestion des objets vectoriels), **raster** (gestion des objets raster), **rgeos** (géotraitements vectoriels), **rgdal** (géotraitements raster)….
- Désormais, la plus grande partie des traitements peut être réalisée avec les 2 seuls packages **sf** (pour les géométries vectorielles) et **raster**. Le package **dplyr** est pour sa part utilisé pour la gestion des tables attributaires (sélection, jointures, agrégations…).
- L'ensemble des opérations présentées dans cet exercice sont rassemblées au sein d'un script **R01_GIS.r** disponible dans le jeu de données qui accompagne le présent document. Les numéro des paragraphes de ces notes d'exercices peuvent être utilisés comme point de repère pour retrouver les lignes de code dans le script.

```
#*****************************************************************
# 3.7. Lescture de tables (fichier csv)
#*****************************************************************

# Lire le fichier  "stat_pop_2018.csv"
stat_pop =read.csv("stat_population_2018.csv",
                   sep=";",stringsAsFactors = F)
names(stat_pop)

summary(stat_pop)
```

- Les manipulations sont réalisées dans l'environnement de R Studio, avec la version 3.6.1 de R.
- 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.
- **Remarque** : le script **R01_GIS.r** a été créé avec l'encodage « UTF-8 ». Si la version de R Studio dans laquelle le script est affiché utilise un autre encodage, certains caractères accentués ne s'afficheront pas correctement. Pour résoudre ce problème, il suffit de rouvrir le script avec la commande [File] → [Reopen with encoding …] et de sélectionner l'encodage « UTF-8 ».



3. Présentation du package sf

3.1 Installation des packages et chargement des librairies

- Installer et charger les librairies suivantes : sf, sp, raster, dplyr, units, purrr et lwgeom.

```
#************************************************************
# 3.1. Installation des packages et chargement des librairies
#************************************************************

# installation des packages manquants
pkgs = c(
  "sf",
  "sp",
  "raster",
  "dplyr",
  "units",
  "purrr",
  "lwgeom",
  "gdalUtils")
to_install = !pkgs %in% installed.packages()
if(any(to_install)) {
  install.packages(pkgs[to_install])
}

# chargement des librairies
library(sp)
library(sf)
library(raster)
library(dplyr)
library(units) # gestion des unités de mesure
library(purrr) # fonctions de programmation
library(lwgeom) # fonctions topologiques
```

3.2 Lecture et écriture de données vectorielles (objet sf)

3.2.1 Lecture de données vectorielles

- Avant toute chose, il convient de définir le répertoire de travail avec la fonction ***setwd().*** Celle-ci-doit être adaptée au répertoire dans lequel sont rangées les données de l'exercice.

```
setwd("C:/tmp/R_GIS_01/data")
```

- La lecture d'une couche vectorielle pour créer un objet sf utilise la fonction ***st_read()***.

```
# Lire une couche vectorielle (-> objet sf)
comm=st_read("communes.shp",stringsAsFactors = F)
```

Remarque : l'option « stringsAsFactors = FALSE » évite de convertir les colonnes de type texte en colonnes de type « Factor ».

Remarque : l'utilisation de la fonction setwd() pour charger les fichiers peut s'avérer problématique lorsque les données utilisée dans un script sont rangées dans des répertoires différents. Une alternative consiste à définir des variables qui contiennent les chemins absolus des fichiers.

```
# définir les chemins absolus des fichiers
path0="C:/tmp/R_GIS_01/data"
file_in=paste0(path0,"/communes.shp")
comm=st_read("communes.shp",stringsAsFactors = F)
```

3.2.2 Sauvegarde de données vectorielles

- La sauvegarde d'un objet sf sous forme de shapefile s'obtient avec la fonction ***st_write()***. L'option « delete_layer = TRUE » autorise l'écrasement de fichiers préexistants.

```
st_write(comm,"communes_1.shp",delete_layer=TRUE)
# delete_layer=TRUE : permet d'écraser les fichiers existants
```

- Les messages d'avertissement qui accompagnent l'exécution de cette commande peuvent être affichés avec la commande warnings(). Ils informent que la sauvegarde de la couche vectorielle dans le shapefile s'est faite avec des arrondis dans les valeurs de l'attribut SHAPE_AREA en raison du nombre important de décimales présentes dans les valeurs originales.

```
There were 50 or more warnings (use warnings() to see the first 50)
> warnings()
Messages d'avis :
1: In CPL_write_ogr(obj, dsn, layer, driver, as.character(dataset_options),  ... :
  GDAL Message 1: Value 135723300.748 of field SHAPE_Area of feature 0 not successfully written.
 Possibly due to too larger number with respect to field width
```

3.3 Lecture de données vectorielles → objet sp

- La lecture d'une couche vectorielle pour créer un objet sp utilise la fonction ***raster::shapefile()***

```
# Lire une couche vectorielle (-> objet sp)
comm2 = raster::shapefile("communes.shp")
```

- Remarque : la syntaxe ***raster ::shapefile()*** signifie que la fonction ***shapefile()*** appartient au package **raster**. Le nom du package auquel appartient la fonction n'est pas obligatoire :

```
comm2 = shapefile("communes.shp")
```

3.4 Différence entre objets sf et sp

- Un objet sf prend la forme d'un dataframe qui contient un champ « de géométrie » dans lequel sont stockées les coordonnées délimitant les objets. Par défaut, ce champ est baptisé « geometry ».

```
> class(comm)
[1] "sf"         "data.frame"
> names(comm)
 [1] "OBJECTID"   "ADMUKEY"    "ADPRKEY"    "ADMULG"     "ADMUNAFR"
 [6] "ADMUNADU"   "ADMUNAGE"   "HIGHLIGHT"  "SHAPE_Leng" "SHAPE_Area"
[11] "geometry"
```

- Un objet sp prend la forme d'un spatial dataframe. Les données géométriques sont stockées indépendamment du dataframe qui contient la table d'attributs.

```
> class(comm2)
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
> names(comm2)
 [1] "OBJECTID"   "ADMUKEY"    "ADPRKEY"    "ADMULG"     "ADMUNAFR"
 [6] "ADMUNADU"   "ADMUNAGE"   "HIGHLIGHT"  "SHAPE_Leng" "SHAPE_Area"

> class(comm)
[1] "sf"         "data.frame"
> class(comm2)
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

3.5 Conversion entre objets sf et sp

- La conversion entre classes sf et sp s'effectue avec la fonction ***as()***.

```
> # sf -> sp
> comm_sp = as(comm, "Spatial")
> class(comm_sp)
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"

> # sp -> sf
> comm_sf =  as(comm2, "sf")
> class(comm_sf)
[1] "sf"         "data.frame"
```

- Dans la suite de l'exercice, seul l'objet sf sera exploré pour gérer des données vectorielles.

3.6 Système de coordonnées

3.6.1 Identifier le système de coordonnées (CRS) d'un objet sf

- La fonction ***st_crs()*** permet d'identifier le système de coordonnées d'un objet sf.

```
> st_crs(comm)
Coordinate Reference System:
  User input: Belge 1972 / Belgian Lambert 72
  wkt:
PROJCRS["Belge 1972 / Belgian Lambert 72",
    BASEGEOGCRS["Belge 1972",
        DATUM["Reseau National Belge 1972",
            ELLIPSOID["International 1924",6378388,297,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4313]],
    CONVERSION["Belgian Lambert 72",
        METHOD["Lambert Conic Conformal (2SP)",
            ID["EPSG",9802]],
        PARAMETER["Latitude of false origin",90,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8821]],
        PARAMETER["Longitude of false origin",4.36748666666667,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8822]],
        PARAMETER["Latitude of 1st standard parallel",51.1666672333333,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8823]],
        PARAMETER["Latitude of 2nd standard parallel",49.8333339,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8824]],
        PARAMETER["Easting at false origin",150000.013,
            LENGTHUNIT["metre",1],
            ID["EPSG",8826]],
        PARAMETER["Northing at false origin",5400088.438,
            LENGTHUNIT["metre",1],
            ID["EPSG",8827]]],
    CS[Cartesian,2],
        AXIS["easting (X)",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["northing (Y)",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    USAGE[
        SCOPE["unknown"],
        AREA["Belgium - onshore"],
        BBOX[49.5,2.5,51.51,6.4]],
    ID["EPSG",31370]]
```

- Cette fonction renvoie un objet de type « CRS ». Le système de coordonnées y est décliné selon 2 modalités : (1) une chaîne de caractère contenant le nom du système de coordonnées et (2) une seconde chaîne de caractère contenant le descriptif du système de coordonnées avec une représentation en mode WKT. Cette dernière contient un attribut [ID] dans lequel on retrouve le code EPSG du système de coordonnées.

3.6.2 *Modifier le système de coordonnées (CRS) d'un objet sf*

- On peut modifier le « CRS » d'un objet sf dès lors que celui-ci est absent ou incorrect.
- Par exemple, le système de coordonnées de la couche **localites.shp** n'est pas défini. L'opérateur sait que cette couche a été produite dans le système de coordonnées EPSG : 31370.
- La fonction ***st_crs()*** peut être utilisée pour lui attribuer le CRS correspondant au code EPSG : 31370.

```
> loc=read_sf("localites.shp")
> st_crs(loc)
Coordinate Reference System: NA
> # attribuer le CRS epsg:31370 à la couche loc
> st_crs(loc)=31370
> st_crs(loc)
Coordinate Reference System:
  User input: EPSG:31370
  wkt:
PROJCRS["Belge 1972 / Belgian Lambert 72",
    BASEGEOGCRS["Belge 1972",
        DATUM["Reseau National Belge 1972",
            ELLIPSOID["International 1924",6378388,297,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4313]],
    CONVERSION["Belgian Lambert 72",
        METHOD["Lambert Conic Conformal (2SP)",
            ID["EPSG",9802]],
```

3.6.3 *Reprojeter une couche vectorielle*

- La fonction ***st_transform()*** permet de reprojeter une couche vectorielle contenue dans un objet sf. Il suffit d'indiquer le code EPSG « cible ».

```
> loc_wgs84 = st_transform(loc, 4326)
> st_crs(loc_wgs84)
Coordinate Reference System:
  User input: EPSG:4326
  wkt:
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
```

3.6.4 *Emprise d'une couche vectorielle*

- L'emprise d'une couche vectorielle peut être appréhendée avec la fonction ***st_bbox()*** ou la fonction ***extent()***.

```
> st_bbox(loc)
     xmin      ymin      xmax      ymax
 45721.92  22377.02 291375.30 172090.27
> st_bbox(loc_wgs84)
     xmin      ymin      xmax      ymax
 2.887875 49.507861  6.355449 50.849862
>
> extent(loc)
class       : Extent
xmin        : 45721.92
xmax        : 291375.3
ymin        : 22377.02
ymax        : 172090.3
```

3.7 Lecture et écriture de données tabulaires (dataframe)

3.7.1 *Lecture et écriture de fichiers .csv*

- Le format d'entrée des données tabulaires le plus couramment utilisé est le format .csv. Ces fichiers peuvent être chargés dans l'environnement R sous forme de dataframe avec la fonction ***read.csv()***
- Lire le fichier **stat_pop_2018.csv** qui contient les statistiques de population par commune pour l'année 2018 (source : www.statbel.fgov.be).

```
# 3.7.1. Lecture et écriture de fichiers .csv

stat_pop =read.csv("stat_population_2018.csv",
                   sep=";",stringsAsFactors = F)
names(stat_pop)

summary(stat_pop)
```

```
> summary(stat_pop)
     INS                Lieu              Hommes            Femmes            Total
 Length:650         Length:650         Min.   :     50   Min.   :     38   Min.   :      88
 Class :character   Class :character   1st Qu.:   3938   1st Qu.:   3969   1st Qu.:    7886
 Mode  :character   Mode  :character   Median :   6562   Median :   6747   Median :   13309
                                       Mean   :  42419   Mean   :  43775   Mean   :   86194
                                       3rd Qu.:  12388   3rd Qu.:  12973   3rd Qu.:   25193
                                       Max.   :5597906   Max.   :5778164   Max.   :11376070
                                       NA's   :4         NA's   :4         NA's   :4
```

- L'option ***dec*** permet de préciser la nature du séparateur décimal.

```
# Lire le fichier  "stat_agricole_2018.csv"
# le séparateur décimal de ce fichier est la virgule

stat_agri=read.csv("stat_agricole_2018.csv",
                   sep=";",dec=",", stringsAsFactors = F)
head(stat_agri)
```

```
> head(stat_agri)
    ins     commune nb_exploitation surf_agr_utile terre_arable prairie_perm pct_terre_arable
1 11001 AARTSELAAR              12          32040        17975        11837             56.1
2 11002      ANVERS              39         147747        69849        77730            47.28
3 11004   BOECHOUT               40          45039        21681        15318            48.14
4 11005       BOOM                0              0            0            0
5 11007   BORSBEEK                0              0            0            0
6 11008 BRASSCHAAT                5           3922         1437         2216            36.64
  pct_prairie nb_bovins nb_porcins
```

- La sauvegarde d'un dataframe dans 1 fichier .csv s'effectue avec la fonction ***write.table()***.

```
# Ecrire  le dataframe stat_agri dans 1 fichier .csv
f_out="stat.csv"
write.table(stat_agri,file=f_out,col.names = T,row.names = F,sep=";",dec=".")
```

3.7.2 Lecture et écriture de fichiers .xlsx

- Il est possible d'accéder directement aux feuilles d'un fichier Excel sans passer par le format .csv. Plusieurs packages permettent de lire et écrire des fichiers Excel. Dans l'exemple qui suit, nous utilisons le package ***xlsx***.
- Lire les 2 feuilles contenues dans le fichier **stat_2018.xlsx**.

```
# Lire le fichier  "stat_2018.xlsx"
f_in="stat_2018.xlsx"
stat_pop=read.xlsx(f_in,1)
stat_agri=read.xlsx(f_in,2)
```

- Le choix de la feuille à lire s'effectue en précisant son numéro d'ordre dans le classeur ou son nom.
- La sauvegarde de dataframe dans un fichier .xlsx s'opère avec la fonction ***write.xlsx()***. A noter que l'option append = TRUE permet d'ajouter une feuille à un fichier existant.

```
# Créer un fichier .xlsx pour y copier les df stat_agri et stat_pop
f_out="stat_2018_out.xlsx"
write.xlsx(stat_agri,f_out,sheetName = "s_agricole")
write.xlsx(stat_pop,f_out,sheetName = "s_agricole",append=TRUE)
```

3.8 Sélectionner et/ou renommer des champs dans un objet sf

- L'objet **comm** relatif aux communes de Wallonie, contient plusieurs champs qui sont inutiles pour la suite de l'exercice.

```
> names(comm)
 [1] "OBJECTID"   "ADMUKEY"    "ADPRKEY"    "ADMULG"     "ADMUNAFR"
 [6] "ADMUNADU"   "ADMUNAGE"   "HIGHLIGHT"  "SHAPE_Leng" "SHAPE_Area"
[11] "geometry"
```

- La fonction ***dplyr ::select()*** est utilisée pour sélectionner certains champs et, éventuellement, renommer une partie de ceux-ci. La commande ci-dessous ne conserve que les 5 premiers champs du dataframe de l'objet **comm**. Il est important de noter que le champ [geometry] est conservé par défaut, sans devoir le faire apparaître dans la liste des champs à conserver.

```
# Ne conserver que les 5 premiers champs de "comm"
comm1 = dplyr::select(comm,OBJECTID:ADMUNAFR)
names(comm1)

[1] "OBJECTID" "ADMUKEY"  "ADPRKEY"  "ADMULG"   "ADMUNAFR" "geometry"
```

- L'exemple qui suit opère la même sélection et renomme certains champs.

```
# Même sélection de champs + renommer certains d'entre eux
comm1 = dplyr::select(comm,OBJECTID:ADMUNAFR,
                      INS = ADMUKEY, nom = ADMUNAFR)
names(comm1)

[1] "OBJECTID" "INS"      "ADPRKEY"  "ADMULG"   "nom"      "geometry"
```

- La fonction **dplyr::rename** peut être utilisée pour renommer un seul champ.

```
# Renommer un seul champ
comm1 <- rename(comm1, "langue" = "ADMULG")
```

- La fonction ***purrr :set_names()*** est utilisée pour renommer l'entièreté des champs d'un dataframe.

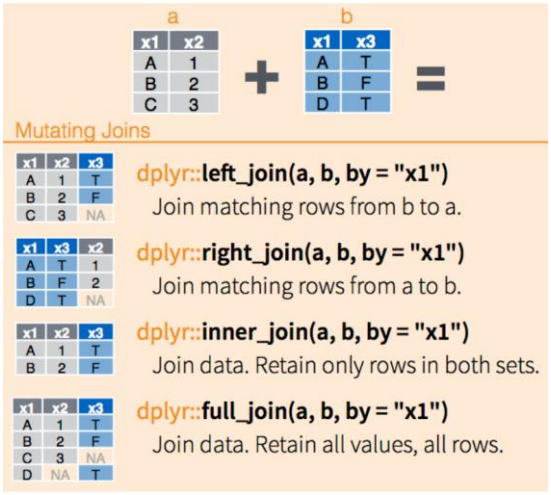
```
# Renommer tous les champs du df
#install.packages("purrr")
library(purrr)
new_names = c("OBJECTID", "INS", "province", "langue", "nom", "geom")
comm1=purrr::set_names(comm1,new_names)
names(comm1)

[1] "OBJECTID" "INS"      "province" "langue"   "nom"      "geom"
```

- Remarque : le champ [geometry] doit être repris dans la liste des champs à renommer.

3.9 Jointures de tables

- La librairie **dplyr** contient plusieurs outils de jointure de table. Appliquées à un objet de type sf, les jointures conservent les propriétés géométriques de ce dernier, lorsqu'il constitue le premier argument de la fonction (destination de la jointure).
- La figure suivante résume le fonctionnement des différents types de jointure disponibles dans **dplyr** (https://mikoontz.github.io/data-carpentry-week/lesson_joins.html).

  Ces jointures fonctionnent selon le modèle relationnel et s'apparentent au langage SQL.



☞ Réaliser une jointure entre l'objet sf **comm1** représentant les communes de Wallonie et le dataframe **stat_pop** contenant les statistiques de population par commune pour la Belgique en 2018.

- La première étape dans la mise en œuvre d'une jointure est d'identifier les champs utilisés pour établir une relation entre les 2 tables à joindre. Dans le cas présent il s'agit des champs [INS].

```
> names(comm1)
[1] "OBJECTID" "INS"      "province" "langue"   "nom"      "geom"
> names(stat_pop)
[1] "INS"    "Lieu"   "Hommes" "Femmes" "Total"
```

- Il faut également s'assurer que les 2 champs contiennent les mêmes types de données. Et procéder aux éventuels ajustements.

```
> class(comm1$INS)
[1] "character"
> class(stat_pop$INS)
[1] "integer"
> class(comm1$INS)

stat_pop$INS=as.character(stat_pop$INS)
```

- Il n'est pas obligatoire que les champs à joindre portent le même nom.
- La jointure utilisée dans le cas présent est une jointure gauche : elle permet de conserver tous les enregistrements de la couche **comm1** et d'y ajouter les éléments de la table **stat_pop** qui présentent une correspondance au niveau du champ [INS].

```
comm2 = left_join(comm1,stat_pop, by = c("INS" = "INS"))
names(comm2)

[1] "OBJECTID" "INS"      "province" "langue"   "nom"      "Lieu"
[7] "Hommes"   "Femmes"   "Total"    "geometry"
```

La fonction **summary()** permet de vérifier qu'il n'y a pas de données manquantes dans les champs ajoutés à l'objet **comm2**.

```
> summary(comm2)
    OBJECTID          INS              province           langue              nom
 Min.   :  1.00   Length:262         Length:262         Length:262         Length:262
 1st Qu.: 66.25   Class :character   Class :character   Class :character   Class :character
 Median :131.50   Mode  :character   Mode  :character   Mode  :character   Mode  :character
 Mean   :131.50
 3rd Qu.:196.75
 Max.   :262.00
     Lieu              Hommes          Femmes           Total                geometry
 Length:262         Min.   :  701   Min.   :  706   Min.   :  1407   MULTIPOLYGON :262
 Class :character   1st Qu.: 2608   1st Qu.: 2622   1st Qu.: 5236   epsg:31370   :  0
 Mode  :character   Median : 4192   Median : 4282   Median : 8505   +proj=lcc ...:  0
                    Mean   : 6756   Mean   : 7077   Mean   : 13834
                    3rd Qu.: 7146   3rd Qu.: 7544   3rd Qu.: 14664
                    Max.   :98474   Max.   :103342   Max.   :201816
```

3.10 Sélection par attributs et par localisation

3.10.1 La fonction dplyr::filter()

- Les sélections au sein d'un objet sf peuvent être réalisées avec la fonction ***dplyr::filter()***. Cette commande utilise comme premier argument l'objet de référence (sur lequel porte la sélection) et comme second argument une condition : c'est 1 vecteur « logical » (VRAI/FAUX) de longueur équivalente à l'objet de référence. Seules les lignes de l'objet de référence pour lesquelles la condition est « VRAI » sont conservées.
- La différence entre sélection par attributs et sélection par localisation porte sur la manière de construire le vecteur « logical ».

3.10.2 Sélection par attributs

☞ Créer un nouvel objet sf qui contient uniquement les communes de la province de Namur.

- Identifier la valeur du champ [province] correspondant à la province de Namur

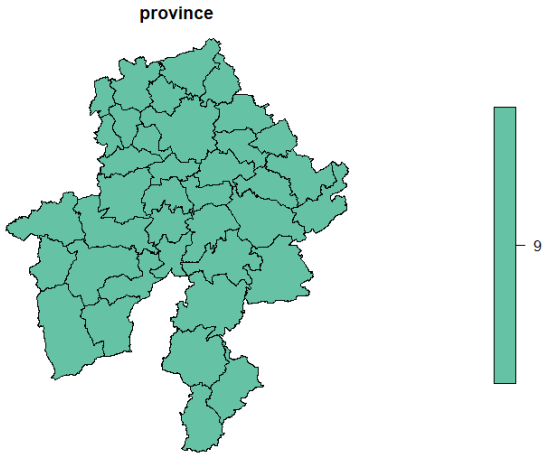
```
plot(comm2[,"province"])
# la province de Nmaur correspond aux code "9"
```

- Créer une requête sous la forme d'un vecteur de valeur TRUE/FALSE

```
query = comm2$province == 9
class(query)
```

- Utiliser ensuite ce vecteur comme critère de sélection des objets à conserver.

```
comm_nam = dplyr::filter(comm2,query)
plot(comm_nam[3])
```



3.10.3 Sélection par localisation (spatial subsetting)

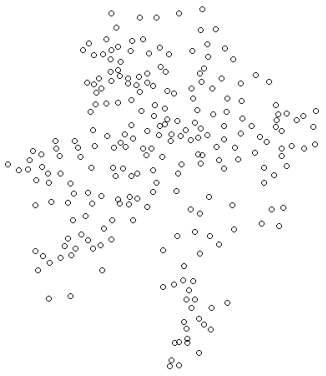
- Les fonctions de construction de requêtes spatiales comparent 2 listes de géométries (par exemple 2 objets sf) de dimension n1 et n2. Elles renvoient une matrice de dimension n1 x n2 qui contient des valeurs VRAI/FAUX traduisant le fait que la relation spatiale testée est VRAI/FAUX pour la paire d'objets i x j (i appartenant à la série 1 et j à la série 2).
- Le plus souvent la seconde liste ne contient qu'un élément par rapport auquel est testée la relation spatiale pour les objets de la première liste.
- La figure suivante liste les principales fonctions utilisées pour exprimer des relations spatiales (https://r-spatial.github.io/sf/reference/geos_binary_pred.html)

```
st_intersects(x, y, sparse = TRUE, ...)

st_disjoint(x, y = x, sparse = TRUE, prepared = TRUE)

st_touches(x, y, sparse = TRUE, prepared = TRUE)

st_crosses(x, y, sparse = TRUE, prepared = TRUE)

st_within(x, y, sparse = TRUE, prepared = TRUE)

st_contains(x, y, sparse = TRUE, prepared = TRUE)

st_contains_properly(x, y, sparse = TRUE, prepared = TRUE)
```

```
st_overlaps(x, y, sparse = TRUE, prepared = TRUE)

st_equals(x, y, sparse = TRUE, prepared = FALSE)

st_covers(x, y, sparse = TRUE, prepared = TRUE)

st_covered_by(x, y, sparse = TRUE, prepared = TRUE)

st_equals_exact(x, y, par, sparse = TRUE, prepared = FALSE)

st_is_within_distance(x, y, dist, sparse = TRUE)
```

Sélectionner les localités situées dans la province de Namur.

```
# sélectionner les localités situées dans la province de Namur
loc_nam = filter(loc,st_intersects(loc,prov_namur,sparse=F))
plot(loc_nam[4])
```



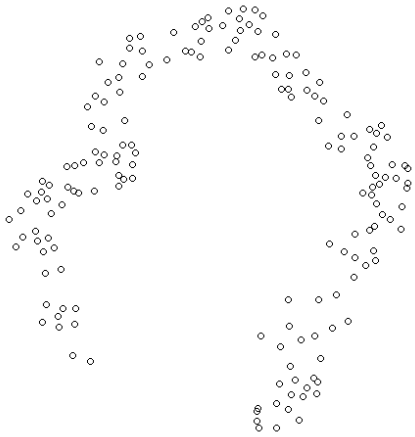
- Remarque : l'option « sparse=F » est obligatoire lorsque la fonction « filter » utilise un critère de sélection de type spatial.

Sélectionner les localités situées **en dehors** de la province de Namur.

```
# sélectionner les localités situées hors de la province de Namur
loc_hors_nam = filter(loc,st_disjoint(loc,prov_nam,sparse=F))
plot(loc_hors_nam[4])
```

Sélectionner les localités situées **en dehors** de la province de Namur ET à moins de 10 km de celle-ci.

```
# sélectionner les localités situées hors de la province de Namur
# ET qui se trouvent à moins de 10 km de celle-ci
loc_hors_nam_LT10km = filter(loc_hors_nam,
        st_is_within_distance(loc_hors_nam,prov_nam,10000,sparse=F))
plot(loc_hors_nam_LT10km[4])
```



3.11 Création de nouveaux attributs dans un objet sf

3.11.1 Calculer la surface de polygones

- La surface des polygones est calculée avec la fonction ***st_area()***.

```
# surfaces en m²
comm_nam$surf=st_area(comm_nam)
head(comm_nam$surf)

Units: [m^2]
[1]  76562633 104756950 122831258  65692871 166366697  95239434

> class(comm_nam$surf)
[1] "units"
```

- Le vecteur généré par la fonction ***st_area()*** est de type « units », c'est-à-dire que les valeurs numériques qu'il contient sont liées à une unité de mesure bien identifiée. Par défaut les unités de mesure de surfaces découlent des unités dans lesquelles sont définies les coordonnées xy. Dans le cas présent, il s'agit de m².
- Cette manière de fonctionner offre une garantie dans la cohérence des calculs qui sont réalisés en aval, en évitant une confusion entre unités de mesure (m² vs ha vs km²).

☞ Calculer la surface des polygones de l'objet **comm_nam** et exprimer le résultat en hectares

---

- Lorsque l'on souhaite modifier les unités par défaut, il faut compléter le calcul de la surface (***st_area()***), par une définition des unités à utiliser. Cette seconde opération est prise en charge par la fonction ***units::set_units()***.

```
# surfaces en ha - syntaxe avec imbrication des fonctions
comm_nam$surf_ha=set_units(st_area(comm_nam),ha)
head(comm_nam$surf_ha)
```

```
> head(comm_nam$surf_ha)
Units: [ha]
[1]  7656.263 10475.695 12283.126  6569.287 16636.670  9523.943
```

- On observe que dans la commande précédente, les 2 fonctions sont utilisées de manière imbriquée. Cette syntaxe présente plusieurs inconvénients :
  - Elle peut devenir difficilement lisible en cas d'imbrications multiples.
  - Les opérations apparaissent dans l'ordre inverse de leur exécution
  - Il est parfois difficile de voir à quel paramètre se rapporte quelle fonction.
- Une solution peut consister à effectuer les opérations les unes après les autres. Le sccript gagne en lisibilité, mais devient plus long.

```
# surfaces en ha - syntaxe avec fonctions séparées
comm_nam$surf_ha=st_area(comm_nam)
comm_nam$surf_ha=set_units(comm_nam$surf_ha,ha)
```

- Une troisième solution consiste à adopter une syntaxe en « mode pipe ». L'opérateur ***pipe*** est noté ***%>%***. Il fonctionne de la manière suivante : lorsqu'on exécute une commande

```
expression %>% fonction(param2, param3, ...)
```

Le résultat de l'expression située à gauche du pipe est passé comme premier argument à la fonction située à droite du pipe. La commande présentée ci-dessus revient à exécuter ceci :

```
fonction(expression, param2, param3, ...)
```

☞ Calculer la surface des polygones de l'objet **comm_nam** en km², en utilisant la syntaxe « mode pipe »

```
# surfaces en km² - syntaxe en "mode pipe"
comm_nam$surf_km2 = comm_nam %>% st_area() %>% set_units(km^2)
head(comm_nam$surf_km2)

Units: [km^2]
[1]  76.56263 104.75695 122.83126  65.69287 166.36670  95.23943
```

- Remarque : pour s'affranchir des unités associées à la fonction ***st_area()***, il suffit de transformer les résultats en valeur numérique avec la fonction ***as.numeric()***.

```
# surfaces en km² - suppression des unités
comm_nam$surf_km2 = as.numeric(st_area(comm_nam)/1000000)
head(comm_nam$surf_km2)

[1]  76.56263 104.75695 122.83126  65.69287 166.36670  95.23943
```

3.11.2 Copier les coordonnées dans la table d'attributs (couche de points)

- Les coordonnées des points constituant les géométries d'un objet sf sont stockées dans la colonne [geometry]. Dans le cas d'une couche de points, on peut être intéressé à les faire apparaître dans le dataframe. Cette opération est réalisée avec la fonction ***st_coordinates()***.

☞ Recopier les coordonnées des points de la couche **eoliennes.shp** dans la table d'attributs.

```
# 3.11.2 Ajouter les coordonnées des points dans la table d'attributs
eol=st_read("eoliennes.shp")
eol$x=st_coordinates(eol)[,1]
eol$y=st_coordinates(eol)[,2]
head(eol)
```

```
> head(eol)
Simple feature collection with 6 features and 4 fields
geometry type:  POINT
dimension:      XY
bbox:           xmin: 158954.3 ymin: 109492.8 xmax: 160383.1 ymax: 110946.5
epsg (SRID):    31370
proj4string:    +proj=lcc +lat_1=51.16666723333333 +lat_2=49.8333339 +lat_0=90
 +lon_0=4.367486666666666 +x_0=150000.013 +y_0=5400088.438 +ellps=intl +towgs8
4=-106.8686,52.2978,-103.7239,0.3366,-0.457,1.8422,-1.2747 +units=m +no_defs
  ID_eol id_parc                  geometry        x         y
1      1      32 POINT (158954.3 110311.7) 158954.3 110311.7
2      2      32 POINT (159649.2 110455.3) 159649.2 110455.3
3      3      32 POINT (160045.6 110946.5) 160045.6 110946.5
4      4      32 POINT (159236.6 109492.8) 159236.6 109492.8
5      5      32 POINT (160383.1 110335.1) 160383.1 110335.1
6      6      32   POINT (159628 109849.4) 159628.0 109849.4
```

3.11.3 Création d'un nouveau champ non géométrique par jointure de table

☞ Créer un nouveau champ dans l'objet **comm_nam** pour stocker le nom de l'arrondissement dont relève la commune.

- Cet exemple permet d'illustrer la combinaison d'une jointure de table avec différentes opération sur les champs du dataframe contenu dans l'objet sf.
- Les noms des arrondissements sont contenus dans le fichier ***arrondissements_2018.csv***.

```
# Lire le fichier arrondissement_2018.csv
arrond=read.csv("arrondissements_2018.csv", sep=";"
                                ,stringsAsFactors = F)
head(arrond)
```

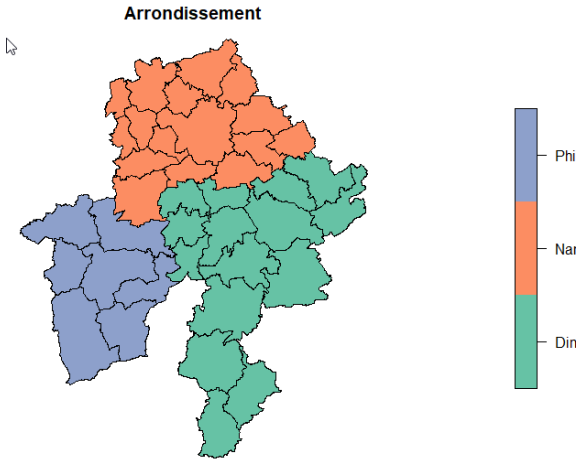
```
> head(arrond)
    INS     Arrondissement
1 11000             Anvers
2 12000            Malines
3 13000           Turnhout
4 21000 Bruxelles-Capitale
5 23000       Hal-Vilvorde
6 24000             Louvain
```

- Les arrondissements, tout comme les communes, sont identifiés par un code « INS ». Le lien entre communes et arrondissements peut être établi en considérant que les codes INS des arrondissements sont des multiples de 1000 (11000 pour l'arrondissement d'Anvers), et que les codes « INS » communes relevant d'un arrondissement correspondent à la série de valeurs entières supérieures ou égales au code de l'arrondissement (les communes de l'arrondissement d'Anvers possèdent les codes « INS » : 11001, 11002, 11003…)

```
> comm_nam$INS
 [1] "91059" "91064" "91072" "91103" "91114" "91120" "91141" "91142" "91143"
[10] "92003" "92006" "92035" "92045" "92048" "92054" "92087" "92094" "92097"
[19] "92101" "92114" "92137" "92138" "92140" "92141" "92142" "93010" "93014"
[28] "93018" "93022" "93056" "93088" "93090" "91005" "91013" "91015" "91030"
[37] "91034" "91054"
>
> class(comm_nam$INS)
[1] "character"
```

- Sur base de ces éléments, proposer une méthode permettant de répondre à la question posée.
- **La réponse est présentée à la page suivante …**

```
# step 1 : créer un champ comm_nam$id_arrond compatible avec arrond$INS
comm_nam$id_arr=as.integer(as.numeric(comm_nam$INS)/1000)*1000

# step 2 : créer 1 jointure pour insérer le nom de l'arrond. dans comm_nam
comm_nam=left_join(comm_nam,arrond,by = c("id_arr" = "INS"))

# vérifier le résultat
names(comm_nam)
plot(comm_nam[,"Arrondissement"])
```



3.12 Création de tableaux de synthèse (summarize)

- Il existe plusieurs outils permettant l'agrégation de données tabulaires. Par souci de simplification, seul l'outil **dplyr::summarize()** sera présenté.
- Lorsque cette fonction est appliquée à un objet sf, le résultat est un objet sf. La création du tableau de synthèse s'accompagne généralement d'une fusion des objets agrégés.

- Remarque : les fonctions ***summarize()*** et ***summarise()*** sont identiques

---

☞ Calculer pour chaque arrondissement de la province de Namur : le nombre de communes, la population totale, ainsi que la densité de population moyenne par commune

---

- La première étape consiste à calculer la densité de population (habitants/km²) des communes.

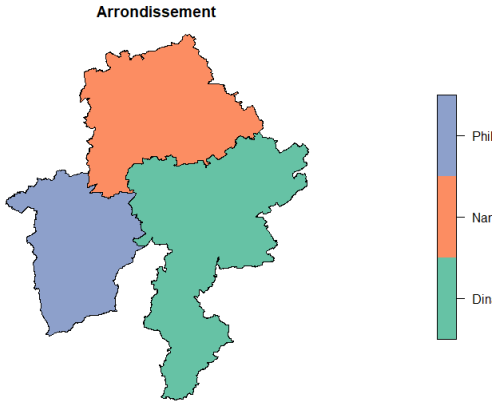
```
# calculer la densité de popumlation par commune
comm_nam$pop_dens=comm_nam$Total/comm_nam$surf_km2
summary(comm_nam$pop_dens)

> summary(comm_nam$pop_dens)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  26.03   58.66  106.78  151.88  163.85  823.24

> names(comm_nam)
 [1] "OBJECTID"       "INS"            "province"       "langue"
 [5] "nom"            "Lieu"           "Hommes"         "Femmes"
 [9] "Total"          "surf_ha"        "surf"           "surf_km2"
[13] "id_arr"         "Arrondissement" "geometry"       "pop_dens"
```

- La synthèse est ensuite réalisée avec la fonction ***summarize()*** complétée de la fonction ***group_by()*** pour définir le critère d'agrégation. La commande est écrite en utilisant la syntaxe en « mode pipe ».

```
# aggréger les données
arr_nam = comm_nam %>%
            group_by(Arrondissement) %>%
            summarize(nb_comm = n(),
                      pop_tot = sum(Total),
                      pop_dens_moy = mean(pop_dens))
class(arr_nam)

> class(arr_nam)
[1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

- On constate que la structure de l'objet résultant comporte 2 classes supplémentaires : « tbl_df » et « tbl ». Celles-ci constituent des variantes du dataframe déjà présent dans l'objet. Elles sont générées lors de l'utilisation de la librairie ***dplyr*** . Ces classes supplémentaires n'altèrent en rien les caractéristiques géométriques de l'objet sf.
- L'affichage de celui-ci permet de constater que les polygones correspondant aux communes ont été fusionnés par arrondissement.

```
plot(arr_nam[1])
```



- On peut transformer ce résultat en simple dataframe (tableau). Il suffit pour cela de supprimer les classes inutiles.

```
# convertir l'objet en dataframe
# (abandon des classes "sf", "tbl_df" et "tbl")
df_arr_nam = as.data.frame(st_drop_geometry(arr_nam))
class(df_arr_nam)
df_arr_nam
```

```
[1] "data.frame"
```

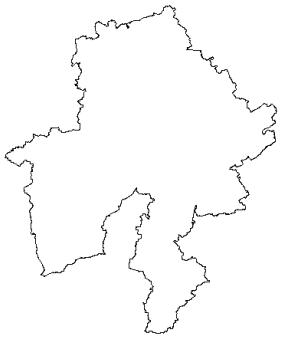
	Arrondissement	nb_comm	pop_tot	pop_dens_moy
1	Dinant	15	110610	75.11397
2	Namur	16	316058	259.02575
3	Philippeville	7	66405	71.45894

☞ Produire un objet sf correspondant à la province de Namur

- En l'absence de l'option ***group_by()***, la méthode summarize() va opérer l'agrégation sur tous les éléments contenus dans l'objets sf traité.

```
# agréger toutes les communes de la province de Namur
prov_nam = comm_nam %>%
  summarize(nb_comm = n(),
            pop_tot = sum(surf))
names(prov_nam)
plot(prov_nam[3])
```

```
[1] "nb_comm"  "pop_tot"  "geometry"
```



- Si on est uniquement intéressé par la création de l'objet géométrique, la commande suivante peut aussi être utilisée.

```
# summarize sans calcul statistique
prov_nam = comm_nam %>%
  summarize()
names(prov_nam)
```

3.13 Fusion de plusieurs objets sf

☞ Produire un objet sf correspondant à la fusion des fichiers **100kmE39N29.shp** et **100kmE39N30.shp**.

```
# Fusionner les shapefiles 100kmE39N29.shp et 100kmE39N30.shp
clc1=st_read("100kmE39N30.shp")
clc2=st_read("100kmE39N29.shp")
nrow(clc1)
nrow(clc2)

> nrow(clc1)
[1] 7838
> nrow(clc2)
[1] 4448
```

- La fusion d'objets sf est réalisée avec la fonction ***rbind()***.

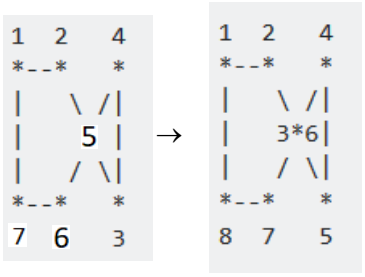
```
clc=rbind(clc1,clc2)
nrow(clc)

> nrow(clc)
[1] 12286
```

3.14 Vérification de la validité de couches vectorielles

- Préalablement à certains géotraitements, il peut s'avérer utile de vérifier la validité topologique d'un objet sf. Ce test peut être réalisé avec la fonction ***st_is_valid()***. Les éventuelles erreurs peuvent être corrigées avec la fonction ***st_make_valid()*** de la librairie ***lwgeom***.
- Les principales erreurs rencontrées concernent des géométries vides (polygone constitué de 0 point) ou des ***auto-intersections***. Ce dernier phénomène se produit lorsqu'un polygone s'auto-intersecte en un seul point comme c'est le cas dans la figure suivante.

Auto-intersection du polygone jaune

- En cas d'auto-intersection, la fonction ***st_make_valid()*** duplique le point d'auto-intersection afin de rendre le polygone valide en regard des règles topologiques.




```
# Vérifier la validité géométrique d'un objet sf
test=st_is_valid(clc,reason=T)
table(test)
```

```
test
Ring Self-intersection[138335.192255512 33749.5392231001]
                                                        1
Ring Self-intersection[165739.791611671 131755.226867391]
                                                        1
Ring Self-intersection[171615.857753845 123459.326357162]
                                                        1
Ring Self-intersection[177483.442720612 47169.8963334616]
                                                        1
Ring Self-intersection[180272.248601168 27383.9014600664]
                                                        1
Ring Self-intersection[198029.614379534 38046.4615226965]
                                                        1
 Ring Self-intersection[200684.45123619 96534.6097705895]
                                                        1
Ring Self-intersection[204215.883388184 122885.843701713]
                                                        1
 Ring Self-intersection[211134.71675667 55559.1473038448]
                                                        1
Ring Self-intersection[224086.061415774 88897.2288439469]
                                                        1
Ring Self-intersection[227366.756253684 47253.2251866199]
                                                        1
Ring Self-intersection[233113.899044576 45113.7809174098]
                                                        1
                                        Valid Geometry
                                                 12274
```

```
clc_valid=st_make_valid(clc)
test=st_is_valid(clc_valid,reason=T)
table(test)
```

```
> test=st_is_valid(clc_valid,reason=T)
> table(test)
test
Valid Geometry
         12286
```

3.15 Intersection

- L'intersection de 2 objets sf génère un nouvel objet sf qui contient des entités résultant de l'intersection géométrique des entités présentes dans les 2 objets. Les entités créées héritent en outre des attributs des entités « parentes ». Cette opération est prise en charge par la fonction ***st_intersection()***.

- **Remarque** : cette fonction ne doit pas être confondue avec la fonction ***st_intersects()*** utilisée pour réaliser des requêtes spatiales lors de sélection par localisation

---

Réaliser l'intersection entre l'objet **comm_nam**, contenant les communes de la province de Namur et un objet sf contenant le shapefil **routes.shp**. En déduire la longueur de routes par commune.

- Charger le shapefile **routes.shp** dans un objet sf.
- Réaliser l'intersection des 2 objets sf.

```
int_route_comm=st_intersection(routes,comm_nam)
```

- Calculer ensuite la longueur des segments de route en km.

```
# calculer la longueur des segments de route
int_route_comm$long_km =
  int_route_comm %>% st_length() %>% set_units(km)
head(int_route_comm$long_km)

Units: [km]
[1] 5.5169493 6.0319452 0.0537802 0.5732917 1.5862168 7.0602682
```

- Agréger les segments de route par commune et calculer la longueur de route par commune.

```
# Calculer la longueur de route par commune
route_comm = int_route_comm %>%
  group_by(nom) %>%
  summarize(long_route = sum(long_km))
tableau = as.data.frame(st_drop_geometry(route_comm))
tableau

> tableau
                nom      long_route
1           Andenne 17.1395446 [km]
2             Anhée  8.1333088 [km]
3           Assesse 18.3135066 [km]
4         Beauraing 22.3487613 [km]
5            Bièvre  9.3183805 [km]
6       Cerfontaine  8.4833200 [km]
7             Ciney 31.0994018 [km]
8            Couvin 19.8001671 [km]
9            Dinant 25.7825854 [km]
10           Doische  2.7229284 [km]
11           Eghezée 19.8347455 [km]
```

☞ Réaliser l'intersection entre l'objet **comm_nam**, contenant les communes de la province de Namur et l'objet **clc_valid** contenant la couche Corine Land Cover (§ 3.14). Produire un tableau à 2 entrées « commune x CODE01 » (CODE01 : classes d'occupation du sol de niveau 1) pour calculer la répartition des surfaces par commune et par classe d'occupation.

```
# 3.15.1 Intersection de 2 couches (lignes et polygones)
# Intersection des communes de la province de Namur et de la couche CLC
# Produire un tableau de surface commune x classe CLC niveau 01 (CODE_01)
# CODE01 : 1=urbain, 2=agri, 3=foret, 4=z. humides, 5=eau

# Intersection de 2 couches de polygones
comm_nam_clc=st_intersection(clc_valid,comm_nam)
comm_nam_clc$surf=st_area(comm_nam_clc)
names(comm_nam_clc)

# Tableau de synthèse communes x code01
comm_nam_clc01= comm_nam_clc %>%
              group_by(nom,CODE_01) %>%
              summarize(surf=sum(surf))
names(comm_nam_clc01)
tableau=as.data.frame(st_drop_geometry(comm_nam_clc01))
tableau
```

```
> tableau
                nom CODE_01              surf
1          Andenne       1  23340284.6 [m^2]
2          Andenne       2  46573163.6 [m^2]
3          Andenne       3  14494306.7 [m^2]
4          Andenne       5   1729671.1 [m^2]
5            Anhée       1   8600668.8 [m^2]
6            Anhée       2  34462829.0 [m^2]
7            Anhée       3  22593723.2 [m^2]
8            Anhée       5    327851.1 [m^2]
9          Assesse       1   7251159.1 [m^2]
```

3.16 Buffers

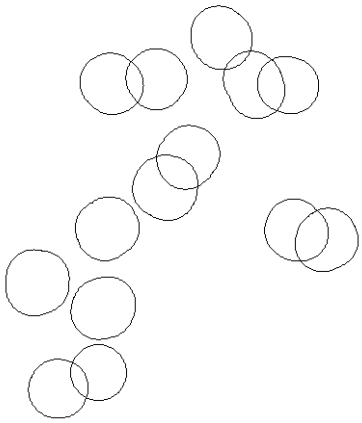
- La création de buffers autour des éléments d'un objet sf s'opère avec la fonction **st_buffer()**.

☞ Construire des buffers de 5 km autour des mâts éoliens décrits dans la couche **eoliennes.shp**. Regrouper ensuite ces buffers par parc éolien (champ [parc_id]).

```
# Construire un buffer de 5 km autour de
# chaque éolienne de la couche eoliennes.shp

eol=st_read("eoliennes.shp")
eol_buff=st_buffer(eol,dist=5000)
plot(eol_buff[5])
```

```
# Fusionner les buffers par parcs éoliens (champ [id_parc])
parc_buff = eol_buff %>%
  group_by(id_parc) %>%
  summarize(nb_mat = n())
plot(parc_buff[3])
```

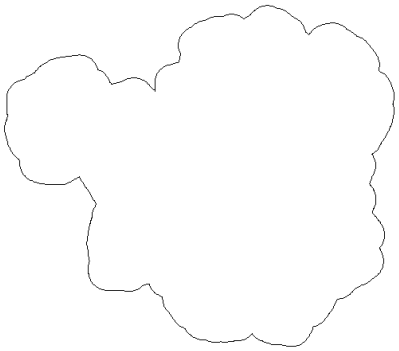
eol_buff



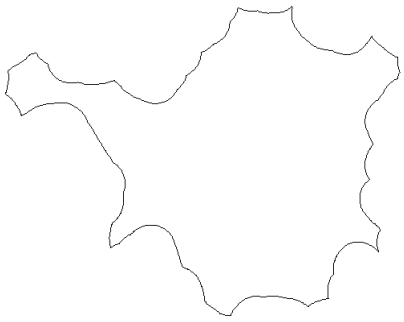
parc_buff

☞ Construire un buffer de 1 km autour de la ville de Namur. Construire un autre bufffer de 1 km à l'intérieur de la ville de Namur.

```
# Buffer de 5 km autour de la ville de Namur
namur=filter(comm2,comm2$nom=="Namur")
nam_buff_1km=st_buffer(namur,dist=1000)
plot(nam_buff_1km[1])

nam_buff_1km_inside=st_buffer(namur,dist=-1000)
plot(nam_buff_1km_inside["geometry"])
```



nam_buff_1km



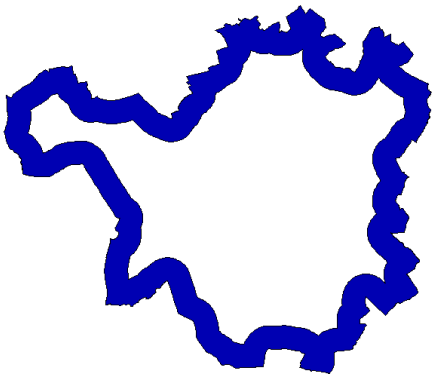
nam_buff_1km_inside

3.17 Difference

- La fonction ***st_difference()*** supprime les parties de géométries contenues dans 1 objet sf qui sont recouvertes par les géométries d'un autre objet sf.

☞ Construire un anneau délimitant la partie du territoire de la ville de Namur située à moins de 1 km de la limite extérieure de la commune.

```
# 3.17 Difference
nam_anneau=st_difference(namur,nam_buff_1km_inside)
plot(nam_anneau[1])
```



3.18 Calcul de la distance au plus proche voisin

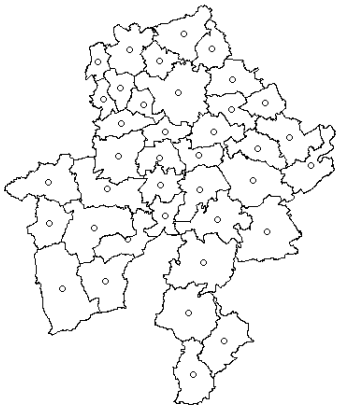
```
#*****************************************************************
# 3.18. Recherche du plus proche voisin et calcul de distance
#*****************************************************************
eol = st_read("eoliennes.shp")

power_line = st_read("power_lines.shp")

# rechercher la ligne électr. la + proche de chaque éolienne
eol2=st_join(eol, power_line, join = st_nearest_feature)
head(eol2)

# pour chaque éolienne, calcul de la distance à cete ligne électr.
for(i in 1:nrow(eol2)){
  eol2$d2pow[i]= min(st_distance(eol2[i,], power_line))
}

# vérifier le résultat dans QGIS
write_sf(eol2,"dist_eol_powerline.shp")

eol$d2power = st_distance(eol,power_line)
eol$nn = power_line$full_id[st_nearest_feature(eol,power_line)]
head(eol2)

st_write(eol,"eol2.shp",delete_layer=TRUE)
```

3.19 Conversion entre types géométriques

3.19.1 Création de centroïdes pour 1 couche de lignes ou de polygones

- La fonction ***st_centroid()*** génère les centroïdes des éléments (lignes ou polygones) contenus dans 1 objet sf.
- La fonction ***st_point_on_surface()*** réalise la même opération, mais garanti en outre que chaque centroïde est situé à l'intérieur de l'élément qu'il représente.

```
# 3.19.1. Création de centroïdes
comm_nam_centro=st_centroid(comm_nam)
plot(st_geometry(comm_nam))
plot(st_geometry(comm_nam_centro),add=TRUE)

# pour imposer la présence du centroïdes à l'intérieur de l'objet
comm_centro2=st_point_on_surface(comm)
plot(comm_centro2[1])
```

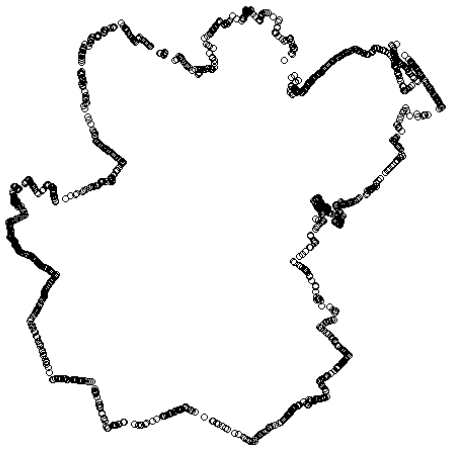


3.19.2 Convertir des polygones en lignes ou en points

La fonction cast() est utilisée pour convertir des objets d'un type de géométrie vers un autre type : MULTIPOLYGON → POLYGON, POLYGON → LINESTRING, POLYGON → POINT.

```
#3.19.2. Convertir des polygones en lignes

# convertir les polygones de la couche comm en lignes
# la fonction réalise 1 conversion 1 polygone -> 1 ligne !!!!
# la couche comm étant constituée de MULTIPOLYGONS,
# il faut au préalable transformer ceux-ci en POLYGONS simples

class(comm$geometry)

> class(comm$geometry)
[1] "sfc_MULTIPOLYGON" "sfc"
```

```
comm2=st_cast(comm,"POLYGON")
comm3=st_cast(comm2,"LINESTRING")

# vérifier dans QGIS
# la frontière séparant 2 communes est "dédoublée"
st_write(comm3,"comm_line.shp")


# Convertir des lignes/polygones -> points

gembloux=filter(comm3,comm3$ADMUNAFR=="Gembloux")
gembloux_pnt=st_cast(gembloux,"POINT")
plot(st_geometry(gembloux_pnt))
```

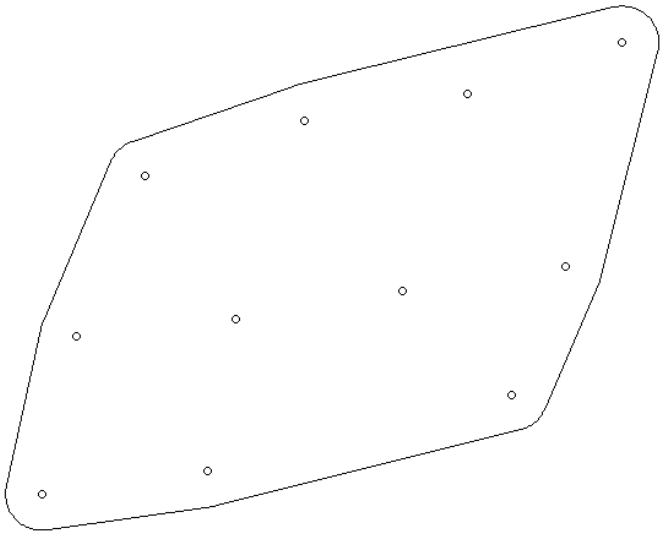


3.20 Enveloppes convexes

☞ Utiliser les données de la couche **eoliennes.shp** pour générer des polygones de type « enveloppe convexe » pour représenter les limites des différents parcs éoliens (identifiés par le champ [id_parc]). Les mâts doivent se trouver à une distance >= 100 m de la limite du parc qui les contient.

```
#*****************************************************************
# 3.20 Enveloppes convexes
#*****************************************************************
# Délimiter les parcs éoliens à l'aide d'enveloppes convexes
# Les mâts devant se trouver à une distance >= 100 m
# de la limite du parc

# step 1 : regrouper les mâts par parc (géométries multipoints)
eol_parc = eol %>%
  group_by(id_parc) %>%
  summarize(nb = n())
head(eol_parc)
```

```
# step 2 : faire une boucle sur les parcs et générer l'enveloppe convexe
#          a chaque itération ajouter le polygone produit à l'objet sf "final"
#          celui-ci est créé lors de la première itération
for (i in 1:nrow(eol_parc)){
  id_parc=eol_parc$id_parc[i]
  buff=st_buffer(eol_parc[i,],dist=100)
  hull=st_convex_hull(buff)
  if(i==1){
    parc_hull=hull
  }else{
    parc_hull=rbind(parc_hull,hull)
  }
}


# Afficher le résultat pour le parc n°7
plot(st_geometry(parc_hull[parc_hull$id_parc==7,]))
plot(st_geometry(eol[eol$id_parc==7,]),add=TRUE)
```



3.21 Conversions dataframe → sf et sf → dataframe

☞ Utiliser les données de la couche **eoliennes.shp** pour générer des polygones de type « enveloppe convexe » pour représenter les limites des différents parcs éoliens (identifiés par le champ [id_parc]). Les mâts doivent se trouver à une distance >= 100 m de la limite du parc qui les contient.

3.21.1 Convertir un dataframe avec des données « xy » en une couche de points

- Le premier exemple illustre la conversion d'un dataframe contenant des colonnes avec des données « x » et « y » en une couche de points stockée dans 1 objet de type sf. Cette conversion s'opère avec la fonction ***st_as_sf()***. Il convient de définir les colonnes qui contiennent les coordonnées, de même que le système de coordonnées dans lequel elles sont définies. On utilise pour cela la valeur de code EPSG du système de coordonnées.

```
# 3.21.1 Conversion df -> sf
# Convertir le fichier localite_wallonie.csv en 1 couche de points

loc_wall=read.csv("localites_wallonie.csv",sep=";",stringsAsFactors = F)
names(loc_wall)

loc_wall=st_as_sf(loc_wall,coords=c("X","Y"),crs=31370)
plot(loc_wall$geometry)
names(loc_wall)
plot(loc_wall$geometry)
```

3.21.2 Convertir une couche de points en dataframe

- Dans certains cas, on souhaite abandonner le champ contenant les géométries des objets. Cette suppression est réalisée avec la fonction ***st_drop_geometry()***. Il convient de noter que la transformation du dataframe en objet sp a pour conséquence que les colonnes « x » et « y » disparaissent du dataframe pour se retrouver dans la colonne « geometry ». Si l'on veut conserver les coordonnées « x » et « y », il convient de les réintégrer dans le dataframe avant l'abandon de la colonne « geometry ».

```
# 3.21.2 Conversion sf -> df

# Supprimer la colonne de géométrie contenue dans l'objet loc_wall
df_loc_wall=st_drop_geometry(loc_wall)
names(df_loc_wall)

# les coordonnées ont disparu

loc_wall$X=st_coordinates(loc_wall)[,1]
loc_wall$Y=st_coordinates(loc_wall)[,2]
df_loc_wall=st_drop_geometry(loc_wall)
names(df_loc_wall)
```

3.22 Création de géométries sur mesure

- Pour certaines applications, il est utile de pouvoir générer des objets géométriques « sur mesure ». Dans ce cas, il convient de repartir des objets géométriques de base de la classe **sfg** (single feature geometry) qui sont ensuite transformés en objets de classe **sfc** (simple feature column) qui se différencient des précédents par le fait que l'objet possède un attribut définissant le système de coordonnées. Une troisième étape permet de convertir la classe **sfc** en classe **sf** en ajoutant une table d'attributs sous la forme d'un dataframe.

sfg (objet géométrique simple) → **sfc** (objet géométrique simple + CRS) →

sf (objet géométrique simple + CRS + attributs)

- Généralement cette séquence est réalisée objet par objet au sein d'une boucle, et les éléments **sf** produits en bout de chaîne sont « empilés » à l'aide de la commande *rbind()*.

☞ Générer 50 quadrats de 1km² répartis aléatoirement au sein de la Wallonie.

- Pour résoudre ce problème, la première étape consiste à générer les centres des quadrats à l'aide de la fonction ***st_sample()*** avec l'option type = « random ». Cette fonction génère des objets de la classe **sfc** au sein du territoire wallon (défini par la couche **comm**). Ces éléments sont ensuite transformés en éléments de type **sf** avec la fonction ***st_as_sf()***.

```
# 3.22.1. générer 50 quadrats de 1 km²  répartis au sein de la Wallonie
comm=st_read("communes.shp",stringsAsFactors = F)

# step 1 : générer les centres des quadrats
centers=st_sample(comm, size=50, type = "random", exact = TRUE)
class(centers)
centers=st_as_sf(centers)
names(centers)
names(centers)[1]="geometry"
st_geometry(centers)="geometry"
class(centers)
centers$id=seq.int(nrow(centers))

st_write(centers,"centers.shp")
```

- L'étape suivante prend la forme d'une boucle dans laquelle les quadrats sont générés les uns après les autres en reprenant la séquence présentée en introduction : un objet **sfg** est généré. Il s'agit d'une ligne représentant le périmètre du quadrat. Elle est créée avec la fonction ***st_linestring()***. L'objet **sfg** est ensuite transformé en objet **sfc** avec la fonction ***st_sfc()*** dans laquelle est défini le système de coordonnées. Finalement, la table d'attribut (dataframe) est adjointe via la fonction ***st_sf()***.

- Les différents quadrats ainsi produits sont rassemblés dans 1 seule objet de type **sf** avec la fonction ***rbind()***.

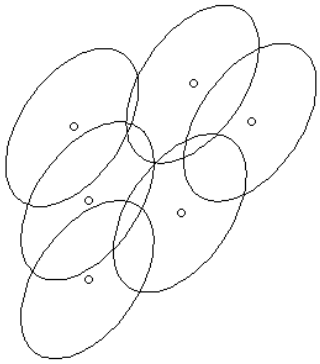
```
for (i in 1:nrow(centers)){
  # centre du quadrat
  x0=st_coordinates(centers)[i,1]
  y0=st_coordinates(centers)[i,2]
  # création d'un objet sfg (1 géomtrie)
  sfg = st_linestring(rbind(c(x0-500, y0-500), c(x0-500, y0+500),c(x0+500, y0+500),
                            c(x0+500, y0-500),c(x0-500, y0-500)))
  # création d'un objet sfc (1 géomtrie + 1 CRS)
  sfc=st_sfc(sfg,crs=31370)
  # création d'un df
  df <- data.frame(id = i)
  # création d'un objet sf (sfc + df)
  sf=st_sf(cbind(sfc, df))
  # empiler les quadrats dans l'objet quadrat
  quadrat=rbind(quadrat,sf)
}
```

- La dernière étape vise à convertir les éléments de type ligne en polygones avec la fonction ***st_cast()***.

```
# convertir la géométrie des objets (lignes -> polygones)
class(quadrat$geometry)
quadrat=st_cast(quadrat,"POLYGON")
plot(quadrat$geometry)
class(quadrat$geometry)
```

☞ Définir l'emprise théorique d'un parc éolien (situé à Ernage) dont la position des mâts est définie par la couche **eol_ernage.shp**.

- Le modèle théorique pour définir l'emprise d'une éolienne est celui d'une ellipse dont les demi-axes correspondent respectivement à 7 fois et 4 fois le diamètre du rotor de l'éolienne. Le grand axe de l'ellipse est orienté dans la direction des vents dominants. Ce type de représentation est utilisé pour planifier la disposition des éoliennes d'un même parc afin d'optimiser la production. Idéalement les éoliennes doivent se trouver en dehors des ellipses des autres éoliennes.



```
library(DescTools) # utilisée pour appliquer une rotation aux ellipses

# Lecture des localisations des éoliennes
eol=st_read("eoliennes_ernage.shp")

# paramètres de base
d_rotor = 80 # diamètre du rotor en m
a=7*d_rotor
b=4*d_rotor
azimut=235 # vents dominants
```

- Les éoliennes sont traitées au sein d'une boucle dans laquelle sont réalisées successivement les opérations suivantes :
  - Création d'une ellipse sans orientation
  - Orientation des sommets de l'ellipse selon l'orientation des vents. Le résultat se présente sous la forme d'une liste avec les coordonnées x et y transformées.
  - Transformation de la liste en dataframe puis en objet de type **sf**. Contenant les points situés sur le périmètre de l'ellipse
  - Transformation des points en polygone avec les fonctions ***summarize()*** et ***st_cast()***.
  - Empilement des polygones dans la variable finale avec la fonction ***rbind()***.

```r
# boucle sur les éoliennes
for (i in 1:nrow(eol)){

  # générer une ellipse centrée sur l'éolienne
  ell0=st_ellipse(eol[i,],a,b,res=100)
  plot(ell0)
  class(ell0)

  # orienter l'ellipse par rapport aux vents dominants
  ell0=DescTools::Rotate(st_coordinates(ell0),
                         theta=azimut/180*pi)
  class(ell0) # les données se présentent sous le forme d'une liste
  names(ell0)
  # convertir la liste -> df
  df=as.data.frame(ell0$x)
  df
  names(df)="x"
  df$y=ell0$y
  # convertir df -> sf
  ell0=st_as_sf(df,coords=c("x","y"),crs=3812)
  plot(ell0$geometry)
  ell0$id=1

  # regrouper les points pour en faire 1 polygone
  pol = ell0 %>%
    group_by(id) %>%
    summarize(do_union = FALSE) %>%
    st_cast("POLYGON")

  # empiler les résultats dans 1 sf
  emprise_eol=rbind(emprise_eol,pol)
}
# afficher le résultat final
plot(emprise_eol$geometry)
plot(eol$geometry,add=T)
```

4. Introduction au package raster

4.1 Introduction

- Les données raster sont généralement composées d'un en-tête (header) et d'une matrice (i lignes x j colonnes) de pixels de forme rectangulaire. L'en-tête contient les informations suivantes :
  - le système de coordonnées
  - les coordonnées de l'origine du raster (coin supérieur gauche)
  - la dimension des pixels dans les 2 directions
  - le nombre de lignes et de colonnes

4.2 Lecture et écriture de données raster

4.2.1 Lecture et écriture d'objets raster simples

- Avant toute chose, il convient de définir le répertoire de travail avec la fonction ***setwd().*** Celle-ci-doit être adaptée au répertoire dans lequel sont rangées les données de l'exercice.
- La lecture d'une couche raster s'effectue avec la fonction ***raster()***.

```
# 4.2.1. Lecture et écriture d'une couche raster simple
mnt250=raster("mnt_250m.tif")
mnt250
```

```
> mnt250
class       : RasterLayer
dimensions  : 1014, 1232, 1249248  (nrow, ncol, ncell)
resolution  : 250, 250  (x, y)
extent      : 4000, 312000, 6500, 260000  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=lcc +lat_1=51.16666723333333 +lat_2=49.8333339 +lat_0=90 +
lon_0=4.367486666666666 +x_0=150000.013 +y_0=5400088.438 +ellps=intl +towgs84=
-106.8686,52.2978,-103.7239,0.3366,-0.457,1.8422,-1.2747 +units=m +no_defs
data source : C:/tmp/R_GIS_02/data/mnt_250m.tif
names       : mnt_250m
values      : -197.413, 702.726  (min, max)
```

- La sauvegarde d'une couche raster s'effectue avec la fonction ***writeRaster()***.

```
writeRaster(mnt,"mnt_copie.tif",overwrite=TRUE)
```

- **Remarque** : l'option « overwrite = TRUE » permet d'écraser des fichiers préexistants.
- L'option « datatype » est utilisée lorsque l'on veut préciser le type de codage pour la couche de sortie. Le tableau suivant présente les différents types de données à considérer avec cette option.

Data type	Minimum value	Maximum value
LOG1S	FALSE (0)	TRUE (1)
INT1S	-127	127
INT1U	0	255
INT2S	-32,767	32,767
INT2U	0	65,534
INT4S	-2,147,483,647	2,147,483,647
INT4U	0	4,294,967,296
FLT4S	-3.4e+38	3.4e+38
FLT8S	-1.7e+308	1.7e+308

```
# écriture d'un raster en forçant le type de données
writeRaster(mnt250,"mnt_copie2.tif",overwrite=TRUE, datatype="INT2S")

# utiliser la fonction gdalinfo pour afficher les propriétés du fichier
library(gdalUtils)
gdalinfo("mnt_copie2.tif",verbose=TRUE)
```

```
"Center      (  158000.000,  133250.000) (  4d28'48.97\"E, 50d
"Band 1 Block=1232x3 Type=Int16, ColorInterp=Gray"
"  Min=-197.000 Max=
```

4.2.2 Lecture et écriture d'objets raster multi-bandes (RasterStack et RasterBrick)

- Les objets **RasterStack** et **RasterBrick** permettent de stocker des données raster multibandes.
- Un **RasterStack** peut être assimilé à une liste d'objets raster, ceux-ci pouvant provenir de fichiers différents. Par contre un **RasterBrick** doit être construit au départ d'un seul fichier contenant les différentes couches.

```
# 4.2.2. Lire plusieurs fichers raster-> RasterStack
# créer 1 stack avec les données worldclim de t°moyenne mensuelles

# step 1 : créer la liste des noms de fichiers contenant les
# données de précipitation (tavg_01.tif, tavg_02.tif, ...)
path_worldclim = paste0(getwd(),"/worldclim")
list_file=list.files(path_worldclim,pattern="*.tif$",full.names=TRUE)
list_file=list_file[grep("tavg", list_file)]
list_file

> list_file
 [1] "C:/tmp/R_GIS_02/data/worldclim/tavg_01.tif"
 [2] "C:/tmp/R_GIS_02/data/worldclim/tavg_02.tif"
 [3] "C:/tmp/R_GIS_02/data/worldclim/tavg_03.tif"
 [4] "C:/tmp/R_GIS_02/data/worldclim/tavg_04.tif"
 [5] "C:/tmp/R_GIS_02/data/worldclim/tavg_05.tif"
 [6] "C:/tmp/R_GIS_02/data/worldclim/tavg_06.tif"
 [7] "C:/tmp/R_GIS_02/data/worldclim/tavg_07.tif"
 [8] "C:/tmp/R_GIS_02/data/worldclim/tavg_08.tif"
 [9] "C:/tmp/R_GIS_02/data/worldclim/tavg_09.tif"
[10] "C:/tmp/R_GIS_02/data/worldclim/tavg_10.tif"
[11] "C:/tmp/R_GIS_02/data/worldclim/tavg_11.tif"
[12] "C:/tmp/R_GIS_02/data/worldclim/tavg_12.tif"

tmoy <- stack(list_file)
tmoy
```

```
> tmoy
class       : RasterStack
dimensions  : 245, 470, 115150, 12  (nrow, ncol, ncell, nlayers)
resolution  : 0.008333333, 0.008333333  (x, y)
extent      : 2.516667, 6.433333, 49.48333, 51.525  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
names       : tavg_01, tavg_02, tavg_03, tavg_04, tavg_05, tavg_06, tavg_07, t
avg_08, tavg_09, tavg_10, tavg_11, tavg_12
min values  :    -0.8,    -0.7,     2.0,     4.9,     9.6,    12.2,    14.3,
  14.4,    11.1,     7.1,     2.5,     0.2
max values  :     4.3,     4.3,     7.0,     9.8,    14.2,    17.0,    19.1,
  18.8,    16.1,    12.3,     8.7,     5.8
```

- La sauvegarde d'un **RasterStack** dans un fichier multi-bandes utilise la même commande ***writeRaster()***.

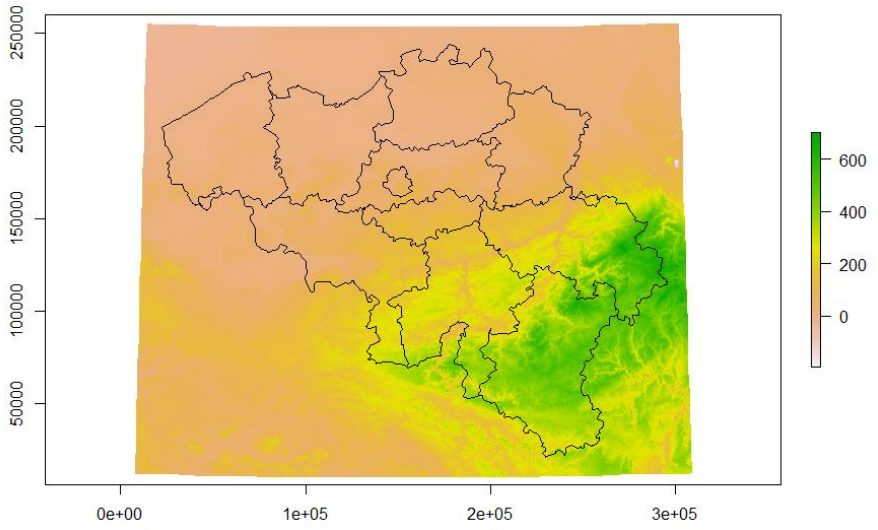
```
# Sauvegarde d'un RasterStack dans 1 fichier multibandes
writeRaster(tmoy,"tmoy.tif",overwrite=TRUE)
```

- Le chargement d'un fichier multi-bandes dans un RasterBrick se fait avec la commande ***brick()***.

```
# Chargement d'un raster multibandes dans un RasterBrick
tmoy2=brick("tmoy.tif")
```

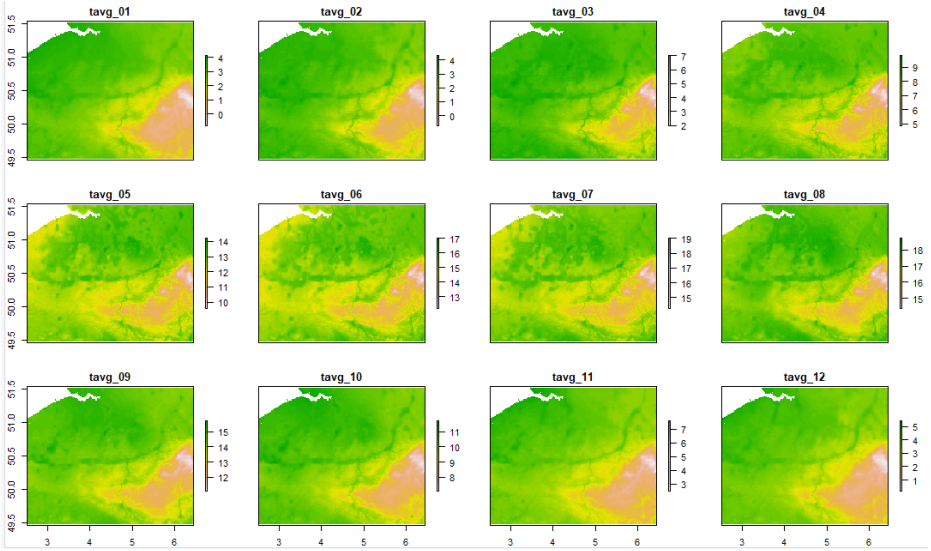
4.3 Affichage de données raster (*plot( )*)

```
#*************************************************************
# 4.3. Affichage d'un raster + superposition d'un objet sf
#*************************************************************

plot(mnt250)

# superposer une couche vectorielle
prov=st_read("nuts02_BE.shp")
prov_l72=st_transform(prov,31370)
plot(st_geometry(prov_l72),add=TRUE)
```



- L'affichage d'un RasterStack avec la fonction ***plot()*** affiche toutes les bandes du RasterStack.

```
plot(tmoy)
```



4.4 Modification d'objets raster

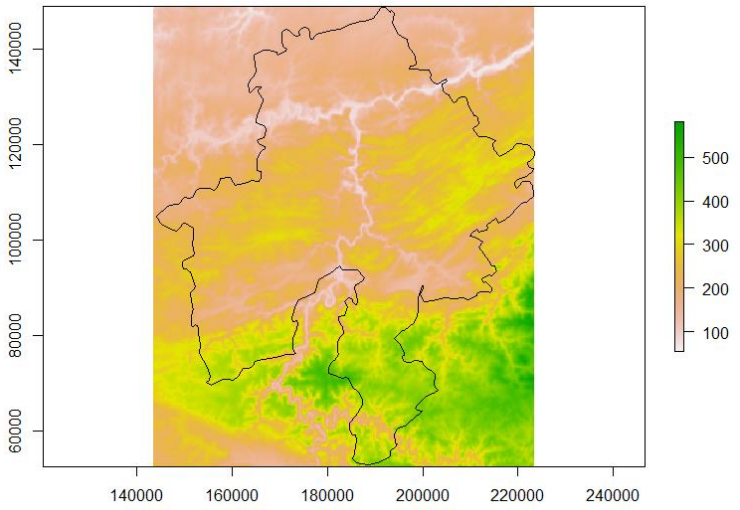
4.4.1 Rognage (crop( )) avec définition d'une emprise (extent( ))

- La modification d'objets raster nécessite généralement de définir une nouvelle emprise (Extent) caractérisée par des coordonnées xmin, xmax, ymin et ymax. Cette emprise peut être directement récupérée depuis n'importe quel objet géographique vectoriel ou raster. Il convient cependant d'être attentif au fait que les dimensions de l'emprise (xmax-xmin et ymax-ymin) soient compatibles avec la taille des pixels.

```
# 4.4.1 Rogner un raster (crop)
# Découper le raster mnt250 avec l'emprise de la province de Namur

# step1. déterminer l'emprise du nouveau raster
prov_nam=filter(prov_l72,prov_l72$NUTS_ID=="BE35")
ext=extent(prov_nam)
ext

# step2. Il faut faire en sorte que l'emprise soit compatbile
# avec la résolution du raster (ici 250 m)
res0=250
ext[1]=as.integer(ext[1]/res0)*res0-res0
ext[2]=as.integer(ext[2]/res0)*res0+res0
ext[3]=as.integer(ext[3]/res0)*res0-res0
ext[4]=as.integer(ext[4]/res0)*res0+res0
ext

> ext
class       : Extent
xmin        : 143750
xmax        : 223500
ymin        : 52500
ymax        : 149000
```

- Le rognage (fonction ***crop()***) consiste à extraire une partie d'un raster aux limites d'une emprise géographique.

```
# step3 : rognage du raster
mnt250_nam=crop(mnt250,ext)
mnt250_nam
plot(st_geometry(prov_nam),add=TRUE)
```



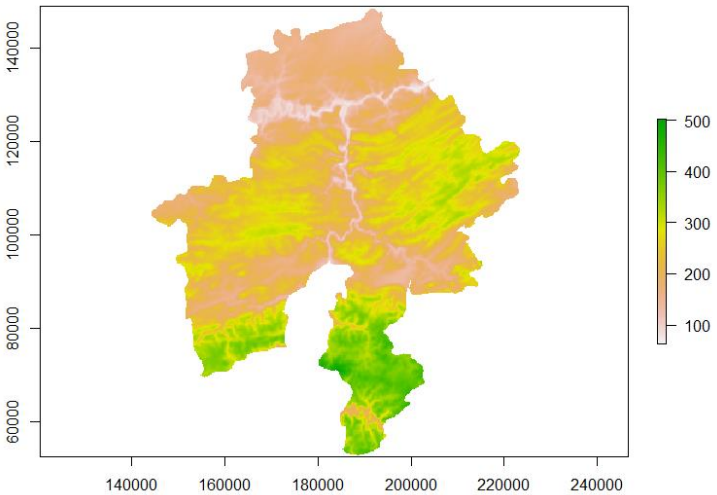
- Remarque : la définition d'une emprise peut se faire directement avec 1 vecteur de 4 coordonnées (xmin, xmax, ymin, ymax)

```
# remarque : définir un extent directement avec 1 liste de valeurs numériques
ext = extent(143750, 223500, 52500, 149000)
```

4.4.2 Masquage (mask( ))

- Le masquage (fonction ***mask()***) consiste à remplacer les valeurs situées en dehors de la couche de masquage par des nodata.

```
# 4.4.2 Masquer un raster (mask)
# Masquer le raster mnt250_nam avec le masque de la province de Namur
mnt250_nam2=mask(mnt250_nam,prov_nam)
plot(mnt250_nam2)
```



4.4.3 Reprojection et rééchantillonnage d'un raster (projectRaster( ))

☞ Reprojeter et rééchantilloner le RasterStack **tmoy** pour que son système de coordonnées et sa géométrie (orgine, nombre de lignes, nombre de colonnes et résolution) correspondent à ceux du rasterlayer **mnt250**.

```
# 4.4.3. Reprojeter et rééchantillonner un raster
#        pour le faire correspondre
#        à la géométrie d'un autre raster

tmoy250=projectRaster(tmoy, mnt250, method="bilinear")
```

- L'affichage des propriétés des 2 objets (mnt250 et tmoy250) permet de constater que leurs caractéristiques géométriques sont identiques. Ces 2 séries de couches raster pourront être utilisées simultanément dans des géotraitements.

```
> mnt250
class       : RasterLayer
dimensions  : 1014, 1232, 1249248  (nrow, ncol, ncell)
resolution  : 250, 250  (x, y)
extent      : 4000, 312000, 6500, 260000  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=lcc +lat_1=51.16666723333333 +lat_2=49.8333339 +lat_0=90 +
lon_0=4.367486666666666 +x_0=150000.013 +y_0=5400088.438 +ellps=intl +towgs84=
-106.8686,52.2978,-103.7239,0.3366,-0.457,1.8422,-1.2747 +units=m +no_defs
data source : C:/tmp/R_GIS_02/data/mnt_250m.tif
names       : mnt_250m
values      : -197.413, 702.726  (min, max)
```

```
> tmoy250
class       : RasterBrick
dimensions  : 1014, 1232, 1249248, 12  (nrow, ncol, ncell, nlayers)
resolution  : 250, 250  (x, y)
extent      : 4000, 312000, 6500, 260000  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=lcc +lat_1=51.16666723333333 +lat_2=49.8333339 +lat_0=90 +
lon_0=4.367486666666666 +x_0=150000.013 +y_0=5400088.438 +ellps=intl +towgs84=
-106.8686,52.2978,-103.7239,0.3366,-0.457,1.8422,-1.2747 +units=m +no_defs
data source : C:/Users/lejeune.p/AppData/Local/Temp/RtmpAD2GB2/raster/r_tmp_20
19-09-23_142231_13672_49386.grd
names       :    tavg_01,    tavg_02,    tavg_03,    tavg_04,    tavg_05,    t
avg_06,    tavg_07,    tavg_08,    tavg_09,    tavg_10,    tavg_11,    tavg_12

min values  : -0.8162772, -0.6810343,  2.0000000,  4.9000001,  9.6000004, 12.2
180455, 14.3052484, 14.4181594, 11.1181600,  7.1105650,  2.5000000,  0.2000000

max values  :   4.300000,   4.300000,   7.000000,   9.799646,  14.200000,  17.
000000,  19.091111,  18.778261,  16.100000,  12.300000,   8.700000,   5.800000
```

4.4.4 Rééchantillonner un raster sans reprojection (resample())

- La fonction ***resample()*** peut également être utilisée lorsque l'on vise uniquement à rééchantillonner un raster pour le rendre compatible avec un autre raster défini dans le même SCR.

```
# 4.4.4. Rééchantillonner un raster (même SCR)

mnt1km=mnt250
res(mnt1km)=1000
# après avoir modifié la résolution, le raster ne contient plus de données
summary(mnt1km)
# rééchantillonner mnt250 -> mnt1km
mnt1km=resample(mnt250, mnt1km, method="bilinear")
mnt1km
```

4.4.5 Fonction aggregate et disaggregate : rééchantilloner ou suréchantilloner un raster

- La fonction ***aggregate()*** est utilisée pour modifier la résolution d'un facteur entier. Ainsi un facteur 2 permet de convertir un raster ayant des pixels de 5 m en un raster avec des pixels de 10 m. Une fonction est définie (par exemple fun=mean) pour préciser les modalités d'aggrégation.
- A l'inverse, la fonction disaggregate() suréchantillonne un raster d'entrée d'un facteur entier. Un facteur 2 convertira un raster ayant des pixels de 10 m en un raster avec des pixels de 5 m.

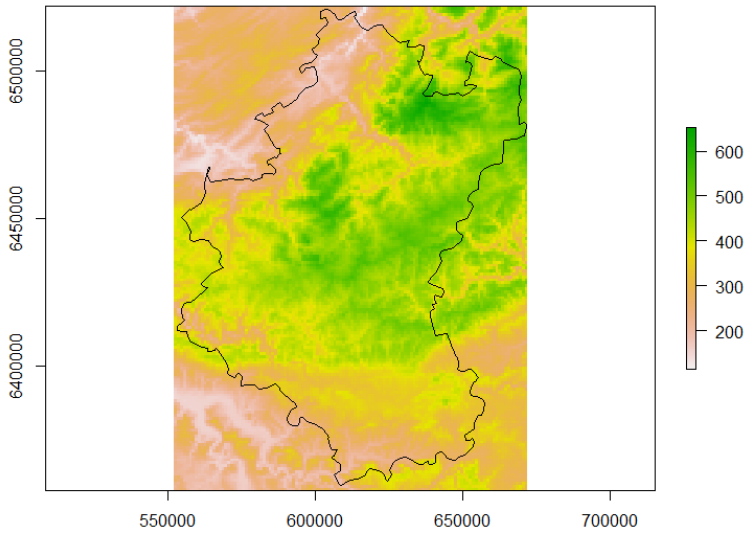
```
# 4.4.5 Aggregate et Disaggregate

mnt500=aggregate(mnt250,fact=2,fun=mean,expand=TRUE)
mnt500

mnt250_v2=disaggregate(mnt500,fact=2)
mnt250_v2
```

4.4.6 Création d'un raster « template »

- Lorsque l'on souhaite mettre en place une analyse spatiale à l'aide de données raster, il est primordial de s'assurer de la cohérence spatiale des différentes couches utilisées dans l'analyse. Dans cette optique, le plus simple est de préparer un raster de référence qui respecte les critères géométriques voulus : système de coordonnées, emprise, résolution.

☞ Créer de toute pièce un raster répondant aux spécifications géométriques suivantes : utiliser le système de coordonnées Pseudo-Meractor (EPSG : 3035), couvrir complètement la province de Luxembourg, avoir une résolution de 1 km. Utiliser ensuite ce raster de référence pour générer 1 couche raster décrivant l'altitude.

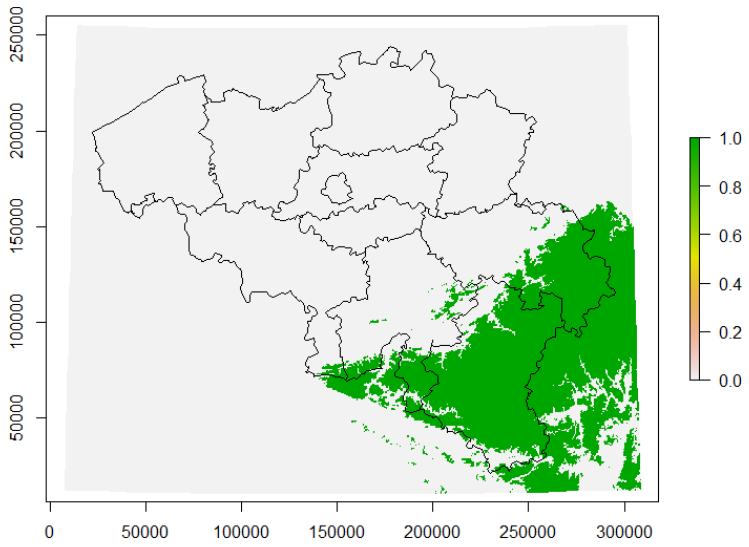
```
# 4.4.6. Créer un raster de toute pièce
# sur base de paramètres géométriques prédéfinis :
# - couvrir la province de Luxembourg
# - CRS : 3837 (pseudo-mercator)
# - taille des pixels : 1 km

# step 1. générer 1 objet raster
r=raster()

# step 2. définir l'emprise (x y multiples de 1000)
prov_3857=st_transform(prov_172,3857)
prov_lux=filter(prov_3857,prov_3857$num_nuts02==19)
plot(st_geometry(prov_lux),add=T)
ext=extent(prov_lux)
ext
res0=1000
ext[1]=as.integer(ext[1]/res0)*res0-res0
ext[2]=as.integer(ext[2]/res0)*res0+res0
ext[3]=as.integer(ext[3]/res0)*res0-res0
ext[4]=as.integer(ext[4]/res0)*res0+res0
ext
extent(r)=ext

# step 3. définir le CRS du raster
proj_3857=st_crs(prov_lux)
proj_3857
projection(r)=proj_3857$proj4string

# step 4. définir la résolution
res(r)=1000

# exemple d'utlisation du raster r pour
# créer un nouveau raster
alt_1km = projectRaster(mnt250, r, method="bilinear")
plot(alt_1km)
plot(st_geometry(prov_lux),add=TRUE)
```



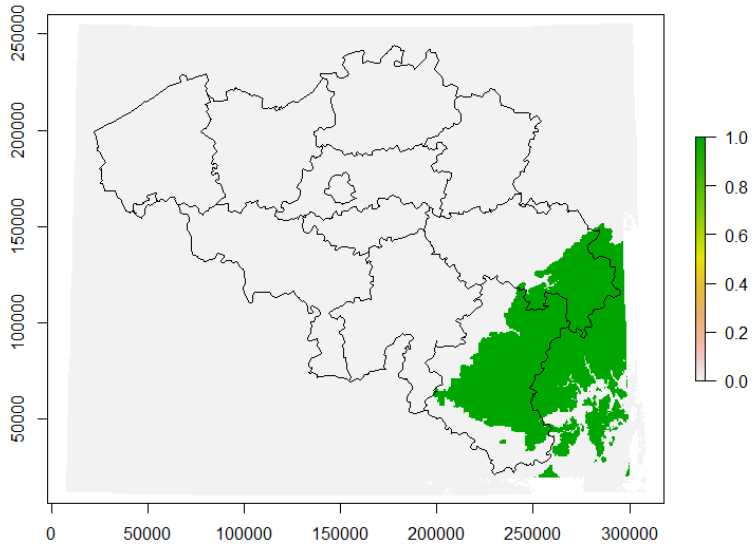
4.5 Algèbre « raster » (calculatrice raster)

4.5.1 Opérations algébriques

```
# 4.5.1 Opérations algébriques

# Requête simple
mnt_gt300 = mnt250>300
plot(mnt_gt300)
plot(st_geometry(prov_l72),add=TRUE)
```



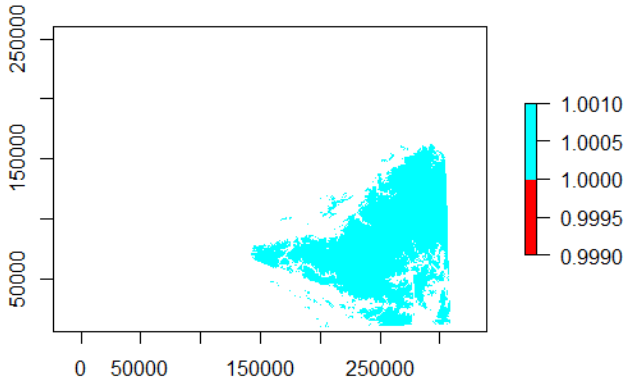

```
# Requête basée sur plusieur rasters
r = (tmoy250[[1]]<1) & mnt250>300
plot(r)
plot(st_geometry(prov_l72),add=TRUE)
```



4.5.2 Accès aux valeurs d'un raster : fonction values()

- La fonction ***values()*** permet d'accéder au contenu d'un raster sous la forme d'un vecteur de valeurs numériques. Elle constitue un moyen simple d'effectuer certaines opérations sur les données contenues dans un raster. L'accès à ces valeurs peut également s'envisager avec la notation r[ ] où r représente un objet raster.

```
# 4.5.2 Accès aux valeurs numériques d'un raster

m1=mnt_gt300
values(m1)[values(m1)== 0] <- NA
plot(m1,col=rainbow(2))

m1=mnt_gt300
m1[m1[]==0]=NA
plot(m1,col=rainbow(2))
```

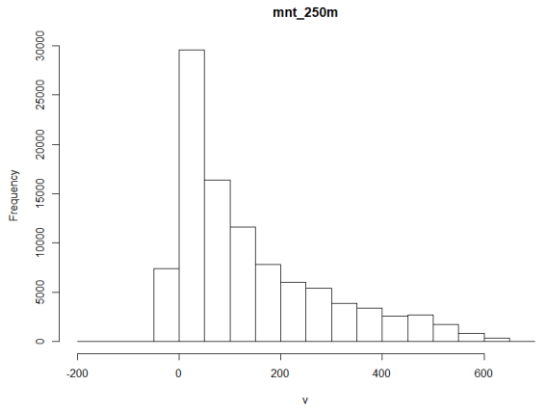


4.6 Statistiques de couches raster

4.6.1 Histogramme des valeurs d'un raster (hist( ))

- En première approche, la fonction *hist()* permet de produire un histogramme des valeurs prises par les pixels d'un raster.

```
# affichage de l'histogramme des valeurs du raster
hist(mnt250)
```



4.6.2 CellStats()

- La fonction ***cellStats()*** renvoie une valeur statistique (sum, mean, min, max, sd…) décrivant l'ensemble des pixels d'un raster.

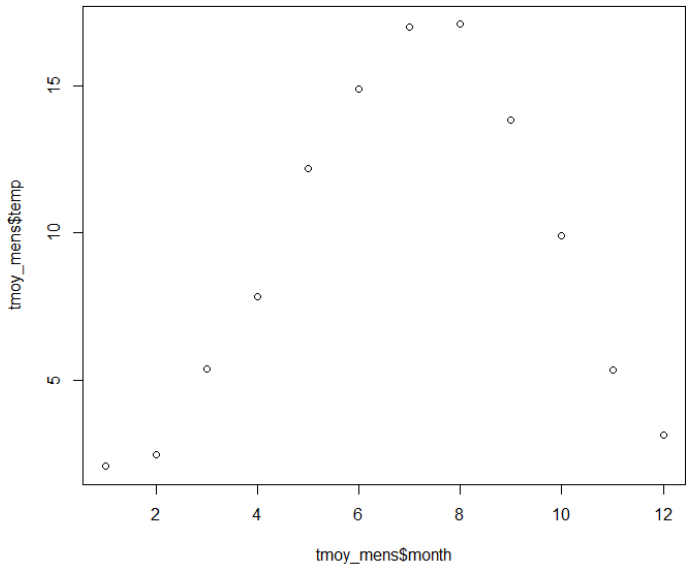
```
# 4.6.2 cellStats() appliquée à 1 raster simple
# -> renvoi 1 valeur numérique simple
cellStats(mnt250,stat=mean)
cellStats(mnt250,stat=max)

> cellStats(mnt250,stat=mean)
[1] 140.0499
> cellStats(mnt250,stat=max)
[1] 702.726
```

- Lorsque ***cellStats()*** est appliquée à un **RasterStack**, ou un **RasterBrick**, le résultat est un vecteur de valeurs numériques.

```
# 4.6.2 cellStats() appliquée à 1 stack
# -> renvoi 1 vecteur de valeurs numériques (1 par couche)
# températures moyennes mensuelles pour la province de Namur

# step 1. extraire les t° pour la province de Namur
tmoy_nam250=crop(tmoy250,prov_nam250)
tmoy_nam250=mask(tmoy_nam250,prov_nam250)
plot(tmoy_nam250[[1]])

# step 2. calculer les statistiques
tmoy_mens=cellStats(tmoy_nam250,stat=mean)
tmoy_mens
```

```
> tmoy_mens
             temp month
tavg_01  2.056582     1
tavg_02  2.442902     2
tavg_03  5.388252     3
tavg_04  7.850094     4
tavg_05 12.178450     5
tavg_06 14.886875     6
tavg_07 16.993726     7
tavg_08 17.100301     8
tavg_09 13.836481     9
tavg_10  9.901102    10
tavg_11  5.328314    11
tavg_12  3.133245    12
```

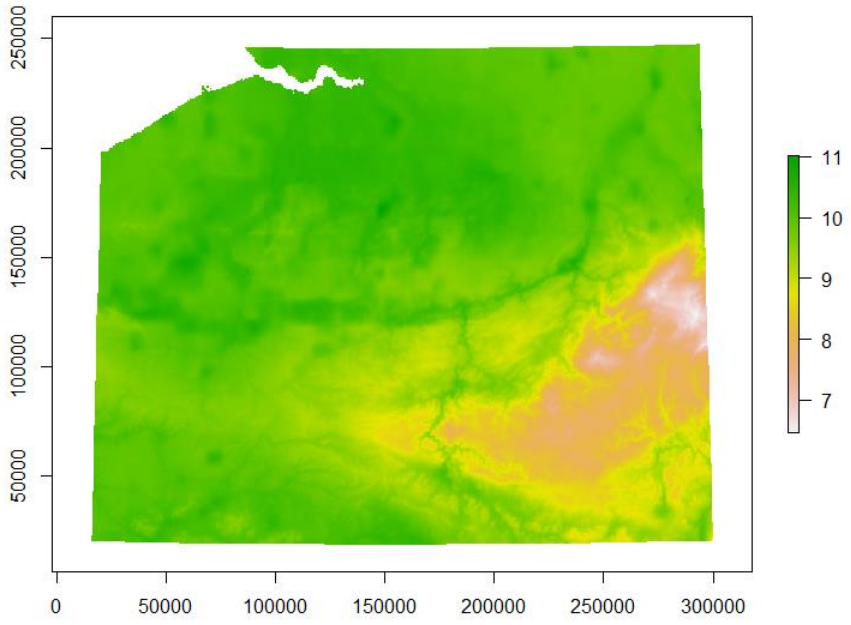
```
# step 3. convertir le vecteur en df pour créer 1 graphique
tmoy_mens=as.data.frame(tmoy_mens)
names(tmoy_mens)[1]="temp"
tmoy_mens$month=names(cellStats(tmoy_nam250,stat=mean))
tmoy_mens$month=as.numeric(substr(tmoy_mens$month,6,7))
plot(tmoy_mens$month,tmoy_mens$temp)
```



4.6.3 Calc() : calcul de paramètres statistiques à l'échelle pixel

- La fonction ***calc()*** permet de produire un raster exprimant, à l'échelle pixel, des paramètres statistiques relatifs à un empilement de rasters.

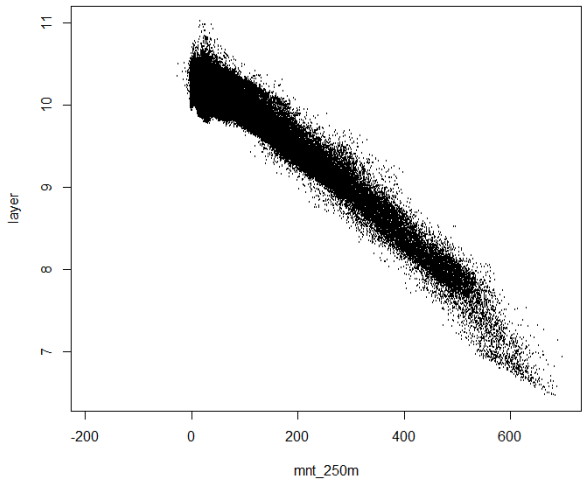
```
# 4.6.3  fonction calc() appliqué à 1 stack
# pour calculer des statistiques au niveau pixel
# -> renvoi 1 raster mono-bande
# exemple : calculer la température moyenne annuelle pour la Belgique

tmoy_ann <- calc(tmoy250, mean)
plot(tmoy_ann)
```



4.6.4 *Corrélation entre 2 rasters*

- La fonction ***cor()*** calcule la corrélation entre 2 rasters présentant les mêmes caractéristiques géométriques.

```
# 4.6.4 calculer une corrélation entre 2 rasters
plot(mnt250,tmoy_ann)
cor(values(mnt250),values(tmoy_ann),use = "na.or.complete")
```




```
> cor(values(mnt250),values(tmoy_ann),use = "na.or.complete")
[1] -0.9680481
```

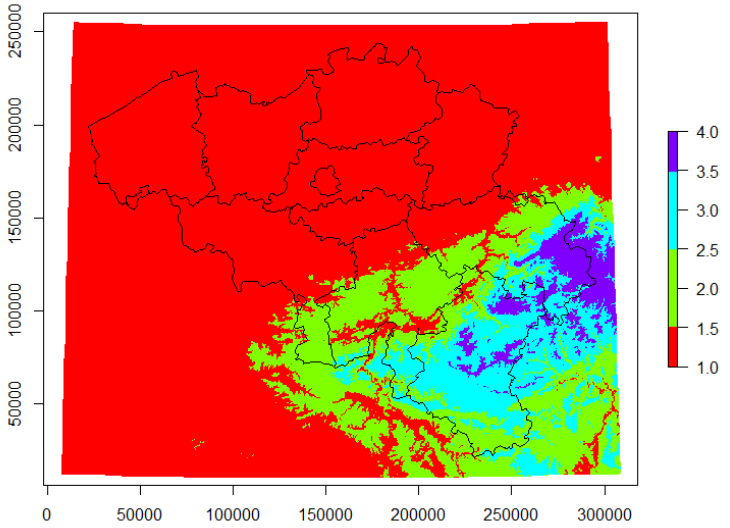
4.6.5 *Statistiques zonales*

- La fonction ***stat()*** produit des statistiques zonales calculées sur un raster par rapport à des zones définies dans un autre raster.

```
# 4.6.5. statistiques zonales
# la couche décrivant les zones doit être un raster

# calculer l'altitude moyenne pour chaque province
prov250=fasterize(prov_l72,mnt250,"OBJECTID")

alt_moy=zonal(mnt250, prov250, fun='mean', digits=0, na.rm=TRUE)
alt_moy

> alt_moy
      zone      mean
 [1,]   67  45.94317
 [2,]   68  17.09462
 [3,]   69  62.37957
 [4,]   71  15.92018
 [5,]   75  14.18369
 [6,]   77 311.09254
 [7,]   80 111.13730
 [8,]   82  59.54476
 [9,]   90 236.66536
[10,]   98 108.98672
[11,]  594 394.35123
```

4.7 Reclassification (*reclassify( )*)

```
#*****************************************************************
# 4.7. Reclassification
#*****************************************************************

# créer une carte avec 4 classes d'altitude
# 1 : <= 200, 2 : 200 < <= 350, 3 : 350 < <= 500, 4 : 500 <
classes=c(-Inf,200,1, 200,350,2, 350,500,3, 500,Inf,4)
cl_alt=reclassify(mnt250,classes)
library(colourvalues)
plot(cl_alt,col = rainbow(4))
plot(st_geometry(prov_l72),add=TRUE)
```

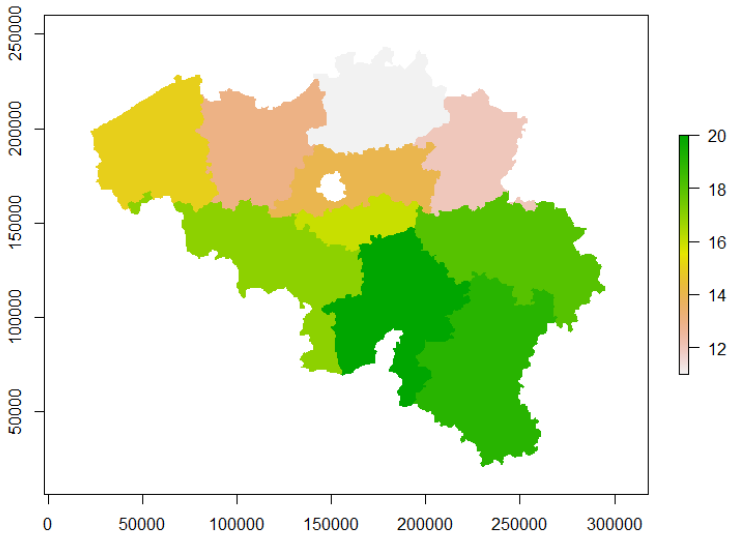


4.8 Convertir une couche vectorielle en couche raster (*rasterize( )*)

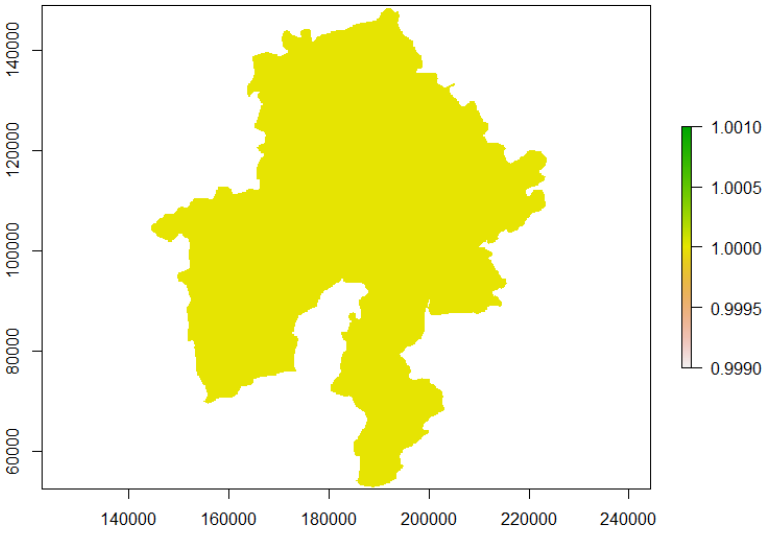
- La conversion d'une couche vectorielle en couche raster est réalisée avec la fonction ***rasterize()***. Elle nécessite de définir un raster de référence dont les caractéristiques géométriques seront utilisée pour créer le nouveau raster. Il convient également de définir les modalités de codage des pixels. Deux options sont proposées :

  - Utiliser les valeurs d'un attribut de la couche vectorielle, de préférence numérique ;
  - Utiliser une valeur unique, par exemple la valeur 1

```
#*****************************************************************
# 4.8. Convertir couche vectorielle -> raster
#*****************************************************************

# coder les pixesl avec 1 attribut numérique
prov250=rasterize(prov_l72,mnt250,"num_nuts02")
plot(prov250)
```




```
# coder les pixels avec 1 valeur unique
prov_nam250=rasterize(prov_nam,mnt250_nam,1)
prov_nam250
```



4.9 Distance euclidienne (*distance( )*)

- La création de couche de distance par rapport à une série d'objets nécessite au préalable la conversion de ceux-ci en mode raster. La fonction ***distance()*** calcule, pour chaque pixel contenant une valeur NA, la distance au pixel ayant une valeur différente de NA le plus proche.

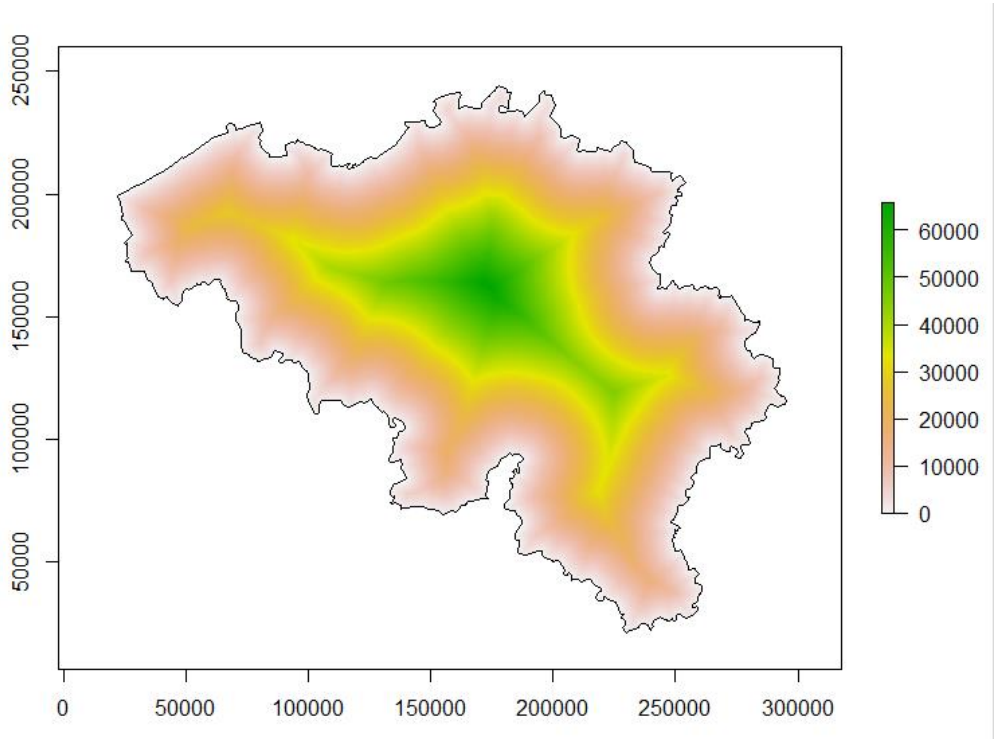
☞ Créer une couche raster exprimant la distance euclidienne par rapport à la frontière belge. Utiliser comme couche raster de référence (résolution, emprise) la couche **mnt250**. Appliquer au résultat final le masque du territoire belge (les distances sont uniquement définies à l'intérieur du territoire belge). La couche vectorielle de départ est **prov_l72** (§ 4.3) qui contient les provinces belges

---

Les différentes étapes à suivre peuvent se résumer comme suit :

- Créer une couche de polygones correspondant au territoire belge (fusionner les provinces).
- Convertir la couche de polygones en une couche de lignes.
- Convertir la couche de lignes en un raster.
- Construire la couche de distances.

```
#****************************************************************
# 4.9. Distance euclidienne
#****************************************************************

# créer une carte de distance par rapport à la frontière de la Belgique

# step 1. fusionner les provinces -> Belgique
bel=summarize(prov_l72)

# step 2. Convertir un sf "polygon" en sf "linestring"
bel_border=st_cast(bel,"LINESTRING
                   ")
# step 3. convertir sf -> raster
r_bel_border=rasterize(bel_border,mnt250,1)

# step 4. Construire la couche de distance
d2bord=raster::distance(r_bel_border)

d2bord=mask(d2bord,bel)
plot(d2bord)
plot(st_geometry(bel),add=TRUE)
```



4.10 Conversion de couches raster en couches vectorielles

- La fonction ***rasterToPoints()*** est utilisée pour convertir un raster en une couche de points. Cette fonction comporte une fonction (fun) de sélection des pixels qui sont convertis en points. Dans l'exemple présenté ci-dessous, seul le pixel présentant la valeur maximale est converti.

```
# 4.10.1 raster -> points

# Créer une couche vectorielle contenant le point situé en Belgique
# et qui est le plus éloigné de la frontière belge

# step 1 : distance maximale à la frontière
dmax=cellStats(d2bord,stat=max)

# step 2 : extraire les coordonnées du point le + éloigné
p <- rasterToPoints(d2bord, fun=function(x){x==dmax})

# step 3 : convertir les coordonnées en objet sf
point <- st_as_sf(as.data.frame(p), coords = c("x", "y"), crs = 31370)
plot(st_geometry(point),add=TRUE)
```

- La fonction ***rasterToPolygons()*** fonctionne sur le même principe mais transforme le raster en polygones. L'option « dissolve=T » rassemble les polygones d'une même valeur en 1 seul polygone.

```
# 4.10.2 raster -> polygons

# Créer une couche vectorielle contenant la partie du territoire située
# à moins de 20 km de la frontière

# step 1. Sélectionner les pixel situés à moins de 20 km de la frontière
bord20km= d2bord < 20000
plot(bord20km)
bord20km[bord20km[]==0]=NA

# step 2. Convertir le raster en polygones
pol=rasterToPolygons(bord20km,dissolve=T,n=4,na.rm=TRUE)
```

4.11 Extraire de l'information d'une couche raster (*extract()*)

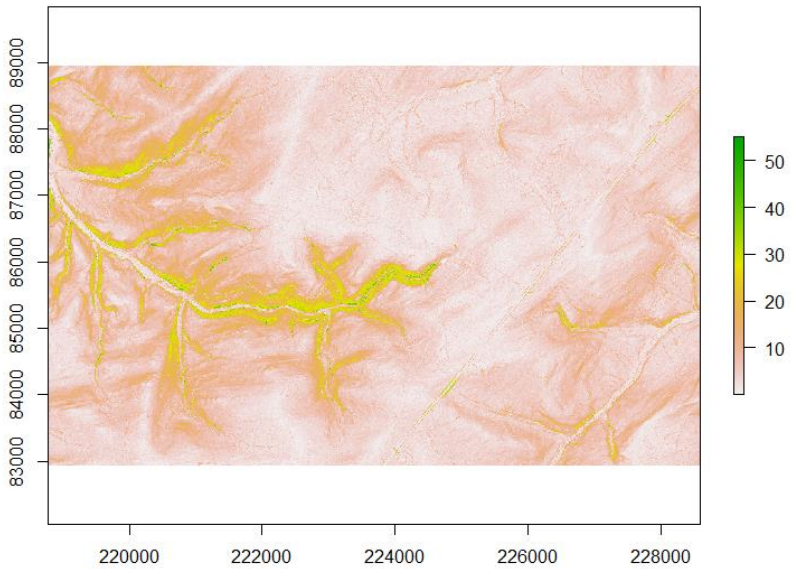
- La fonction ***extract()*** est utilisée pour extraire de l'information d'une couche raster pour des objets contenus dans une couche vectorielle (objet sf)
- Dans les 2 exemples qui suivent, la fonction ***extract()*** est utilisée, respectivement pour extraire l'altitude des points de la couche **eol** (localisation d'éoliennes) et pour calculer l'altitude moyenne des polygones de la couche **comm_bw** correspondant aux communes de la province du Brabant wallon.

```
#************************************************************
# 4.11. Extraire des valeurs d'une couche raster
#************************************************************

# 4.11.1 Extraire des valeurs ponctuelles

eol=st_read("eoliennes.shp")
mnt=raster("mnt_250m.tif")
eol$alt=raster::extract(mnt,eol)

# 4.11.2 Extraire des valeurs moyennes pour des polygones
#        (équivalent à zonal, mais sur des données vecrtorielles)
comm_bw=st_read("communes.shp",query =
                  "select * from communes where ADPRKEY='25'")
comm_bw$alt=extract(mnt,comm_bw,fun=mean,df=TRUE)
head(comm_bw)
```

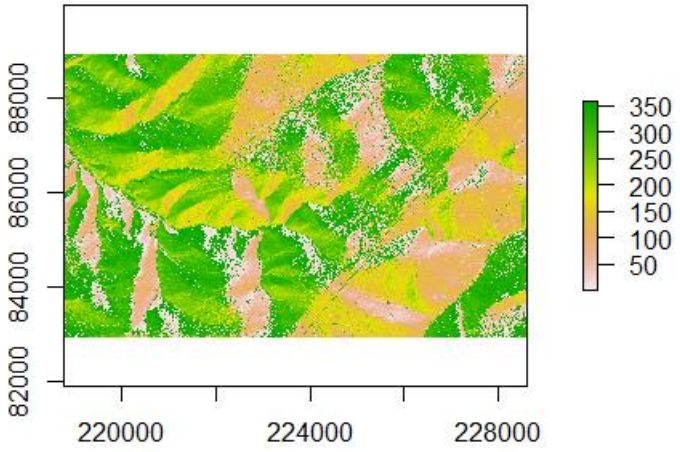
4.12 Produits dérivés d'un MNT (pente, exposition, hillshade)

- La fonction ***terrain()*** est dédiée à la production de couches rasters dérivées d'un MNT. Elle permet notamment de générer une couche de pente (opt= « slope »), ou d'exposition (opt=« aspect »).
- L'option ***« unit »*** conditionne les unités dans lesquelles sont calculées les variables. Les modalités disponibles sont « degrees », « radians » ou « tangent ». Cette dernière génère des pentes exprimées en pourcents.
- L'option « neighbors » (valeur 4 ou 8) désigne l'algorithme.

```
#************************************************************
# 4.12. Couches dérivées d'un MNT
#************************************************************
mnt=raster("mnt_2m.tif")

# 4.12.1 Pente
pente=terrain(mnt,opt="slope",unit="degrees",neighbors=8)
plot(pente)
```



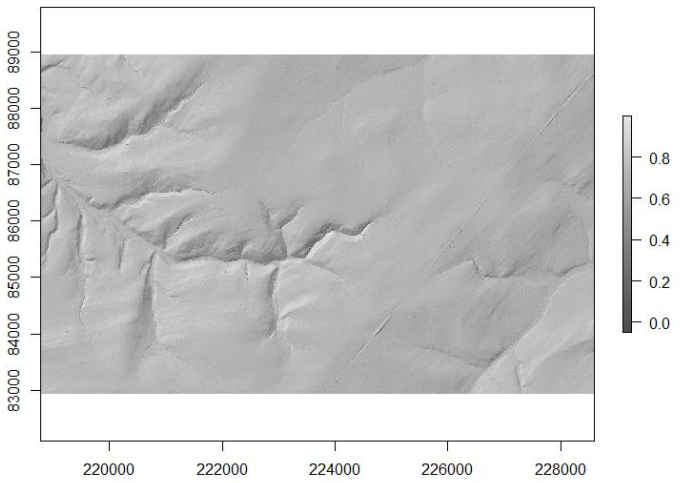
```
# 4.12.2 Exposition
expo=terrain(mnt,opt='aspect',unit="degrees",neighbors=8)
plot(expo)
```



- La fonction ***hillshade()*** génère une couche ombrée au départ des couches de pente et d'exposition. Les paramètres à définir concernent l'angle d'élévation du soleil et la direction d'illumination. Les valeurs classiquement utilisées sont 45° pour l'élévation et 315° pour la direction d'illumination.

  Remarque : les couches de pente et d'exposition doivent être produites en « radians »

```
# 4.12.3 Hillshade
pente_rad=terrain(mnt,opt='slope',unit="radians",neighbors=8)
expo_rad=terrain(mnt,opt='aspect',unit="radians",neighbors=8)
hillsh=hillShade(pente_rad, expo_rad,angle=45,direction =315)
plot(hillsh,col = gray.colors(255,gamma = 0.5))
```



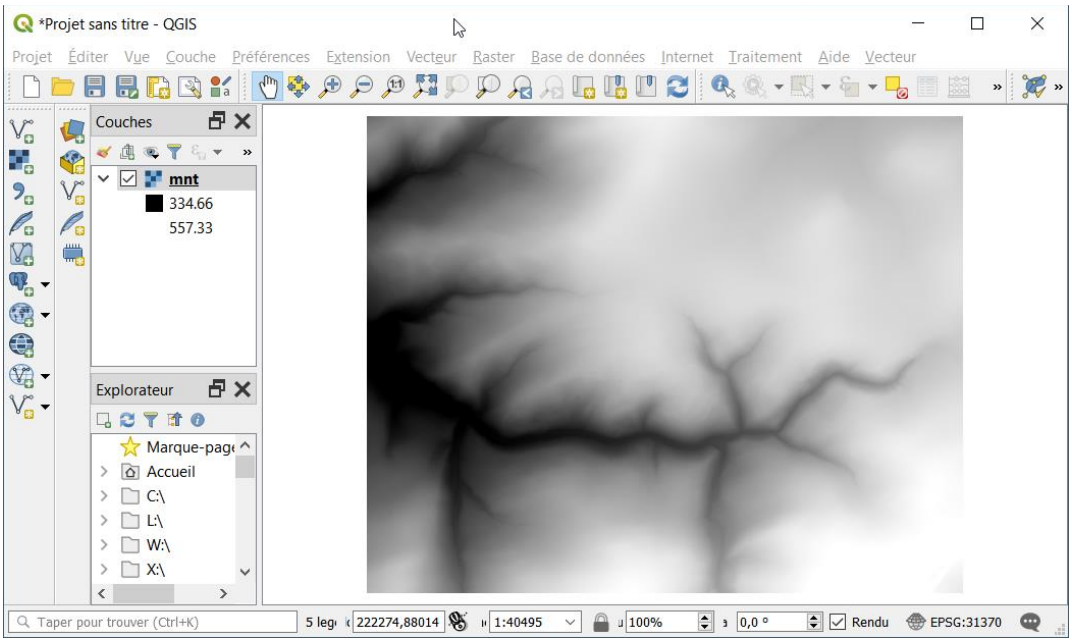
4.13 Création de rasters virtuels (vrt)

- La fonction ***gdalbuildvrt()*** issue du package ***gdalUtils*** est utilisée pour construire des rasters virtuels. Les rasters virtuels sont des outils très intéressants pour la gestion de données raster, que ce soit pour la création de **mosaïques** (assemblage de rasters contigus couvrant une zone donnée) ou la création d'**empilements** (stack) de plusieurs couches rasters ayant les mêmes caractéristiques géométriques (nombre de lignes, nombre de colonnes, taille de pixel).

4.13.1 Mosaïquage de rasters mono-bandes (MNT)

- Les 16 fichiers .tif présents dans le répertoire **\tuile_mnt** correspondent à un MNT découpé en tuiles de 1 km x 1 km.
- Dans l'exemple qui suit, ces 16 tuiles sont assemblées en 1 raster virtuel permettant de gérer les 16 tuiles sous la forme d'un seul raster.
- La construction de ce raster virtuel s'opère en 2 étapes : il convient d'abord de générer une liste des fichiers à assembler avant de créer le fichier **.vrt**.

```
path0="C:/tmp/R_GIS_01/data"
library(gdalUtils)

# Mosaïquage de rasters mono-bandes (exemple = MNT)

path_mnt=paste0(path0,"/tuiles_mnt")
# step 1 : créer 1 liste des fichiers à assembler
list_file=list.files(path_mnt, pattern="*.tif$",full.names = TRUE)
nb_tuile=length(list_file)

# step 2 : générer le raster virtuel
file_vrt=paste0(path0,"/mnt.vrt")
gdalbuildvrt(gdalfile=list_file, output.vrt= file_vrt,overwrite=TRUE)
```

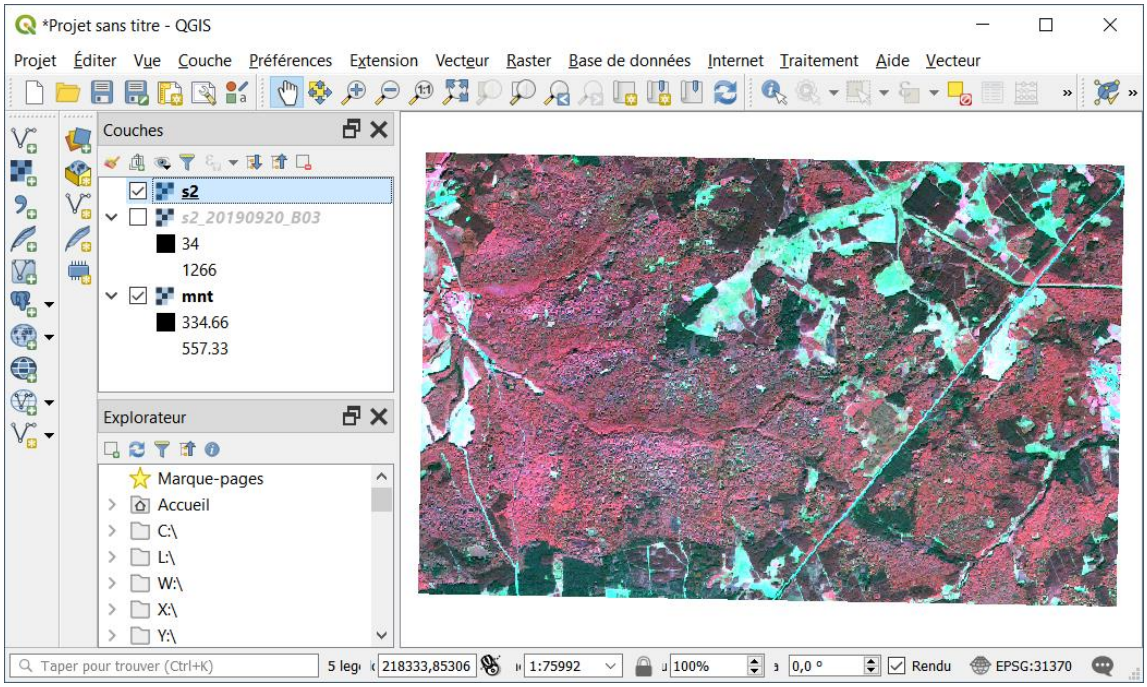


4.13.2 Empilement de rasters mono bandes pour générer un raster multi bandes

- Dans l'exemple qui suit, les bandes B03, B04 et B08 d'une image Sentinel 2 sont assemblées pour former un raster multispectral utilisable pour l'affichage d'une composition colorée dans QGIS. Les fichiers utilisés dans cet exemple sont extraits de l'image S2A_MSIL2A_20190920T104021_N0213_R008_T31UFR_20190920T120330.SAFE téléchargée sur le site https://scihub.copernicus.eu/).

```
# Empilement de rasters mono-bandes -> raster multi-bandes

path_s2=paste0(path0,"/sentinel2")
# step 1 : créer 1 liste des fichiers à assembler
list_file=list.files(path_s2, pattern="*.tif$",full.names = TRUE)

# step 2 : générer le raster virtuel : !! SEPARATE = TRUE
file_vrt=paste0(path0,"/s2.vrt")
gdalbuildvrt(gdalfile=list_file, output.vrt= file_vrt,
             overwrite=TRUE,separate=TRUE)
```

- **Remarque importante** : l'empilement des bandes est lié à l'option « separate = TRUE » dans la fonction ***gdalbuildvrt()***.

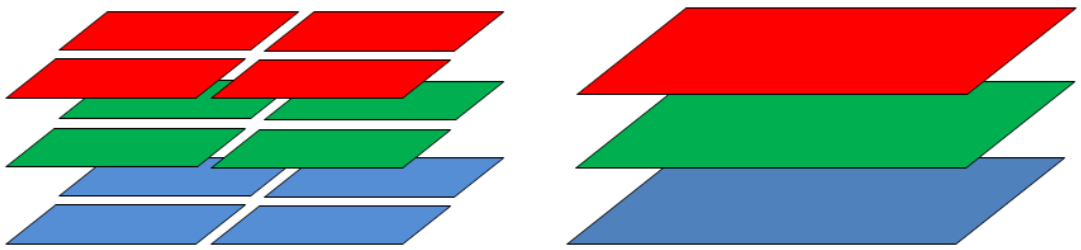


4.13.3 Empilement de mosaïques monobandes extraites de mosaïques multi-bandes

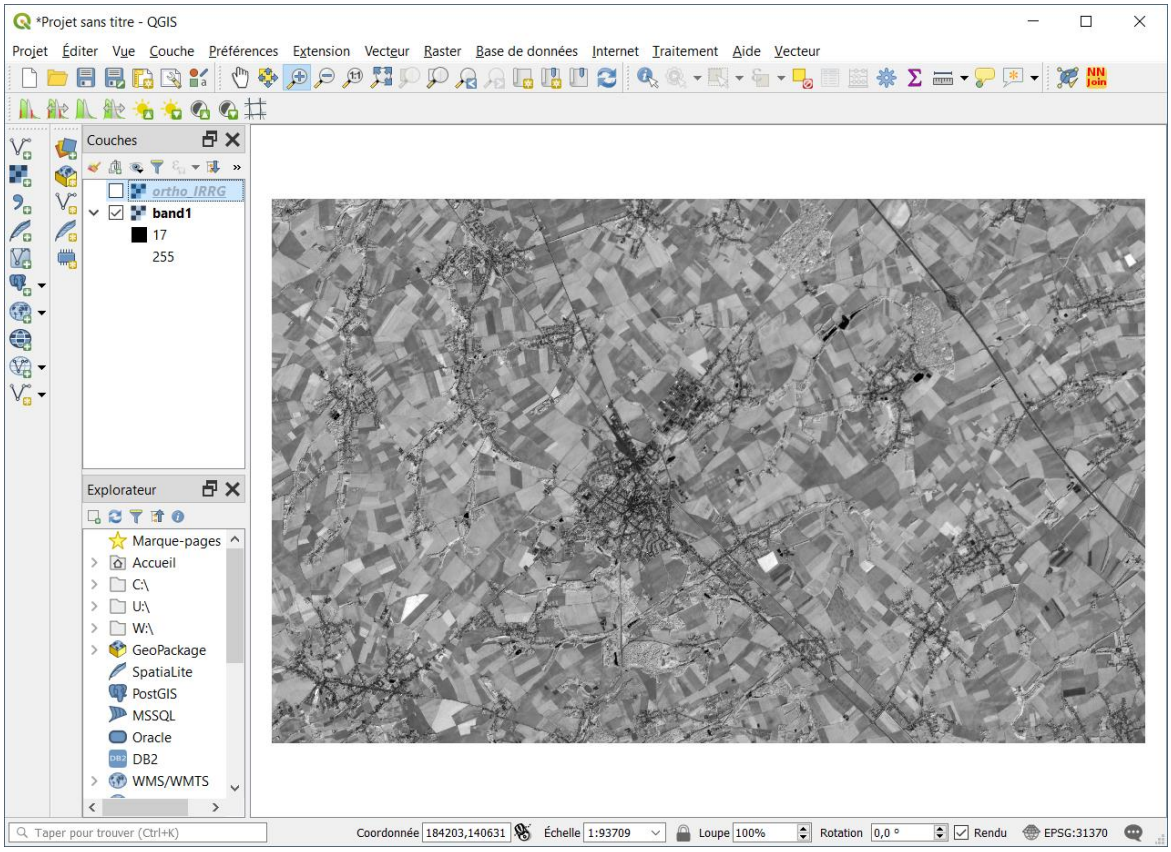
- Dans l'exemple qui suit, 1 couche ortho-images couvrant la région de Namur-Gembloux est constituée de 8 images multispectrales (bande1 : NIR, bande 2 : Green, bande 3 : Red). Les données se trouvent dans le répertoire **\tuiles_ortho**.
- Cette collection peut être visualisée dans son ensemble à l'aide du fichier tuiles_ortho.vrt.
- L'objectif visé est de générer 3 mosaïques mono-spectrales (1 pour chaque bande) et ensuite d'assembler ces 3 mosaïques dans 1 vrt multi-spectral.
- Cette transformation permet de disposer de fichiers « mono-bandes » couvrant l'ensemble de la zone d'étude, alors que dans les données de départ, chaque bande est distribuée au sein de 8 fichiers.
- Le traitement s'effectue en 2 étapes. On procède d'abord au mosaïquage des différentes bandes pour produire 3 fichiers vrt. Ces 3 fichiers sont ensuite empilés pour permettre l'affichage d'une composition colorée ou de réaliser une analyse nécessitant cette mise en forme.

```
# Exemple 3 : conversion d'une mosaïque d'images multi-bandes -> empilment d'images monobandes

# On procède en 2 étapes : on crée des mosaïques de chacunes des bandes (.vrt monobandes)
# et ensuite on empile ces vrt en 1 vrt multi-bandes

path_ortho=paste0(path0,"/tuiles_ortho")

# step 1 : mosaïquer les bandes IR,R,G des différentes images dans 3 fichiers .vrt
list_vrt=list() # liste qui va contenir les noms des 3 fichiers .vrt
list_file=list.files(path_ortho, pattern="*.tif$",full.names = TRUE)
list_file
for (i in 1:3){
  file_vrt=paste0(path_ortho,"/band",i,".vrt")
  gdalbuildvrt(gdalfile=list_file, b=i, output.vrt= file_vrt,
               overwrite=TRUE,separate=FALSE)
  list_vrt=rbind(list_vrt,file_vrt)
}
list_vrt=list_vrt[,1] # crée une liste de fichiers

# step 2 : empiler les 3 vrt pour créer un vrt multispectral
file_vrt=paste0(path_ortho,"/ortho_IRRG.vrt")
gdalbuildvrt(gdalfile=list_vrt, output.vrt= file_vrt,
             overwrite=TRUE,separate=TRUE)
```

- La figure ci-dessous illustre le passage de 4 images à 3 bandes vers 3 images à 1 bande.



- La figure ci-dessous correspond à la mosaïque de la bande 1 (NIR) sur l'ensemble de la zone d'intérêt.


*Projet sans titre - QGIS
Projet Éditer Vue Couche Préférences Extension Vecteur Raster Base de données Internet Traitement Aide Vecteur
Couches
ortho_IRRG
band1
17
255
Explorateur
Marque-pages
Accueil
C:\
U:\
W:\
GeoPackage
SpatiaLite
PostGIS
MSSQL
Oracle
DB2
WMS/WMTS
Taper pour trouver (Ctrl+K)
Coordonnée 184203,140631
Échelle 1:93709
Loupe 100%
Rotation 0,0 °
Rendu
EPSG:31370

5. Résumé des fonctions R liées aux géotraitements

Fonctions	Descriptif	Réf
st_read()	Lecture d'une couche vectorielle dans 1 objet **sf**	3.2
st_write()	Sauvegarde d'un objet **sf** dans une couche vectorielle	3.2
shapefile()	Lecture d'une couche vectorielle dans 1 objet **sp**	3.3
as(, « Spatial »)	Conversion d'un objet **sf** en un objet **sp**	3.5
st_crs()	Affichage du CRS d'un objet **sf**	3.6.1
st_crs() =	Modification du CRS d'un objet **sf**	3.6.2
st_transform()	Reprojection d'un objet **sf** (changement de CRS)	3.6.3
st_bbox()	Affichage de la bounding box d'un objet **sf**	3.6.4
extent()	Emprise d'un objet **sf**	3.6.4
read.csv()	Lecture d'un fichier csv (création d'un dataframe)	3.7.1
write.table()	Sauvegarde d'un dataframe dans 1 fichier .csv ou .txt	3.7.1
read.xlsx()	Lecture d'un fichier Excel (création d'un dataframe)	3.7.2
write.xlsx()	Lecture d'un fichier Excel (création d'un dataframe)	3.7.2
select()	Sélectionner et/ou renommer des champs dans un dataframe (création d'un nouveau dataframe)	3.8
rename()	Renommer 1 champ dans un dataframe	3.8
setnames()	Renommer tous les champs d'un dataframe	3.8
left_join	Jointure de tables	3.9
select()	Sélection par attributs	3.10.2
select()	Sélection par localisation	3.10.3
st_intersects()	Relation d'intersection dans 1 sélection par localisation	3.10.3
st_disjoint()	Relation d'absence d'intersection (disjointure) dans 1 sélection par localisation	3.10.3
st_is_within_distance()	Relation de proximité dans 1 sélection par localisation	3.10.3
st_area()	Calcul de la surface des éléments d'un objet **sf**	3.11.1
set_units()	Gestion des unités pour les attributs d'un objet **sf**	3.11.1
st_coordinates()	Copier les coordonnées dans le dataframe d'un objet **sf**	3.11.2
summarize()	Création de tableaux de synthèse	3.12
	Fusion d'éléments dans un objet **sf**	3.12
rbind()	Fusion de plusieurs objets **sf**	3.13
st_is_valid()	Vérification de la validité topologique d'un objet **sf**	3.14
st_make_valid()	Correction des problèmes topologiques d'un objet **sf**	3.14
st_intersection()	Intersection de 2 objets **sf**	3.15
st_buffer()	Création de buffers autour des éléments d'un objet **sf**	3.16

st_difference()	Différence géométrique entre 2 objets **sf**.	3.17
st_join()	Jointure spatiale	3.18
st_nearest feature()	Opérateur « + proche voisin » d'une jointure spatiale	3.18
st_distance()	Calcul de distance entre les éléments de 2 objets **sf**	3.18
st_centroid()	Création des centroïdes pour les éléments d'un objet **sf**	3.19.1
st_point_on_surface()	Force les centroïdes à se trouver à l'intérieur des géométries	3.19.1
st_cast()	Convertir un **sf** de polygones → **sf** de lignes ou un **sf** de polygones/lignes → **sf** de points	3.19.2
st_convex_hull()	Création d'enveloppes convexes	3.20
st_as_sf()	Conversion d'un dataframe vers 1 objet sf en considérant les colonnes contenant des coordonnées x et y	3.21.1
st_coordinates()	Extrait les coordonnées d'un objet **sf**	3.21.2
st_drop_geometry()	Converti un objet sf en dataframe en supprimant sa géométrie	3.21.2
st_sample()	Création de points répartis aléatoirement au sein d'une zone de référence	3.22
st_linestring()	Création d'objets géométriques de base (sfg) de type linéaire	
sf_sfc()	Transformation d'objets sfg en objets sfc	
st_sf	Transformation d'objets sfc en objets sf	
st_cast()	Transformation du type de géométrie pour des objets sf	
st_ellipse()	Création d'ellipses	
Rotate()	Application d'une rotation à 1 objet géométrique	
raster()	Lecture d'un fichier raster → création d'un objet raster simple (**RasterLayer**)	4.2.1
writeRaster()	Sauvegarde d'un objet raster dans un fichier .tif	4.2.1
stack()	Lecture de plusieurs fichiers rasters → création d'un objet raster multicouche (**RasterStack**)	4.2.2
brick()	Lecture d'un fichier rasters multibande → création d'un objet raster multicouche (**RasterBrick**)	4.2.3
extent()	Emprise spatiale d'un raster	4.4.1
crop()	Rognage d'un objet raster	4.4.1
mask	Application d'un masque à un objet raster	4.4.2
projectRaster()	Reprojecter et rééchantillonner un objet raster	4.4.3
resample()	Rééchantillonner sans changer le CRS	4.4.4
aggregate()	Réduire la résolution d'un facteur fixe	4.4.5
disaggregate()	Augmenter la résolution d'un facteur fixe	4.4.5
raster()	Création d'un raster « template »	4.4.6
=	Opérations algébriques sur les objets raster	4.5.1
values()	Accès aux valeurs numériques d'un raster	4.5.2

hist() cellStats() calc() cor() zonal()	Histogramme des valeurs d'un raster Paramètres statistiques d'un objet raster Calcul de paramètres statistiques à l'échelle du pixel Calcul d'une corrélation entre 2 rasters Statistiques zonales	4.6.1 4.6.2 4.6.3 4.6.4 4.6.5
reclassify()	Reclassification	4.7
rasterize()	Conversion d'une couche vectorielle en couche raster	4.8
distance()	Calcul de la distance euclidienne par rapport aux cellules non vides d'un raster	4.9
rasterToPoints() rasterToPolygons()	Conversion couche raster → couche de points Conversion couche raster → couche de polygones	4.10.1 4.10.2
extract()	Extraire de l'information d'une couche raster (→ points ou polygones)	4.11
terrain() hillshade()	MNT → couches de pente ou d'exposition MNT → couche d'ombrage	4.12.1 4.12.2
gdalbuildvrt ()	Construction de rasters virtuels	4.13